A Methodology for Formal Hardware Verification, with Application to Microprocessors

Derek Lee Beatty
August, 1993

CMU–CS–93–190

Computer Science Department
Carnegie Mellon University
5000 Forbes Avenue
Pittsburgh, PA 15213-3891 USA

Thesis committee:
Randal E. Bryant, Chair
Edmund M. Clarke, Jr.
Allan L. Fisher
Carl-Johan H. Seger (Univ. of British Columbia)

This research was sponsored in part by an NSF fellowship and in part by the Avionics Laboratory, Wright Research and Development Center, Aeronautical Systems Division (AFSC), U. S. Air Force, Wright-Patterson AFB, OH 45433–6543 under Contract F33615–90–C–1465, Arpa Order No. 7597.

Keywords: computer-aided engineering (CAE), computer-aided design (CAD), VLSI design and validation, formal hardware verification methodologies, hardware description languages (HDLs), specification techniques, pre- and post-conditions, computation by abstract agents, string functions, marked strings, automata theory, discrete-event simulation, switch-level simulation, symbolic simulation, COSMOS symbolic switch-level simulator, partially-ordered system models, trajectory evaluation, symbolic computation, algebraic manipulation, binary-decision diagrams (BDDs), verification methodologies, microprocessor validation, Hector microprocessor

Do not copy this page!

Replace this sheet with the signature page before copying.

Abstract

Microprocessors are now ubiquitous, but their design is not without difficulties. Numerous microprocessors have been introduced only to find—sometimes years later—that they contain mistakes. The need for better ways of checking designs is clear. This research develops a methodology for formally verifying data-intensive circuits against abstract state machines defined by assertions, and applies it to show that a microprocessor circuit implements its intended instruction set.

"Implementation" is a formal relation between input-output sequences: of an abstract state machine, and of a circuit. This simple abstract model of behavior is reconciled with concepts from digital-systems design including dynamic logic, pipelining, multi-phase clocks, and a separate memory system. The methodology structures proof of implementation using a series of decompositions. It exposes internal system state. It dissects sequences into their component transitions (using a new formalism called marked strings to reason about overlapped operation). For processor verification it also abstracts away most of the memory state. Making these simplifications from global I/O behavior once, in the methodology, allows the focus when verifying a circuit to be on the localized effects of single state transitions.

Verifying a circuit then consists of a few steps. The user specifies intended behavior by giving high-level assertions denoting state transitions (much like a Hoare logic of an assembly language), plus an additional mapping of abstract state onto circuit state incorporating the temporal movement of state through pipeline registers. The verifier makes consistency checks on the state mapping, then simulates the circuit symbolically using patterns generated by mapping the state transitions onto the circuit. A circuit passing these tests correctly implements the abstract state machine. The verifier is based upon reduced, ordered, Binary-Decision Diagrams (BDDs), and operates at the switch level, but these details are not required by the methodology.

The verifier has been used to check initialization, response to interrupt, and execution of several instructions of a pre-existing, 16-bit nMOS microprocessor called Hector, which is roughly similar to the DEC PDP–11. The specification was hand translated from a pre-existing instruction-set simulator, and the circuit was extracted from a description of the mask geometry used for actual chip fabrication.

in memory of
Charlotte Money Beatty
and of
Clinton Douglas Hester

Acknowledgements

Although a doctoral thesis is the work of an individual, only the help of many others makes one possible. The department has been a great place to work, and the facilities staff kept equipment running and files backed up (and responded promptly [on a holiday!] when I thought `sup -d` meant "debug" and it really meant "delete"). I cannot overstate the help of my committee: my advisor Randy Bryant, and Ed Clarke, Allan Fisher, and Carl Seger. Randy, particularly, steered me toward the important problems associated with the methodology, and away from diversions too numerous to name. Had he not convinced me to throw out my original theory, the middle of this thesis would have been much longer (!) and its proofs more difficult. If any elegance can be found in this work it is due to his guidance. Ed suggested the pictures that accompany many of the proofs. Carl's enthusiasm for doing things right saved me from several errors. The opportunity to visit him and Sheila for a week during Randy's sabbatical (despite their own move) was a high point of that year. Allan encouraged me when I needed encouragement, and advised on other endeavors as well as research.

I owe much to others as well. I learned too late to keep a list of names. So although I thank colleagues and friends far and near: Peter Andrews, Dimitri Avresky, Joel Bartlett (and Digital Equipment Corporation), Soumitra Bose, Chris and Aksel Bothner-By, Jerry Burch, Karl Brace, Kyeongsoon Cho, Luc Claesen, Kevin Deak, Masahiro Fujita, Duke Groebe, Nagisa Ishiura, Mark Horowitz, Larry Huang, Sea-Way Jan, Jeff Joyce, Manpreet Khaira, David Long, Roland Luthi, David Maynard, Ken McMillan, Manish Pandey, Carl Pixley, Richard Rami, Rick Rudell (and Synopsys), Carlo Sequin, Tom Sheffler, Bebb Wheeler Stone, Victor Yodaiken, Gary York, Jerry Warren, Dan Weise, Judson Wiley III for their individual contributions, I must apologize to those I have surely forgotten, and to all mentioned for an alphabetical list instead of specific thanks. My office mates, Richard Wallace, Olin Shivers, Allan Heydon, Hank Wan, Herman Herman, and Nina Zumel, put up with my quirks. I honed many arguments against the stone of their skepticism. Special thanks go to Tom Miller for giving me access to the Hector design, and for permission to reproduce Figures 8.1, 8.3, and 8.2. While typesetting this thesis my favorite book was Paul W. Abrahams' *TEX for the Impatient*.

Don Hester was my first mentor, and deserves special thanks for his faith and advice over half my life. It was a privilege to have been a friend of his son Clint. My father Lee has always encouraged me, as did my mother Charlotte during her life, and now in memory.

Finally, to someone who too often came last as I conducted this work I owe my greatest thanks: my wife Cheryl Dragel, who made time for me despite law school and the bar exam. Her love sustained me during the dark times, and made life sweeter during the good.

Summary Contents

Contents

List of Figures

Part I

Preliminaries

Chapter 1

Introduction

Microprocessors are complicated artifacts. When a new microprocessor is introduced, it contains design errors. No manufacturer is immune. The need for better ways of checking designs, before they are fabricated, packaged, and sold, is clear.

This thesis addresses the processor correctness problem and provides a step toward a solution. It considers the following issues:

- the question of an appropriate criterion for correctness,
- how to specify the instruction set that a processor is supposed to provide,
- how to expand such a specification to include details of a particular circuit implementation, and
- how to check such an expanded specification using a symbolic simulator.

The goal of the work described here is to develop a methodology for formally verifying pipelined hardware that can accommodate existing practice. This requires that the methodology make use of existing principles and models, formalizing existing informal usage when possible. The touchstone of many hardware verification attempts in the past has been a microprocessor, so the methodology has been developed with microprocessor verification in mind.

The design of a microprocessor is a bridge between software and digital circuit theory. Consequently, a methodology for verifying a microprocessor must encompass some formalism from each domain. From the software domain we adapt the notion of axiomatic semantics—that the meaning of each statement in an imperative programming language can be described precisely by an assertion giving the assumptions needed in order to execute the statement, and the guarantees established by executing it. From the hardware domain we adapt a circuit model.[1]

[1] We have chosen the switch-level model. This reflects our heritage. The choice of models is somewhat arbitrary. Insofar as the methodology is concerned, a gate-level model would have worked as well.

The key requirement of the circuit model is that it be a model within which those circuits we wish to consider—in this case microprocessors—are actually designed.

Also from the hardware domain we recognize the notions of timing diagrams and of pipelining. Formalisms that play the role of or express these concepts will play an important role in our methodology. Finally, from the hardware domain we adopt a unifying model of computation: Moore machines.

The methodology itself is carefully designed so as to properly account for several subtleties of reasoning about circuit behavior. Nonetheless, many of the key arguments are made at an abstract level, The thesis demonstrates the validity of the methodology by applying it to a real microprocessor, Hector, which I did **not** design. However, the main topic of the thesis is the methodology rather than the case study. A "methodology" is a collection of methods, rules, and principles.

The Hector microprocessor was designed before the research described here commenced, and was in no way designed with formal verification in mind. Hector was intended to be a realistic pedagogical example of a microprocessor. A more recent version of Hector (not treated in this thesis) has been proposed as part of a data collection and telemetry system to be implanted into experimental animals. Although not a particularly modern architecture, Hector serves well as an example for verification, because of its size and availability. Despite its comparatively small size, it poses significant challenges to verification. Its circuit design was not deliberately made simple to facilitate verification. Verifying its complex instruction set involves such problems as interactions of addressing modes, and a variable-length instruction encoding. The specification of Hector was derived from an instruction-level simulation of Hector written in the C programming language. The switch-level model of Hector was extracted from the mask geometry used to fabricate the actual chips.

The goal of verification is to show that two descriptions of a system are consistent. One description, called the "specification," is taken to be correct. The correctness of a second description, called the "realization," is then checked. If the realization is correct (with respect to the given specification!), then we say that the realization "implements" the specification. Some authors refer to the realization with the term "implementation." I avoid this, so as to reserve the term to denote a formal *relation* between two machines.

1.1 Overview of the thesis

This thesis consists of three main parts, with two appendices.

The first part is introductory. Chapter 1 is this introduction, which describes the problem treated, as well as the thesis itself. Chapter 2 is a "manifesto" giving intentions and motives by way of an extended example. The example is necessarily treated at a superficial level, but it includes cross references to more detailed treatment further in the thesis.

The next part develops the methodology. Each of its chapters has both a practical and a theoretical content. Chapter 3 treats the specification of the behavior of data-intensive systems, in particular, processor instruction sets. It also illustrates the way in which such specifications can be viewed as nondeterministic state machines. Chapter 4 describes the model of circuit realizations used. It is a generalization of the switch-level model for digital MOS circuits, which is an accepted model in the design community. This chapter also shows that such circuit models can be viewed as nondeterministic state machines. Chapter 5 discusses the notion of correctness. The ultimate notion of correctness is fairly simple, but accuracy and intuitive appeal were not sacrificed for additional simplicity.

Up until this point, the chapters may seem somewhat disconnected, but chapter 6 pulls its three predecessors together into the heart of the thesis. It proves the central theorem of the thesis, the Overlap Theorem, which shows shows how a specification can be mapped onto a circuit simulation model to yield patterns that can be efficiently checked. A circuit passing such checks is correct according to the previous definition of implementation.

Chapter 7 discusses several issues that must be addressed in order to apply the methodology, including a way of mapping symbolic statements about instruction sets into symbolic simulation patterns. It also discusses the decomposition of a computer into its processor and its memory system. Instead of verifying an entire computer consisting of a processor and a memory system, our goal is to verify a processor. To do so, we assume the memory system system correct and check only the processor. This greatly reduces the amount of state information that must be considered, and also allows the claim that this thesis treats *processor* verification. While this is not properly within the core of the methodology, it is a key to making it practical.

The last part concludes the thesis with a case study. Chapter 8 describes the Hector microprocessor design and discusses its formal specification. Chapter 9 describes the Hector chip, the mapping of the formal specification, and the verification of Hector instructions. Chapter 10 contains a summary of this research and its successes and failures, and outlines directions for continued work.

In addition to the body of its text, this thesis contains two appendices. Appendix A consists of an axiomatic development of an algebraic structure called **marked strings**. The key idea in this thesis is that abstract states are represented in real systems by *intervals* of computation, which can be overlapped in a carefully controlled way. Marked strings provide a mechanism for formalizing the overlap.

Appendix B contains a specification for the Hector microprocessor, typeset directly from the input files for the verifier.

Discussions of related work appear throughout the thesis. A general survey of hardware verification, and a brief synopsis of microprocessor verification, appears in Section 1.5 (p. 9). Discussions of hardware description languages, and of assertions, appear in Section 3.5.2 (p. 95). Discussion of the switch level model appears throughout Chapter 4. Its application to formal modeling is mentioned in Section 4.3 (p. 106). Works on abstraction, and notions of just what the term "implementation" should mean, are discussed in Section 5.3 (p. 120). The relation of this methodology to Hoare logic is explained in Section 6.5 (p. 140). The Hector microprocessor, and processor specification, are discussed in Section 8.4 (p. 204). Finally, a more detailed survey of microprocessor verification appears in Section 9.5 (p. 228).

1.2 Microprocessor design errors

Newly-introduced microprocessors contain design errors. Early, hand-designed processors contained errors, and even modern computer-aided designs of great economic importance continue to have errors.

Today, the "bug lists" accompanying microprocessor designs are usually closely guarded industrial secrets,[2] though such was not always the case. The engineering prototype of the Motorola 68000 processor was accompanied by a list acknowledging 12 bugs and 3 "architectural enhancements." Some are simple details, such as flag bits set improperly, or exceptions that are improperly enabled or disabled. But they include a vague, ominous warning that when the `TAS` (test-and-set) instruction occurs with certain addressing modes, the processor "does not execute the instruction stream properly."

The desirability of a more reliable means to check microprocessor designs before fabrication is quite evident from bug reports in industry newsletters [97, 98]. Though the 486 microprocessor's bugs have gained notoriety recently, this problem is long standing. The 6502 microprocessor used in early personal computers suffered an instruction-fetch bug. It was never corrected; programmers had to take care to avoid it. National Semiconductor's `PACE` design, one of the first 16-bit microprocessors, never saw widespread use. It is rumored that this was because it was so buggy. Subsequent microprocessors—such as the Z8000 and 32016—were buggy for years. The 88000 had floating-point problems for more than a year; it was recently announced that there remains only one significant bug in this chip (excluding clarification of its bus interface). Bugs in the R3000 and the 29000 have been published recently. Though the 32GX32 is a derivative of the 32532, which was available for several years; five bugs were known to remain in 1990, all in its debugging facilities.

At least one microprocessor manual includes legal language prohibiting use of the device in surgical implants and life support devices [189, title page].[3]

[2] Although MIPS published the R4000 bug list [184].

[3] "[Our] products are not authorized for use as components in life support devices or systems

Modern microprocessors are not immune. For example, the Motorola 68040 [96] had at least three known problems. First, the test equipment used to test the fabricated devices could not test to the full current spec of the chip's drivers, so there is no confidence that they can indeed source or sink the specified amount of current. This requires that the circuit design surrounding the processor in a system be more conservative than would otherwise be required, to guarantee proper operation. Second, the cache snooping logic gives erroneous information for an instruction-cache fill. (Fortunately, there is little need to snoop the i-cache, since multiprocessor algorithms based on self-modifying code are quite rare!) Finally, the `MOVE16` instruction, which is supposed to copy a 16-byte value, does not work. It is unclear which, if any of these errors have been corrected in later revisions of the design.

Intel Corporation's microprocessors suffer similar problems. For example, early versions of the 386 microprocessor would stop if a DMA occurred during a floating-point instruction. The last rumored bug found in the 386 allegedly came to light in October 1990, though I have been unable to document this. It is said that early i860 chips even had a bug in a shift instruction.

The superscalar SPARC processor was released later than anticipated due to debugging difficulties.

1.3 Brief catalog of some recent microprocessor bugs

A glance at the trade and popular press reveals dozens of articles on microprocessor flaws [80].

A brief synopsis of the articles underscores the importance of the problem.

Intel Because of its prominence and importance, Intel Corporation is most visible. But Intel's share of problems is only proportionate. The publicity surrounds two floating-point bugs in the Intel 486 microprocessor, discovered by Compaq in November 1989, a bus interface bug discovered in January 1990, and some thermal problems in a high-speed version, in August 1991.

July, 1989 Engineers at Alsys (a UK corporation) claim to have found a bug in the 286 processor; Intel disagrees.

---

intended for surgical implant into the body or intended to support or sustain life. Buyer agrees to notify [us] of any such intended use use whereupon [our company] shall determine availability and suitability of its product or products for the use intended."

September, 1989 There are signs of possibly serious bugs in the D-step 80386 [220].

October, 1989 IBM ships a 486 upgrade board; Intel admits bugs but claims that they do not affect applications [62]. Compaq finds bugs in the Intel 486 [110, 140, 248]; the news is reported in major newspapers [170, 252]. Many companies' product plans become uncertain [215]. Intel says the bug will cause no delays in production, and rushes a new metallization mask to its fabrication facilities [206].

November, 1989 A magazine warns its readers to avoid the early 486 chips [255]. System designers report that the bugs in the 486 cause problems. One company (in health care!) plans to use 486 chips despite bugs. The 486 bugs affect the stock prices of Compaq, which delays products because of the bugs [164]. Compiler writers wait for the bugs to be fixed [143]. A respected industry analyst condemns vendors who announce 486-based computers while known bugs remain in the chip [214]. Intel says it has fixed the 486 defect [153].

December, 1989 Intel plans free replacement of buggy 486's [122]. IBM announces a 486-based PS/2, having previously pulled its 486 upgrade for the PS/2 off the market, due to bugs [169]. Intel resumes 486 shipments [200, 218].

January, 1990 In the previous year, many 486 systems have been announced, but few shipped, due to the bugs. A *new* bug is discovered in the 486, related to interrupts and the bus interface; and makes the papers, though only minor delays are expected [193, 250].

February, 1990 More vendors announce 486 boxes despite bugs [225]. The bug is revealed: an EISA chip set runs some bus cycles in wrong order [223]. The 486 flaw is said to be "minor."

March, 1991 Longtime rival AMD claims that Intel's low-power 80386SL is full of bugs.

April, 1991 In an interview, Intel engineers identify locating bugs as one of their chief problems [63].

May, 1991 There is more discussion on bugs in the low-power 80386SL, which had been announced the previous October.

August, 1991 The 50MHz 486 sometimes overheats. Intel stops production of the part [171, 70, 104, 224, 249].

September, 1991 There is more discussion of the Intel 486 testing problem [38] which is delaying products [111].

November, 1991 Zenith is said to be pleased with Intel's 386SL, despite its bugs [152].

July, 1992 Intel delays debut of "P-5," its new microprocessor, to refine its process but also to have more time to find bugs [221, 99, 251].

Motorola

September, 1990 The M68040 is delayed for a "few," "innocuous" bugs [195]. Sun and Intel gather market share due to Motorola's delay [253].

November, 1990 Volume shipments of the '040 finally begin [216].

January, 1991 Motorola claims that production has reached parity with demand [230].

March, 1991 Two full years after the 68040 was announced, ramp up is underway [7].

Others

September, 1990 A bug in the TI TMS38053 "Falcon" microcontroller is claimed to be responsible for problems with a token-ring network [138].

July, 1992 The Inmos T9000 is finally imminent, after having been delayed due to a pipeline control flaw [190].

1.3.1 Relation to this thesis

Clearly, avoiding design errors is the ideal. In its absence, better methods of catching processor design errors early are desirable. In fact, since compilers may be increasingly using instruction sequences in ways not envisioned by the machines' architects [121], it is becoming even more important that all aspects of instruction behavior be rigorously checked.

The work described in this thesis is a step along the road toward bug-free microprocessors. The pages that follow develop and explore a methodology for verifying functional properties of microprocessors. Some of the bugs that plague microprocessor designs are beyond the scope of this work. There are many potential errors which are not apparent when modeling a system at the switch or higher level. For example, the Intel overheating bug is not a functional property. Sometimes errors are induced by capacitive coupling between transistor active areas and overlaying metallization. However, functional errors are important to detect because they are design errors that will affect every instance of the design.

1.4 Verifier errors

Any kind of test is subject to four possible kinds of errors. Two are particularly important. A bad verifier might indicate that a bad circuit is correct, or a verifier might indicate that a good circuit is incorrect. Alternatively, a verifier might indicate that a good circuit is correct, or that a bad circuit is incorrect, but do so for the wrong reason. The first two types of errors can be termed "false positives" and "false negatives," respectively. Ideally, none of these errors will occur. However, if a verifier is based at any stage upon a proof procedure which is not complete (in the logical sense), then it will be subject to false negatives. It can be worthwhile to tolerate occasional, rare false negatives if there is a corresponding tradeoff that yields improved performance. A verifier that is subject only to false negatives is conservative. That is, when it says that a circuit is correct, the circuit is indeed correct.

On the other hand, a verifier that is subject to false positives is a catastrophe. It cannot be relied upon. When it says that a circuit is correct, we do not know whether the circuit is actually correct, or whether a false positive has occurred. False positives must be avoided.

The remaining two types of errors are less problematic. If a verifier indicates correctness, but for the wrong reasons, no harm is done. If a verifier indicates incorrectness for the wrong reason, we may waste time in trying to debug a circuit, but at least we will not have committed the blunder of claiming that a buggy circuit has been formally verified.

We will be careful to develop a conservative methodology.

1.5 Related work

The Soviet telephony engineer A. K. Kutti [156] made perhaps the earliest attempt to distinguish the specification of the intended behavior of a sequential circuit from the design of the circuit itself. He specified behavior by state tables. Unfortunately, his work was obscure and it was decades before it was re-invented independently [186].

1.5.1 "Verification"

In the contemporary CAD (computer-aided-design) community (academic as well as commercial), the term "verification" does not always denote an entirely rigorous approach based on a formalism.

For example, Ernst and Bhasker [100] describe a system called *Satya* that can be used used to "verify" a high-level synthesis system, under a set of restrictions (including that there is no pipelining). The goal of the system is to show equivalence between an "algorithmic-level" description and one at the logic level. The heart of *Satya* is simulation driven by a random-pattern generator. The algorithmic-level description is annotated with the output assignment generated by the synthesis system in order to facilitate comparison.

1.5.2 Formal hardware verification

There are a number of surveys of formal hardware verification techniques [68, 67, 124, 123, 72]. McFarland's tutorial [172] contains a number of examples from early techniques, but its coverage of more recent work in this fast-moving field is unfortunately weak. Work related to model checking is condensed in [75]. Leonard surveys computer specification in general [159]. Yoeli's tutorial collects a number of classic papers [95]. There has been some side-by-side comparison of different theorem-prover techniques [6, 229, 228].

Combinational verification

There has been much work on the verification of combinational circuits. Many approaches consist of recursive algorithms based on Shannon expansions (e.g., [126]). Many such older algorithms are now subsumed by graph algorithms on BDDs. BDDs (reduced, ordered binary-decision diagrams) are a canonical, graph representation for Boolean functions [48, 54].

There has been a temptation among some to claim that it is straightforward to extend techniques suitable for verifying combinational circuits to sequential circuits. Such claims should generally be viewed skeptically, but there are successful cases, such as when the latches and timing of the specification and realization agree [18].

Verification of combinational logic has been used for years within IBM. Roth [208] described a verification strategy for clocked designs. Automatically synthesized designs were compared with hand-optimized designs using an algorithm that exploited circuit structure. Roth later reviewed IBM's verification efforts on the 3081 design [207], and estimated that 8.5 years were saved by using logic synthesis and comparing the synthesized logic with optimized logic produced by hardware designers.

Grammar-based approach

A number of researchers have taken approaches based on graph grammars [106, 9, 3]. Such approaches generally restrict the set of possible circuits to those that can be generated by the grammar, and are limited to combinational circuits. They can also sometimes be used to check adherence to electrical design rules [21].

Model checking

Model checking is a powerful approach for verifying certain classes of circuits. Properties are expressed in a logic. A system is verified against such a specification by checking that it is a model (in the logician's sense of the word) of the specification formula.

Bochmann [20] proposed analysis of circuits using temporal logic. Clarke and his colleagues [73] have advocated model-checking and developed powerful algorithms and implementations. McMillan [173] developed the symbolic version. Coudert [85] developed a symbolic model checker for a restricted class of formulas. Bose and Fisher [26] developed a symbolic model checker based on a Cosmos switch-level model. Their modeling approach was to introduce a state variable for each input and each storage node in the network, and thereby construct a complete model of an excitation function, so unlike the work in this thesis, there was no partitioning of the specification into assertions.

Clarke and colleagues sketch symbolic model checking in [73]. Specifications are given in a powerful branching-time temporal logic known as CTL (computation tree logic). CTL contains both path formulas, which express properties of computations, and state formulas, defined relative to individual states. A formula is valid in a model if it is satisfied by all starting states of the transition system.

The basic idea behind CTL model checking is to "unwind" the state-transition graph of a system into an infinite tree. Each node of this tree then represents a unique state at a unique time. Properties that involve time can then be phrased as graph problems over this tree, and computed with graph algorithms such as reachability analysis. In the symbolic formulation, the checking algorithm is recursive in the formula structure, and is a series of fixed point calculations on relations expressed as BDDs.

By encoding the labels on states using Boolean variables, transition relation can be represented by BDDs. The model checking problem for CTL over labeled transition graphs can be solved by an algorithm that is basically a series of fixed-point calculations on relations expressed by BDDs, recursive in the formula structure. Incorporating fairness constraints, which allow specification of some additional properties, require only a slight modification of the basic procedure.

Because CTL is a powerful logic, CTL model checking and its extensions can be used to verify properties such as the absence of deadlock that cannot even be described in the limited specification notation developed in this thesis.

CTL model checking has been used to verify a simple pipelined data path. However, it has not concentrated on developing a general methodology for verifying pipelined systems. In the data-path verification, the ostensible specification is a register-transfer language, but it was significantly transformed during the verification. Temporal operators appeared directly in the specification, yielding path formulas, by taking a register-transfer specification and producing a "temporal interpretation for RTL specifications." This transformation is textual, and was not checked. After the transformation, some equivalences, which are also expressed as CTL formulas, were checked to ensure that a set of substitutions was safe. Performing the substitutions yielded a set of equivalent state formulas. The model-checking procedure was actually performed on these state formulas.

It is conceivable that by formalizing the RTL, such work could be cast in the framework developed in this thesis, and the transformations could then be viewed as implementation mappings.

Trace theory

Dill [93] developed a form of trace theory as a basis for verification of speed-independent asynchronous circuits. Burch [61] did some subsequent work, generalizing the mathematics. In trace theory, behaviors are considered to be traces, or sequences of events, where an event is a signal transition on a wire. Systems are modeled by sets of traces. Dill's model of computation is a trace structure, which includes both successful and unsuccessful or failure traces. The two sets are not disjoint because it is possible to hide symbols and thereby destroy the distinction as to within which set a trace belongs. Less abstractly, the failure traces are necessary to reject cases where the environment in which a circuit is used provides inputs too quickly for the circuit to respond.

Since the work in this thesis is intended for synchronous systems, and models valuations rather than transitions, the need to make such a distinction does not arise.

Dill's notion of "conformation" is analogous to our notion of implementation.[4]

Dill requires that trace structures be receptive—that a circuit cannot constrain the inputs that it receives from its environment. Again, we have nothing precisely the same because it is not applicable to our synchronous model, but one of our requirements—that mappings from specifications to realizations be surjective on inputs—is similar.

[4] Dill chose "conforms to" to avoid the lattice-theoretic technical term "meets" or the logic-theoretical term "satisfies." The notion of conformity developed here is unrelated; I chose the word for its geometric sense in imagining the overlap of timing diagrams.

State machine comparison

Not all researchers work with logic. To some, "formal verification" denotes the comparison of finite-state machines. There are standard algorithms for comparing finite-state machines [134]. Equivalence of two machines can be checked by forming a product machine. In essence, the inputs of the two machines are connected in parallel, and the equivalence-checking procedure searches the state space for reachable states in which the outputs disagree. The search may be conducted breadth-first or depth-first.

For example, Supowit and Friedman [232] describe a method for verifying the functional equivalence of two different sequential circuits. The circuits need not have the same number of inputs or the same latency. The criterion for correctness is essentially input-output behavior, with each circuit starting in a known state.

The method operates by constructing a series of generalized automata. The first automaton is the original circuit. In the second automaton, one transition represents a single application of the circuit's real inputs. The third automaton represents an application of the circuit's conceptual inputs. For example, a bit-serial adder circuit might have two real inputs, but when operating on 8-bit words, it would have 16 conceptual inputs. Once this final automaton has been generated for each of the two circuits, the automata are compared using a variant of a standard algorithm. As originally described, this technique handles circuits with only a single-bit output. However, Corella generalized it [81].

Recent work on state-machine comparison has focussed on symbolic representations. Those based on BDDs are most successful. Coudert [84] presents a symbolic formulation of the product-machine construction, which represents sets of states using the images (codomains) of functions which are represented as BDDs. Touati and colleagues [234] presents heuristics that improve performance.

Hwang and Newton [139] verify state machines composed of gates and latches against specifications described as transition tables by finding a circuit state that covers the starting state of the specification. The notion of a cover is defined analogously to the combinational notion used in logic minimization.

Theorem proving

In contrast to those who have abandoned logic to work solely with models, others work within logic. To some, "formal" denotes "mechanized." While such work has strong mathematical rigor, there is a danger that complicated logical statements whose validity is mechanically established by such mechanized proof will bear little relation to circuit behavior. For example, it is possible to "prove" that a "gate" which connects its inputs together and its output to ground (see Figure 5.2, p. 114) implements an exclusive-OR function.[5] With extremely complicated constraints and implications, this becomes a difficult and subtle problem [240].

The two most common contemporary hardware verification approaches based on theorem proving are to work within either first-order logic using the Boyer-Moore proof system, or to work in higher-order logic using either Gordon's HOL system or another approach.

The original use of theorem proving to verify hardware was Wagner's thesis [238]. Hanes [128] described a system typical of early verifiers based on theorem provers. Functional specifications were written in a programming language. These descriptions were translated into a notation used by a logic design system, in which the circuit design was also specified. A theorem prover based on term rewriting then applied equivalence transformations to show the two representations equivalent.

Barrow [11, 10] was among the early researchers to consider formal verification of VLSI circuits. He acknowledges an "intellectual debt" to Michael Gordon. Many ideas found in Barrow's system "VERIFY" appear in others' later work. The problem he addresses is correspondence between state machines, which may be related by either identity (but expressed by equations in different form), structural homomorphism (i.e., there is a mapping from variables of one machine to those of the other), or behavioral homomorphism (mapping from state sequences of one to state sequences of the other). Barrow's goal was to produce a useful system dealing with real designs.

Srinivas and Agrawal [226] described a prolog-based system similar to VERIFY.

Goguen [114] has used a functional programming language called OBJ as a theorem prover. His approach to hardware verification is similar to that used in HOL, but he must use extra equations in place of existential quantification. Other uses of functional languages that have associated proof systems include the verification of Cayuga by Bickford and Srivas [227].

HOL One of the better-known proof systems in use for formal reasoning about hardware is HOL (sometimes pronounced "hole"), a mechanization of typed higher-order logic. Typed higher order logic is sometimes known as simple type theory [5] and is a strongly-typed lambda calculus. This logic is mathematically foundational, in the sense that a large portion of mathematics can be formalized within type theory. Foundational theories are good frameworks for developing systems because they have been designed for flexibility.

Hanna and Daeche [129] were apparently the first to note the utility of mechanized higher-order logic for reasoning about hardware. They have demonstrated

[5] If the correctness condition is phrased as "when the inputs differ the output is high," then the antecedent of the implication can never be satisfied by the circuit, hence the entire implication is valid.

detailed reasoning about the implementation of an edge-triggered flip-flop using gates. More recently they have proposed a system based on a different formulation of type theory [130]. Mike Gordon has been an enthusiastic proponent of this approach. He and his colleagues have generally concentrated on reasoning at much higher levels of abstraction than the gate level.

The HOL system was developed by Gordon [118, 115, 116] based on an earlier system called LCF-LSM. HOL has seen significant use in the research community [144, 65, 64, 92, 146, 161, 148, 147, 162, 176, 71, 120, 149, 245, etc.].

The HOL proof system is guaranteed to be sound by the type system of its implementation: a theorem is a type in a strongly-typed language. The only way to construct an object of type "theorem" is to invoke a procedure that corresponds to one of the inference rules of the logic.

As its name implies, HOL allows the manipulation of "higher-order" objects—e.g., functions which operate on other functions. This allows time-varying signals to be modeled as functions from the integers (which represent points in time) to signal values. Hardware elements can then be described by predicates over signals (i.e., predicates over functions from integers to values). For example, an inverter with unit delay would be defined by the equation

$$\mathrm{Inv}(i, o) = \forall t_{\blacksquare} o(t+1) = \neg i(t)$$

where i and o are functions from integers to truth values, and $\neg$ denotes logical negation.

Internal structure can be hidden by existential quantification. A buffer constructed of two inverters in series could be defined by

$$\mathrm{Buf}(i, o) = \exists x_{\blacksquare}\, \mathrm{Inv}(i, x) \wedge \mathrm{Inv}(x, o)$$

which effectively hides the point x of series connection.

Behavior can be described in a similar way using logical predicates. Proving correctness consists of showing that one conjunction of logical predicates (which describe circuit structure) implies another conjunction of predicates (which describe desired system behavior), e.g., $\mathrm{Imp}(a, b) \supset \mathrm{Spec}(a, b)$.

Several abstraction techniques can be expressed within this logic [175, 176]. Quantification to hide structure can be considered an abstraction. Data abstractions are dealt with by abstraction functions.

Temporal abstractions are dealt with by mapping abstract time points to points of detailed time, by defining a monotonic function f on the integers. This function can be composed with functions denoting signals to perform temporal abstraction. For example, using this mechanism, the proof condition above becomes $\mathrm{Imp}(a, b) \supset \mathrm{Spec}(a \circ f, b \circ f)$. Such temporal abstraction functions may be constructed by using the signals of the implementation. For example, a temporal abstraction function might be defined so as to "pick out" the output of a system only when an "output valid" signal is asserted.

This approach is usable for systems where the timing relationship between the realization and its specification is one of granularity—for example, to show that a microcoded machine implements a computer whose timing is given in terms of atomic instructions. However, it is far from obvious how this might apply to pipelined systems. Considering the limitations of this approach to temporal abstraction led to the inspiration for the marked strings in this thesis.

Since the HOL system is a very general approach based on logic, without recourse to system models themselves, there is no clear methodology which could establish the validity conditions necessary for a proof to be a meaningful statement about hardware. This lack of a methodology is a drawback of the approach. For example, a simple microprocessor that was "verified" using HOL lacked a reset signal.[6]

The most common use of HOL sometimes seems to have been verifying versions of Tamarack, but there is actually a wider variety of work. For example, Dhingra [92] used HOL to provide a formal basis for a set of "rules of thumb" for dynamic CMOS circuit design, and exhibited a digital PLL (phase-locked loop) as a case study. He identified the formal relation of the switch level to higher levels of hardware description as an area for future work.

Boyer-Moore (Nqthm) The other popular contemporary research into verification using theorem provers is based on the Boyer-Moore prover, a mechanization of quantifier-free first-order logic. This logic is simpler than higher-order logic, and consequently its mechanization is more of a theorem prover than HOL, which is more of a proof assistant. One comparison of HOL and Boyer-Moore found the entire process of using Boyer-Moore was much faster than that of using HOL [6]. On the other hand, another concluded that the HOL approach was more attractive [229, 228].

Since the Boyer-Moore logic is a first-order logic, and sequences are modeled within HOL as higher-order entities, it is not possible to model sequential behavior within Boyer-Moore using the style of HOL. Instead, sequential behavior is described by writing recursive functions. Most of the parameters of each such function represent system state, and recursive invocations represent the updating of the state variable. Each function also has a parameter which represents time (e.g., as a list of all the points in time for the future), and the recursive invocation progress through time (e.g., by deleting the first element of this list). A proof of correctness consists of showing the equivalence of two functions, one of which has less state (i.e., the specification) than the other.

It is not easy to see how this approach could be used to deal with pipelining.

[6]It was fortuitous that the chip actually powered up in a meaningful state, and would work.

Possibly a pipelined realization could be expressed as a set of mutually recursive functions, where the mutual recursions represented the interaction of pipe stages. The unpipelined specification would then be a single recursive function. The proof of their equivalence would then require disentangling the mutual recursion, which might be quite difficult.

The Boyer-Moore prover has been used in several efforts at hardware verification, particularly by those most familiar with it [137, 17, 16, 25, 24].

Other theorem provers Other implementations of higher-order logic that have been used include Veritas+ [130] and Nuprl [13], which are based on intuistionistic type theory.

Another approach to hardware verification based on theorem proving is the use of a prover such as Clio [227], SBL [213], or OBJ3 [114], which were developed for reasoning about programs written in (lazy) functional languages, which can express (infinite) sequences directly.

Symbolic simulation

Symbolic simulation is an old technique, which became much more practical with the advent of BDDs. Darringer [88] presented symbolic execution as a program verification technique adaptable as a hardware verification technique. This style of symbolic execution consists mainly of tracing the possible execution paths, and Darringer applies it to microcode verification. He makes use of a simulation relation between two machines, and proves that machines started in corresponding states will always proceed in correspondence. He also shows how to apply symbolic execution to combinational logic verification, by building a gate-level simulator then simplifying the equations it implements.

Cory [82, 83] discusses the comparison of designs, expressed in a HDL, using "conventional" symbolic simulation. The practicality of such symbolic simulation was very limited at the time.

Bryant's introduction of reduced, ordered BDDs for symbolic switch-level simulation [40, 41] for circuit verification [42, 52, 47] renewed interest in symbolic execution; Bryant and colleagues have verified some simple circuits [45, 46, 15]. Reeves [205] also constructed a symbolic verifier based on these techniques. Hu *et al.* [135] describe a system for translating a higher-level specification notation into low-level BDDs, but they do not give a formal justification for the translation. This thesis was motivated by the lack of a methodology for establishing correctness conditions for such symbolic simulations.

Bull's industrial tool Priam [18] performs symbolic simulation of two descriptions written in a hardware description language called LDS. One description, provided by the user, serves as a specification, while the other, extracted from a circuit, serves as a realization. The latches and timing of the two descriptions must agree, and Priam checks the two descriptions for equivalence, by symbolic simulation using a form of BDDs. The chief difficulty in doing this is that the LDS language allows the possibility of conditional assignment statements. Priam solves this by maintaining both the value for each variable, and a validity condition that indicates the cases in which the value is actually valid. This technique is used not only to check that values are equivalent, but also that each signal is assigned a value exactly once, regardless of execution path. (This corresponds to the absence of conflicting drivers or floating nodes in the design.) Priam has been used within Bull to verify actual designs, and a VHDL-based version has been marketed.

Jain and Gopalakrishnan [142, 141] examined the use of parametric Boolean formulas in symbolic simulation. In their view of symbolic simulation, which they implement with Cosmos, circuit state is established, inputs are applied, and then outputs and state are checked. This will work for circuits which have no input constraints or state invariants, but potentially fails in their presence. To handle such constraints, they propose replacing the Boolean variables representing inputs and initial state with Boolean expressions which form parametric representations of the input values allowed by the constraints. Their present method of deriving such parametric representations is lengthy, and they are seeking better approaches. The methodology developed in this thesis can be viewed in part as a systematic way of deriving such parametric forms. We get parametric forms as a result of applying mappings to components of the specification.

1.5.3 Microprocessor verification

Several researchers have verified microprocessors. All have been designed with verification in mind. Here we give a brief overview of this work; a more detailed survey appears in Chapter 9.

FM8501 Warren Hunt verified FM8501, a microprocessor designed for that purpose, in his thesis. Crocker and his colleagues subsequently re-verified it. Hunt and colleagues have also verified a simple design called Kit, and layered a "short stack" of correctness proofs for compilers and programs above it [16].

Cayuga Bickford and Srivas verified mini-Cayuga, a simplified version of a pipelined processor. Sekar and Srivas verified a simplified version of Wirth's Lilith.

Tamarack Mike Gordon illustrated his early ideas on hardware verification using a simple computer which has been verified, usually using higher-order logic, several times, by Jeff Joyce and others [117, 11, 10, 144, 146, 150, 81, 158]. Elaborated versions of this design are known as Tamarack. Windley later improved upon the structure of the proofs [245, 244].

Viper The Viper microprocessor has received publicity as the "first verified microprocessor." It was not entirely verified because of funding constraints, and because the project had exhausted most of its research content. Nonetheless, Viper is claimed by its proponents and advertisers to be the "first verified microprocessor."

MTI

A processor called MTI has been verified as part of an evaluation of verification techniques [23]. The processor is not pipelined.

SECD

Graham [120] has verified an implementation of Landin's SECD architecture, which is designed to execute a functional programming language.

1.6 Contributions of this thesis

This thesis sets out a simple, intuitive notion of what it means to say that a circuit correctly implements a specification. It describes a new specification notation sufficiently powerful for expressing instruction sets, and sufficiently restrictive to allow efficient checking. It also presents a new methodology for verifying processor implements against such specifications. It also presents the formalization of the methodology, along with a new algebraic structure, **marked strings**, needed in the formalization. It also empirically validates its claims: a prototype tool implementing the methodology, the verifier, is applied to a real, pre-existing, 16-bit, microcoded, CISC microprocessor, modeled at the switch level, in order to verify initialization, an interrupt response, and execution of several instructions.

Chapter 2

Verifying some simple sequential circuits

The verification methodology presented in this thesis is complicated of necessity. Many layers of abstraction separate the semantics of an instruction set from the rectangles of a chip design. In order to explain the methodology, this chapter contains some examples. They are simple familiar circuits whose correctness is not in question. Hence, they can be used in a discussion that concentrates primarily on the methodology itself, rather than the circuit examples. Nonetheless, using examples keeps the discussion concrete.

This chapter should leave the reader with a good grasp of the strategy of the methodology. The mathematical details will be vague or nonexistent, and there will be no proofs, so the correctness of the approach may be in question. For the reader who insists on knowing some particular detail, there are cross-references to the more detailed chapters which follow. Thus, this chapter can also serve as an extended outline of much of the thesis.

This chapter presents two simple sequential circuits as motivating examples. The circuits were chosen for simplicity of illustration. The first is a simple latch, implemented several ways. The second is a stack circuit that can be found in several textbooks [174, pp. 72–75], [243, pp. 364–366], [191, pp. 245–249]. This stack has been formally verified [15].

The first section illustrates a latch. Despite the circuit's simplicity, it illustrates many concepts, including

- timing is determined by the operation being performed by the circuit, and consequently, clocks are just inputs,
- the notion of input conformability,
- the notion of marking the nominal beginning and ending of an operation, state storage defined relative to these marks, inputs defined relative to the

beginning mark, and outputs defined relative to the ending mark,

- circuit nodes which serve for both state storage and output,
- abstract specification of circuit behavior, and the structuring of a specification by assertions, and
- a variety of implementations of one specification.

The second section uses a more complex circuit, a stack. Most of the preceding concepts are also illustrated by this example, which also shows

- a more detailed example of input conformability,
- a more complete example of the methodology in action, and
- an example of system decomposition.

2.1 Latch

The simplest sequential circuit is arguably the latch. Despite this simplicity, there are many ways to implement latches. This section begins by describing an extremely simple nMOS latch[1], shown in Figure 2.1. Though somewhat contrived, it serves well as an illustration.

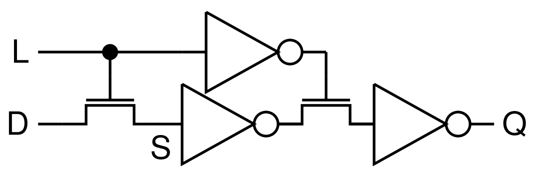


Figure 2.1: A simple latch

The latch has two inputs. A control input L is pulsed to load the latch. A data input D provides the data to be loaded. When the latch is holding data, the latched value is retained by the capacitance on node S. The latch has a single output, Q.

By taking an abstract view, we can say that a latch performs two operations: load and hold. Stylized timing diagrams for these operations on our circuit are shown in Figure 2.2. We will shortly explain the parts of the diagram in detail.

[1]suggested by Manpreet Khaira of Intel Corporation

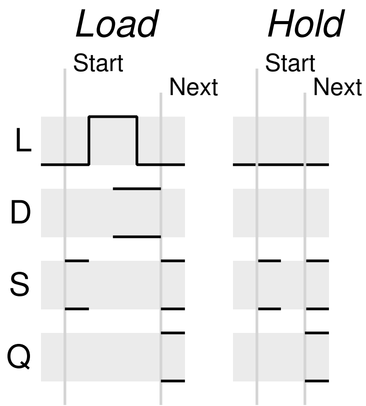


Figure 2.2: Timing diagrams for simple latch. For each signal, a double horizontal line indicates that either a high or a low value might be present; the absence of any line indicates that we don't know or don't care about the value.

Our approach to verification is through simulation. To verify a circuit, we will simulate it exhaustively, and check that it performs correctly in all possible conditions. We choose to simulate the circuit at the switch level, though this choice is pragmatic rather than fundamental. Assuming then that we know what it means to simulate a circuit, we must consider several issues.

- What do we mean by *exhaustively?* (And how can we hope to cover all these cases?)
- How do we establish these conditions in our simulation model?
- How do we check that the circuit's results are correct?

While considering these, we must keep in mind that our goal is *formal* verification. That is, our goal is to establish correctness results with the reliability of mathematics.

To answer these questions, we must examine the nature of circuits and of our simulation model of them. This will provide us with some of their properties, which yield requirements on our methodology of verification. We will begin this by considering the latch and its timing diagram more closely, looking at different parts of it in turn.

Our first observation is that the basic timing of these operations is defined by the control signal. Figure 2.3 repeats a portion of the original timing diagram.

From this diagram we can observe three things: timing is defined by the operation, clock signals are ordinary inputs, and in order to define timing in this way,

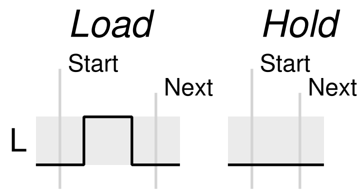


Figure 2.3: Timing of the control signal of the latch. Note that the two operations take different lengths of time.

the beginning and the ending of each operation must somehow be marked.

Each operation that the circuit is to perform has its own timing. In our example, the circuit can perform two operations. A load operation commences with the control signal low. The signal undergoes a transition to the high logic level, where it stays for some length of time before returning to the low level. In contrast, for the hold operation, the control signal simply stays low, and the circuit dictates no reason why this operation must have the same duration as the load operation[2]. The figure shows this operation as taking less time than the other one.

Since operations dictate their own timing, there is no need to distinguish clock signals from other inputs. In a synchronous circuit provided with a regular clock, circuit operations will have a fixed timing relationship with the clock signals. However, rather than saying that operations are defined by driving control signals to particular values on certain clock phases, we can just as well express the relationship of the clock signals to circuit operation. Doing so provides a strong advantage: clock signals are no longer treated differently from other input signals. This affords considerable simplification. Regardless of the clocking discipline we have a uniform way of describing clocking. Whether the clock is single-phase or 4-phase, we can treat all circuits alike.

Having gotten rid of clocks as the way of distinguishing different points in time, however, we must provide a substitute. We can do this most simply by identifying the *nominal start* and *nominal end* of each operation. In the figures here, we have indicated these with gray lines, labeled "start" and "next." To perform a sequence of operations, conceptually we align the *start* marker of each operation with the *next* marker of its predecessor.

In performing this alignment, we must ensure that when we align markers, we

[2]However, when the circuit is used in an actual system, the context in which the circuit appears might dictate that both operations take the same time. For example, the control signal might be a gated clock. Here the load operation would correspond to the presence of a pulse, and the hold to its absence, during the active phase of a regularly occurring clock. Still, the circuit itself is not required to be used in this manner.

also align signals. That is, if one operation requires that a signal be high at a particular time, we cannot allow another operation to require it to be low at the very same time. For the latch it is rather obvious that this condition is satisfied. A subsequent circuit example will begin to expose the subtleties of the requirement. This idea, that conflict must be avoided, will be formalized in our notion of input conformity.

The markers, two per operation, provide a convenient way to define the timing of each operation. The signals that comprise the operation can be identified relative to the start marker. The next marker can be placed relative to the start marker by noting the nominal duration of the operation.

Moreover, these markers provide a convenient way to define the timing of state storage. Consider the classical view of finite-state system operation. The system begins in one state, and is presented with some input. Then time advances in an atomic step, after which the system is in another (or possibly the same) state. We wish to reconcile this view, a simple abstract model of state transition, with the reality of state representation in circuits.

In a circuit such as our latch we can consider the state of the circuit at a particular time, such as at the nominal beginning of an operation. In general the state of a system will not be so well-defined at any particular time. Instead, some parts of the system may be stable during some clock phases, while other parts will not be stable until different phases occur. Since we have replaced the traditional standard of reference, the clocks, with a pair of markers, we will define the timing of state relative to a marker. The question is then, "Which marker?," and the answer is both. The precondition state, that is the state before an operation takes place, is relative to the start marker. The postcondition or "after" state is relative to the next maker. The timing of the precondition state for our latch is illustrated in Figure 2.4a (p. 30).

In this figure, we have drawn two lines for the data signal, one at the high logic level and the other at the low. This is because, though we intend for some particular value to be applied to the circuit during the indicated time, we do not know what this particular value will be. This is in contrast to the control signal, where we know the particular value to be applied.

Since in classical models inputs are treated similarly to predecessor state, to treat them in a uniform way we must establish their timing relative to the start marker. Figure 2.4c illustrates this.

The timing of output data is normally determined relative to the next marker[3], as in Figure 2.4d. However, many circuits make no disjoint distinction between output and stored state. In such cases, the timing used will depend on the role being considered. When the node is being considered as an output or successor

[3] Our notion of machines is the Moore model, which makes explicit the delay between cause and effect inherent in any realizable physical system.

state, its timing will be taken relative to the next marker, and when it is being considered as predecessor state, it will be taken relative to the start marker.

The preceding elements were combined in the timing diagram of Figure 2.2 (p. 22). Using this diagram, we can analyze sequences of latch operations. For example, Figure 2.5 (p. 30) shows a load operation followed by a hold operation. Notice how the diagrams of the two separate operations are combined to yield the complete diagram. The end of the first operation coincides with the beginning of the second. Where signals on circuit nodes overlap in the diagram, their values match. Any longer sequence of latch operations can also be expressed by a timing diagram constructed of the two component operations from Figure 2.2.

Having looked in some detail at the way in which the circuit's timing operates, we need also to consider the latch from a more abstract point of view.

The fundamental motivation for formal verification is to increase confidence in the correctness of real systems. One way in which it can assist in achieving confidence is that it provides a check between two independent representations of conceptions of a system. A positive verification result does not guarantee that a system is actually what we intend it to be, since it is possible to make the same mistake twice, and check an incorrect circuit against an incorrect specification. However, the likelihood of this occurrence is smaller[4] than the likelihood of a single error.

But verification can increase confidence in another way as well. If one representation is somehow simpler than the other, we can reason, whether formally or informally, more easily about it. In the course of such reasoning we may reach some opinion of the suitability of the system. This is the more crucial aspect of verification: that it provide us with some leverage with which to move the mountains of detail that the latch example has shown are necessary to describe accurately the operation of even very simple circuits. Therefore, we wish to specify the behavior of circuits in an abstract way. The specification model should be detailed enough to represent what we find essential or interesting about circuits, but abstract enough to avoid what we find distracting.

2.1.1 Specification

The specification model we have in mind is an abstract Moore machine, structured as a set of symbolic assertions. (Recall that in a Moore machine, the outputs are determined by current state, but not by inputs.) Essentially, each assertion corresponds to an operation that the system is to perform. Each assertion is symbolic so that the text of a single assertion can concisely represent the operation for all

[4] We do not intend any study of sources of such error, but this reduced probability will hold provided there is *any* statistical independence between the mechanisms that introduce error in the two representations. The reduction in probability of course may be insignificant if the independence is small.

possible data values. Describing the specification machine in this way decouples the different operations of a circuit into different places in the specification.

Let us specify the abstract behavior of the latch we have been considering. Informally, the latch has an input and it has an output and it stores a bit. It implements two operations, load and hold. A load operation takes an input value and stores it. A hold operation maintains the stored value. After either operation, the stored value also appears on the output.

Noticing that the stored value appears on the output after every possible operation, we are faced with our first decision. Should the specification identify the stored value with the output? The case can be made either way. Distinguishing the stored value and the output is more in keeping with our circuit. However, identifying the stored value with the output simplifies the specification. Making this identification, if we let D denote the input and Q the stored/output value, we can specify the latch with the two assertions:

$$((\text{operation} = \text{load}) \wedge (D = b)) \stackrel{\delta}{\Rightarrow} (Q = b) \tag{2.1}$$

$$((\text{operation} = \text{hold}) \wedge (Q = c)) \stackrel{\delta}{\Rightarrow} (Q = c) \tag{2.2}$$

Each of these assertions corresponds to one operation of the latch. In each, the left-hand side of the $\stackrel{\delta}{\Rightarrow}$ relation denotes the conditions on the system in an (abstract) state, and the right-hand side denotes the conditions in its successor state. If the system is in a configuration satisfying the logical formula on the left-hand side, then after its transition it will be in a configuration satisfying the formula on the right. Thus, the symbol $\stackrel{\delta}{\Rightarrow}$ denotes both logical implication and the (abstract) passage of time. The symbols b and c are variables that denotes arbitrary bits. The symbols Q and D and the name "operation" are variables corresponding to the state and inputs of the system being described.

Part of the development our methodology will be to show that specifications expressed in such a language define an abstract machine, and that the checks we make between specifications and circuits entail a formal relationship between machines.

2.1.2 Mapping

For now, though, we need to relate the latch specification of equations 2.1 and 2.2 to the latch timing diagram of Figure 2.2.

Fortunately, this becomes straightforward if we refer to the elements included in the diagram, distinguished in Figures 2.3 and 2.4. When we compare them with the inner-most parenthesized subformulas of equations 2.1 and 2.2, a simple correspondence can be found. The subformulas (operation = load) and (operation = hold) correspond to the elements shown in Figure 2.3. The subformula $(D = b)$ corresponds to the element shown at the bottom of Figure 2.4c, where the two horizontal lines show that the bit b might take on either 0 or 1.

The subformula $(Q = b)$ or $(Q = c)$ corresponds to the elements shown in Figures 2.4a, 2.4b and 2.4d, that is, it corresponds to both nodes S and Q having the indicated value over the indicated time. The indicated time is the only tricky part. When the subformula appears on the left of the assertion's $\overset{\delta}{\Rightarrow}$ symbol, the timing is taken relative to the first marker, that is, as shown in Figure 2.4a. When the subformula it appears on the right-hand side of the $\overset{\delta}{\Rightarrow}$, the timing is taken relative to the second marker, as in Figures 2.4b and 2.4d.

From this example we can see what we need to be able to describe in a mapping language: we must map subformulas like the innermost parenthesized ones of the assertions above, which give particular values to particular abstract state variables, into a representation of values on particular nodes of the circuit, with time relative to some sort of marker. For those subformulas that correspond to inputs, we must also represent another marker, indicating the duration of the corresponding operation.

Given such a set of assertions and mappings, we can verify a circuit by using the mappings to construct symbolic simulation patterns for each assertion, then simulating the circuit to see that it passes the tests defined by the patterns.

2.1.3 Different implementations

There are many other ways to implement a latch, and we would like for all of them to satisfy our latch specification.

If our specifications are sufficiently abstract, they will include no particular details of a specific circuit implementation. This will leave us free to verify more than one circuit against a particular high-level specification. On the other hand, as we saw in relating the abstract specification to the timing diagram, part of our specification does have to consider circuit details precisely. We will need to keep these details separate from the high-level specification of behavior, if we are to consider verifying different circuits from one specification. But this is exactly what the mappings do: provide a bridge between the circuit and the high-level specification, which are isolated from one another.

For example, the preceding latch maintained its stored value through charge storage in a capacitor. Ideal capacitors can store arbitrary amounts of charge for any length of time. Actual devices lack both such ideal properties, but the lack of the latter is important here. Since real capacitors have nonzero conductance, charge bleeds off and eventually this latch "forgets" its stored value[5]. A latch

[5]Although the particular switch-level simulator we have used in this research happens to neglect this phenomenon.

that uses weak feedback to maintain stored state fixes this particular flaw[6], but since the circuit is supposed to be in essence the same, it ought also to meet our specification. For example, Weste and Eshraghian [243] call the CMOS circuit of Figure 2.6 a “D flip-flop.”

The timing of this circuit is similar to that of the original latch, so its timing diagram will look similar, but it is not the same. The node S of the original latch has no counterpart here. Or, if you like, the nodes S and Q of the original latch have the same counterpart, node Q of this latch. Thus to verify such a latch we would have to modify the mapping from the abstract specification so that it maps onto this circuit. In this case, the modified mapping would be simpler than the original. Figure 2.7 illustrates the timing of the two operations in this latch.

This barely begins to catalog the possible latch implementations, as a glance through any book on digital integrated circuit design would show.

2.1.4 Specification language

Let us return to the latch specification of assertions 2.1 and 2.2 from page 26 to see what they imply for the design of a formal language to support such a style of specification. (These implications will be explored more fully in Chapter 3.)

Although these simple assertions do not use all of the features of a fully developed specification language, or even all of its essential features, we can still make several cogent observations from them. First, each assertion consists of two parts, or formulas, separated by the $\stackrel{\delta}{\Rightarrow}$ sign.[7] We call the first formula the assertion's antecedent, or precondition. We call the second one its consequent, or postcondition. Implicit in this division is the notion that the antecedent is to hold at one time, while the consequent is to hold at some successor time. We can also observe that in these assertions, the antecedent is composed of the conjunction of two parts, or conjuncts, each of which is a smaller subformula.

Examining one of these simple subformulas reveals a subtlety of this style of specification. Consider the consequent of the first assertion, $Q = b$. It is deceptively simple. Yet the letter Q denotes state of the system, whereas the letter b does not. Instead, b is a Boolean variable that appears only in the assertion so that we can describe the case where b denotes the bit 0, and the case where it denotes the bit 1, in a single expression. Each of the other primitive subformulas has the same form: on the left of the $=$ sign is a state variable denoting system state, where on the right is an expression over a set of case variables denoting state *values* being considered.

These two kinds of variables, state variables and case variables, play different roles in the specification. The state variables define the inputs, outputs, and state

[6]though now the circuit designer must be careful that the “weak” inverter is actually weak

[7]The symbol $\stackrel{\delta}{\Rightarrow}$ can be read “then implies” or “next implies.”

of the system being specified. They are global in scope; each can appear in any assertion. However, the context in which state variables can be used is restricted: each primitive subformula contains exactly one state variable. The case variables, in contrast, define the cases in which an assertion applies. Case variables are used to build up expressions of the values of state variables[8]. Case variables are local to assertions in scope; if the same letter is used as a case variable in two different assertions, the two occurrences are completely independent.

We conclude this section with a preview of what is to come: a specification of the latch, in Figure 2.8. The notation used will be described in Chapter 3. Here we have made explicit the distinction between state variables and case variables.

2.1.5 Summary

This latch example has illustrated several characteristics of this view of synchronous hardware. A sequential circuit performs different operations. Each of these operations has its own timing: a nominal duration, demarcated by a nominal beginning and a nominal ending. The signals needed to make the circuit perform the operation, however, need not be restricted to this interval, and indeed we will soon see that they often cannot be so confined. The important thing is to ensure that they are consistently defined. In particular, the circuit's clock signals are simply inputs which often must follow a particular pattern both before and after the nominal bounds of the operation. These markers defining an operation's timing are also useful references by which the state of the system is measured.

To verify a circuit, we need a specification of its intended behavior (Chapter 3). If the specification is more abstract than the circuit, some formal relation between the specification and the circuit must be given (Chapter 5). This can take the form of a nondeterministic, or set-valued, mapping from the specification to the circuit. This relation then relates transitions of the abstract specification to circuit operation by relating abstract instants to the markers distinguishing the circuit's operations, and relating abstract state at these instants to circuit state timed relative to the markers. Parameterized symbolic representations of sets of specification transitions are related to parameterized symbolic representations of intended circuit operation, and circuits can be checked against these by symbolic simulation (Chapter 6).

[8] and are also used in another combining form called *case restriction*, not present in this example

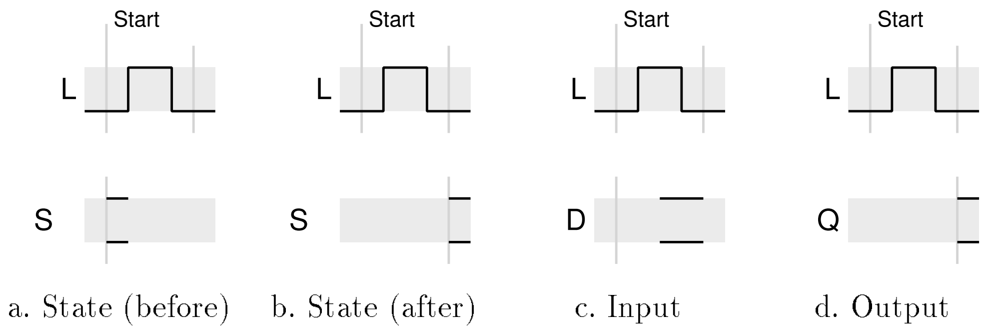


Figure 2.4: Timing for *load* operation of latch. Each aspect of the operation is shown separately.

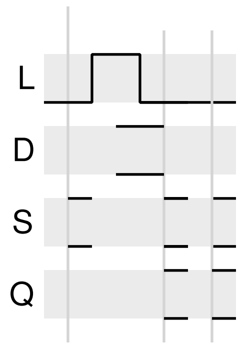


Figure 2.5: Two successive latch operations. Compare to Figure 2.2.

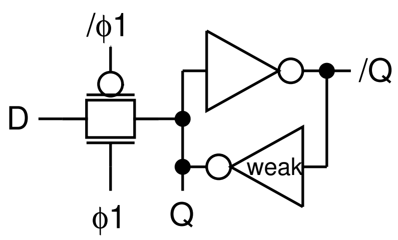


Figure 2.6: Equivalent latch. This circuit performs the same function as that of Figure 2.1, but its timing details differ.

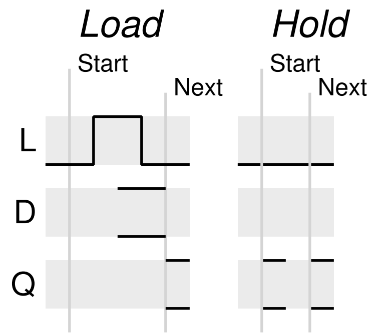


Figure 2.7: Timing diagram of equivalent latch. Compare to Figure 2.2 to see the difference in the timing details.

```
SMALversion 0 specification latch
types
  value word 1;
  operation enumeration load, hold.
state
  op : operation;
  D : value; Q : value.
assertions
  op = load /\ D = b:value  ==>  Q = b;
  op = hold /\ Q = c:value  ==>  Q = c.
```

Figure 2.8: Abstract specification of a latch. The notation will be fully described in Chapter 3.

2.2 The stack from Mead and Conway

The preceding section used a very simple sequential circuit to present some elements of a rather complicated verification methodology. This section gives a slightly more complex circuit to better motivate some of the complexity.

Mead and Conway's classic text [174] presents an example of the design of a complete subsystem. The example is a stack, constructed from a bidirectional dynamic shift register. This circuit has been well-studied [243, 191]. Here we give the circuit and its operation. We also identify a shortcoming that motivates our notion of input conformity[9].

The stack is composed of a series of the cells shown in Figure 2.9.

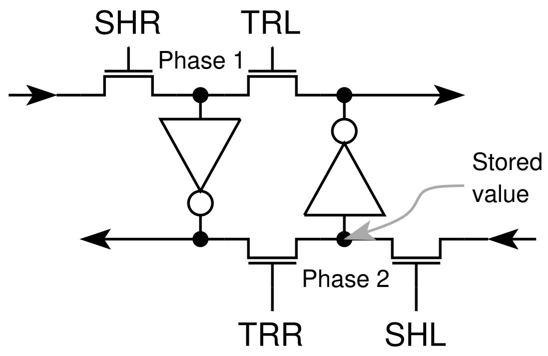


Figure 2.9: Mead and Conway's stack cell. A stack is constructed by composing these cells horizontally. The top of the stack is at the left-hand end. Values are stored in inverted form on the indicated node.

The top pass transistors of the cell may conduct during the first phase of a two-phase non-overlapping clock. The bottom pass transistors may conduct during the second phase. Data bits are stored (in inverted form) on the indicated node.

A stack can do three things: push data, pop data, or hold data. By turning these transistors on at suitable times, the three operations of a stack can be performed[10]. Figure 2.10 gives the timing of the stack cell as originally presented. Push is implemented by asserting SHR during the first clock phase, and then asserting TRR during the second phase. This loads the data input from the left-hand side of the cell. Hold is implemented by asserting TRL during the first phase, and

[9] Mead and Conway [174] as well as Weste and Eshraghian [243] glossed over the shortcoming. Mukherjee [191] noted it, but neglected to point out that a push followed by a pop serves as a delay.

[10] Mukherjee uses the names PUSH1, POP1, PUSH2, and POP2 respectively for SHR, TRL, TRR, and SHL.

then asserting TRR during the second. But pop is implemented by asserting SHL during the second phase, and then asserting TRL during the first. This outputs the data to the left side of the cell.

Outputs and stored values are actually inverted in this stack, but we will use the uncomplemented values in the timing diagrams that follow, to simplify them.

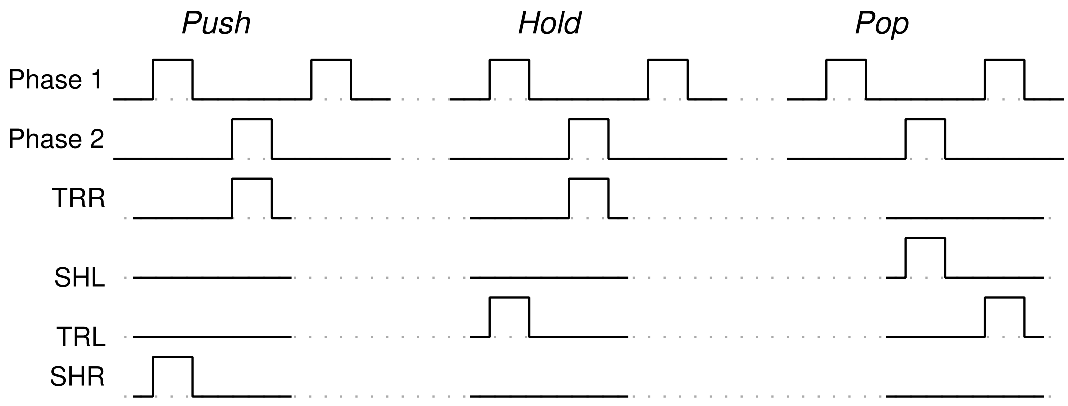


Figure 2.10: Timing diagram for stack cell. Each of the three different stack operations is shown.

The four control signals can be generated by the control logic of Figure 2.11. The control circuit generates the four stack control signals from a pair of control signals multiplexed on a single input line. The circuit simply demultiplexes the signals and uses them to gate the appropriate clocks. This controller introduces a delay of one clock phase. Figure 2.12 illustrates the timing needed for the control circuit in order to produce the timing shown above for the stack cell. A push operation is performed by holding the op signal high during phase 2, then low during the following phase 1. A hold is performed by holding the signal low during both phases. But a pop is performed by holding the signal high during a phase 1, then low during the following phase 2.

However, it is difficult to understand how to use the stack when its operation is presented this way. The timing of the pop operation is inconsistent with the timing of the other two operations. It is not even clear that this stack can actually be used for any purpose. The only way that a circuit can do anything useful is to produce an output, and the only stack operation that produces an output is pop. Yet the pop operation cannot follow either the push or the hold operation.

For example, suppose we want to perform the operation sequence hold; pop. Attempting to align the pop operation to follow *hold* is shown in Figure 2.13. There the clock signals are in conflict during the shaded interval. One operation

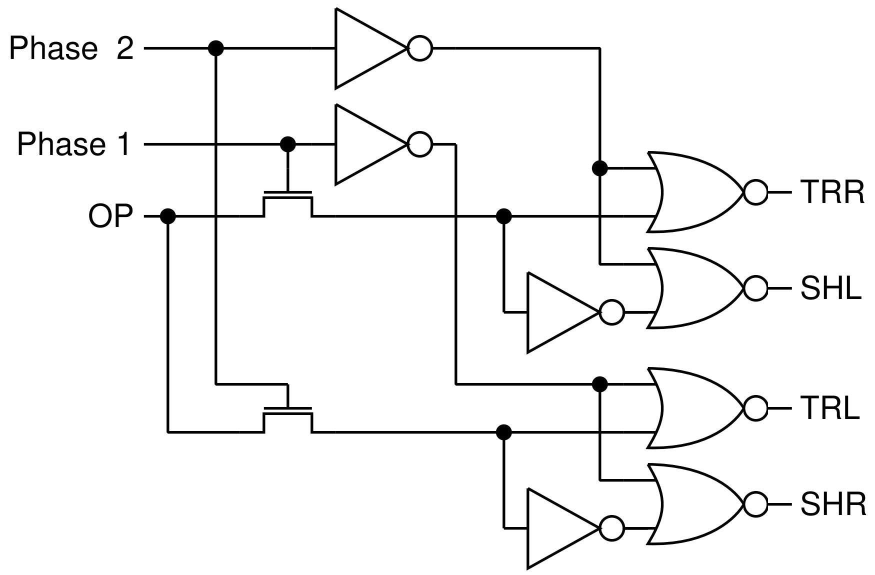


Figure 2.11: Mead and Conway's stack control circuit. The control input is time-demultiplexed and used to gate the clock, producing the cell control outputs.

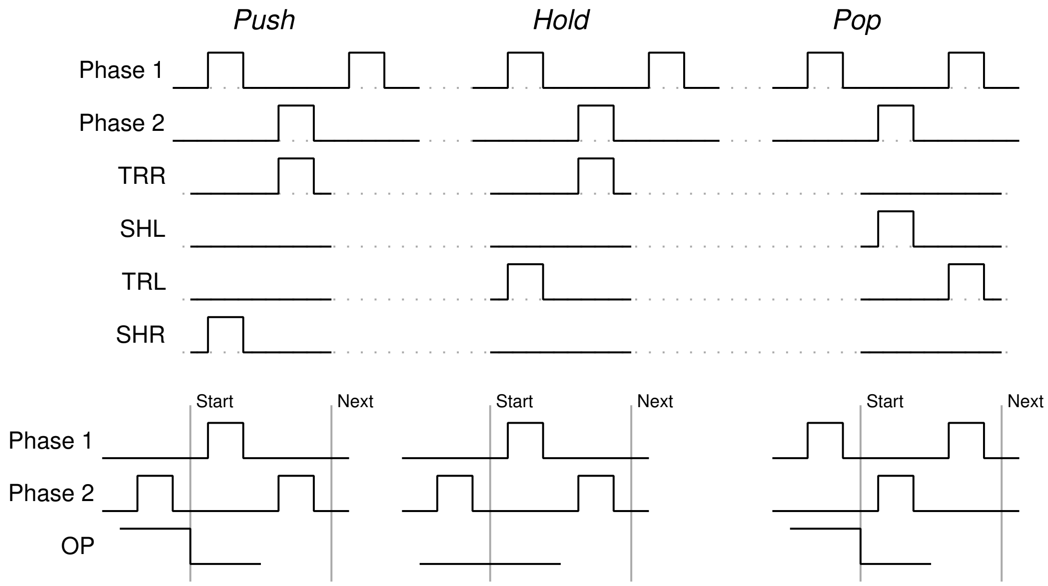


Figure 2.12: Timing diagram for multiplexed stack control, after Mead and Conway. The control inputs are shown at the bottom, and the cell timing is shown at the top.

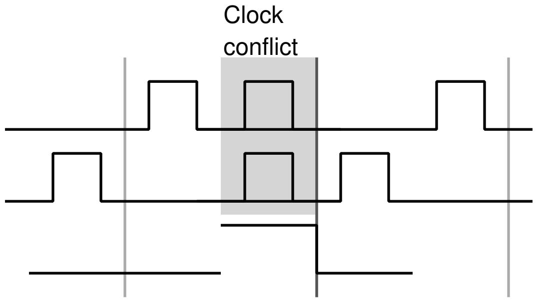


Figure 2.13: Attempt to align pop to follow hold operation. The inconsistent specification causes a signal conflict.

requires that clock phase one occur while the other operation requires that clock phase two occur at the same time. Adjusting the start and finish times of the operations so that the clocks can be aligned causes problems with the op control signal. Either there is a conflict of values on this signal, or there is a period when no value is specified for this signal—and if this second proposed adjustment is made, then we would find a conflict if we examine a pop followed by a push. The timing is inconsistent. We wish to allow any possible sequence of stack operations.

Fortunately, we can adjust our idea of a pop operation to have timing consistent with the other operations. We call such consistency **conformity**—overlapping portions of different operations must conform with one another: they must agree. With this revision, a more appealing view of the stack's control timing is given in Figure 2.14.

For the timing shown here, all abstract input sequences are conformable. That is, for any sequence of push, pop, and hold operations, the signals that represent them can be applied in succession to the circuit. Figure 2.15 illustrates the timing for all three operations, including IO and stored state, and Figure 2.16 shows that the operations can be combined to form a sequence.

Observe that both an abstract notion—a sequence of operations—and a more concrete notion—the circuit's actual timing—come into play in this discussion. When we discuss conformity in more detail, we will again see that we must consider both aspects.

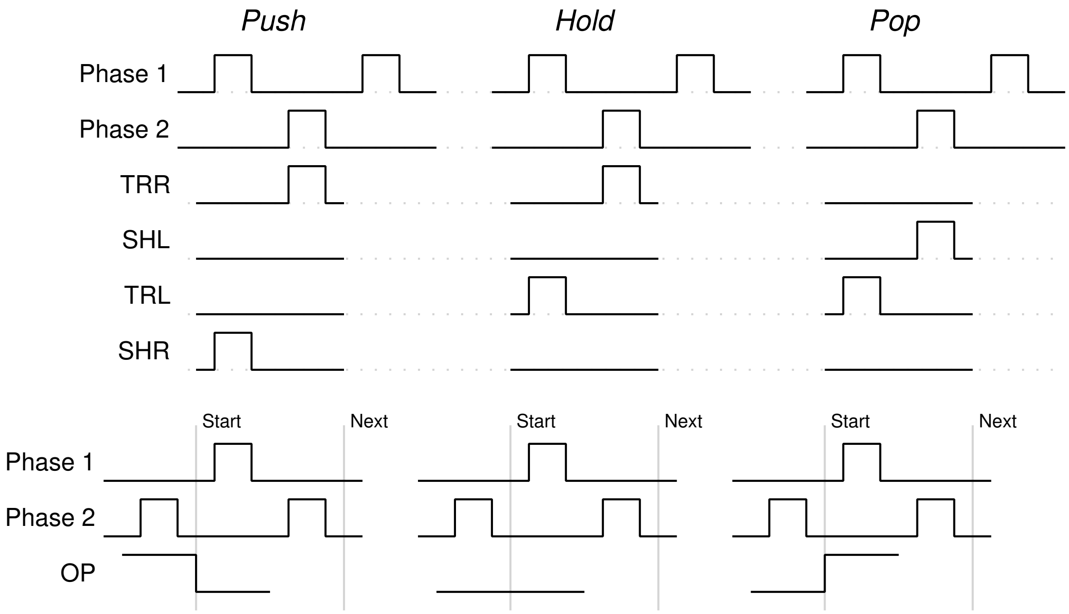


Figure 2.14: Revised timing diagram for multiplexed stack control. This version eliminates the conflict identified in Figure 2.13.

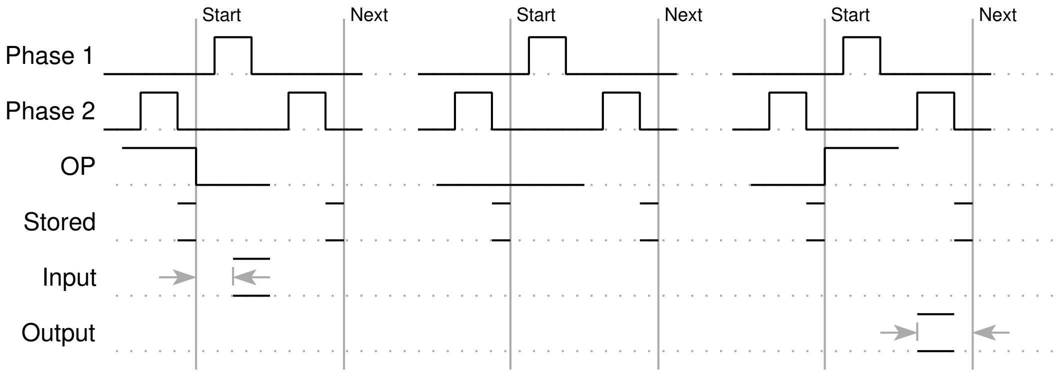


Figure 2.15: Timing diagram for stack operations. The three operations are shown in the style of Figure 2.2. The grey arrows indicate the marker relative to which signal timing is measured.

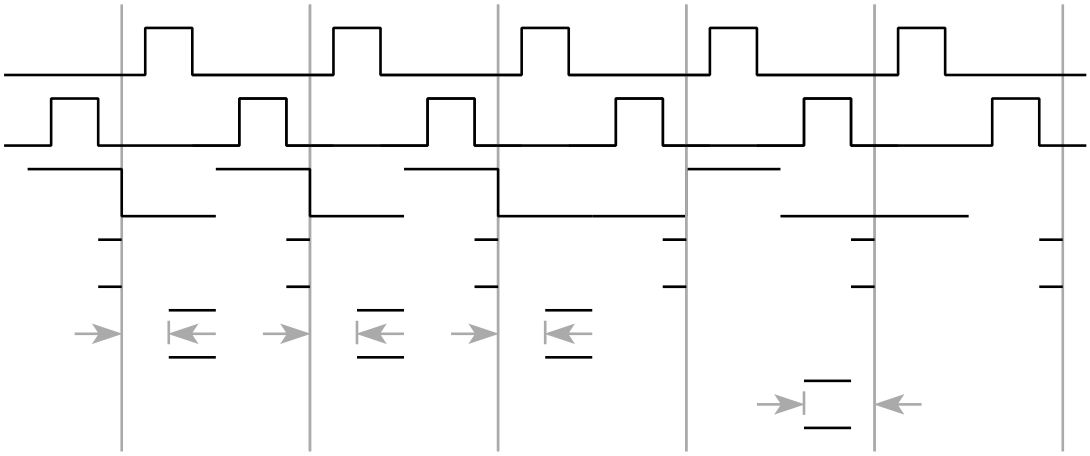


Figure 2.16: Sequence of stack operations: push, push, push, pop, hold. Analogous to Figure 2.5. The grey arrows indicate the boundaries relative to which state is analyzed.

2.2.1 Verification

We will use this sequence from Figure 2.16: three push operations, a pop, and a hold, to provide a more complete illustration of the methodology.

operation	push	push	push	pop	hold
input	0	1	0	–	–
output	–	–	–	–	0

Figure 2.17: IO behavior of abstract stack. Data is supplied during each of the push operations, and an output appears following the pop operation.

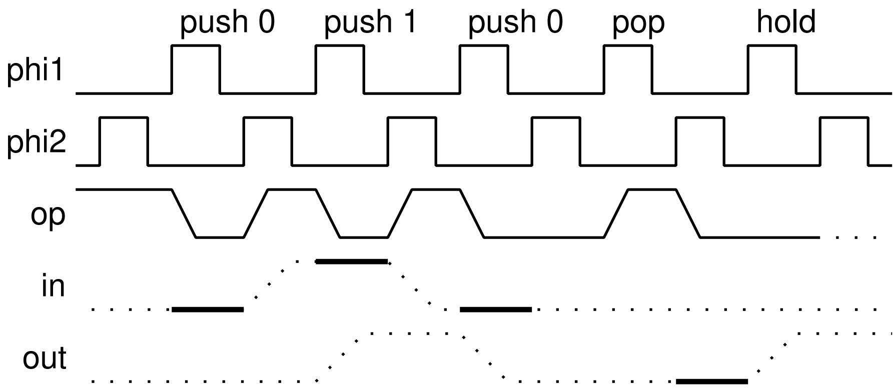


Figure 2.18: IO behavior of stack circuit. The time during which inputs must be stable, or during which outputs will be stable, is indicated by the heavy lines.

Figure 2.17 shows the input-output behavior of an abstract model of a stack for this sequence of operations, and one particular pattern of input data. An input value must be provided for each of the push operations. The pop operation produces an output. Our Moore-machine model of sequential behavior dictates that this output will be present at the succeeding time, that of the final hold operation.

Suppose this abstract model is to be our specification of the circuit. To say that the circuit meets the specification, we must relate the IO behavior of the abstract model in Figure 2.17 to the IO behavior of the circuit given by the timing diagram in Figure 2.18. Using this relation, we must show that the when the circuit receives circuit inputs that correspond to these abstract inputs, it produces circuit outputs that corresponds to this abstract output. We adopt a definition of implementation based solely on input-output behaviors since the only way that any environment can distinguish between a good circuit and a bad one is to interact with it through its IO.

Figure 2.18 shows the behavior of the stack circuit for this input sequence. The input and output signals are shown with dotted lines, which represent their values over time. The solid portions of these lines represent the intervals where the values are actually of interest: the three bits given as input to the push operations, and the output bit following the pop operation.

Having a notion of implementation based solely on input-output behaviors does not dictate that the analysis used to prove this property must be based entirely on IO. In fact, it has long been established that it is not possible to uniquely identify a finite-state system using a fixed test of its IO behavior [185].

operation	push	push	push	pop	hold
input	0	1	0	–	–
output	–	–	–	–	0
cell 0	–	0	1	0	1
cell 1	–	–	0	1	0
cell 2	–	–	–	0	–

Figure 2.19: Behavior of abstract stack, including state. State appears below the horizontal line.

Figure 2.19 shows the behavior of the abstract stack with its internal state exposed. Figure 2.20 shows the corresponding timing diagram with the circuit's internal state exposed. (Grey arrows indicate the data transfer.)

This is the natural way to think about a stack. With internal state exposed, showing that the circuit implements the specification becomes much easier. This is because exposing state makes it clear that system behavior is constructed out of transitions. Thus, in order to show that the circuit implements the specification regardless of the input sequence, it suffices to show that it implements the specification for each transition. Continuing with our example, each pair of columns in Figure 2.19 represents a transition of the specification. The first column of the pair gives inputs and state before the transition. The second column gives outputs and state after the transition. On the other hand, a portion of the timing diagram in Figure 2.20 represents each corresponding fragment of circuit operation. The individual fragments are broken apart for illustration in Figure 2.21.

Figure 2.21 shows the individual transitions of the circuit for the first four operations of the sequence, which correspond to the four pairs of columns from Figure 2.19. If we check that each circuit transition implements the appropriate abstract transition, then we need only a way of combining transitions to yield

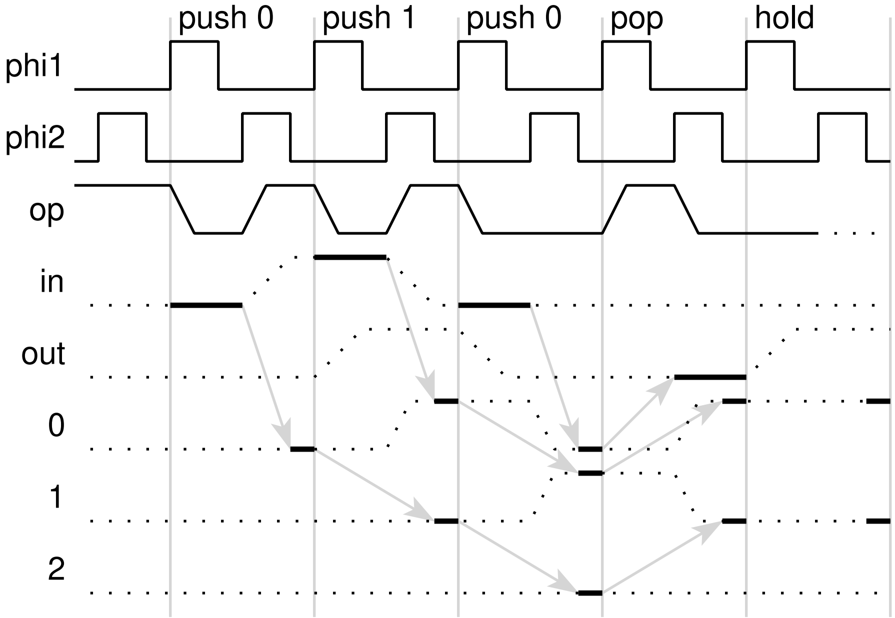


Figure 2.20: Behavior of stack circuit, including state. The state consists of the last three signals shown.

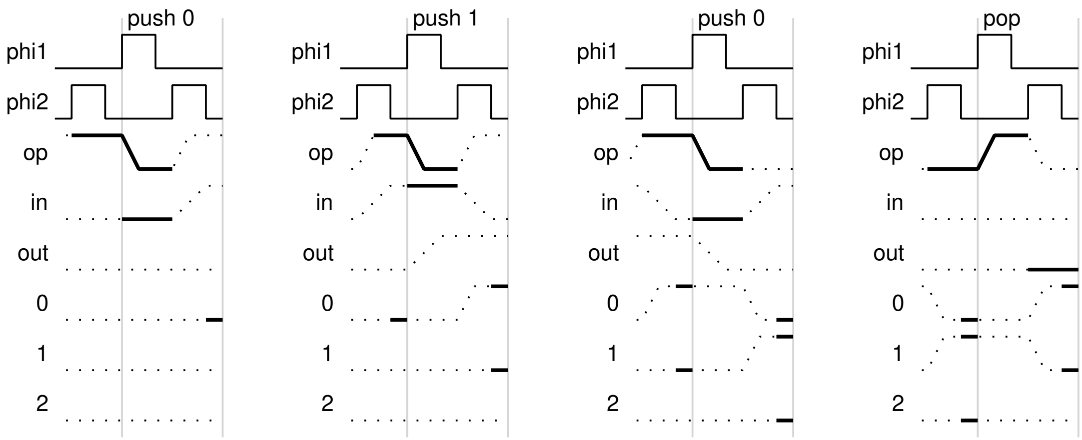


Figure 2.21: Transitions of stack circuit

behaviors. Then we will have verified that the entire sequence of circuit operations implements the sequence of abstract operations. Observe from Figure 2.21 that we can construct the timing diagram of Figure 2.20 by overlapping individual transitions, aligning the vertical grey lines.

Breaking behaviors down into transitions eases the task of verification, but even a small system has a large number of possible transitions, more than can be analyzed exhaustively. Since separately checking each transition in a large system is still too onerous a task, we will develop symbolic representations. Each will capture many transitions. It is these symbolic **assertions** that we will check. From this, we will be able to deduce that the individual transitions are implemented by the circuit, and hence that the circuit implements the specification, for all behaviors (i.e., sequences).

Continuing our stack example, we recall that there are three possible operations. Each operation has a similar effect, whatever the context that it may occur in. That is, a push operation in one stack configuration will be very similar to a push operation in any other stack configuration: the input data will be loaded, and the contents of each location transferred to the next location. Similarly, a pop operation will always produce the first location's contents as an output, and transfer data from each location to the previous one, and a hold operation will retain data in place, regardless of the particular data values.

This simple observation forms the basis for our expression of several transitions as a single assertion. For example, the sequence we have been considering in our example contains three push operations, but they can all be described by Figure 2.22. Using a symbolic simulator, we can check circuit operation against

operation	push	–
input	a	–
output	–	–
cell 0	b_0	a
cell 1	b_1	b_0
cell 2	–	b_1

Figure 2.22: Symbolic transition for push operation. This table has the same form as the table of Figure 2.19, but the bits (0's and 1's) have been replaced by symbolic values.

such a symbolic assertion. This effectively checks many transitions at once. The diagram of Figure 2.23 illustrates the operation of a symbolic simulation model of the circuit for this symbolic transition.

We can make additional refinements to the assertions, and the basic principle will remain the same. For example, rather than include both b_0 and b_1 values in

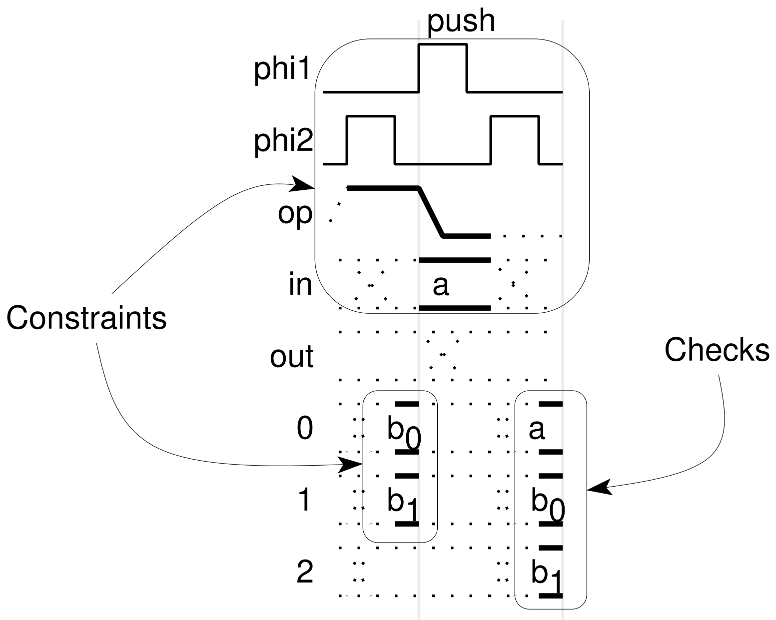


Figure 2.23: Symbolic patterns to verify push operation. Bits have been replaced by symbolic values, and simulated circuit state is constrained or checked at only the times indicated by heavy lines. Constraints and checks are as indicated.

the assertion, we can use two assertions, each with a single variable b_i. Then we could combine the two assertions *symbolically* to yield a single assertion containing a single data variable b, and an index variable i to distinguish the cases under consideration. It is difficult to see the utility of this on such a small example. In larger systems this technique, which we call **symbolic indexing**, can reduce the number of variables needed from the size of a memory to its logarithm. Symbolic indexing will be discussed further in section 7.2.

We illustrate part of this technique in Figure 2.24, which shows two assertions that together represent the same transitions as the previous assertion. It is worth

operation	push	–	push	–
input	a	–	a	–
output	–	–	–	–
cell 0	b_0	a	–	a
cell 1	–	b_0	b_1	–
cell 2	–	–	–	b_1

Figure 2.24: Two assertions, covering different storage bits. By intersection, together they have the same meaning as the assertion of Figure 2.22.

examining more closely how these two assertions represent the same transitions as the previous assertion: by intersection. The original assertion represented a set of transitions. Specifically, it constrained system behavior by requiring that each system configuration that matched the first column must lead to a successor configuration that matched the second column. For configurations that did not match the first column, however, their successors remained unconstrained. Each of the two revised assertions also represents a set of transitions, in a similar way. The transitions that match the first column of both revised assertions are precisely the transitions that match the first column of the original assertion. Those are the configurations whose successors are constrained. Similarly, the the transitions that match the second column of both revised assertions are precisely those that match the second column of the original assertion. Thus, the two assertions together describe the same transitions as the first assertion.

The preceding discussion gives an example of the development of a specification. We saw how a set of assertions can be used to describe the behavior of a system. And we also saw how the assertions can be checked individually, when fragments of circuit behavior (the elements of Figure 2.21) are combined to yield entire behaviors (Figure 2.20) by aligning markers.

The role of the theory developed later in this thesis is to support a verification methodology that follows the sketch above. We will develop a formal notion of implementation as an input-output relationship, and show that we can establish such a relationship by exposing internal system state. Then we will show that such behaviors, with state exposed, can be divided into transitions in the specification and "marked strings" in the circuit, so that a relation between transitions implies a relation between entire behaviors. Finally, we will show that this relation can be checked by checking assertions.

The entire proof that a circuit implements its specification then has several parts. The first part imposes requirements on whomever is verifying the circuit. Someone must expose internal state, formalize the relation between specification state and circuit state, describe the specification using assertions, and (using an automated tool) check the circuit against each of the assertions. The theory provides the rest of the proof: that the sequences of circuit operation corresponding to the checked assertions can be stitched together to yield circuit behaviors, and that when the internal state is hidden to yield circuit IO behaviors, a formal relationship, implementation, exists between the them and the IO behaviors of the original specification.

2.3 Verifying decomposed systems

Specifying a microprocessor directly is difficult. Furthermore, apart from verification, the specification itself is not particularly useful. However, a properly-written specification of a computer consisting of a processor and a memory system is much more interesting: it reflects the semantics of the computer's instruction set. Thus, it is important to be able to verify decomposed systems: to take a specification for the entire system and an assumption of the correctness of one part (e.g., the memory system), and verify the correctness of the other part (e.g., the microprocessor).

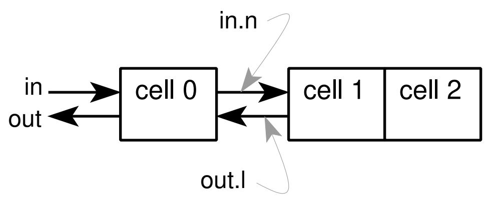


Figure 2.25: Decomposition of 3-bit stack into 2-bit stack and additional cell

However, when we examine this figure closely, we see that something is amiss. The individual components, that is the third cell and the two-bit stack, cannot follow a strict Moore machine model. For example, consider the pop operation. During this operation, the value on the top of the two-bit stack (i.e., the value in

operation	push	push	push	pop	hold
input	0	1	0	–	–
output	–	–	–	–	0
cell 0	–	0	1	0	1
in.n	–	0	1	0	–
out.l	–	–	–	–	1
cell 1	–	–	0	1	0
cell 2	–	–	0	0	–

Figure 2.26: Hypothetical abstract sequence of decomposed stack. The stack cannot actually be decomposed at this abstract level, as seen by the violation of the Moore model by the top cell during the last transition.

We can illustrate the verification of a decomposed system using our now-familiar stack. Consider a 3-bit stack constructed from a 2-bit stack (cells 1 and 2) and an additional stack cell (cell 0), as shown in Figure 2.25. Here the 2-bit stack plays the role of the memory system, and the additional cell plays the role of the microprocessor. In addition to its input, output, and storage locations, such a composite system will have two additional locations of interest: the connections between the new stack cell and the original 2-bit stack. We will refer to the input to the two-bit stack in.n (the "input to the next cell") and the output of the two-bit stack as out.l (the "output of the last cell"). These signals are internal to the 3-bit stack, so we would expect to treat them like any other internal state of the system. If we include these additional signals in our abstract view of the stack, we might expect its behavior on our example sequence to be something like that of Figure 2.26.

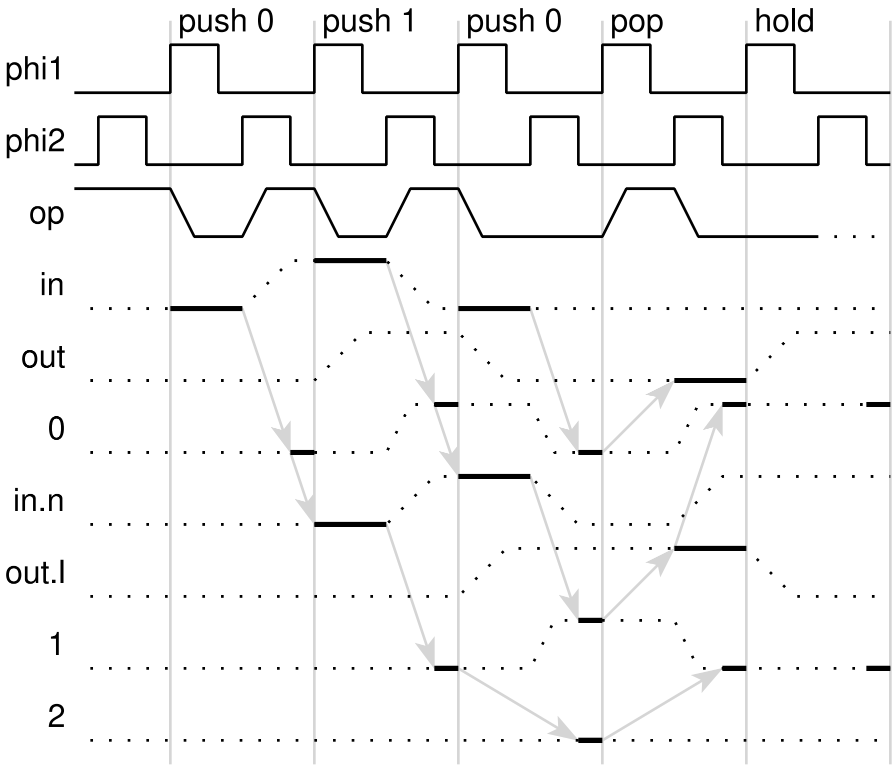


Figure 2.27: Behavior of decomposed stack circuit. This diagram is similar to that of Figure 2.20, with internal connections exposed. Grey arrows illustrate data transfer.

cell 1) must be transferred to the third cell (i.e., into cell 0) via the out.l line. Since this is an output produced by the pop operation of the two-bit stack, it must occur in the successor configuration of the pop transition (the last column of the figure). But since it is also an input of the pop operation on the extra cell, it must occur in the predecessor configuration of the pop transition (the next-to-last column).

The problem is that the stack cannot be decomposed as a Moore machine. The stack as a whole can be considered to be a Moore machine (where the transitions are individual stack operations), but we cannot retain this view of the stack as we divide it into its components. We must take a finer-grained look at the decomposed stack to understand its operation. For example, when we look at its timing diagram, in Figure 2.27, we can see that it does in fact operate as we would expect.

We wish to show that the decomposed three-bit stack implements the specification, when we assume that the two-bit stack implements its specification. To do so, we must examine the role of the signals between the two subsystems, and the use of our assumption about the 2-bit stack. First, we need to look again at the two signals in.n and out.l with which the cell communicates with the rest of the system. From the standpoint of the rest of the system (the 2-bit stack), in.n is an input and out.l is an output. However, from the point of view of the extra cell, which is the component we are actually trying to verify, in.n is an output, and out.l is an input. So (compared to when verifying the 2-bit stack) these two wires have changed roles.

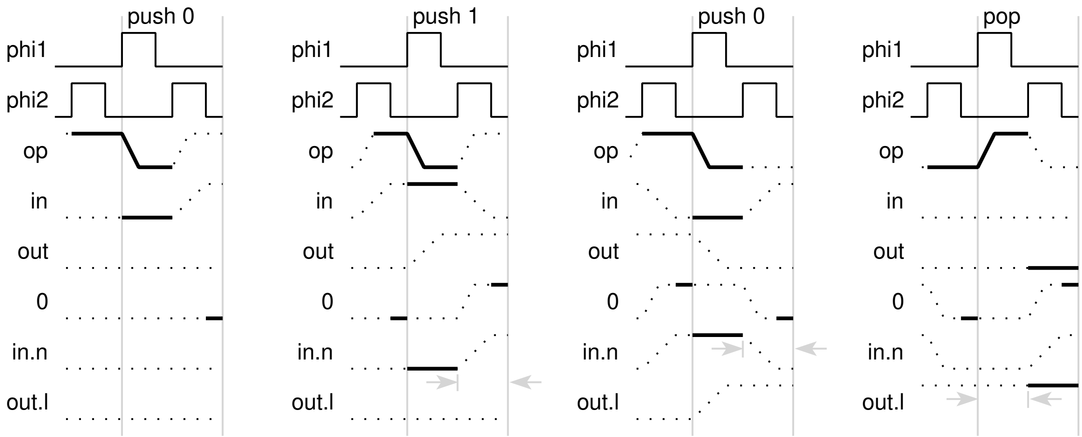


Figure 2.28: Transitions of decomposed stack cell. Note that in.n is an output and out.l is an input. Gray arrows indicate the boundaries with respect to which timing is measured.

If we assume that the two-bit stack is correct, we are assuming that when it gets its inputs as shown, it produces its outputs as shown. Thus, to verify the additional cell, we need to check that its outputs are as shown (i.e., we must ensure that our assumption applies) but we may assume that its inputs are as shown (i.e., we can indeed make the assumption, since we have ensured that is applicable). To verify cell 0 then we make use of the patterns that we would use if were were verifying the 2-bit stack, except that we change them slightly at the interface between the 2-bit stack, which we are now assuming correct, and the cell we are now verifying: there, the antecedent and consequent exchange their roles. We can ignore the portion of these patterns that consider the internal nodes of the 2-bit stack—although they would be important in *verifying* the 2-bit stack, we are now *assuming* that the 2-bit stack is correct. Thus, the patterns we would use to verify the stack cell for our sequence of operations are as shown in Figure 2.28.

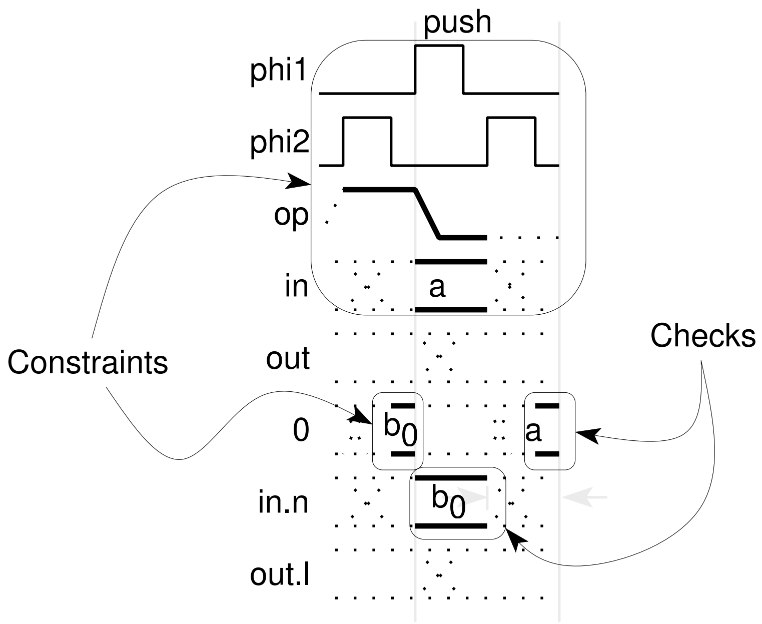


Figure 2.29: Symbolic pattern for verifying push operation of decomposed stack cell. Gray arrows indicate the boundaries with respect to which timing is measured.

When we generalize from individual transitions to assertions, we get symbolic patterns; Figure 2.29 shows the pattern corresponding to the push assertion.

Decomposition is formalized in Chapter 7.

2.4 Chapter summary

We have explored two simple sequential circuits to see what it means to say that they are correct. First, we described a very simple latch. Nonetheless, we found several aspects which required attention in order to treat the example in a general way.

We then described a stack and its timing, identifying the need to ensure input conformity. Then we sketched the steps involved in verifying the stack, using one particular input sequence as an example. After considering the entire stack, we considered decomposing it into a smaller stack and an additional cell, and sketched the verification of the decomposed system.

These simple circuits have served as an introduction to the methodology. The succeeding chapters will develop the methodology in more abstract form, and evaluate it in more detail by means of application to a microprocessor.

We will begin by introducing some mathematics, then discussing the specification of systems at an abstract level, using assertions.

Part II

Methodology

Chapter 3

Machines and declarative specification

In order to verify the behavior of a system, we must first specify it. This chapter discusses specification of the behavior of data-intensive systems. It explains how a machine is defined by a set of assertions, gives some examples, and describes a language in which to write assertions and the semantics of its key elements.

Before delving into such a discussion, however, some mathematical preliminaries are in order.

3.1 Mathematical preliminaries

This section reviews a few of the mathematical ideas we use in our presentation. Within this thesis, all definitions are numbered in one sequence, while propositions, lemmas, and theorems are numbered in another. Some proofs are deferred; when such proofs are later given, the original statement is repeated with the original number.

Definition 1 (Closure) *If G is a set and $\bullet$ an associative binary operator, $[G]_\bullet$ denotes the $\bullet$-closure of G. An element g is in $[G]_\bullet$ iff there exist finitely many g_i in G, such that $g = g_1 \bullet g_2 \bullet \cdots \bullet g_n$.*

We say that G is a set of $\bullet$-generators of $[G]_\bullet$.

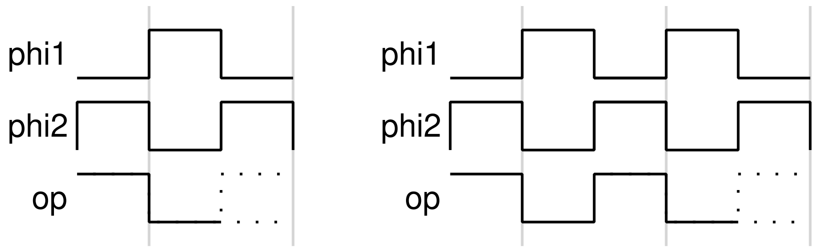
We write a dot $\centerdot$ to delimit a bounded quantifier, e.g., $\forall x \in S \centerdot P(x)$. The letter λ is as used in the lambda calculus [5]. That is, it binds a parameter and defines a function. For example, the function defined by the equation $f(x) = x^2$ or the expression $f\colon x \mapsto x^2$ can also be defined as $\lambda x(x^2)$.

When we discuss a language, the language under discussion is the "object language" and the language we are using (English and mathematics) is the "metalanguage."

3.1.1 Strings and marked strings

The concept of a string over an alphabet should be familiar. An alphabet A is a set of elements called symbols, and a string is an ordered sequence of zero or more symbols. The set of finite strings over A is written A^*. The empty string, that is the one containing no symbols, is denoted by the Greek letter ϵ. We write a list of symbols from an alphabet to denote the string containing those symbols in the order given. When we write two strings adjacently, we intend their concatenation.

In addition to ordinary strings, we will make extensive use of marked strings, that is, strings over an alphabet augmented by a distinguished marker symbol. A theory of marked strings is formally established in detail in Appendix A. The basics of marked strings are given here; proofs are deferred.



a. Timing diagram b. Two overlapped copies

Figure 3.1: Overlap of timing diagram

The basic reason to define marked strings is to provide a formal model analogous to the controlled, aligned overlapping of two timing diagrams. Consider as an example the idealized timing diagram shown in Figure 3.1a. Three signals are constrained during the first two intervals of time, but during the final, third interval of time, only the first two signals are constrained. Suppose that this timing diagram represents an operation that some circuit performs. Two light gray vertical lines indicate the beginning and ending of this operation. Notice that although we want to say that the operation begins at the first gray line, some of the signals have already been determined during the interval leading up to this instant. We wish to be able to determine the representation for the operation performed twice in succession.

Figure 3.1b represents the repetition of the operation—the first diagram twice in succession. The vertical lines show the times when the first operation begins, when the first ends and the second begins, and when the second ends. We will formalize the construction of this diagram from copies of the previous one. Of course, if we were to combine still more copies, we should get a larger diagram,

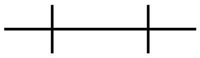


Figure 3.2: Skeleton of a marked string. The two vertical lines represent the first and last markers in the marked string. The first segment of the horizontal line represents the part of the string preceding the first marker. It will not contain any markers. The next segment represents the part between the first and last markers. (It may itself contain other markers.) The final segment represents the part following the last marker. It also contains no markers. Any of these three parts may be empty.

but the final result should not depend on the order in which we put together the smaller pieces.

Suppose that the letter a represents the combination of values[1] $(0, 1, 1)$, that the letter b represents $(1, 0, 0)$, and the letter c represents $(0, 1, 0)$. Furthermore, suppose that the prime symbol, $'$, represents the gray marker line. Then the first diagram can be represented by the set of strings $\{\, a'bc', a'ba' \,\}$ and the second can be represented by $\{\, a'ba'bc', a'ba'ba' \,\}$. The following sections develop a theory that allows us to do this, and to find the representation of the second from two copies of the representation of the first, using an operation we call "overlapped concatenation."

Another simple kind of diagram is useful in understanding marked strings. Figure 3.2 shows the "skeleton" of a marked string.

Let A be an alphabet. Define A'^* to be the set of marked strings over A, where a marked string is a string over $A' = A \cup \{\, \prime \,\}$ and the symbol $\prime$ (to be read "mark") does not appear in alphabet A. The symbol ϵ denotes the empty string. Note that $\epsilon \in A'^*$. We say that a marked string is k-marked if it contains k occurrences of the marker symbol $\prime$, and k+-marked if it contains k or more occurrences. We say that a string is double-marked if it is 2-marked.

It is important to note that the prime symbol is a marker. That its, a' is not a variable distinct from a. It is an a followed by a $\prime$.

We will need an error indicator, which we denote by $\top$. The functions we define will be strict with respect to $\top$. Thus we will consider functions over the universe $A'^* \cup \{\, \top \,\}$. Without further mention we will abuse notation and write A'^* when strictly speaking we mean $A'^* \cup \{\, \top \,\}$. We express the usual concatenation by adjacency, but we let it be strict on $\top$. That is, $x\top = \top = \top x$ whenever $x \in A'^*$.

We will define marked strings inductively.

[1]on the nodes phi1, phi2, and op respectively

Definition 2 (Marked strings) *The set A'^* of marked strings over alphabet A is defined inductively by the equations:*

$$\begin{aligned} \epsilon &\in A'^* \\ \top &\in A'^* \\ {}'x &\in A'^* \quad \textit{for } x \in A'^* \\ ax &\in A'^* \quad \textit{for } a \in A \textit{ and } x \in A'^* \end{aligned}$$

For example, ϵ (the empty string) and a and abc and $a'bc'$ and $a'''b$ and $''''$ are all marked strings, and $\top$ (the error indicator) is also a marked string. Since concatenation is strict on $\top$, expressions such as ${}'\top$ are not marked strings. The length of a marked string x is written $|x|$, and we do not count the markers when measuring the length.

Marked strings share many prefix and suffix properties which are symmetrical. Defining the reversal of a string makes it convenient to exploit this symmetry.

Definition 3 (Reversal) *If $\hat{x}$ is a marked string, its reversal $\hat{x}^R$ is given inductively by the equations:*

$$\begin{aligned} \epsilon^R &= \epsilon \\ \top^R &= \top \\ ({}'x)^R &= \left(x^R\right)' \\ (ax)^R &= x^R a \end{aligned}$$

For example, $(a'bc')^R = {}'cb'a$.

We can build up lattices of marked strings.

Definition 4 (Marked prefix and suffix orders) *The relation $\sqsubseteq_p$ is given by induction.*

$$\begin{aligned} \epsilon &\sqsubseteq_p x \\ x &\sqsubseteq_p \top \\ ax &\sqsubseteq_p ay \quad \textit{whenever } x \sqsubseteq_p y \\ {}'x &\sqsubseteq_p {}'y \quad \textit{whenever } x \sqsubseteq_p y \\ x &\sqsubseteq_p {}'y \quad \textit{whenever } x \sqsubseteq_p y \end{aligned}$$

The relation $x \sqsubseteq_s y$ is defined to hold exactly when $x^R \sqsubseteq_p y^R$ does.

For example, the relations $\epsilon \sqsubseteq_p a$ and $a \sqsubseteq_p ab$ and $ab \sqsubseteq_p a'b$ all hold.

The relation $\sqsubseteq_p$ is similar to the familiar prefix ordering of strings where a string x is "less than" another string y if x is a prefix of y. The innovation here is the addition of the marker. Intuitively, inserting markers anywhere in a string produces a larger one.

We can define an operation that lines two marked strings up at their left-hand edges and joins them together.

Definition 5 (Marked prefix and suffix joins) *The binary operator* $\sqcup_p$ *is given inductively.*

$$
\begin{aligned}
x \sqcup_p \top &= \top \\
\top \sqcup_p x &= \top \\
\epsilon \sqcup_p x &= x \\
x \sqcup_p \epsilon &= x \\
{}'x \sqcup_p {}'y &= {}'(x \sqcup_p y) \\
{}'x \sqcup_p y &= {}'(x \sqcup_p y) \\
x \sqcup_p {}'y &= {}'(x \sqcup_p y) \\
ax \sqcup_p ay &= a(x \sqcup_p y) \\
ax \sqcup_p by &= \top
\end{aligned}
$$

The operator $x \sqcup_s y$ *is defined as* $(x^R \sqcup_p y^R)^R$.

Observe that we have covered the entire set A'^*.

For example, $a'b \sqcup_p a'b = a'b$ while $aa'b \sqcup_p a'b = \top$. On the other hand, $a'bb \sqcup_p a'b = a'bb$. Finally, $'abb' \sqcup_p a'b' = 'a'b'b'$.

The operator $\sqcup_p$ is similar to the familiar least upper bound on prefix ordering. Again, the innovation is the marker. In the ordinary prefix ordering, the least upper bound of two strings is the longer of the two, if the shorter is its prefix, and is $\top$ if the shorter is not a prefix of the longer. Adding markers to the ordering requires inserting of the markers from both of the operand strings. In fact, the same correspondence between joins and associated orders holds with the markers included.

We can also define the prefix meet of marked strings. It is not particularly useful, except in developing the lattice theory.

Theorem 1 *The structures* $\langle A'^*, \sqcup_p, \sqcap_p \rangle$ *and* $\langle A'^*, \sqcup_s, \sqcap_s \rangle$ *are lattices.*

The proof consists of showing that the meets, joins, and partial orders have the proper relationship. Theorem 1 allows us to use lattice theory when needed in proofs.

We can break marked strings into convenient pieces at the first or last marker. We will soon use the pieces to define an overlapped concatenation operator. Figure 3.3 shows these parts.

Definition 6 (Clean prefix and suffix) *If* $\hat{x}$ *is a marked string, its clean prefix (or "first" part)* $\mathrm{F}(\hat{x})$ *is given inductively:*

$$
\begin{aligned}
\mathrm{F}(\epsilon) &= \epsilon \\
\mathrm{F}(\top) &= \top \\
\mathrm{F}(ax) &= a\,\mathrm{F}(x) \\
\mathrm{F}({}'x) &= \epsilon
\end{aligned}
$$

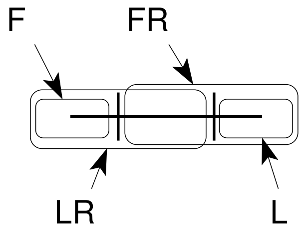


Figure 3.3: The parts of a marked string, in skeleton form. The function F extracts the part before the first marker. The function L extracts the part after the last marker. The function FR extracts the "rest" left over from F, and LR extracts the "rest" left for from L.

If $\hat{x}$ *is a marked string, its clean suffix (or "last" part)* $\mathrm{L}(\hat{x})$ *is defined to be* $(\mathrm{F}(\hat{x}^R))^R$.

Definition 7 (Marked suffix and prefix) *If* $\hat{x}$ *is a marked string, its marked suffix (the "rest" left after removing the "first" part and the demarcating marker)* $\mathrm{FR}(\hat{x})$ *is given inductively:*

$$\begin{aligned}
\mathrm{FR}(\epsilon) &= \epsilon \\
\mathrm{FR}(\top) &= \top \\
\mathrm{FR}({\prime}x) &= x \\
\mathrm{FR}(ax) &= \mathrm{FR}(x)
\end{aligned}$$

If $\hat{x}$ *is a marked string, its marked prefix (the "rest" left after removing the "last" part and marker)* $\mathrm{LR}(\hat{x})$ *is defined to be* $(\mathrm{FR}(\hat{x}^R))^R$.

For example, the equations $\mathrm{F}(a'b'c) = a$ and $\mathrm{L}(a'b'c) = c$ and $\mathrm{FR}(a'b'c) = b'c$ and $\mathrm{LR}(a'b'c) = a'b$ all hold.

Using these parts, we can define an overlapped concatenation operator which aligns the first marker of the last string with the last marker of the first string. The formal definition may at first be a bit opaque, but the skeletons shown in Figure 3.4 illustrate the intuition behind the definition.

Definition 8 (Overlapped concatenation) *The overlapped concatenation of marked strings* x *and* y *is written* $x /\!\!/ y$ *and is defined to be the marked string*

$$(\mathrm{LR}(x) \sqcup_s \mathrm{F}(y))'(\mathrm{L}(x) \sqcup_p \mathrm{FR}(y)).$$

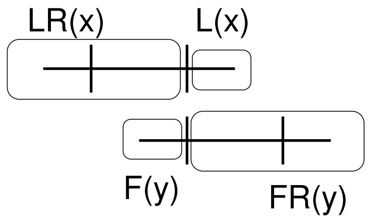


Figure 3.4: Skeletons of overlapped concatenation. The marker in the middle is obtained by aligning the first marker of the last string with the last marker of the first string. The portion before this marker is taken by forming the suffix join of the parts of each constituent that come before this marker. The portion after this marker is taken by forming prefix join of the parts of each constituent that come after the marker.

When there is no conflict, overlapped concatenation yields a real marked string. For example,

$$\begin{aligned} a'ba' /\!\!/ a'bc' &= (\mathrm{LR}(a'ba') \sqcup_s \mathrm{F}(a'bc'))' \, (\mathrm{L}(a'ba') \sqcup_p \mathrm{FR}(a'bc')) \\ &= (a'ba \sqcup_s a)'(\epsilon \sqcup_p bc') \\ &= a'ba'bc' \end{aligned}$$

When there is conflict, it yields $\top$, which can be thought of as an error indicator. For example, expanding $a'bc' /\!\!/ a'ba'$ from the definition yields $(a'bc \sqcup_s a) = \top$ so (since concatenation is strict) $a'bc' /\!\!/ a'ba' = \top$.

Theorem 2 *The operator $/\!\!/$ is associative.*

This may seem obvious, but is necessary to employ a trick in its proof, by showing that two different joins are associative under the proper conditions, and that these conditions apply.

We ultimately wish to construct mappings that are homomorphic over $/\!\!/$. We will need slightly more machinery for this.

This function "cleans" the last marker off of a string.

Definition 9 *The function* CL *is defined by the equation* $\mathrm{CL}(x) = \mathrm{LR}(x)\,\mathrm{L}(x)$.

This function has some properties that we will find useful.

Lemma 3 *If x is 2-marked then $x = \mathrm{CL}(x) /\!\!/ x$.*

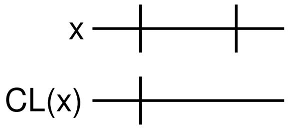


Figure 3.5: Dropping the last marker.

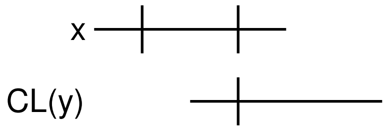


Figure 3.6: Dropping the last marker and aligning in preparation for overlapped concatenation.

The skeletons in Figure 3.5 illustrate the idea behind this lemma.

If a string has at least two marks, it does not matter whether we drop the last marker before or after performing an overlapped concatenation. Figure 3.6 illustrates the idea here.

Lemma 4 *If y has 2 or more marks, then $x /\!/ \mathrm{CL}(y) = \mathrm{CL}(x /\!/ y)$.*

Later we will need to compare marked strings while ignoring certain aspects of them. One such view is compatibility. Two marked strings are compatible if their "first" and "rest" parts are not in conflict. We will also sometimes need to consider the lengths of both these parts.

Definition 10 (Compatibility) *If strings x and y are 2-marked, we will say that they are compatible, and denote this by $x \approx y$, if and only if neither the expression $\mathrm{F}(\mathrm{CL}(x)) \sqcup_s \mathrm{F}(\mathrm{CL}(y))$ nor $\mathrm{FR}(\mathrm{CL}(x)) \sqcup_p \mathrm{FR}(\mathrm{CL}(y))$ is equal to $\top$.*

Definition 11 *If string x is 2-marked, the* measurements *of x, denoted $||x||$ is the pair $(|\mathrm{F}(\mathrm{CL}(x))|, |\mathrm{FR}(\mathrm{CL}(x))|)$. If x is 1-marked, $||x|| = (|\mathrm{F}(x)|, |\mathrm{FR}(x)|)$.*

The usefulness of compatibility is given by the following proposition.

Proposition 5 *If x and y are 2-marked then $x \approx y$ iff $\mathrm{CL}(x) /\!/ y \neq \top$.*

Compatibility and the same measurements imply a useful equality.

Proposition 6 *If two 2-marked strings have the same measurements, that is, if the equality* $||x|| = ||y||$ *holds, and* $x \approx y$*, then* $\mathrm{CL}(x) = \mathrm{CL}(y)$.

In defining the meaning of an assertion mapped onto a circuit, we will need to form sets of incompatible strings.

Definition 12 *If* A *is a set of* 2*-marked strings, all of the same measurements, the notation* $\tilde{A}$ *denotes the set* $\{\, x \mid \forall y \in A \centerdot ||x|| = ||y|| \wedge x \not\approx y \,\}$.

In defining the formal semantics of a mapping language, we will need the following definition, which it is not otherwise required.

Definition 13 *If* x *is a marked string,* Γ *is a superset of the alphabet, and* (f, l) *is a pair of nonnegative integers, define* $\mathrm{ext}^{\Gamma}_{(f,l)}$ *as follows. Let the pair* $(m, n) = ||x||$ *denote the measurements of* x*, let* $\hat{f}$ *be* $\max(m, f)$ *and let* $\hat{l}$ *be* $\max(n, l)$*. Then the extension of marked string* x *to alphabet* Γ *with lengths* (f, l) *is defined by the following equation.*

$$\mathrm{ext}^{\Gamma}_{(f,l)} = \{\, y \mid y \approx x \textit{ and } ||y|| = (\hat{f}, \hat{l}) \,\}$$

Intuitively, the function ext extends a set of marked strings so that every string in the set has measurements of at least (f, l).

3.1.2 Homomorphisms

A homomorphism is a mapping between two different sets each of which has some structure. The structure of the sets must somehow be similar, and the mapping must preserve this structure.

Definition 14 (Homomorphism) *If* A *and* B *are sets, and* $f_A\colon A \times A \to A$ *and* $f_B\colon B \times B \to B$ *are functions, a mapping* $h\colon A \to B$ *is a homomorphism (with respect to* f*) if for every* x *and* y *in* A*, the equality* $h(f_A(x, y)) = f_B(h(x), h(y))$ *holds.*

Often the functions f_A and f_B are denoted by the same symbol, which is written as an infix operator. Although traditional, the functions need not have arity two; the generalization is straightforward. When $f_A = f_B$, and this function has arity one, we say that functions f_A and h commute.

3.1.3 Set-valued functions and nondeterminism

A nondeterministic system, object, agent, or function[2] is one which may exhibit many possible behaviors, for example, by selecting one according to some random variable that we cannot observe. At any particular time, though, it exhibits only one of its several behaviors. We do not know which behavior it might take, so we represent its behavior using the set of all possibilities.[3]

We often write expressions that strictly interpreted seem to apply functions to subsets of their domain, rather than to elements of their domain. Such notations denote the image of the set under the function. In other words, we extend functions freely to sets.

Definition 15 (Set extension) *If S is a subset of the domain of F, $F(S)$ denotes the image of S under the mapping F. More precisely, if F is single-valued and S is any subset of its domain, $F(S)$ denotes the T such that $F\colon S \to T$ is surjective. If F is set-valued, $F\colon S \to 2^T$ is surjective with respect to T if $T = \bigcup_{s\in S} F(s)$.*

We will also extend binary operators on elements to sets, similarly. Sometimes we will include an "error indicator," denoted $\top$, in the domains we consider. In particular, marked strings include this error indicator. In extending operators to sets, we exclude this value as follows. Let S be such a set. For a binary operation $\bullet : S^2 \to S$ over such a set, its extension is $\bullet : (\mathcal{P}(S))^2 \to \mathcal{P}(S)$ defined by the equation:

$$Q \bullet R = \left(\bigcup_{q\in Q} \bigcup_{r\in R} \{q \bullet r\} \right) - \{\top\}$$

Proposition 7 *If $\bullet$ is an associative operator on elements and is strict[4] on $\top$, its extension to sets is also associative.*

Proof: Let capital letters denote sets and small letters denote elements. Suppose the extension is not associative. Then $A \bullet (B \bullet C) \neq (A \bullet B) \bullet C$. Without loss of generality, pick an $e \in A \bullet (B \bullet C)$ but not in $(A \bullet B) \bullet C$. Then there exist a, b, and c, such that $e = a \bullet (b \bullet c)$. But by hypothesis $a \bullet (b \bullet c) = (a \bullet b) \bullet c$ so $e \in (A \bullet B) \bullet C$, contradicting our assumption. ■

The strictness condition is necessary to ensure that when we throw the $\top$ element out of sets, it stays thrown out.

---

[2]The reason we prefer set-valued functions over relations will become clear in section 5.2.2

[3]This blurs the distinction between set-valued functions and relations. (In fact, in some formulations of type theory [4] the two are not distinguished.)

[4]strictness was defined in the text just before Definition 2, p. 53

3.1.4 Set-valued homomorphisms

A similar definition of homomorphism can be used when the functions are set-valued, i.e., nondeterministic. This requires that the functions commute for every nondeterministic possibility. All the cases below can simply be considered as the extension of the original definition to sets.

If h is set-valued, then h is a homomorphism if for every $\hat{z} \in h(f_A(x, y))$ there are $\hat{x}$ and $\hat{y}$ in the sets $h(x)$ and $h(y)$ such that $\hat{z}$ is equal to $f_B(\hat{x}, \hat{y})$, and for every $\hat{x}$ and $\hat{y}$ in the sets $h(x)$ and $h(y)$ there is a $\hat{z}$ in the set $h(f_A(x, y))$ such that $\hat{z}$ is equal to the value $f_B(\hat{x}, \hat{y})$.

If f (i.e., f_A or f_B or both) is set-valued, then h is a homomorphism if for every z in the set $f_A(x, y)$, the element $h(z)$ is in the set $f_B(h(x), h(y))$, and for ever $\hat{z}$ in the set $f_B(h(x), h(y))$ there is a z in the set $f_A(x, y)$ equal to $h(\hat{z})$.

If f and h are both set-valued, then h is a homomorphism if for every z in the set $f_A(x, y)$ and every $\hat{z}$ in the set $h(z)$ there are values $\hat{x}$ and $\hat{y}$ in the sets $h(x)$ and $h(y)$ such that $\hat{z}$ is in the set $f_B(\hat{x}, \hat{y})$, and if for every $\hat{x}$ and $\hat{y}$ in the sets $h(x)$ and $h(y)$, for every $\hat{z}$ in the set $f_B(\hat{x}, \hat{y})$ there is a z in the set $f_A(x, y)$ such that $\hat{z}$ is in the set $h(z)$.

3.1.5 Compositions of homomorphisms

The composition of two homomorphisms is also a homomorphism. For the standard case, this is obvious. It is also true if either or both of the functions is set-valued. The details are completely standard, and there are no tricks involved.

3.1.6 Partial functions

Partial functions are those which do not yield a value for some inputs. When we model nondeterminism with set-valued functions, we note that the empty set is in the range of nondeterministic partial functions. It is not in the range of nondeterministic total functions.

A binary operator that is partially defined is associative if the result of applying the operator in every order is defined exactly when the result is defined in any order, and if so has the same value for all orders. In other words, using the notation of operators extended to sets, $/\!\!/$ is associative if $(a /\!\!/ b) /\!\!/ c = a /\!\!/ (b /\!\!/ c)$.

3.2 Agents and machines

Having established our ground, we can turn to the systems we are interested in. Before we can consider specifications of systems, we must have some idea what kind of systems we wish to specify—we must select a model of computation.

This section begins by making some basic definitions regarding abstract agents. An agent is something having inputs and producing outputs. (Abstractly it is a nondeterministic function with a few special properties.)

3.2.1 Abstract agents

We define an agent M to be a set-valued[5] function as follows. Its domain is called the agent's set of inputs, $\mathrm{ins}(M)$. Its codomain is called the agent's set of behaviors, $\mathrm{beh}(M)$. From the behaviors we must be able to return to the inputs with a projection function Π. Thus $\mathrm{ins}(M) = \Pi\,\mathrm{beh}(M)$. Furthermore, if $v \in \mathrm{beh}(M)$ then $v = M(\Pi v)$. We refer to $\{\, v \mid i = \Pi v \,\}$ as the set of behaviors consistent with input i.

Intuitively, an agent can be thought of as a function that includes a copy of its input in its output.[6] Figure 3.7 illustrates this idea.

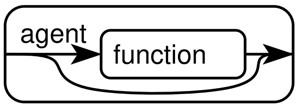


Figure 3.7: Formation of an agent from a function. The agent takes an input (from the left) and produces an output that includes a copy of the input.

All of this together is called the agent's signature: the agent M itself, and the sets $\mathrm{ins}(M)$ and $\mathrm{beh}(M)$, and the projection function Π.

3.2.2 Sequential machines

We can relate this abstract view of an agent to a more familiar model of sequential machines. Let a *sequence machine* (or simply *machine*) be an agent M whose $\mathrm{ins}(M) = \mathrm{inp}(M)^*$ and whose $\mathrm{beh}(M) = \mathrm{config}(M)^*$, where $\mathrm{inp}(M)$ is a set called the input alphabet and $\mathrm{config}(M)$ is a set called the configuration alphabet, and $\mathrm{inp}(M) = \Pi\,\mathrm{config}(M)$, with Π on $\mathrm{beh}(M)$ its pointwise extension.

Since sequence machines are agents, implementation is defined between them.

Mealy machines and *Moore machines* are sequence machines whose behavior is determined by a relation[7] called the transition relation:

$$\mathrm{steps}(M) \subseteq \mathrm{config}(M) \times \mathrm{config}(M)$$

[5]it is set-valued to represent nondeterminism

[6]Keeping a copy is a technical detail which will simplify some later proofs.

[7]We use a transition relation rather than a transition function so that we can construct it by set intersection.

For such machines, we define $M(i)$ to be the set[8]

$$\{ v \mid i = \Pi v \wedge \forall k \in [0, |i| - 2] \centerdot (v^k, v^{k+1}) \in \text{steps}(M) \}$$

where $|i|$ is the length of the input sequence i, and v^k denotes the k-th element of sequence v. We will be concerned with Moore machines, although other machine models can also be related to the notion of agents.

Nondeterministic Moore machines

Moore machines are straightforward. Since $\text{inp}(M) = \Pi\,\text{config}(M)$, and Π is a projection, we can say that $\text{config}(M) = \text{inp}(M) \times \text{res}(M)$ where $\text{res}(M)$ represents the "results" of M, i.e., its states and outputs. The we can define Π_r to be the projection such that $\text{res}(M) = \Pi_r\,\text{config}(M)$. A Moore machine is a machine such that $(v_a, v_c) \in \text{steps}(M)$ and $\Pi_r v_c = \Pi_r \hat{v}_c$ implies $(v_a, \hat{v}_c) \in \text{steps}(M)$. In other words, the transition relation determines the next state, but not the next input.

This definition of a Moore machine given here differs from the standard description, which is usually given in terms of a next-state function and an output function. The skeptical reader might take a moment to verify that our definition is in fact equivalent to the more standard one.

Mealy machines

Though we will not use Mealy machines in this thesis, they can be similarly defined. A Mealy machine is more complex, because it distinguishes state from output. Formally the results, $\text{res}(M)$, are split into outputs and states, that is, $\text{res}(M) = \text{out}(M) \times \text{states}(M)$. We can define projections Π_o and Π_s according to the equations $\text{out}(M) = \Pi_o\,\text{res}(M)$ and $\text{states}(M) = \Pi_s\,\text{res}(M)$ respectively. We then define a relation $\text{outrel}(M)$, which is any subset of $\text{config}(M)$ such that for every $i \in \text{inp}(M)$ and every $s \in \text{states}(M)$ there is an $o \in \text{out}(M)$ such that $(i, s, o) \in \text{outrel}(M)$. In other words, for each input and state there is at least one output. Finally, we say that a Mealy machine is one such that for any $(v_a, v_c) \in \text{steps}(M)$ and for any $\hat{v}_c \in \text{outrel}(M)$, for every $\Pi_s v_c = \Pi_s \hat{v}_c$ it is the case that $(v_a, \hat{v}_c) \in \text{steps}(M)$.

In other words, the set $\text{res}(M)$ denotes the outputs and feedback of the machine, but excludes its inputs. Since a Mealy machine distinguishes outputs from state, we use Π_o in our model to identify the outputs, and Π_s to identify the state. The condition that defines a Mealy machine is that the inputs and current state at any time determine the possible outputs at the same time, and the state at the next time. So the transition relation determines the next state, but not the next input or output.

[8]The number 2 occurs since we number sequences beginning with 0, and there are one fewer transitions (fence rails) than configurations (fence posts).

Henceforth we will confine our attention to Moore machines. This ensures that our model reflects the delay between stimulus and response present in any physical system.[9] We will often speak of states when strictly speaking we mean configurations (i.e., states and inputs).

3.3 Core of a specification language

Having considered what a machine is, we turn now to a way of expressing the desired behaviors of machines. We will express desired behaviors by defining machines that exhibit them, using a declarative specification language. Fundamentally, the specification language is rather simple. A specification consists of the declaration of some state variables, followed by some assertions. The state variables determine the state space[10] of the machine being specified. The assertions determine its transition relation.

Each assertion consists of two parts, an antecedent and a consequent. Each of these is a formula written in a restricted logic, and contains occurrences of two kinds of variables: the state variables, and the case variables.[11] Case variables are variables which appear in the specification, but are not a component of the state space. Their role is simply to keep specifications concise.

The antecedent and consequent are formulas, which are restricted to a few simple forms: either 1) an equivalence, containing exactly one state variable, which is its left-hand side, or 2) an implication whose antecedent involves only case variables, or 3) a conjunction of sub-formulas, or 4) an existential quantification of a case variable.

These restrictions are needed to apply trajectory evaluation (see section 7.4, p. 172) to check specifications. They do restrict the form of specifications. For example, to express the equivalence $a = b$ when a and b are state variables, it is necessary to write $a = c \wedge b = c$ where c is a case variable. This ensures that no additional variables need be introduced before trajectory evaluation. In practice, for many circuits the restrictions are not burdensome. Disjunction $f \vee g$ can be synthesized by $\exists b.b \rightarrow f \wedge \overline{f} \rightarrow g$. The only logical connective then lacking is negation.

This section gives the meaning of the key constructs of a specification language for finite-state systems. Many features of the full specification language, such as its function definition facility, are rather standard. While such features are necessary in a practical language, their formal treatment does not contribute markedly to the understanding of a verification methodology.

---

[9] This observation is due to Victor Yodaiken.

[10] Strictly speaking, they determine the *configuration* space.

[11] The reason to distinguish these two kinds of variables was discussed in section 2.1.4 on page 28.

Thus, we begin by giving the syntax and semantics for a core subset of the specification language. This subset encompasses the language's interesting features. Although the portions of the language that are not treated here contribute much to the language's usefulness, they do not markedly affect its semantics. Including a complete semantics for the language would obscure the essential point of this section: that a specification denotes a state machine.

3.3.1 A core subset

A specification in our language consists of three sections, according to the syntax illustrated in Figure 3.8.[12] The first section defines some types, the second section defines the state or *configuration* space, and the third section defines the transition relation.

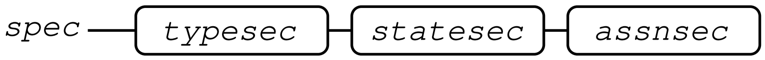


Figure 3.8: Syntax of specifications (language subset). The `spec` is the specification. The `typesec` is the type section. The `statesec` is the state-variable section. The `assnsec` is the assertion section.

The type section gives symbolic names to one or more types, according to the syntax in Figure 3.9.

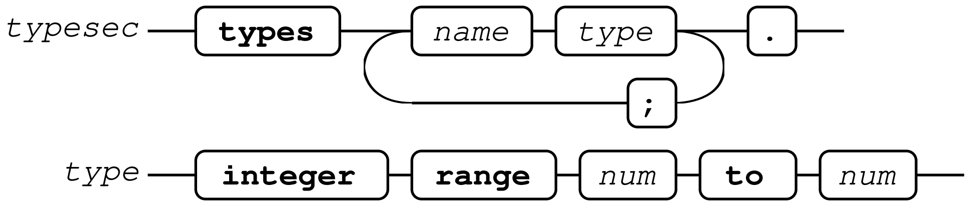


Figure 3.9: Syntax of type definitions (language subset). The `typesec` is the type section. A `num` is a number.

The state-space section names a set of state variables, according to the syntax in Figure 3.10. There are two kinds of state variables: scalars and arrays. Arrays are indicated by the presence of a subscript type.

[12] In this and the related syntax diagrams, literal values are shown in a bold face, while terminal and nonterminal symbols are shown in italics.

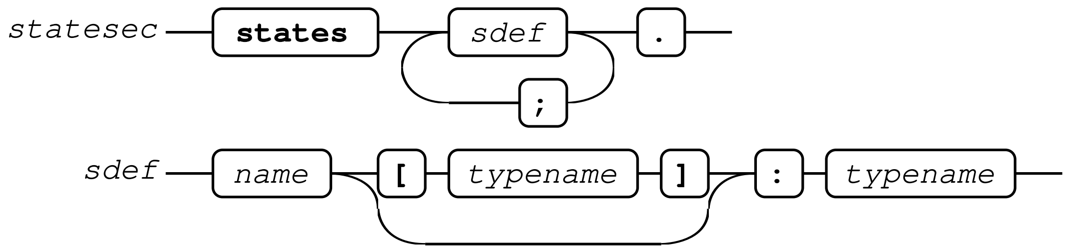


Figure 3.10: Syntax of state variable declarations (language subset). The `statesec` is the state-variable section. A `sdef` is a state-variable definition. A `typename` is a name previously defined in the type section.

The transition-relation section defines a transition relation by means of a set of assertions, according to the syntax in Figure 3.11. Each assertion consists of a set of case-variable declarations,[13] followed by a pair of formulas, the antecedent and the consequent of the assertion. A case-variable declaration consists of a series of declarations, which associate types with variable names.

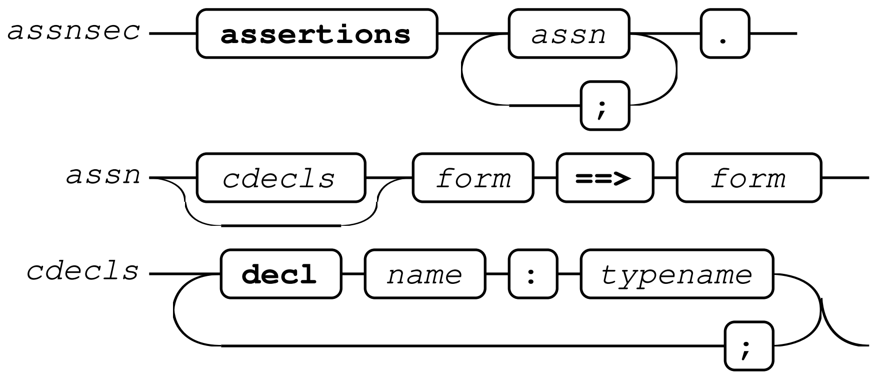


Figure 3.11: Syntax of assertions (language subset). The `assnsec` is the assertion section. An `assn` is an assertion. The `cdecls` are the case-variable declarations. A `form` is a formula.

Each of the formulas in an assertion has a recursive structure according to the syntax in Figure 3.12. A formula can be a case restriction, a conjunction, a primitive scalar or array formula, or an existentially quantified formula.

[13]Chapter 2 discussed the distinction between state variables and case variables.

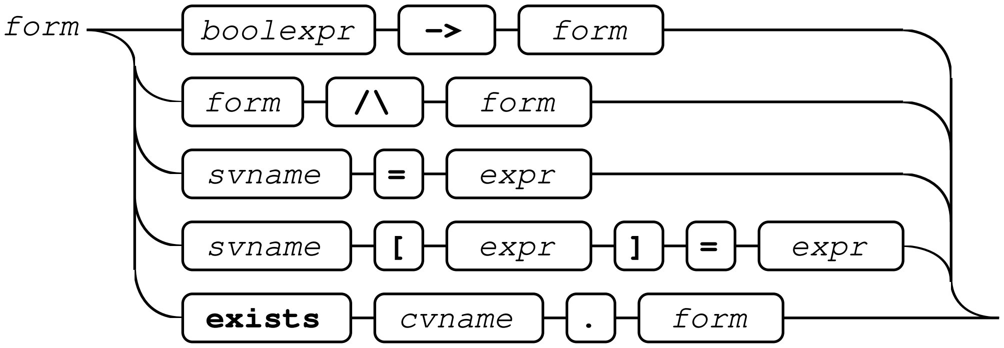


Figure 3.12: Syntax of formulas (language subset). A `form` is a formula. A `boolexpr` is a boolean expression. An `expr` is an expression. A `svname` is a state-variable name. A `cvname` is a case-variable name.

Expressions in the subset are quite simple: they can be constants, case variables, or sums, according to the syntax in Figure 3.13.

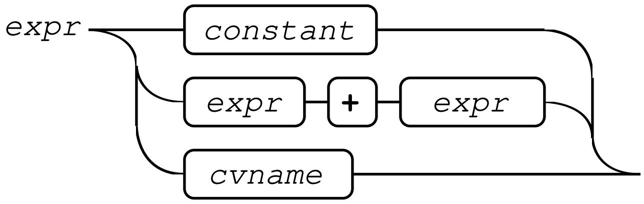


Figure 3.13: Syntax of expressions (language subset)

Example

The language subset retains the unique features of the specification language. With only minor variations, the latch example we saw earlier can be specified with the subset. The latch example of Figure 2.8 is repeated in Figure 3.14 using only the essential subset language. The only difference is that the choice of latch operations must be encoded as a bit rather than given a symbolic name.

The goal of the semantics of the specification language is to provide a formal definition of the transition relation from the syntax of the specification language. We turn now to the semantics. The following section gives the semantics of the

```
types bit integer range 0 to 1.
state op : bit;
       D : bit; Q: bit.
assertions
  decl b : bit;
      op = 1  /\  D = b  ==>  Q = b ;

  decl c : bit;
      op = 0  /\  Q = c  ==>  Q = c .
```

Figure 3.14: Abstract specification of a latch in the subset language. Compare to Figure 2.8.

subset language, then illustrates with an example how this assigns the transition relation of a latch to the text of the latch example.

3.3.2 Semantic equations

We can define the meaning of a specification formally by organizing a set of equations around the syntax of the language. The following equations define the semantics. Types are denoted by sets. The letter E refers to an *environment*, i.e., a mapping from names to types (sets). The letter N refers to the set of *system variables,* i.e., those variables which describe the state space (and input space) of the system being defined. The letter V refers to the current set of *case variables,* i.e., those variables which describe the cases being analyzed or distinguished.

A type expression denotes a set. A type definition augments its current environment E with a new binding. A sequence of type definitions operate in succession. The type definition section starts from the empty environment, i.e., the one containing no bindings.

$$[[\texttt{integer range}\, l\, \texttt{to}\, h]] = \{l, l+1, \ldots, h-1, h\}$$
$$[[name\ type]](E) = E{:}name \mapsto [[type]]$$
$$[[typedef\,;typedefs]](E) = [[typedefs]]([[typedef]](E))$$
$$[[\texttt{types}\, typedefs\ .]] = [[typedefs]](\emptyset)$$

A scalar system-variable declaration augments its current environment with a new binding which maps the name of the new variable to its type (i.e., set). It also augments the set of state variables with the name of the new variable. A vector declaration does the same. However, the type is more complicated; it is the vector's value type, raised to a Cartesian power given by the size of its index type. In other words, it "contains" an element of the value type for each location in the vector. A sequence of variable declarations operates in succession. The state-variable section starts with the empty set of state variables.

$$
\begin{aligned}
&[[\mathit{name} : \mathit{typename}]](E,N) = (E : \text{name} \mapsto E(\text{typename}), N \cup \{\text{name}\}) \\
&[[\mathit{name}\,[\,\mathit{typename}_i\,] : \mathit{typename}_v]](E,N) = \\
&\qquad \Big(E : \mathit{name} \to E(\mathit{typename}_v)^{|E(\mathit{typename}_i)|}, N \cup \{\mathit{name}\}\Big) \\
&[[\mathit{sdef}\,;\mathit{sdefs}]](E,N) = [[\mathit{sdefs}]]([[\mathit{sdef}]](E,N)) \\
&[[\texttt{state}\ \ \mathit{sdefs}\ \ .]](E) = [[\mathit{sdefs}]](E,\emptyset)
\end{aligned}
$$

A case-variable declaration augments its current environment with a new binding which maps the name of the new variable to its type. It also augments the set of case variables with the name of the new variable. A sequence of case variable declarations operates in succession.

$$
\begin{aligned}
&[[\texttt{decl}\ \ \mathit{name} : \mathit{typename}]](E,N,V) = \\
&\qquad (E{:}\mathit{name} \mapsto E(\mathit{typename}), N, V \cup \{\mathit{name}\}) \\
&[[\mathit{cdecl}\ \ ;\ \ \mathit{cdecls}]](E,N,V) = [[\mathit{cdecls}]]([[\mathit{cdecl}]](E,N,V))
\end{aligned}
$$

Thus far we have required only types and variables. We will soon have need of *assignments,* i.e., valuations for sets of variables. Ultimately, however, we wish to define state machines. The most natural way to think of the state in a state machine whose state is determined by several values, is as a cross product S which is the cross product of the possible values that each state element can take on. It is convenient to abbreviate the set of possible valuations, and the set of possible states, using Ψ and S respectively.

$$
\begin{aligned}
\Psi(E,N) &= \Big\{\psi{:}N \to \bigcup_{n\in N} E(n){:}\{n\} \to E(n)\Big\} \\
S(E,N) &= \times_{n\in N} E(n)
\end{aligned}
$$

Since the valuations are intended to be states, we need to show that the two functions above are isomorphic. This is clear since each state in S chooses for each $n \in N$ an element of $E(n)$, and this is exactly what any assignment in Ψ does.

We can define a similar notion for the case variables and the set of cases that they define.

$$
\begin{aligned}
\Phi(E,V) &= \Big\{\phi{:}V \to \bigcup_{v\in V} E(v){:}\{v\} \to E(v)\Big\} \\
C(E,V) &= \times_{v\in V} E(v)
\end{aligned}
$$

Now knowing what a valuation is, we can define the meaning of expressions. An expression consisting of a case variable name is interpreted relative to some particular case, i.e., relative to a valuation of the case variables. A constant expression needs no interpretation. A compound expression is interpreted by interpreting its pieces and applying its operator.

$$[[cvname]](E,N,V,\phi) = \phi(cvname)$$
$$[[constant]](E,N,V,\phi) = constant$$
$$[[expr_1 \ + \ expr_2]](E,N,V,\phi) = [[expr_1]](E,N,V,\phi) + [[expr_2]](E,N,V,\phi)$$

Next, we can define the meaning of formulas. A formula is interpreted relative to some particular state, i.e., relative to a valuation of the state variables. We think of a primitive scalar formula as meaning that some state variable—the one mentioned in the formula—has some value—the value of the expression mentioned in the formula. Thus, we say that a simple formula is true in some case and some state when the state variable, interpreted according to the state, has the value given by the expression, interpreted according to the case.

$$[[svname \ = \ expr]](E,N,V,\phi,\psi) \ = \ (\psi(svname) \equiv [[expr]](E,N,V,\phi))$$

A primitive array formula has a similar meaning. Recall that a state assigns to an array variable an array value, i.e., a Cartesian power, or cross product, of constituent scalar values. Thus, we say that a primitive array formula is true if the selected element of the value of the array variable—i.e., the component given by a projection function determined by the value of the *index* expression (interpreted relative to the case)—has the value given by the *value* expression (also interpreted relative to the case).

$$\begin{aligned}&[[avname\,[\,expr_i\,] \ = \ expr_v]] = \\ &\qquad \left(\Pi_{[[expr_i]](E,N,V,\phi)}\psi(avname) \equiv [[expr_v]](E,N,V,\phi)\right)\end{aligned}$$

A case restriction is a form of implication. It is intended to mean that a formula is true in those cases in which some boolean expression, or guard, is true. Thus, we say that a case restriction holds if the guard is false (relative to the case), or if the subformula is true (relative to the case and the state).

$$\begin{aligned}&[[boolexpr \ -> \ form]](E,N,V,\phi,\psi) = \\ &\qquad \overline{[[boolexpr]](E,N,V,\phi)} \vee [[form]](E,N,V,\phi,\psi)\end{aligned}$$

Conjunction is straightforward, since the meaning of a formula is a logical truth value.

$$\begin{aligned}&[[form_1 \ \ /\backslash \ \ form_2]](E,N,V,\phi,\psi) = \\ &\qquad [[form_1]](E,N,V,\phi,\psi) \wedge [[form_2]](E,N,V,\phi,\psi)\end{aligned}$$

An existential quantifier is intended to mean that a formula is true if there is any value for the quantified case variable which makes the subformula true. Since we have constructed the semantics so that the meaning of the subformula is a logical truth value, we can express the quantifier directly in our metalanguage.

$$\begin{aligned}&[[\texttt{exists}\ \ \mathit{cvname}\ \ .\ \ \mathit{form}]](E,N,V,\phi,\psi) = \\ &\qquad \exists v \in E(\mathit{cvname}).[[\mathit{cvname}]](E,N,V,\phi{:}\mathit{cvname} \mapsto v,\psi)\end{aligned}$$

An assertion is the most subtle element of the state-machine language, and it is still rather straightforward. We think of an assertion as being an implication. However, the verification theory underlying the language treats an assertion as a superset of a state machine's transition relation. Thus, we want an assertion to denote a set of transitions—a set of pairs of states.

The meaning of a bare assertion is given relative to the environment in which it appears. This includes a set of bindings E and a set of state variables N and a set of case variables V. In each of the cases defined by the case variables, the assertion denotes a set of transitions, or "case set." The assertion denotes the intersection of all of these case sets.

$$\begin{aligned}&[[\mathit{form}_a \texttt{=>} \mathit{form}_c]](E,N,V) = \\ &\bigcap_{\phi \in C(E,V)} \left\{(\psi_a,\psi_c) \in S(E,N)^2 \mid \overline{[[\mathit{form}_a]](E,N,V,\phi,\psi_a)} \vee [[\mathit{form}_c]](E,N,V,\phi,\psi_c)\right\}\end{aligned}$$

The meaning of a bare assertions was defined relative to a set of case variables. Case declarations establish such sets. The denotation of an assertion together with its case variable declarations is given by applying the case variable declarations in a context having an empty set of case variable declarations, then applying the meaning of the bare assertion to the result. Thus, an assertion with case variable declarations denotes a set of transitions, defined relative to a set of bindings E and a set of state variables N.

$$[[\mathit{cdecls}\ \mathit{bare\_assn}]](E,N) = [[\mathit{bare\_assn}]]\,([[\mathit{cdecls}]]\,(E,N,\emptyset))$$

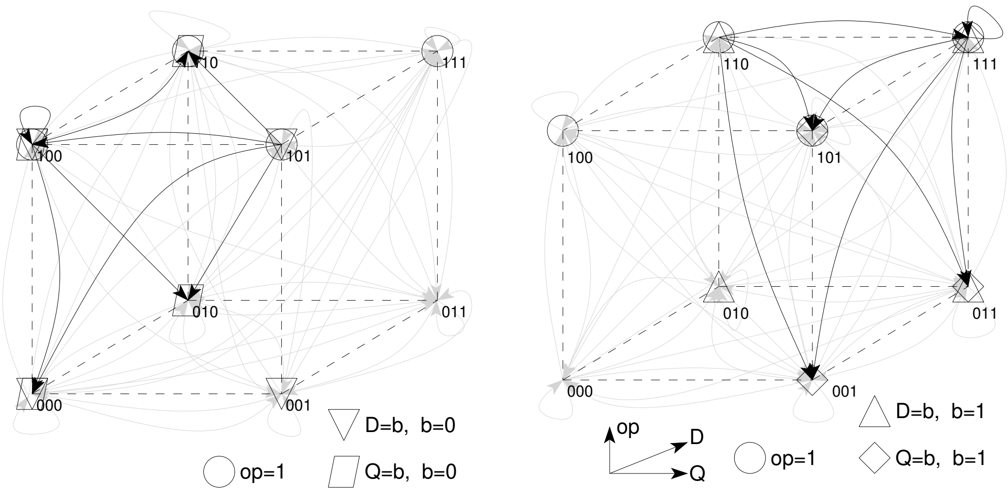
A set of assertions denotes the intersection of the denotations of the individual assertions.

$$[[\mathit{assn}\ \ ;\ \ \mathit{assns}]](E,N) = [[\mathit{assn}]](E,N) \cap [[\mathit{assns}]](E,N)$$

With this machinery in place, the meaning of an entire specification is straightforward. It is given by establishing an initial binding of type names in the type section, augmenting this binding and establishing state variables in the state variable section, and finally determining a set of transitions in the resulting context.

$$[[\mathit{typesec}\ \mathit{statesec}\ \mathit{assnsec}]] = [[\mathit{assnsec}]]\,([[\mathit{statesec}]]\,([[\mathit{typesec}]]))$$

Interpreting a specification using the preceding definitions yields a transition relation defined by a set of assertions. This is exactly the kind of specification that our theory will expect.



a. $b = 0$ case b. $b = 1$ case

Figure 3.15: Transitions associated with individual cases of the "load" assertion of the latch. Transitions given explicitly by the antecedent and consequent are shown in black. Transitions that are implicit from the antecedent are shown in grey.

Example

Given these semantic equations, we can return to the latch specification in Figure 3.14 (page 68) in order to see how it defines a transition relation. The type definition section of the latch specification establishes a mapping from the name `bit` to the set $\{0, 1\}$. The state-variable declaration section augments this mapping so that each of the names `op`, `D`, and `Q` also maps to this set. It also establishes the set of state variable names, $\{\mathtt{op}, \mathtt{D}, \mathtt{Q}\}$. A valuation for the state variables—a state of the specified system—is then a string of three bits giving the values of `op`, `D`, and `Q` respectively.

Figure 3.15 shows the transitions defined by each case of the first assertion.[14] The assertion begins with the declaration of the case variable `b`. Thus there are two cases to consider for this assertion: the case where `b` is 1, and the case where it is 0. Consider the first case. The antecedent of the assertion then is true in the states 100 and 101. The consequent is true in states 000, 010, 100, and 110. Thus the entire assertion, for this case, denotes a total of 56 transitions: 8 for which the antecedent is true, and $6 \cdot 8$ or 48 more for which the antecedent is false. In the second case, the assertion also denotes 56 transitions.

[14]In examining these diagrams, it is less important to see all the details of the first figures than it is to understand how they are combined to produce the final result.

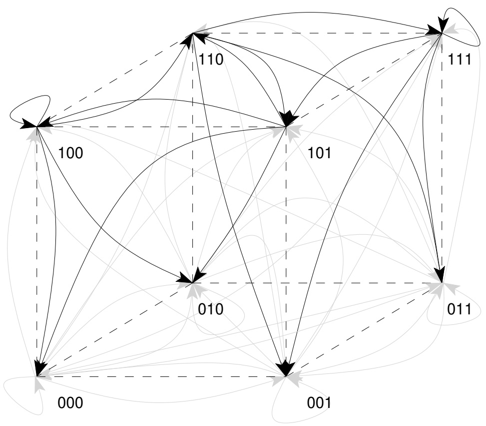


Figure 3.16: Transitions of "load" assertion.

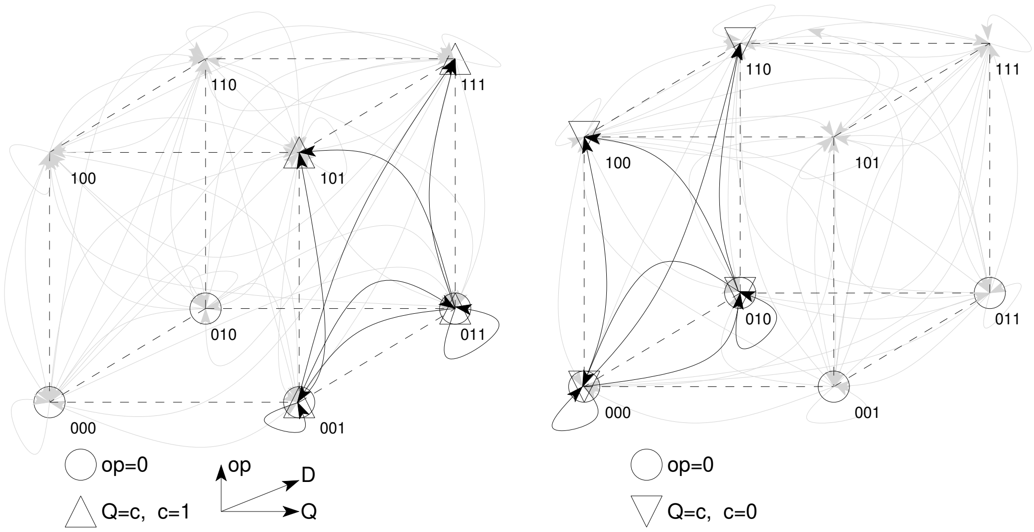
Figure 3.16 shows the transitions defined by the entire assertion—the intersection of the the two cases. The intersection denotes 48 transitions: 16 for which an antecedent is true, plus $4 \cdot 8$ or 32 more for which neither antecedent is true.

Figure 3.17 shows the transitions for each case of the second assertion. In each case there are 56 transitions. When we take their intersection, in Figure 3.18, we also find 48 transitions for this assertion.

Finally, taking the intersection of the sets of transitions from each assertion yields the final transition relation. Figure 3.19 illustrates these transitions.

We can examine this diagram to see that it really represents a latch if we think of the vertices as defining a cube in space. The bottom represents the "hold" operation being applied as an input, while the top represents the "load" operation. The left and right sides represent the two state values. We can immediately see that when the "hold" operation is applied, the state will not change (i.e., there are no transitions at the bottom of the cube that cross from one side to the other). We can also see that when the "load" operation is applied, if the input that is applied is equal to the present state value, the state does not change (i.e., there are no crossing transitions that leave the two upper corners that have self-loops). Finally, we can see that if the "load" operation is applied and the input value differs from the current state, the state will change (i.e., all transitions leaving the other two upper corners cross to the other side of the cube).

Thus, we can see that the entire latch specification denotes the transition relation of a latch.



a. $c = 1$ case b. $c = 0$ case

Figure 3.17: Transitions associated with individual cases of the "hold" assertion of the latch.

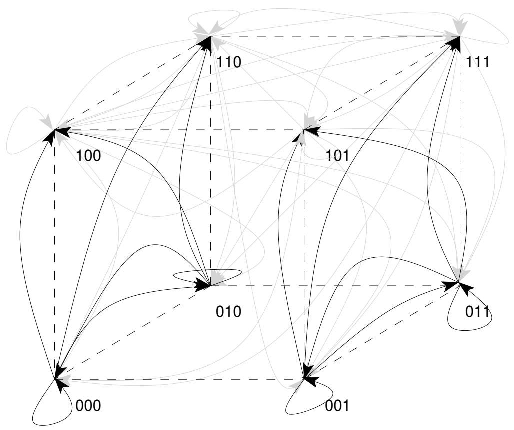


Figure 3.18: Transitions of "hold" assertion

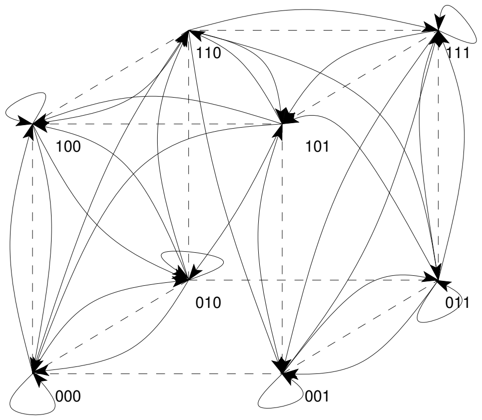


Figure 3.19: Transitions of latch specification

3.3.3 Formalization of assertions

We can relate the semantics of the language to the model of computation we have been discussing, but first we should examine what it is that makes a set of assertions an intuitively attractive and concise device for specification.

Principles underlying the structure of assertions

Assertions structure the set of transitions of a system according to two principles: parameterization and independence. Abstractly an assertion can be thought of as a set of transitions, or equivalently as a pair of sets of states. First, the behavior of many systems can be divided into a few similar parts. (We are mainly concerned with systems whose controllers are simpler than data paths.) For example, a stack has two basic operations, push and pop (plus perhaps a no-op or idle operation, and an initialization operation). We would like to specify each of those similar parts in a parameterized way, independent of the particular data values stored on any particular operation.

Thus, instead of giving the sets A and C of an assertion directly, we give symbolic formula describing them. Such a symbolic assertion represents conceptually many assertions, one for each valuation of the variables that appear in the formulas describing A and C.

Additionally, most systems consist of several parts, only some of which are used on any particular operation. We would like to specify such independent aspects of operation in an independent way. For example, a memory circuit has many storage locations. We would like to specify that each location retains data independently of any operation performed on a different location.

The key to both of these requirements is that we be able to check different sets of transitions separately, and somehow combine the results. As we shall see, assertions provide a mechanism for meeting both requirements. We now formally define an assertion.

Definition 16 (Assertion) *An assertion N over a Moore machine M is a pair A, C of sets where each of A and C is a subset of* $\mathrm{config}(M)$. *The set A is called its antecedent, and the set C is its consequent.*

Subsequently we will abbreviate $\mathrm{config}(M)$ with $\mathbf{U}_M$.

As we discussed in section 3.2.2, the transition relation of M is a subset of $\mathbf{U}_M^2$ (i.e., a subset of $\mathrm{config}(M)^2$). An assertion represents a superset of the transition relation. More specifically, the transition relation is *defined* to be the intersection of all the sets that the individual assertions represent.

Denotation of assertions

An assertion $N = (A, C)$ with antecedent A and consequent C represents a superset of the transition relation of M. We will denote this set by $T(N)$ (to be read "transitions of N"). Formally, it is given by $T(N) = A \times C \cup \overline{A} \times \mathbf{U}_M$. This reflects the intended meaning of an assertion: that when the system is in a configuration within set A, it must next be in a configuration within set C, but if the system starts in a configuration outside A, the assertion imposes no restriction on the configuration it will next be in. This is the meaning of a single assertion.

Let i index the set of assertions. The meaning of a set of assertions is simply the intersection of the meaning of each of its members, $\bigcap_i T(N_i)$.

3.4 Syntax of a specification language

This section defines the full syntax of SMAL, our specification language for writing assertions. (The processor specification in Appendix B of this thesis is given in a very similar notation.)

A specification in SMAL begins with a head, which is followed by its body. The basic elements of a specification are its type declarations, the definition of its state variables, and the assertions which define the state transitions. The syntax is illustrated by a series of figures. The top-level syntax is shown in Figure 3.20. The language is an extension of the core subset given previously. While it contains no essential elements that the subset lacks, it includes more of the typical convenient features found in programming languages, such as function definitions.

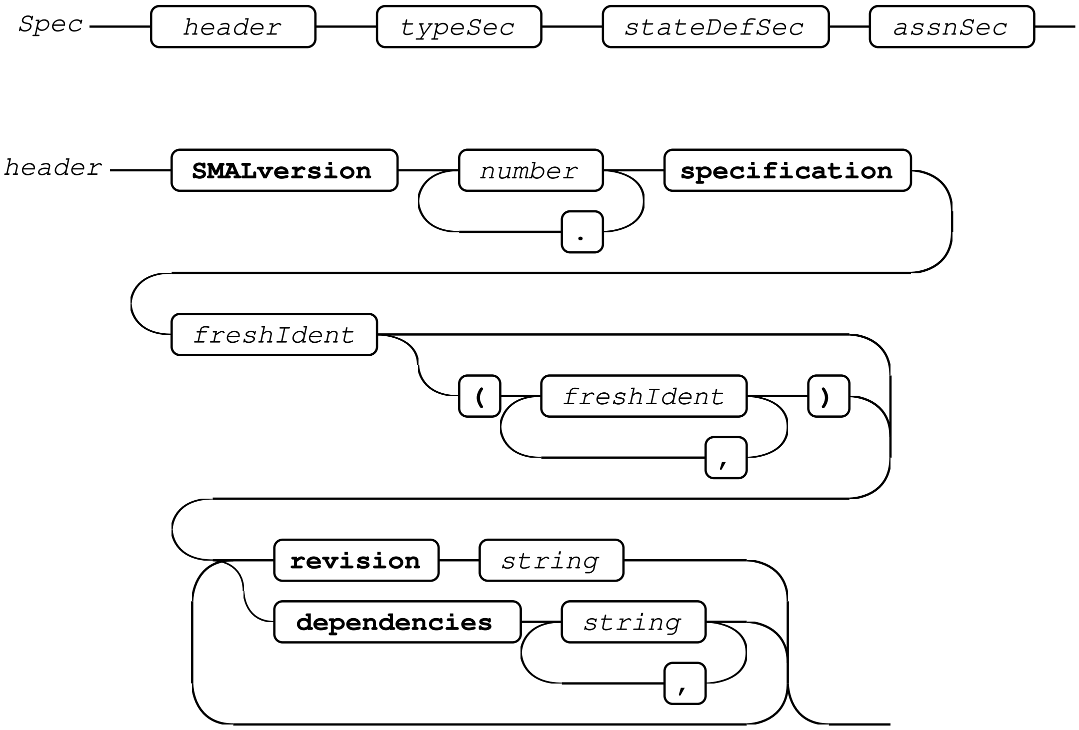


Figure 3.20: Syntax of specifications and their headers. A specification consists of several sections. A header declares parameters and gives some identifying information. The `Spec` is the specification. The `typeSec` is the type-definition section. The `stateDefSec` is the state-variable declaration section. The `assnSec` is the assertion section. A `freshIdent` is a "fresh," i.e., unused, identifier.

3.4.1 Type declarations

The type declarations give names to types. This makes it convenient to declare the types of case variables as they are introduced, by referring to type names. The basic types allowed include words containing different numbers of bits, subranges of the integers, and enumerations, as in Figure 3.21.

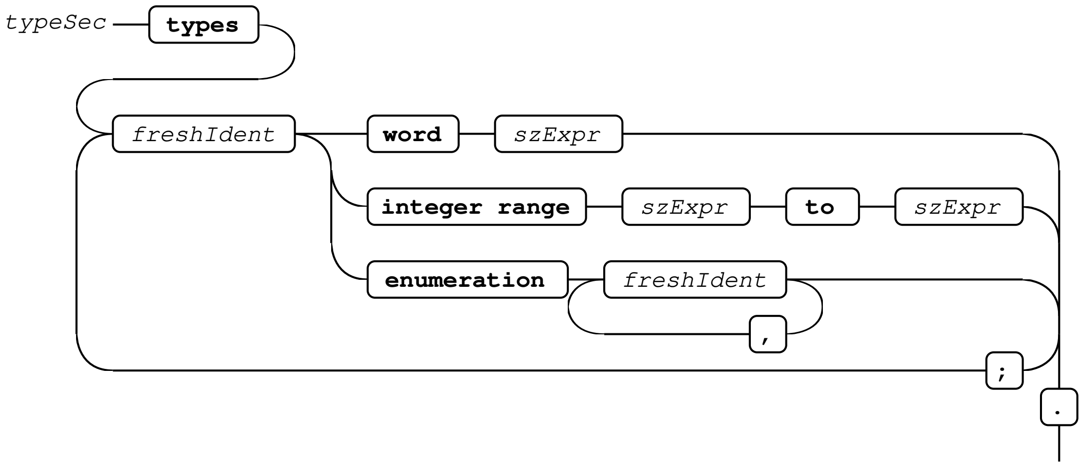


Figure 3.21: Syntax of type declarations. Words, integers, and explicit enumeration types may be defined. A `szExpr` is a restricted, size expression.

The set of expressions that can be used in type definitions is restricted, as shown in Figure 3.22. Only constants, parameters, and a few common functions are allowed. (A library of functions such as the logarithm function log, and conversions between integers and bit strings according to standard encodings, should be provided by the language implementation.) The restrictions ensure that the expressions refer to definite values when the specification is instantiated.

Type checking is a powerful tool for quickly finding many of the errors that are bound to occur in rigorous formal descriptions written by people. Outside computer science, it goes by many names; for example, in engineering it is often called "dimensional analysis." Having type declarations makes it possible to introduce case variables freely, yet do some real checking.footnoteOmitting them might be possible; this would require including a type-inference system [87]. Even with declarations, it is still necessary to perform some interval arithmetic, and algebraic constraint propagation during type checking.

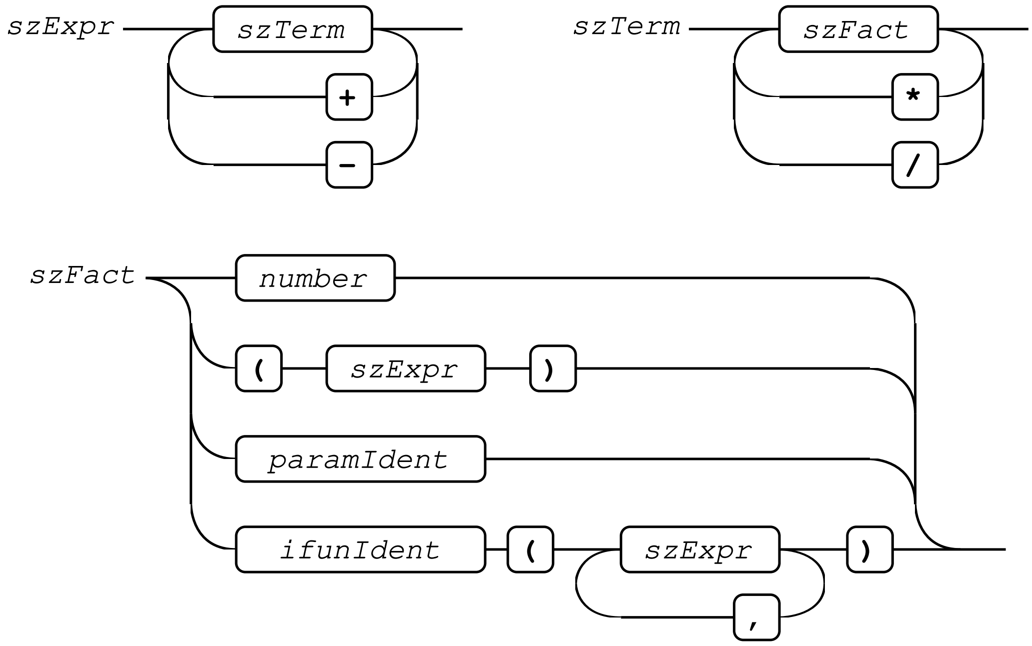


Figure 3.22: Syntax of restricted expressions. Only expressions which evaluate to constants, or which depend only on specification parameters, are allowed. A `szTerm` is a term of a size expression. A `szFactor` is a factor of a size expression. A `paramIdent` is an identifier denoting a parameter. A `ifunIdent` is an identifier denoting an integer-valued function.

3.4.2 State variables

State variables define the state space of the machine being defined. State variables may be either scalar or array. Array variables must be indexed by an integer subrange type. This syntax is shown in Figure 3.23.

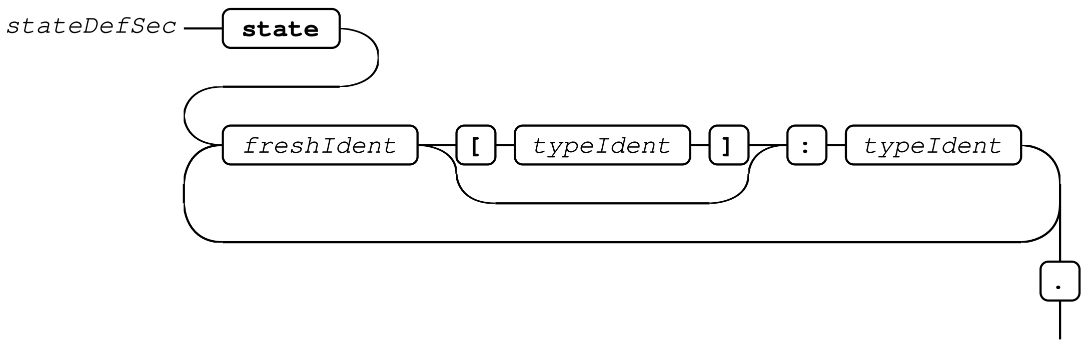


Figure 3.23: Syntax of system state variable section. If a subscript appears, an array is defined. A `typeIdent` is an identifier denoting a type.

3.4.3 Assertions

Assertions are the heart of the specification. Global definitions are useful for keeping them concise, so the syntax in Figure 3.24 allows them to precede the assertions.

Each assertion consists of two formulas, called the antecedent and the consequent, and joined by a symbol $\stackrel{\delta}{\Rightarrow}$. (The δ is left implicit and omitted from the actual syntax.) In this language, the notion of time advancement, inherent in the idea of a state transition, is implicit in this symbol. An assertion can also have its own local definitions. In addition, case analysis is possible at the assertion level.

The syntax of an assertion is given in Figure 3.25.

3.4.4 Formulas

Formulas can be constructed from primitives or from other formulas. A primitive formula always involves exactly one state variable, and at least one expression. Such a formula denotes the set of states in which the state variable has the value given by the expression. If the state variable in a primitive formula is an array, the formula also contains an expression giving the index into the array, and the formula denotes the set of states in which the indicated element of the state-variable array has the indicated value. Formula syntax appears in Figure 3.26.

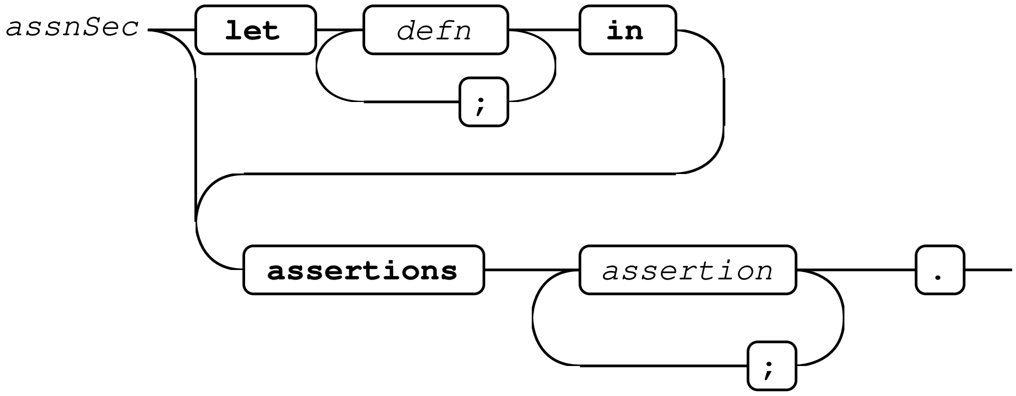


Figure 3.24: Syntax of assertion section. A `defn` is a function or constant definition.

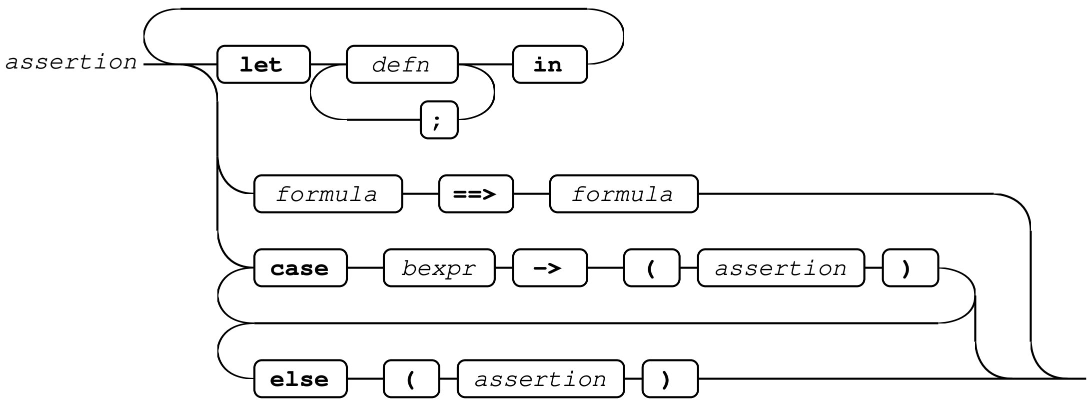


Figure 3.25: Syntax of an assertion. Each assertion may have its own local definitions and case analysis. A `bexpr` is a Boolean expression.

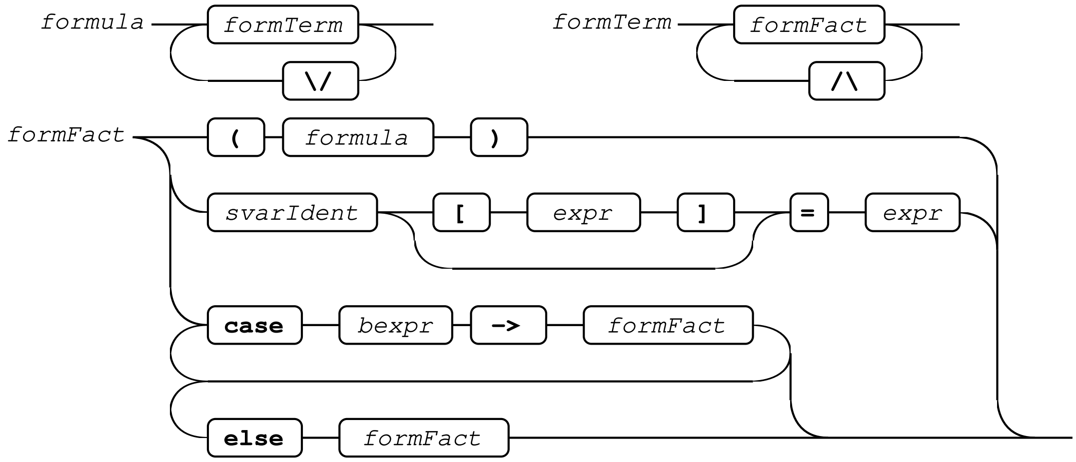


Figure 3.26: Syntax of a formula. A `formTerm` is a term of a formula. A `formFact` is a factor of a formula. An `expr` is an expression. A `svarIdent` is an identifier denoting a state variable. A `bexpr` is a Boolean expression.

Conjunction

Formulas can be combined by conjunction, which corresponds to taking the intersection of the sets they denote. While disjunction is also possible, it is likely to be implemented inefficiently[15] in practice.

Case restriction

Formulas can also be built via case restriction. Case restriction takes a formula and a Boolean condition, and produces a formula that holds when either 1) the Boolean condition is false, or 2) when the original formula holds. The Boolean condition must be composed of expressions, relational operators, and logical connectives. Note, in particular, that state variables cannot appear in the Boolean conditions.

Case restriction corresponds to requiring that a condition hold only in certain of the cases denoted by the case variables.

[15]Disjunction can be implemented by the $\exists$ quantifier. It introduces a Boolean variable, but without good context in which to chose its position in the variable ordering of a BDD-based implementation. This is likely to be inefficient if frequently used. Further discussion of BDDs and variable ordering will appear in Chapter 7.

3.4.5 Expressions

Expressions contain case variables, and denote values. They are not strictly fundamental to the language, in the sense that specifications could be written without them, by expanding to eliminate case variables, and giving constant values explicitly. Expressions are needed to keep the language concise.

The syntax of an expression varies depending on its type. There are four types of expressions. Three of them can occur in several general contexts, as shown in Figure 3.27.

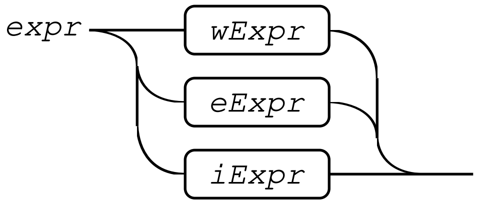


Figure 3.27: Syntax of general expressions. There are three possible kinds. A `wExpr` is a word-valued expression. A `eExpr` is an enumeration-valued expression. A `iExpr` is an integer-valued expression.

Boolean expressions, given in Figure 3.28, are the fourth type of expression. They occur only as guard conditions in **case** statements, and are built from relational operators over other expressions.

Expressions denoting words of bits have the most complicated syntax, in Figure 3.29. The figure looks complicated but it is actually straightforward if read from top to bottom. A word expression may be parenthesized, given by case analysis or a constant, given by a binary operator (either concatenation, or a bitwise Boolean operator) or by a unary operator (negation, *or*-reduction, and *and*-reduction). They may also be variable references, either to new variables declared on-the-spot with their type), or to existing variables. Finally, they may be given as functions of other expressions, or as the extraction of a sub-field of a larger word.

Integer expressions, in Figure 3.30, and enumerated expressions, in Figure 3.31, are straightforward.

3.4.6 Local definitions

In addition to the more fundamental elements described above, it is useful to have local definitions. Local definitions are not usually necessary when specifying circuits that just tend to shuffle data around, such as stacks, queues, register files, and other memories. However, in order to specify circuits that perform computation,

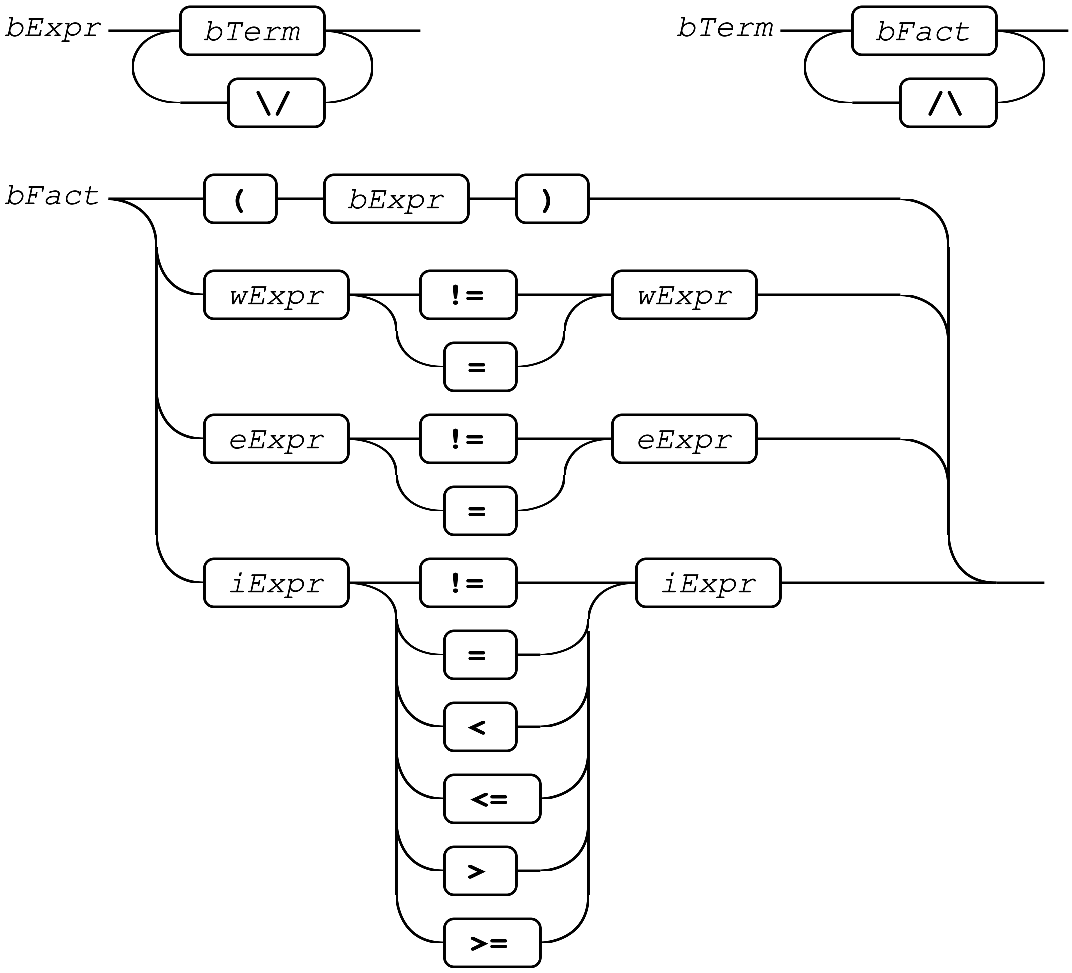


Figure 3.28: Syntax of Boolean expressions. A `bTerm` is a term of a Boolean expression. A `bFact` is a factor of a Boolean expression.

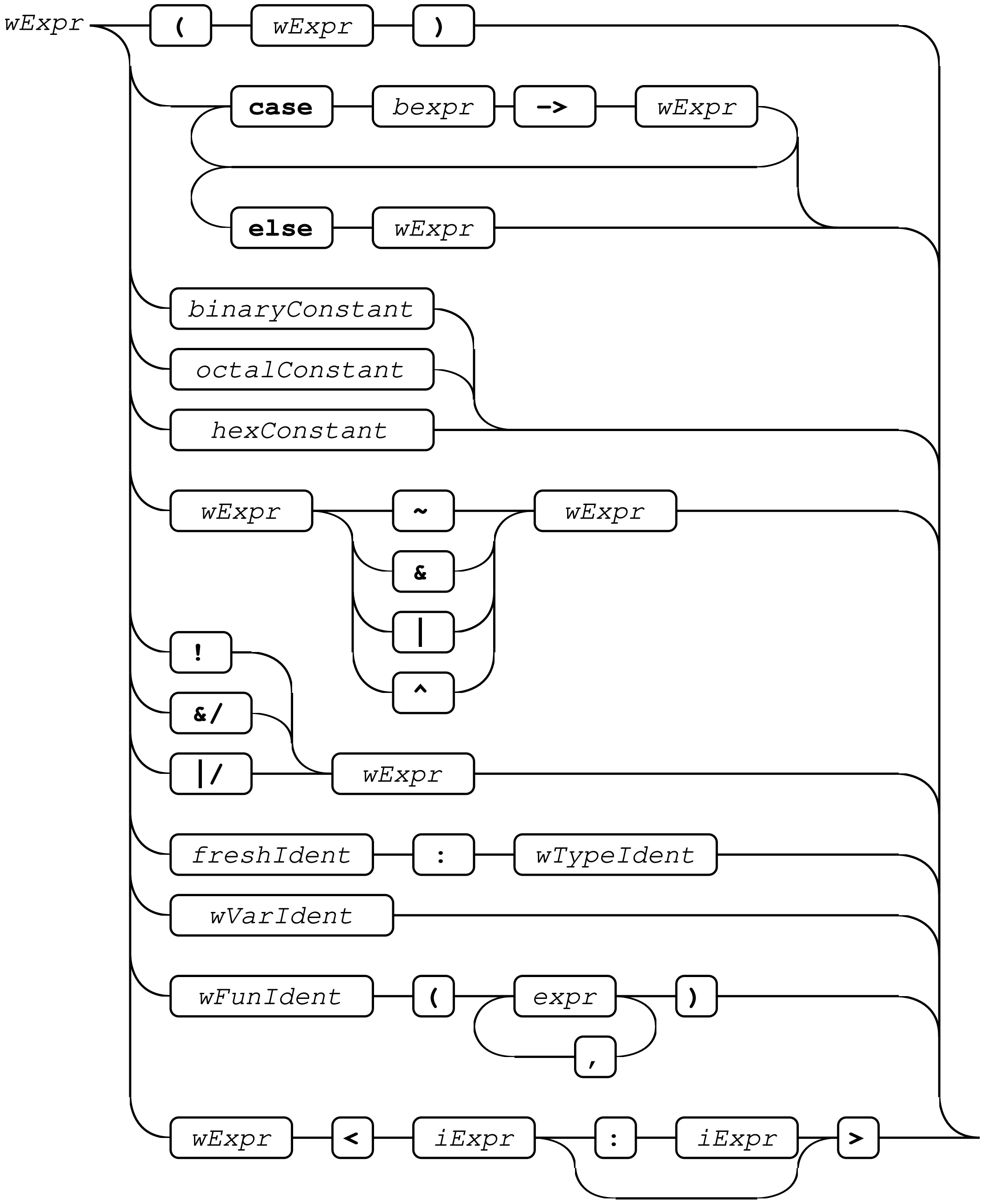


Figure 3.29: Syntax of word expressions. A `wTypeIdent` is an identifier denoting a word type. A `wVarIdent` is an identifier denoting a case variable of word type. A `wFunIdent` is an identifier denoting a function returning a value of word type.

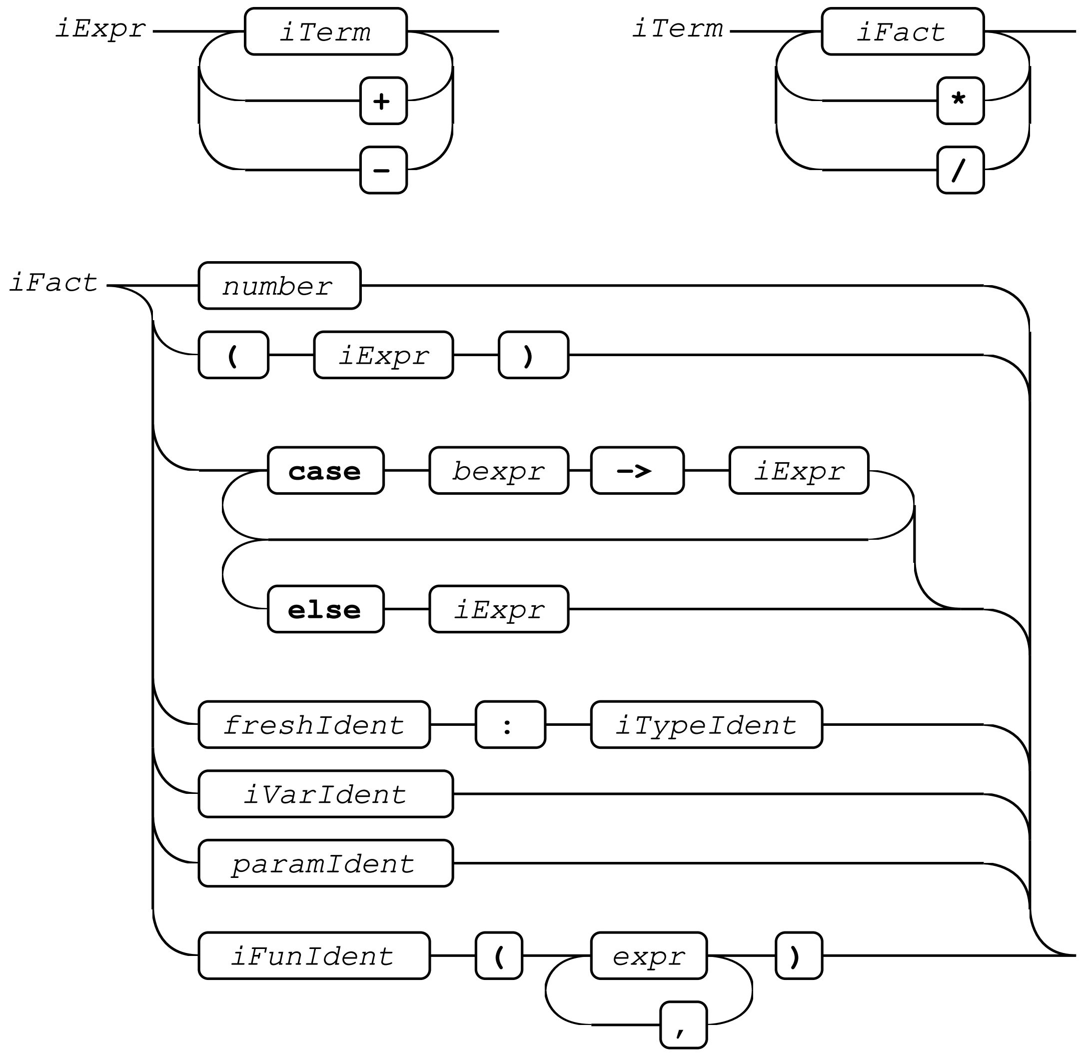


Figure 3.30: Syntax of integer expressions. An `iTerm` is a term of an integer expression. An `iFact` is a factor of an integer expression. An `iTypeIdent` is an identifier denoting an integer type. An `iVarIdent` is an identifier denoting a case variable of word type. A `paramIdent` is a parameter. An `iFunIdent` is an identifier denoting an integer-valued function.

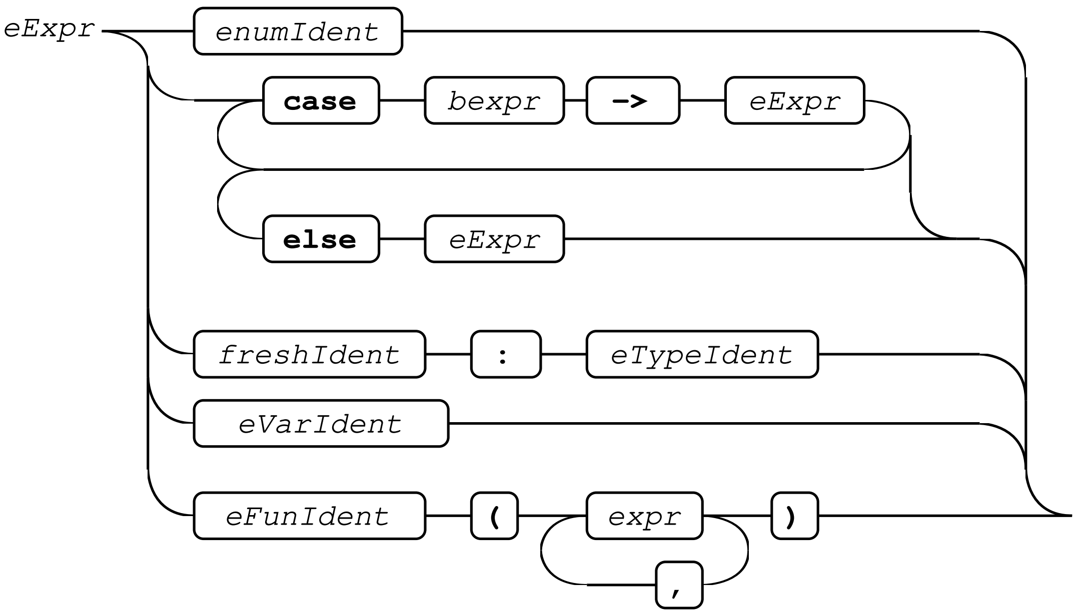


Figure 3.31: Syntax of enumeration expressions. An `enumIdent` is an identifier denoting an enumerated value. An `eTypeIdent` is an identifier denoting an enumerated type. An `eFunIdent` is an identifier denoting an enumeration-valued function.

it is necessary to specify the computation. In such systems, often an operation will update the values of several state variables according to some computational result. To keep the specification concise, local definitions are necessary.

Two separate scopes of definitions are convenient. First, constants and functions of global scope must be available to all assertions. Second, some individual assertions will have need of their own local definitions. Figure 3.32 gives the syntax for definitions.

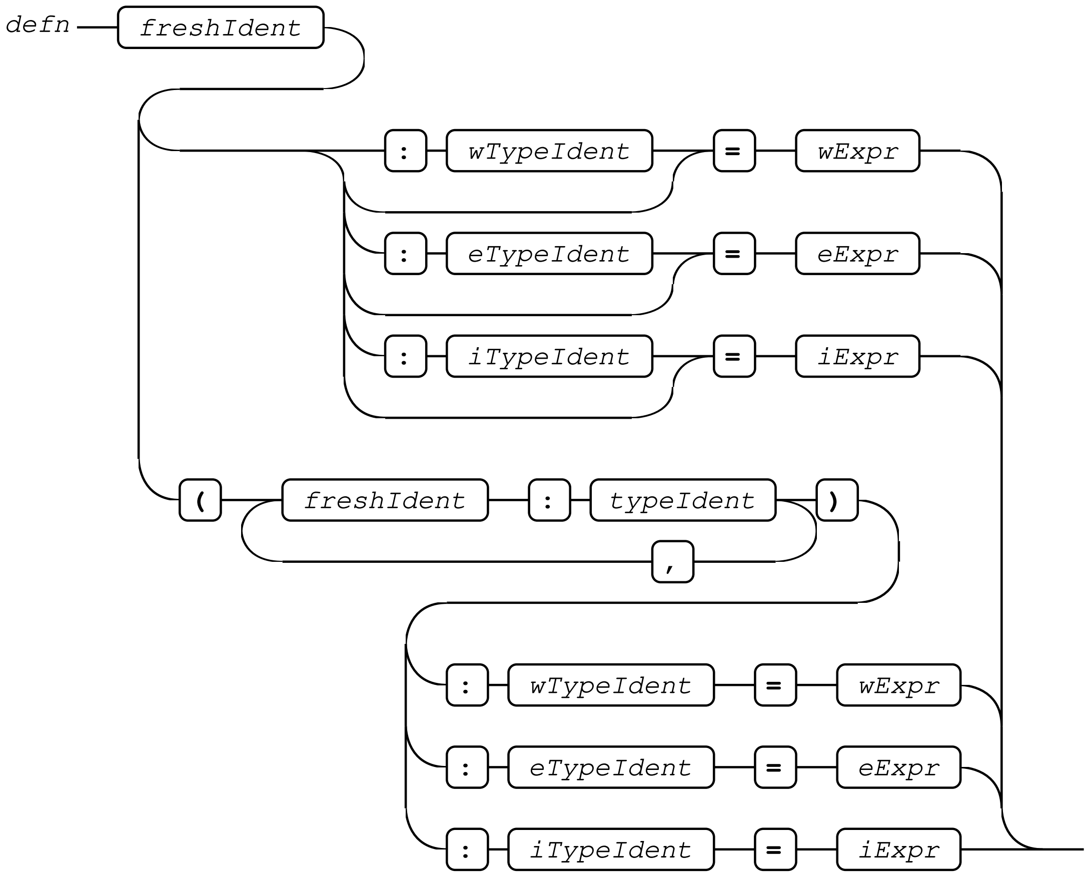


Figure 3.32: Syntax of definitions of constants and functions

Definitions of simple values are useful for many things. Properly used, they provide mnemonic, meaningful symbolic names. For example, in a microprocessor, they can be used to define names for the operation codes. They can also be used in assertions to eliminate the repetition of common sub-expressions, to help keep specifications concise.

Function definitions are also important. For example, in a CISC microprocessor, each addressing mode combination requires its own assertion, but the computations performed by the instructions can be abstracted out into a small set of common functions that the assertions draw upon.

3.4.7 Examples

To make the preceding ideas concrete, this section discusses some examples. We have now seen the latch (that we first saw in section 2.1) several times. A few more examples illustrate the versatility of the language.

Finite state machine

A language intended to express systems as state machines should certainly be able to describe state machines that have been given in a more conventional style. Figure 3.33 is a textbook example of a four-state machine, having four inputs [247, front cover].

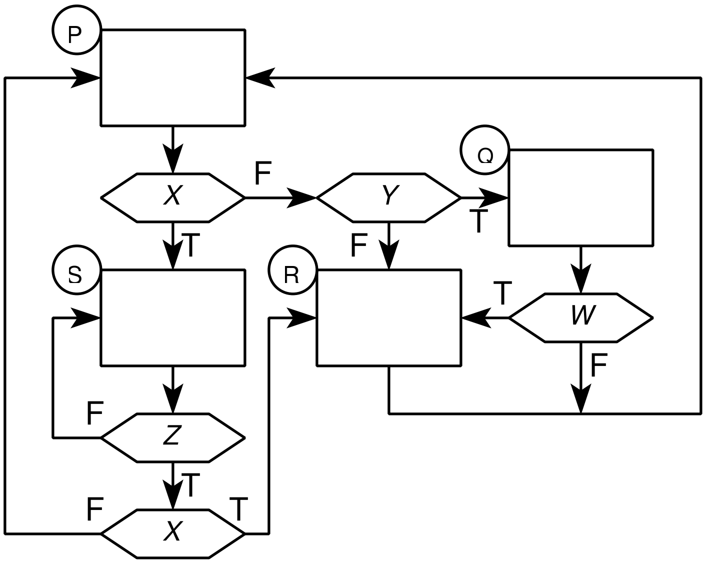


Figure 3.33: The state machine from the cover of Winkel and Prosser [247]

This state machine is easy to describe in SMAL. The specification is shown in Figure 3.34. The state names are enumerated, and the inputs are represented as bits. The state transitions are defined by writing assertions. An assertion can be written to describe each state transition, as in the first three assertions in the figure. Alternatively, branches of the transition structure can be coded more compactly using case analysis. For example, the last assertion in the figure describes all of the possible paths leaving state S.

```
types       boxlabels enum P,Q,R,S.
state       st boxlabels;
            W bit; X bit; Y bit; Z bit.
assertions  st=P /\ X=1        ==> st=S;
            st=P /\ X=0 /\ Y=1 ==> st=Q;
            st=P /\ X=0 /\ Y=0 ==> st=R;
            st=Q /\ W= b:bit   ==> case (b=1) -> st=R else st=P;
            st=R               ==> st=P;
            st=S /\ Z=b:bit /\ X=c:bit
                               ==> case (b=0) -> st=S;
                                   else case (c=0) -> st=P;
                                        else  st=R.
```

Figure 3.34: SMAL definition of the state machine from the cover of Winkel and Prosser [247].

Random-access memory

The goal of SMAL is to describe data-intensive finite-state systems. A simple data-intensive system, but one which has a large state space, is a memory. We can specify a memory using SMAL. The specification is parameterized according to the number of bits in a word, and the number of words that can be addressed.

```
SMAL version 0.0 specification RAM(bits,words).
```

The types define the words of data, the addresses, and the set of possible memory operations.[16]

```
types
    dataWord word bits;       // Word of data stored.
    location integer
      range 0 to words - 1;   // Index identifying location.
    operation
      enumeration read, write // Operations memory can perform.
.
```

The system variables define the I/O signals of the memory, as well as its state. The inputs consist of data, address, and command inputs. The sole output is a data output. The state of the memory is stored in an array.

```
state
```

[16]Here we introduce a comment delimiter of //.

```
    dataIn : dataWord;        // Data given to memory.
    address : location;       // Location address given to memory.
    perform : operation;      // Command given to memory.
    dataOut : dataWord;       // Data produced by memory.
    mem[location] : dataWord // Value is stored in memory.
.
```

The assertions define the operations that the memory can perform. First, writing to the memory should store the provided data.

```
assertions

  // Must be able to put data in a location.
    dataIn=d:dataWord /\ address=i:location /\ perform=write
  ==>
    mem[i]=d
  ;
```

Second, reading from the memory should retrieve the stored data, which should not be destroyed.

```
  // Must be able to read it back out.
    mem[i:location]=d:dataWord /\ address=i /\ perform=read
  ==>
    mem[i]=d /\ output=d
  ;
```

Finally, unaddressed locations should not be corrupted.

```
  // Mustn't perturb idle locations.
    mem[i:location]=d:dataWord /\ address=j /\ perform=op:operation
  ==>
    (i != j) => mem[i]=d
  ;
.
```

This abstract specification of the memory is relatively concise. Similar specifications can be written for similar memories. For example, a dual-ported memory can be specified in terms of the pairs of operations applied on each port.

Stack

Another data-intensive system is one of our initial sources of examples, the stack. As for the RAM, we can parameterize the memory both in the number of locations, k, and the number of bits in a word, w.

```
SMALversion 0.0 specification stack(k, w).
```

The types needed by the stack are similar to those for the memory, but we now need, in addition to a type to represent storage location, a type to represent the number of items stored on the stack. Since the stack can be empty, there are more possible counts of items stored on the stack than there are locations in the stack. This specification of the stack will include an explicit no-op operation.

```
types
  dataWord word w;
  location integer range 0 to k-1;
  fullness integer range 0 to k;
  operation enumeration push, pop, hold
.
```

The state of the stack is similar to the state of the memory, but we must keep track of the number of items stored in the memory.

```
state
  input : dataWord;
  perform : operation;
  depth : fullness;
  output : dataWord;
  stack[:location] : dataWord
.
```

The three stack operations can be defined by separate assertions. The "push" operation places the input on the top of the stack (i.e., location 0 in the storage array), and pushes stored data down toward the bottom of the stack.

```
assertions

 // Push item onto stack (cannot be full).
  case (d:fullness < k)
  -> ( depth=d  /\ input=v:dataWord
      /\ case (i:location < d) -> stack[i]=u:dataWord
      /\ perform(push)
     ==>
      depth=d+1 /\ stack[0]=v /\ case (i < d) -> stack[i+1]=u
     );
```

The "pop" operation places the value from the top of the stack on the output, and pops stored data up from the bottom of the stack.

```
// Pop item off stack (cannot be empty).
 case (d:fullness > 0)
 -> ( depth=d   /\ stack[0]=v:dataWord
     /\ case (0 < i:location < d) -> stack[i]=u:dataWord
     /\ perform(pop)
    ==>
      depth=d-1 /\ output=v   /\ case (0 < i < d) -> stack[i-1]=u
    );
```

The “no-op” operation does not affect any stored data.

```
// No operation.
  depth=d:fullness /\ case (i:location < d) -> stack[i]=u:dataWord
 /\ perform(hold)
==>
  depth=d /\ case (i < d) -> stack[i]=u
```

.

Notice that the specification is careful to exclude cases such as popping an empty stack. Such a specification allows anything to happen in such unspecified cases. If it is necessary that a particular stack exhibit a certain desired behavior in such conditions, that behavior would have to be specified also.

3.5 Related work

3.5.1 Model of computation

Moore machines are a common, established model of computation [185]. The mnemonic notation used for describing machines (e.g., config for the set of configurations) is derived from the notation of Lynch and Tuttle's IO automata [165]. (An early attempt to use IO automata in this research revealed that a simpler model would suffice.) The notion of treating sequential systems as functions from sequences is similar to the string-functional semantics of Bronstein [37, 35, 36], which was further developed by Van Aelten [236, 237]. The idea of using semigroups comes from S. Ginsburg [112]. Ginsburg's generalized notion of an abstract machine is defined in terms of a quasimachine. A quasimachine has inputs and outputs which are semigroups. An abstract machine is a quasimachine whose output semigroup obeys a left cancellation law. In other words, two equivalent computations cannot be made inequivalent by forgetting something that happened long ago.

Process algebra is a model of concurrency originated by R. Milner [181, 182, 183] and very widely developed. The fundamental ideas of process algebra are events (which are either input or outputs), and agents which make state transitions when events occur. Agents can be composed and the events forming their interaction can be hidden. Process algebra is well suited for compositional reasoning about independent agents which communicate (be they gates connected by wires to form an asynchronous circuit, or processes connected by a computer network to form a distributed system) but is not particularly suited for reasoning about functional properties of individual agents, as done in this thesis. Moreover, the work on abstraction in process algebra is poorly suited to data abstraction; to some process algebraicists (e.g., [258]) abstraction is merely the hiding of a set of actions. G. Milne's CIRCAL [180] is an adaptation of process algebra to circuits.

3.5.2 Specification of hardware and processors

Although Davie [89] maintains that it is counter-intuitive to describe a microprocessor as a state machine, we argue that we can achieve a natural specification if the state machine is kept implicit in a declarative, assertion-based notation which describes the instruction set. Boute has advocated declarative description of hardware [29, 28] with a number of varied examples.

There is a large body of work on so-called hardware description languages (HDL's), which are languages used to describe hardware. Most of them are imperative programming languages. Eveking [101, 102] has studied the relationship of conventional HDL's to formal verification. For applications like ours he advocates the use of the notion of interpretation, taken from mathematical logic, as the way to relate different levels of design. His basic approach is to axiomatize HDL's, i.e., to translate constructs of the HDL into statements in the predicate calculus (which he calls "assertions"). This has been done for several levels of description, including the register transfer level and the switch level (incorporating strengths into the value set, after Hayes [131]). Eveking's work has been conducted in the framework of CONLAN, which is a family of HDL's. Eveking advocates the very careful design of HDL's to ensure that the appropriate models can be easily defined. (For example, it is difficult to define synchronous finite-state machines in VHDL, because VHDL lacks the notion of a clock.)

A slightly different approach was taken by Hunt and his colleagues, who recently developed a HDL embedded within the Boyer-Moore logic [34]. Their language is a synchronous HDL—all state is stored in clocked elements. They have used this language in verifying FM9001, a derivative of Hunt's original FM8501 [239]. Interestingly, they developed a simulator to use in early stages of debugging, before attempting formal proof. They also developed a simple translator from their language into a more conventional HDL.

One criticism of their approach—using a specialized prover-based HDL initially, then translating to more conventional form, in contrast to Eveking's approach of putting the desired model of hardware first, then translating to logical formulas—is that it seems less likely to be acceptable to design practice. This point is somewhat moot, however, because contemporary design practice focuses on standardized HDL's such as Verilog and VHDL. Augustin and colleagues have developed an annotation language for VHDL with which to express verification conditions [8].

Assertions

Darringer [88] used assertions in his discussion of verifying hardware descriptions by symbolic execution. Patterson [197] constructed a program verifier and compiler for a high-level microcoding language, based on Floyd's method of inductive assertions [167]. Pitchumani and Stabler [201] extended Floyd's method of inductive assertions to register-transfer programs by treating input signals as arrays, indexed by time.

State deltas are a notation similar in some ways to assertions. They have been used to verify microcode, in a theorem-proving context [160, 168].

Assertions are descriptions, not definitions, of machines. Zave [256] has pointed out that descriptions are better than definitions, since descriptions can be modified by conjoining additional descriptions, while definitions must be made once and for all. Obviously a set of assertions may be extended by addition of new assertions.

3.6 Chapter summary

After establishing a mathematical background, including marked strings, this chapter started by defining an abstract notion of an agent. An agent is a thing having potentially nondeterministic behavior, and it is modeled as a set-valued function, where a copy of the stimulus applied to the agent is (implicitly) retained when examining the agent's response. Then the notion of a machine was introduced. A machine is an agent whose inputs and outputs are sequences. Two particular types of machines, Mealy machines and Moore machines, are sequence machines whose behavior can be determined from a transition relation.

After introducing our model of computation, we introduced our style of specification by giving a subset of a specification language. We used this subset to provide a formal semantics for the language, and we saw that specifications in the language denoted transition relations. We then related the assertions of such specifications back to the model of computation. Finally, we concluded with the syntax of the full language, and examples showing how a finite-state machine, a RAM, and a stack could all be defined in a declarative way.

Specifications are the high level of our verification methodology. We now turn to the low level. Ultimately, of course, we will relate the two.

Chapter 4

Simulation and machines

This chapter discusses switch-level simulation of digital MOS circuits and its relation to abstract machines.

The first section describes the model of realizations. Our realizations are switch-level circuits. Each such circuit can be thought of as defining a Moore machine.

We have chosen the switch level because of previous experience modeling circuits at this level. Use of the switch level model is not a prerequisite of the methodology. Any simulation model could be used, provided that the definition of a Moore machine from excitation functions, given in section 4.2, applies.

There are four requirements that a circuit simulation model must meet in order for our methodology (together with trajectory evaluation) to be applicable to it. The first is that it must be a model in which designs are created. Though this seems an obvious requirement for any approach to verification, it should be repeated because not all models are suitable. For example, verification based on the parsing of circuit graphs according to a graph grammar breaks down when confronted with non-grammatical circuits, and verification tightly coupled to a design hierarchy breaks down if the actual hierarchy, due to implementation constraints, must diverge from the clean, conceptual hierarchy with respect to which verification is easiest. For example, the geometrical hierarchy of a chip design is likely to include busses passing through the middle of a data path element such as an adder, where as conceptually the busses are external to the data path element.

The other three requirements are that the model be monotonic over an "information content" ordering, that it be easily extensible from a value domain to a symbolic domain, and that it be conservative.

A model that operates over a partially ordered set of state values, where the order represents information content, can be quite powerful if the values that carry little information are represented cheaply. Analysis of some "interesting" part of the circuit can proceed, while little effort is expended in dealing with "uninteresting" parts, by maintaining little information about signal values in the "uninteresting" area. The region of interest can be shifted about the system until

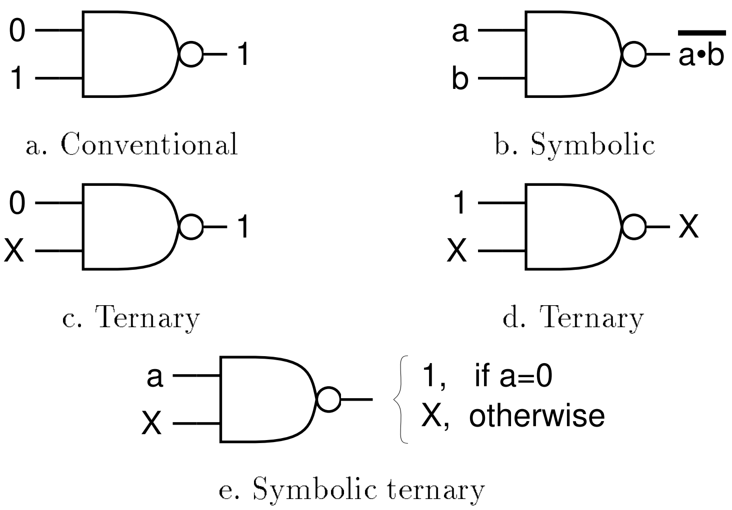


Figure 4.1: Ternary symbolic simulation of a NAND gate.

the entire circuit has been considered. In order for such reasoning to be sound, the model must be monotonic—adding information in one place must not cause information to be lost in another, and vice versa.

The techniques described in this thesis ultimately depend on symbolic simulation,so the simulation model to be used must be easily extensible to symbolic form. In practice this is achieved by using BDDs to represent Boolean functions, and by encoding domains of more than 2 values in terms of several encoding bits. An encoding must be chosen that makes it possible to show that there is monotonicity over the information-content ordering, and which allows the low-information values to be represented cheaply[1] (e.g., by giving all the encoding bits the same value).

Symbolic simulation allows the analysis of many cases at once. Consider the NAND gate shown in Figure 4.1. Conventional simulation is illustrated in part a. Simulator state consists of logic levels. Pure symbolic simulation is illustrated in part b. Here simulator state consists of functions which yield logic levels. The inputs of the gates are the functions $f(a,b) = a$ and $g(a,b) = b$, and its output is the function $h(a,b) = \overline{a \cdot b}$. Partially ordered simulation is illustrated in parts c and d. Here an X value[2] represents the absence of information. In part c we know that if one input is 0, the result will be 1 even if the other input is unknown. In part d we see that if one input is 1 and the other is unknown, the result will be unknown also.

[1]otherwise, "throwing away" information would yield no performance benefit

[2]We emphasize that X is a value, an element of the ternary set $\{0, 1, X\}$, and not a variable.

It is when partially ordered simulation is combined with symbolic simulation that even more interesting results can be achieved. For example, part e shows that when one input is unknown and the other is the function $f(a) = a$, the result is the symbolic case analysis

$$g(a) = \begin{cases} 1, & \text{if } a = 0 \\ X, & \text{otherwise} \end{cases}$$

which captures the information of both part c and part d in a single pattern.

Finally, a model for verification must be conservative. That is, while it may allow errors that would cause us to reject a good circuit, it must not allow errors that would cause a bad circuit to be accepted. The first is merely most undesirable, while the latter is quite catastrophic. For example, if the output in part c were X, this would be a conservative error, but if the output in part d were 1, it would not.

We discuss the switch level in some detail in this chapter as an aid to understanding the microprocessor verification case study found in Chapter 9. The switch level is often very appropriate for modeling MOS circuits, since they are designed at that level. For example, the stack cell of Figure 2.9 can be analyzed only at the switch level (or lower).

4.1 Switch-level model

Here we describe the switch-level model. We supply little detail, using informal terms; this discussion is intended as orientation. For a rigorous presentation of the switch-level model see Bryant [39, 49].

A switch-level model is distinguished from a linear circuit models: each transistor is modeled as a switch that can be on or off. Physically a FET may exhibit a range of conductances, but we model this abstractly, as if we were uncertain of the state of the switch.

Switch-level modeling is appropriate for most digital circuits constructed from MOSFETs (metal-oxide-semiconductor field-effect transistors). The term MOSFET is generally retained for historical reasons, although contemporary silicon designs use polycrystaline semiconductor rather than metal gates, so their transistors are more properly called IGFETs (insulated-gate field-effect transistors) [174, 231]. We use the term "FET" or "transistor" to refer to these devices.

A FET is a three-terminal[3] device, and acts like a voltage-controlled switch. In simplified terms, it consists of two terminals, called the source and the drain (which are often symmetric), on either side of a MOS capacitor. Depending on the charge on the MOS capacitor, a conducting path called the channel can sometimes

[3] Actually there is a fourth terminal, called the substrate, but it is unimportant for our purposes.

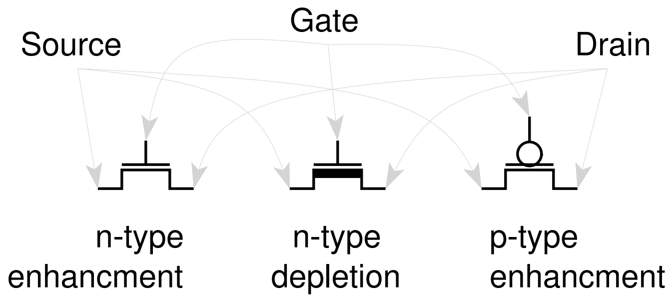


Figure 4.2: Transistor symbols

exist between the source and drain. The distinguished third terminal, called the gate—one of the terminals[4] of the MOS capacitor—is electrically isolated from the other two terminals, and the voltage applied there controls the conductance through the channel.

Figure 4.2 shows the symbols used for three different kinds of transistors. Two particular types of transistors are called nFETs and pFETs. They can be thought of as turning "on" exactly when their gates are at a high or a low voltage, respectively.

Switch-level models distinguish three characteristics of a circuit: its nodes, its FETs, and its sources of electrical charge.

Switching theory has a rich history, but switch-level models for MOS circuits are comparatively new. They were conceived by Bryant [53], while Hayes independently formulated a similar model [131]. In Bryant's original formulation, FETs are modeled as voltage controlled switches. Nodes are divided into three sets: storage nodes, pulled-up nodes, and input nodes. Storage nodes are modeled as sources of arbitrary but bounded amounts of charge. Special pulled-up nodes appear at critical points in ratioed circuits, e.g., the output of an nMOS logic gate. They supply arbitrary unbounded charge, but only if charge is not also being supplied by an input node. Ternary values from the ternary set $\{0, 1, X\}$ represent voltages: 0 represents a low voltage (e.g., 0 volts), 1 a high voltage (e.g, 5 volts or 3.3 volts), and X an unknown or intermediate voltage. Sources of charge are modeled by input nodes, which can supply arbitrary unbounded amounts of charge, at a voltage represented by any of the three logic levels. Stated differently, an input node models an ideal voltage source whose voltage corresponds to the node's present ternary value. This aspect is also present in later refined versions of the switch level model. We will call it the *voltage source assumption*.

[4]The substrate is the other terminal of the MOS capacitor.

This model has several interesting consequences. Since input nodes supply unbounded charge, establishing a conducting path from an input node to any internal node always causes the ternary value of the input node to override any value stored on the internal node.

Since node capacitances are considered be unknown, when two nodes capacitively storing different logic level become connected, both nodes must be set to X. This models the fact that when two capacitors of unknown capacitance, charged to different voltages, are connected together, either might supply an arbitrary amount of charge to the other.

Later refinements to switch-level models introduced two ordered, discrete sets: transistor strengths, and node sizes. Allowing varied transistor strengths allows a conducting path through a strong transistor to override a path through a weak one. This allows pull-up nodes to be eliminated, and replaced by a storage node together with a connection to the power supply through a weak transistor that always conducts. The physical phenomenon modeled is the voltage divider (in the limit as the conductance ratio of voltage divider approaches infinity). We will call this aspect of the refined switch level model the *voltage divider assumption*.

The effect of allowing varied node sizes is to allow the ternary value representing the charge stored on a large node to override the value representing the charge stored on a smaller node. This makes propagation of X values less pessimistic. For example, it allows modeling of structures such as a CMOS logic gate constructed in a precharged, pseudo-nMOS style: the large, precharged output node will sometimes share charge with a tiny, discharged, isolated internal node of the pull-down network, but this will not affect the output node's value. The physical phenomenon modeled is conservation of electrical charge (Kirchoff's current law), in the limit as the capacitance ratio approaches infinity [39]. We will call this aspect the *charge sharing assumption*.

These aspects of the switch-level model are firmly grounded in circuit theory, abstracted by the limit operations.

The switch level model is given a unifying mathematical framework by using a single ordering, where input nodes are given a size greater than transistor strengths, which are in turn greater than storage node sizes. The effect of a network state can then be computed by starting at the highest level and calculating the network response at each level, in turn, before proceeding to the next lower level. The effects calculated at each level do not override effects calculated at a higher level.

This algorithm yields the desired results:

- Since input nodes have the highest strength, and values at lower strengths cannot override values established at higher strengths, input nodes remain at the logic level established by the circuit's environment. This agrees with the voltage source assumption.

- Conducting paths to a storage node through strong transistors from input nodes establish relatively strong values. Paths through weaker transistors and paths from other storage nodes cannot override such values. This agrees with the voltage divider assumption, together with the voltage source assumption.

- Values established by paths from small storage nodes cannot override values established by paths from larger storage nodes. (Note that a path may have a length of zero, i.e., each node comprises such a path.) This agrees with the charge sharing assumption.

Numerous variations of switch level models and simulators have been proposed and implemented; Bryant [44] surveys them. The essential aspects of a switch level model are 1) that it model transistors as switches, and 2) that it operate over at least a three-valued domain. The verification methodology described elsewhere in this thesis further requires a symbolic representation of the model.

4.1.1 Aspects of switch-level models

An important concept in efficient switch-level modeling is that of a transistor group. A transistor group is a set of storage nodes that can share charge, together with the transistors connecting them. More formally: construct the channel graph, a graph with a vertex for each circuit node and an edge for the channel of each transistor. Each connected component of the graph corresponds to a transistor group. Within a transistor group system response can be difficult to analyze, since switches are inherently symmetric and allow charge to flow in both directions. The interaction between different transistor groups is simpler. Each group is affected only by its input nodes and the gates of its transistors; information flows across this boundary in only one direction. Good switch-level simulation algorithms exploit this unidirectional flow to improve efficiency.

Another aspect of circuit modeling is delay. When a transistor switches on, it conducts only a finite current, due to a phenomenon known as saturation. Moreover, the switching of the transistor entails the movement of charge in the MOS capacitor. Thus, a certain time is required before charge is transferred between the source and drain to equalize voltages. There are several ways to model this delay. A simple approach is to consider the transistor as actually switching not when its gate voltage changes, but only after certain delay. Another simplification is to assume that the delays associated with all transistors are the same.

These simplifications lead to the the unit-delay model: the transistor's switching delay is used as the unit of time. Given the logic level at each inputs and storage node, new logic levels are computed for each storage node. As levels are computed, they are stored; at this point the algorithm is computing the effects of all logic levels on transistor conductances. After the new logic level has been computed at each storage node, the switching delay occurs. Then all storage nodes are updated with the new logic levels just computed. The simulation of one unit time is now complete. A simple extension is to say that some transistors switch very quickly, giving them a zero delay.

An additional timing assumption is the phase-level timing assumption. This relates the timing of the circuit's internal operation to the timing of the circuit's environment. It is simplest to assume that after one or more inputs to the circuit change (simultaneously), sufficient time elapses for the circuit to respond—to reach a stable state.[5] This seems a reasonable assumption for synchronous circuits: some inputs may change, and then a clock edge will occur, after which the circuit will have sufficient time to stabilize. Unfortunately it is also unrealistic: some circuits oscillate. To model this, we impose a simple limit. After a certain number of time units (called the *step limit*) as nodes change they can be set to X instead of the computed logic level, with this process repeated until the network becomes stable. (In the worst case all storage nodes would be set to X before stability is reached, but stability is guaranteed to occur.) Although this is not a useful model for circuits such as clock generators that are designed to oscillate, it does model unintended oscillation conservatively. The period of time the circuit is allowed to respond to input changes is called a *phase*.

More sophisticated switch-level timing models are also possible [210, 211]. The verifier used in the case study was not based upon them, though such an extension would be possible.

Two properties of the switch-level model are especially critical for verification: the model is monotone over an "information-content" ordering, and it is conservative. That the model is conservative means that if the model predicts that a circuit will produce an output logic level of 0 or 1, then the circuit will indeed produce this output, rather than some intermediate voltage. A model that is not conservative is of questionable value for verification. That the model is monotone means that if it predicts that, the circuit, with some inputs set to the logic level of X, produces an output logic level of 0 or 1, then if an input that was X is changed, the outputs that are 0 or 1 do not change. Clearly a model that is not monotone is not conservative. We have only argued that the switch-level model is conservative, but it is easy to prove that the switch-level model used in the Cosmos symbolic simulator [55] is monotone.

4.1.2 Symbolic analysis

Several particular details of the switch-level model used in this research are worth noting. In the COSMOS [55] approach to switch-level simulation, the key step is an efficient [51] symbolic Boolean analysis [43] of the switch-level network. The

[5]This is known as the fundamental-mode assumption [132].

problem of determining the response of a switch-level network is formulated in terms of finding paths through the channel graph. The analyzer derives Boolean expressions indicating the conditions under which conducting paths are possibly or definitely formed, and under which one path is blocked by another. Using an encoding representing each logic level by a pair of Boolean values, it forms these expressions into systems of Boolean equations at each strength level. Finally, it solves the equations at each level in turn, using Gaussian elimination. This yields a set of Boolean expressions that capture the network response in full generality. By choosing the proper state encoding, these Boolean expressions are monotone [43].

4.1.3 Simulation

The Boolean expressions produced by symbolic analysis comprise a symbolic representation of the network's response, in one unit time, to its current state and inputs. Given such a representation, the response of a circuit in a particular state to particular a phase is easy to compute: iterate, updating circuit state by evaluating the Boolean expressions, until the state reaches a fixpoint. If the step limit is reached before a fixpoint state occurs, oscillating nodes are set to the logic level X, until a fixpoint is reached. The particular state encoding chosen and the monotonicity of the Boolean expressions guarantee that the phase response of the simulator is monotone.

4.1.4 Symbolic simulation

The result of symbolic analysis is attractive due to the ease of constructing a simulator. This ease may be used to good advantage by constructing a *symbolic* simulator. A conventional simulator, as outlined above, represents the logic levels on input and storage nodes by pairs of bits encoding ternary values. A symbolic simulator extends this by replacing the bits with a representation of arbitrary Boolean functions over some set of variables. Pairs of these functions then encode ternary-valued functions of the set of Boolean variables. In order to test for the fixpoint in computing the phase response of the circuit, it is necessary to test Boolean equivalence. Boolean satisfiability, the canonical NP-complete problem, is trivially reducible to Boolean equivalence. Thus, such a simulator must exhibit poor performance for some class of inputs. In practice symbolic switch-level simulation achieves reasonable performance for many circuits.

4.1.5 Accuracy and precision

It is useful to be careful in distinguishing two terms that are sometimes used carelessly: accuracy and precision. *Accuracy* is defined as "degree of conformity of a measure to a standard or a true value" while *precision* denotes "the degree of refinement with which an operation is performed or a measurement stated." For formal verification, it may be quite useful to trade off precision for some other benefit—faster execution, for example. An imprecise model says little about the behavior of a system. Reducing the precision of a model simply makes the model more conservative. At worst, an imprecise model gives answers that obviously cannot be used because they hold too little information. As a more concrete example, the introduction of X values in oscillating networks, using the step limit, is an example of decreasing the precision of simulation. On the other hand, trading away accuracy is a dangerous game. An inaccurate model is one that yields answers that do not conform to the "true value" but whose incorrectness is not apparent. Bluntly, it yields wrong answers. Too often the term "accuracy" is used as a synonym for "precision."

Models for verification must be accurate, but sometimes they may not need to be precise.

4.2 The Moore machine defined by a circuit

A switch-level circuit can be though of as defining a nondeterministic machine. The machine states are the circuit states that have no X values, and the circuit states with X values represent sets of machine states. For example, the switch-level state X of a circuit having only one node represents the set $\{ 0, 1 \}$ of both possible machine states. Thus, nondeterminism is represented by ternary X values. Intuitively, this often means that some part of the circuit is not being driven, because it is uninteresting—it shouldn't affect another part (one that we are interested in).

This differs in character from nondeterminism that reflects an actual choice—such as decision of the order in which two signals have arrived.

Our ultimate goal is to verify circuits, but our theory is primarily in terms of abstract machines rather than circuits. Here we now connect our abstraction to a lower-level abstraction, commonly used in reasoning about MOS circuits: the switch-level simulation model we have just examined.

Let $\mathcal{T}$ denote the ternary set, namely $\{ 0, 1, X \}$, with a partial order relation $\sqsubseteq_t$ given by $0 \sqsubseteq_t X$ and $1 \sqsubseteq_t X$. Intuitively, $\sqsubseteq_t$ is an uncertainty ordering. Let $\mathcal{B}$ denote the binary subset of $\mathcal{T}$, namely $\{ 0, 1 \}$.

A switch-level circuit R is a pair (N_R, Y_R). It consists of a finite set N_R of distinct elements, called *nodes*, and an *excitation function* $Y_R \colon \mathcal{T}^{|N_R|} \to \mathcal{T}^{|N_R|}$. The nodes are numbered, and we identify them with subscripts. Thus $N = \{ n_1, n_2, \ldots, n_{|N_R|} \}$. We require that the excitation function Y_R be monotone with respect to the point-wise extension of $\sqsubseteq_t$. This is a common requirement of a simulation model; in particular, it is satisfied by the switch-level simulator COSMOS [55] as a consequence of the choice of primitive operations and state encoding.

The *input nodes* are those nodes whose value is not determined by the circuit. Thus the set of input nodes can be given as $\mathrm{IN}_R = \{ n_i \in N_R \mid \forall t \in \mathcal{T}^{|N_R|} \;\; (Y_R(t))_i = X \}$. The remaining nodes form the set of *state nodes* SN_R. The *output nodes* ON_R are a subset of the state nodes. Let $\mathrm{IN}_R = \{ n_1, \ldots, n_{|\mathrm{IN}_R|} \}$ and $\mathrm{SN}_R = \{ n_{|\mathrm{IN}_R|+1}, \ldots, n_{|N_R|} \}$ and $\mathrm{ON}_R = \{ n_{|N_R|-|\mathrm{ON}_R|}, \ldots, n_{|N_R|} \}$. This entails no loss of generality: one could always permute nodes.

Then the signature of the realization machine is defined by letting $\mathrm{inp}(R) = \mathcal{B}^{|\mathrm{IN}_R|}$ and $\mathrm{out}(R) = \mathcal{B}^{|\mathrm{ON}_R|}$ $\mathrm{states}(R) = \mathcal{B}^{|\mathrm{SN}_R|}$. Then let $\mathrm{config}(R) = \mathrm{inp}(R) \times \mathrm{states}(R) \times \mathrm{out}(R)$, so $\mathrm{config}(R)$ is a proper subset of $\mathcal{T}^{|N_R|+|\mathrm{ON}_R|}$. Thus, $\sqsubseteq_t$ defined on $\mathcal{T}^{|N_R|+|\mathrm{ON}_R|}$ can also be defined between an element of $\mathcal{T}^{|N_R|+|\mathrm{ON}_R|}$ and an element of $\mathrm{config}(R)$. Note that as we required in section 3.2.2 (p. 62), there is a projection Π such that $\Pi\,\mathrm{config}(R) = \mathrm{inp}(R)$. We can define $\sqsubseteq_t$ between $\mathcal{T}^{|N_R|}$ and $\mathrm{config}(R)$ by identifying redundant elements of $\mathcal{T}^{|N_R|+|\mathrm{ON}_R|}$, i.e., those corresponding to $\mathrm{out}(R)$, with the associated elements corresponding to elements of $\mathrm{states}(R)$.

Using this ordering, we can complete the definition of the realization machine. Its transition relation $\mathrm{steps}(R)$ is defined to be the set:

$$\Big\{((i_a, s_a, o_a), (i_c, s_c, o_c)) \mid \exists s \in \mathcal{B}^{|N_R|} \centerdot (i_a, s_a, o_a) = s \wedge (i_c, s_c, o_c) \sqsubseteq_t Y_R(s)\Big\}$$

We usually let R denote the machine rather than the switch-level circuit.

We define the outputs nodes to be a subset of the state nodes to ensure that the excitation function can depend only on input nodes and state nodes (i.e., that it cannot depend on outputs which are not states).

Proposition 8 *The machine R defined by a switch-level circuit as in the construction above is a Moore machine.*

Proof: Assume that $(v_a, v_c) \in \mathrm{steps}(R)$ and $\Pi_r v_c = \Pi_r \hat{v}_c$ and show that $(v_a, \hat{v}_c) \in \mathrm{steps}(R)$. Since $(v_a, v_c) \in \mathrm{steps}(R)$, let $(i_a, s_a, o_a) = v_a$ and $(i_c, s_c, o_c) = v_c$. Then there is an $s \in \mathcal{T}^{|N_R|}$ such that $(i_a, s_a, o_a) \sqsubseteq_t s$ and $(i_c, c_c, o_c) \sqsubseteq_t Y_R(s)$.

Let $\hat{v}_c = (\hat{\imath}_c, \hat{a}_c, \hat{o}_c)$. Since $\Pi_r v_c = \Pi_r \hat{v}_c$, $s_c = \hat{s}_c$ and $o_c = \hat{o}_c$. Finally, since Y_R yields X values on input nodes, whatever $\hat{\imath}_c$ may be, $(\hat{\imath}_c, s_c, o_c) \sqsubseteq_t Y_R(s)$. Hence $(v_a, \hat{v}_c) \in \mathrm{steps}(R)$. ∎

4.3 Related work

Many models used in attempting formal reasoning at the switch level (e.g., [257]) have shortcomings, and are unable to model relatively common MOS structures. A notable exception is the model used by Weise's system Silica Pithicus [241], which is quite detailed. It is successful where others fail because it does not simply describe the behavior of a circuit: it also produces a set of constraints upon the environment in which the circuit operates. These constraints become proof obligations in hierarchical reasoning.

The notion of an excitation function used here is quite close to that used by Seger and Brzozowski [59].

Kam and Subrahmanyam [151] have developed a technique for showing that small switch-level circuits implement abstract machines. They use anamos [55] to produce a set of network excitation functions, and derive from them a transition relation which they represent symbolically in the style of Coudert [84]. This relation is transformed by means of a fixpoint calculation, and an existential quantification to hide the clock signals, to get a cycle-to-cycle transition relation. The fixpoint calculation converges if the circuit does not oscillate.

This strategy of computing the fixed point first and then finding the ultimate transition relation seems a bit cumbersome. The same results should be obtainable by simply evaluating a symbolic simulation over one clock cycle, yielding directly the next-state function, and converting that to a relation. This incorporates the clocking information earlier in the analysis, hopefully reducing the difficulty of analysis. This becomes, essentially, the method of Bose and Fisher [27]. Performance should be better as well, because the simplifying circuit constraint of non-overlap of clocks would be incorporated into the symbolic representation much earlier.

Moreover, with symbolic simulation there is a chance of extracting useful information from oscillating circuits. First, if a circuit oscillates only when the clocking constraints are disobeyed, incorporating the clocking earlier obviously helps. Second, if the circuit oscillates otherwise, the fixpoint calculation that occurs during simulation will not converge. But if it exceeds a threshold number of steps, switching to a monotonic conservative approximation of the excitation functions will guarantee convergence. This allows checking properties of (a conservative approximation to) an oscillating circuit. Such circuits might sometimes be acceptable circuit, e.g., if the oscillation were in an unreachable component of its transition graph.

This conjecture—that Kam's technique is essentially equivalent to that sketched above—could be evaluated both experimentally and also by attempting a proof. Proving equivalence would be immediately useful, while proving a discrepancy would imply that either symbolic simulation or the state-machine extraction was wrong in some subtle way.

4.4 Chapter summary

This chapter has discussed properties of a model of digital MOS transistor circuits suitable for verification. The particular model discussed has been the switch level, which was chosen for historical reasons. An introductory explanation of the switch-level model, and its development, was given. In order to apply the verification methodology and checking algorithm that we will be discussing in this thesis, a circuit model must have certain characteristics. It must operate over a partially ordered domain, be monotonic over this order, and symbolic. It must also be conservative—precise, but not necessarily accurate—in other words, it must answer "I don't known" rather than give a wrong answer. We discussed these characteristics and the way in which the switch-level model meets them, and concluded by formally defining the Moore machine described by a simulation model.

Chapter 5

Implementation

This chapter introduces the idea of mappings between agents in order to discuss a formal relationship between abstract agents, called **implementation**.

5.1 Mappings between agents

If M and N are agents, a mapping I from M to N, written $I\colon M \to N$, consists of several parts, which map the various parts of the signature of M into corresponding parts of the signature[1] of N. Thus, it consists of an input mapping $I_{\mathrm{ins}}\colon \mathrm{ins}(M) \to \mathrm{ins}(N)$, and a behavior mapping $I_{\mathrm{beh}}\colon \mathrm{beh}(M) \to \mathrm{beh}(N)$. Of course I_{beh} and I_{ins} must be consistent with the input projection function Π. We will also sometimes wish to consider a specification agent operating on a subset $\mathcal{L}_S \subseteq \mathrm{inp}(S)$ of its possible inputs. Often we will be particularly concerned with mappings where I_{ins} is surjective. (Note that if I_{ins} is surjective from some $\mathcal{L}_S \subseteq \mathrm{ins}(S)$ then it is surjective from $\mathrm{ins}(S)$.) We omit the subscripts ins and beh from I when they are clear from context.

5.2 Implementation between agents

The central property we are concerned with is implementation. First we discuss it informally, before giving the formal definition.

5.2.1 Informal motivation

To say that one machine implements another is to say that the first does what the second does. If the two machines are interchangeable, this is not difficult to define. However, we are concerned with machines at different levels of abstraction.

[1] the signature of an agent was defined on page 62

The second machine—the specification machine—should be more abstract, so that we can reason about it in a concise way. But the first machine—the realization machine—should be more concrete, so that we can build it. Thus the machines, in general, will not be interchangeable.

The precise degree of difference in abstraction and, in fact, even its direction is not important. The realization could indeed be more abstract than the specification[2]. The important thing is that the neither machine can replace the other. Some sort of a nondeterministic mapping is needed to move from one to the other.

Numerous definitions of implementation relations have been proposed. The one we choose has a simple intuitive definition. We take two nondeterministic agents, called the specification and the realization. We say that the realization implements the specification if, for *any* stimulus on which the specification produces a response, the response of the realization to this stimulus is always one of the responses that we *could* see from the specification.

Such a definition lacks the mapping from specification to realization. This is easy to remedy. Note that the mapping could be nondeterministic. For any response of the specification, there is a set of corresponding responses of the realization. Thus we require that the realization produce *any* response that corresponds to *some* response of the specification. Also note that for any stimulus to the specification, there is a set of corresponding stimuli to the realization. Thus we require that the realization work for any such stimulus that the mapping might pick.

Such a definition must be carefully considered, for it allows trivial implementations in two ways. First, it allows a mapping that never maps any stimulus. Then any requirement that some property hold for all mapped inputs will be trivially true. Second, it allows a mapping that maps all responses together. Then the requirement, that every response of the realization be one that is allowed by the specification, is trivially true. We must address these possible trivialities in our notion of implementation.

Implementation should ultimately be defined as an I/O relationship. Ultimately, it is through the behavior of a system that we must decide whether or not it is acceptable. Recall the example of I/O behavior for an abstract stack, Figure 2.17, (p. 38) and for a stack circuit, Figure 2.18. Our definition of implementation should be a relation between sets of objects like these.

5.2.2 Direction of the mapping

The above characterization of implementation in the presence of a mapping is certainly not the only possible one. A more general framework would be in terms of a relation rather than a directed mapping. (The set-valued mappings that we

[2]the only new question then raised would be of sanity

chose to use are mathematically equivalent to relations.) The mapping could also be given in the opposite direction. Since such mappings from the concrete to the abstract yield abstractions, they are often called abstraction functions. (In an unfortunately confusing choice of terminology, they are also sometimes called implementation functions.) There is no clear mathematical reason to choose one direction over another.

In the absence of a fundamental or mathematical reason for preferring one treatment, we are free to adopt a pragmatic reason. We choose to map from the abstract system to the concrete system because this is the direction of mapping that we find easiest to work with.

Mappings from the abstract system to the concrete system are easy to work with for three main reasons. The first is that they are like macros. In the genesis of this work, mappings were originally conceived of as macros. But waving hands and saying "they're just macros" is unsatisfying. If such a formulation is put in practice, a textual specification may look clean and straightforward—but the actual specification is not the text! Instead, the actual specification is the *macro-expansion* of the text, not the text itself. Reasoning, even informally,[3] about the text as if it were the specification is then a dangerous undertaking. Although our mappings have some of the role of macros, they are treated formally, so that the "un-expanded" specification is really the specification to which the formal relationship is established. Thus, reasoning about the abstract specification may be conducted with confidence.

The second reason for mapping "downward" is that mapping from the abstract to the concrete maps from simpler things to more complicated ones. This lets us try to impose order rather than try to discern it from chaos. It becomes particularly important because our models are switch-level circuits, where any circuit node can be modeled as a state variable. Most of the nodes in a circuit store unimportant values. That is, the fact that they store values at all is a side-effect of the way in which they are designed, rather than a fundamental result on which the circuit operation depends. But by allowing them to store values, we admit a uniform treatment of circuit nodes in the system model. This makes a mapping in the opposite direction prone to two problems. First, it requires consideration of many unimportant state variables, complicating the analysis. Second, if we undertake to reduce the number of state variables by considering only the important ones, this requires that we first identify them through a detailed analysis at the circuit level. Mapping in the other direction avoids this difficulty.

Since, by mapping downward, there are a limited number of Boolean variables, those in the specification, and presumably their role is well-understood, this allows

[3] Even if the macros themselves have precise definitions, it will still be tempting to reason about the un-expanded specification, since it is easy to believe that you "know what the macros mean."

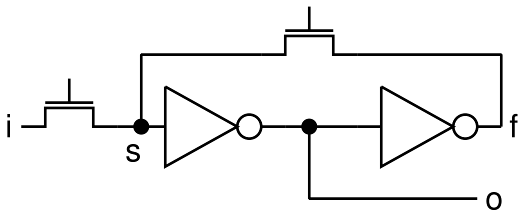


Figure 5.1: Dynamic latch. Meaningful state is stored on node s, but nodes o and f could potentially also be storage nodes.

the users of a verifier based on these techniques to play a large role in choosing an order for the variables used in constructing BDDs. The informal observation that manually chosen variable orders often work well is a part of the folklore surrounding BDDs, and Fujita and his colleagues have reported on experiments confirming this [107].

Finally, such mappings are sufficiently general that they can, at least conceptually, express any relation that could be expressed some other way.

Nonetheless, the direction of our mapping is still in several ways a convention rather than an absolute requirement, and we expect that results similar to some of ours could be obtained if the direction of the mapping were reversed in the formalism.

For example, an algorithm has been suggested by Clarke [76] for checking Hoare-logic assertions on single state transitions using quantified Boolean formulas (QBF). In one sense it is less general than the methodology developed here because it deals only with transitions, not with overlapped operation, but it is more general than trajectory evaluation in the sense that its underlying logic is more expressive. Failure to account for overlap would prohibit this algorithm from being used on a design such as the Hector microprocessor, studied in Chapter 9, but there are other drawbacks as well.

The QBF-based technique requires that the transition function for the system be evaluated for a general state. This can be prohibitively expensive, because of two reasons. First, there are many bits that, in a low-level system model, while they can store state, do not store state. For example, consider the dynamic latch shown in Figure 5.1. Excluding the input i, it contains 3 nodes which can potentially store state: the actual storage node s, the inverted output o, and the output of the feedback amplifier, f. This threefold increase in the number of state bits—cubing the number of states—is but the first step. Note that this triples the number of BDD variables required to represent a general state. Moreover, simulating from a fully symbolic representation of a state requires the simultaneous evaluation of all aspects of circuit operation at once—for example, all the instructions in a processor's instruction set, at once. This may not be possible even when the evaluation of one aspect at a time is tractable.

While a QBF-based method of checking assertions may be useful for verifying some class of systems, it not suitable for microprocessors when reasoning at a detailed level.

5.2.3 Formal definition of implementation

We define implementation formally in terms of three component properties. The first, obedience, requires that the realization behave well. The second, conformity, requires that the input mapping be nontrivial. The last, distinction, requires that the result mapping be nontrivial.[4]

Formally, let S and R be agents, and $I: S \to R$ be a total mapping. Let $\mathcal{L}_S \subseteq \mathrm{ins}(S)$.

Definition 17 (Obedience) *Agent R obeys S on $\mathcal{L}_S$ under I if for every $i_S \in \mathcal{L}_S$, for every $i_R \in I(i_S)$, and every $v_R \in R(i_R)$, there exists a $v_S \in S(i_S)$ such that $v_R \in I(v_S)$.*

In other words, if we encode a specification input in any possible way, everything that the circuit can produce from that input must be something that could be obtained by encoding a result that the specification could have produced from the original input. In such conditions we refer to I as an **obedience function**. When $\mathcal{L}_S$ is equal to $\mathrm{ins}(S)$ we omit mention of $\mathcal{L}_S$, for the definition then holds for any $\mathcal{L}_S \subseteq \mathrm{ins}(S)$.

Definition 18 (Conformity) *The set $\mathcal{L}_S$ conforms under mapping I_{ins} if for every $i_S \in \mathcal{L}_S$ there is at least one $i_R \in I(i_S)$.*

In other words, we must be able to encode every specification input to produce a circuit input.

Definition 19 (Distinction) *The mapping I is distinct for $\mathrm{beh}(S)$ if for every distinct v_S and $\hat{v}_S$ (i.e., $v_S \neq \hat{v}_S$), in $\mathrm{beh}(S)$ the sets $I(v_S)$ and $I(\hat{v}_S)$ are disjoint.*

In other words, the encoding must always allow us to tell different things apart. Note that this does not require the specification to be a reduced machine.

Definition 20 (Implementation) *Agent R implements S on $\mathcal{L}_S$ under I if R obeys S on $\mathcal{L}_S$ under I and $\mathcal{L}_S$ is conforms under mapping I_{ins} and I_{beh} is distinct for $\mathrm{beh}(S)$.*

---

[4] As a mnemonic aid, imagine a fictitious military school on Cape Cod, whose motto might be "Conformity, Obedience, Distinction."

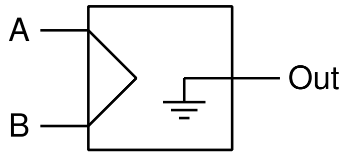


Figure 5.2: Bad implementation of exclusive-OR

When this property holds, we refer to I as an **implementation mapping.**[5]

Obedience

This is the property that we often refer to when we say informally that one machine implements another. In this sense it is the key property. Most of the effort goes to show it.

Conformity

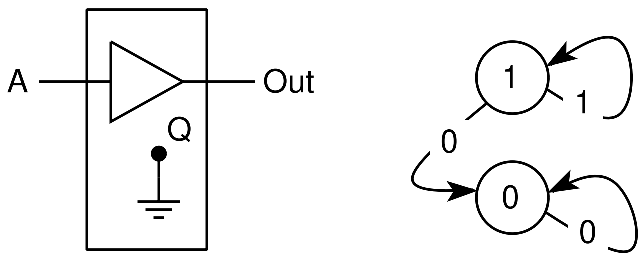
This is the requirement that the input value i_R in Definition 17 always exists. At first glance the presence or absence of such a property of a mapping may almost seem to be so obvious that there is no need to check it. However, our desired application of this theory is to circuit verification. There are subtleties involved in mapping states onto circuits.

Suppose we wish to specify an exclusive-OR gate. We might write this as the assertion $A = a \wedge B = b \overset{\delta}{\Rightarrow} \text{Out} = a \oplus b$.

Suppose we did not require conformity—that we be able to map all inputs. Then the circuit shown in Figure 5.2 would meet our exclusive-OR specification.

This circuit is obviously not an exclusive-OR gate. Since its output is grounded, obviously it computes nothing at all. Since A and B are actually the same node, we can map the specification states described by $A = 0,\ B = 0$ or $A = 1,\ B = 1$ onto the circuit, and for each of these, the desired output is 0—which the circuit produces, since the output is grounded. However, we cannot map the specification states described by $A = 1,\ B = 0$ or $A = 0,\ B = 1$ onto the circuit, since the input node (which goes by the two names A and B) cannot be both high and low. Consequently, in these cases, the antecedent of the implication inherent in the assertion will be false, so the entire implementation will be true. Thus, for all cases, the assertion is valid.

[5]Unfortunately, almost every possible term (except perhaps "downward mapping") has been used by someone to refer to a mapping in the opposite (upward) direction, i.e., from the realization to the specification. For example, Lamport has used both "implementation mapping" [157, p. 103] and "refinement mapping" [1] to refer to mappings in the opposite direction from ours.



a. Circuit b. Specification

Figure 5.3: Bad implementation of a serial AND gate. a. The node Q, which is supposed to store state, is grounded. b. State diagram of specification.

Clearly something is amiss here. The problem is that we cannot map all specification inputs onto the circuit. If we impose the conformity requirement, however, we will notice that not all legal abstract inputs can be mapped onto the circuit.

This is one aspect of conformity, which we can examine without considering sequential behavior. However, when we begin to consider temporal behavior, we find that conformity involves even more subtlety.

Before we examine the temporal aspect of conformity, however, we should look briefly to see if there is an analogous property for the state rather than the input. Conformity is a kind of a totality requirement, and we also have required that our state mapping be total. This is an important requirement as well. Just as for the instantaneous aspect of conformity, we can illustrate the need for totality by using a simple circuit.

Suppose that we have a "serial AND" circuit and we specify the behavior we desire with the two assertions $\text{In} = 1 \wedge Q = 1 \overset{\delta}{\Rightarrow} \text{Out} = 1 \wedge Q = 1$ and $\text{In} = 0 \overset{\delta}{\Rightarrow} \text{Out} = 0 \wedge Q = 0$. Suppose that we did not require the state mapping to be total. If so, then we could verify the circuit shown in Figure 5.3a against this specification.

For the first assertion, we find that we cannot map the antecedent onto the circuit, since the grounded node Q cannot be set to 1. Thus this assertion succeeds. The second assertion also succeeds. Yet clearly the circuit does not behave as intended by the specification. The specification describes a system with two states, $Q = 1$ and $Q = 0$, as indicated in Figure 5.3b, yet the circuit has only one state, $Q = 0$.

This situation arose because we did not require that the state mapping be total, so the antecedent of the assertion, once it had been mapped onto the circuit, failed in cases where we had not expected it to fail. Often, if a state mapping were not total, we would be able to detect a problem because a consequent would fail for some assertion. However, we can construct pathological specifications such as this one in which the failing state $Q = 1$ has no incident edges in the transition graph (other than self-edges), so it does not occur in a consequent (other than one in which it also occurred in the antecedent). Since we cannot rely on failure in the consequent to detect such unmappable states, we require that the mapping be total.

This has the effect of outlawing circuit antecedent failure in cases when the antecedent succeeds at the abstract specification level.

In addition to their single-state or instantaneous aspect, failures can have a temporal aspect. This is an even more subtle aspect of conformity. It can also be illustrated by simple examples. However, rather than introduce yet another special-purpose circuit example, we can refer to the stack example of Chapter 2. In Figure 2.13 (p. 35) we have already seen an example where conformity failed for a sequence of stack operations, when we attempted to have a "pop" operation follow a "hold" operation. When mappings take on a temporal aspect, we must ensure that conformity holds over sequences. In the case of the original definition of the stack operations, conformity did not hold.

This is an example of a general phenomenon. To achieve good performance, most circuits overlap successive operations. It is quite conceivable that a mapping I might "work" for short sequences but fail to work on long ones, due to conflicting requirements on an overlapping interval. Therefore we include an explicit check that the overlapping portions of inputs must conform to one another.

Distinction

This property rejects trivial implementations. Without it we would be able to say that a trivial machine, which has only one output, implements every machine, because its behavior—the only one it can possibly produce—is always in the image of any other machine's behavior, since there is only one possible mapping—the mapping onto this behavior.

5.2.4 Example

Since our notions of agents and of implementation are rather general, they can be illustrated in a more general context than is our ultimate application. This emphasizes that the definition of implementation is a natural one. As an illustration of the notion of implementation, consider the analog buffer shown in Figure 5.4.

Now suppose that we wish to implement a buffer, but that its input is differential. That is, the input value is encoded as the *difference* in voltage between two wires. We can implement a differential-input buffer using a differential amplifier, or "diff amp." Figure 5.5 illustrates a diff amp constructed from an operational

Figure 5.4: Specification of an analog buffer, and its DC transfer functions.

amplifier.[6]

The DC I/O behavior of the analog buffer of Figure 5.4 can be expressed by the transfer function graphed in the same figure. This serves as a specification of the intended behavior of a buffer.

The graph in Figure 5.5 shows the I/O behavior of the differential amplifier, i.e., the realization. An input of the specification is a vertical line in the space of Figure 5.4. If we take any such vertical line, it corresponds to some vertical plane in the space shown in the second figure. For example, the vertical plane shown in the back corner of Figure 5.5 corresponds to the vertical line shown in the first figure. This vertical plane represents a set of realization inputs: each of the vertical lines comprising this plane represents a different realization input, each of which corresponds to the original specification input.

If we take any output of the specification, that is any horizontal line on the ordinate of the first figure, it corresponds to a horizontal plane at a corresponding location in the space of the second figure.

The line in Figure 5.4, and the shaded plane in Figure 5.5, represent the respective behaviors of the specification and the realization.

Consider an example of our notion of implementation in the context of these figures. The input mapping takes voltages to voltage differences. The output mapping is the identity. Given these mappings, implementation first requires that for any input to the specification (e.g., the vertical line shown in Figure 5.4), any corresponding (i.e., mapped) input to the realization (e.g., the vertical plane shown in Figure 5.5), every possible realization output (e.g., the horizontal plane, containing the white line, in Figure 5.5) must correspond with (i.e., be contained in the image under the mapping of) a possible output of the specification for the original input (e.g., the horizontal line in Figure 5.4. And indeed we see that plane of the white line in Figure 5.5 lies at the 5V level, the same as the horizontal line in Figure 5.4.

This has illustrated the notion of implementation when there is some nondeterminism in the input mapping. Any pair of realization input voltages having the proper differential formed an acceptable encoding of a specification input. Suppose there were some nondeterminism in the specification as well. A specification might allow its output the tolerance of a slight deviation from the nominal, correct value. In the illustrated example, it might require that the output fall in the region shown by the dashed lines in the specification figure. In this case, the realization implements the specification provided that it yields less deviation than the specification allows—exactly as we would expect—as indicated by the region defined by the dashed lines in the realization figure. The realization implements the specification

[6] First-order analysis of an op amp follows from the two principles that the voltages on the two input terminals will be equal, and the currents into the inputs will be zero, plus Ohm's and Kirchoff's laws.

Figure 5.5: Realization of an analog buffer, using a differential amplifier, and its DC transfer functions.

provided that its output falls within the allowed tolerance.

5.3 Related work

5.3.1 Roles of abstraction

The role of abstraction in hardware verification has received disparate treatments. For example, someone unfamiliar with formal verification might wonder whether the thesis work of Melham [176] and Long [163] share common ground. Some of this confusion arises because while the term "abstraction" is very general, it has seen two different specific uses in formal hardware verification. To understand the distinction, it is necessary to distinguish between *design verification* and *implementation verification* [66].

Unfortunately, the distinction between design verification and implementation verification is itself not always clear. Implementation verification is the process of establishing that "what has been implemented" agrees with "what has been specified." Design verification is the process of establishing that "what has been specified" agrees with "what is desired." Design verification is sometimes called "property checking" [202] which is a more descriptive term.

The kind of abstraction, then, depends on the kind of verification. Implementation verification is the process of establishing a relationship between two agents of computation, while property checking is the process of establishing properties of an agent. For implementation verification the role of abstraction is to relate two agents. Abstraction is essential to implementation verification, for there are *a priori* two agents to be related.

In contrast, for property checking, the role of abstraction is to simplify both an agent and the statement of some property, so as that if the simplified agent has the simplified property, the original agent will have the original property. Abstraction can be incidental to property checking—it would suffice to check the original property of the original agent. The focus in abstraction and property checking must be different: the abstract (simplified) agent may not be given, so part of the task is to discover or construct it. Since model checking is property checking,[7] work on abstraction in this context [154, 163], focuses on different issues.

[7] "Model checking" would be an even more descriptive term than "property checking" because it is actually the *model* (i.e., agent) that is being checked, not the *property*, which is taken to be the one desired, but this would be confusing, because the word "model" in "model checking" is normally taken to mean the technical sense used by logicians.

5.3.2 Abstraction functions

Abstraction techniques for implementation verification are often based on abstraction functions which map from the concrete to the more abstract—opposite to the direction of the mappings that we use. Abstraction functions were introduced for program verification by Hoare, for verifying the implementation of abstract data types [133].

Mappings from the abstract to the more concrete have been considered in program verification. Flon and Misra [105] discuss cases where concrete specifications are derivable from abstract axioms.

Melham's PhD [176] deals with abstraction in implementation verification, using the HOL proof system, particularly by describing ways of constructing abstraction functions.

Burch [61] dealt with abstraction for design verification in trace theory by defining pairs of abstraction functions to be applied to the specification and the realization, respectively. If the specification abstraction function underestimates behaviors, and the realization abstraction function overestimates behaviors, a conservative, approximate check for language containment is possible even when the two abstraction functions differ.

5.3.3 Input-output relationships

The most appealing equivalence between processes seems to be observation equivalence. In the process-algebra context, this is often established by establishing a bisimulation relation. Though the notion of bisimulation is often attributed to Milner [181], it is attributed by Milner to Park [196], and has roots in A. Ginzburg and M. Yoeli's notion of weak homomorphism [113]. Cory's thesis [83] describes similar criteria for evaluating the consistency of two representations of sequential machines. These criteria consider only localized state transitions, and Cory gives no clear intuitive notion of what it means for a circuit to be correct. His approach to relating different levels of abstraction when timing details differ is to add "translators" which act as buffers to map between levels of abstraction.[8] Unfortunately, Cory's thesis does not significantly relate symbolic execution to these consistency criteria.

The correctness criteria in string-functional semantics [37, 35, 36, 236, 237] are also input-output relationships.

[8] This is similar to Bose and Fisher's definition of an abstraction function by means of a circuit [27].

5.4 Chapter summary

This chapter has defined a notion of implementation between two agents operating at different levels of abstraction. Since the levels of abstraction differ, this is not a simple equivalence or implication. Instead, a relation between the two levels is necessary. We chose to express this relation as a nondeterministic "downward" mapping. Such mappings are sufficiently general to express any mathematical relation, and we had particular reasons for preferring this direction. (Some of these reasons will become more concrete in subsequent chapters.) Since the notion of implementation is rather general, we were able to illustrate it using a pair of analog circuits and their DC transfer functions.

Chapter 6

Proving implementation

This chapter shows one way of proving that one machine implements another. The first machine, or realization, is expressed as a switch-level simulation model. The second machine, or specification, is expressed as a set of symbolic assertions. The proof procedure consists of mapping each assertion into a symbolic simulation pattern, and checking the circuit against it.

This chapter develops a theory which shows that such a check establishes implementation.

6.1 Relation between agents

6.1.1 Mappings

Since specifications and circuits are given at different levels of abstraction, we have mappings between them. A promising start to a theory of implementations that encompasses mappings is to look at mappings themselves.

Input-preserving mappings

One class of mappings is of particular interest: mappings that preserve inputs. These are mappings between agents M and N where $\mathrm{ins}(M) = \mathrm{ins}(N)$ and I_{ins} is simply the identity mapping.

We can also allow nondeterministic input-preserving mappings.

A class of input-preserving mappings that are particularly important are the machine homomorphisms.

Definition 21 (Machine Homomorphism) *A machine homomorphism H is an input-preserving mapping from agent M to agent N which is homomorphic with respect to M and N.*

That is, if H is a machine homomorphism, $H(M(i)) = N(H(i))$. And since H preserves inputs, $H(M(i)) = N(i)$.

Machine homomorphism implies obedience

This gives us an example of the obedience property of Definition 17 (p. 113).

Proposition 9 *If H is a machine homomorphism from M to N, then N obeys M with respect to H.*

Proof: From the definition of a machine homomorphism, $i = H(i)$, and $N(i) = H(M(i))$. Thus $N(i) \subseteq H(M(i))$ so H fits the conditions on the mapping I in the definition of obedience. ■

6.1.2 Accepting agents

We have described an agent as a function that takes an input to an output. Another view is to consider an agent as examining possible input-output pairs and approving those it likes.

Definition 22 (Accepted Set) *If M is an agent, the set of behaviors accepted by M, denoted* $\mathrm{accept}(M)$, *is defined to be* $\{ v \in \mathrm{beh}(M) \mid v \in M(\Pi v) \}$.

Containment

From the elements of implementation given in Chapter 5, together with the notion of acceptance sets, we can reach an alternate formulation of implementation as containment. A realization obeys a specification if the set that it accepts is a subset of the set that the specification accepts. (In the presence of a mapping I this might be stated formally as $\mathrm{accept}(R) \subseteq I(\mathrm{accept}(S))$. Similarly, an analog of our conformity property, that for every i_S an i_R must exist, might also be stated as $\Pi\,\mathrm{accept}(R) \supseteq I(\Pi\,\mathrm{accept}(S))$, where Π is the projection onto inputs, as we mentioned when defining agents, in Section 3.2.1, p. 62.)

Equivalent to obedience

Containment of accepted sets is an abstract notion of implementation. Under a simple condition this more abstract idea is equivalent to our previous notion of obedience.[1]

[1] Note that if sets are replaced by characteristic predicates, this containment statement is similar to conditions such as $\chi_R \Rightarrow \chi_S$ as used in theorem-proving based verification efforts.

Theorem 10 *If* $I_{\text{ins}}\colon \mathcal{L}_S \to \text{ins}(R)$ *is surjective, then* R *obeys* S *with respect to* I *if and only if* $\text{accept}(R) \subseteq I(\text{accept}(S))$.

Proof: We will assume that I_{ins} is surjective. We must show that $\text{accept}(R) \subseteq I(\text{accept}(S))$ if and only if, for every $i_S \in \mathcal{L}_S$, every $i_R \in I_{\text{ins}}(i_S)$, and every $v_R \in R(i_R)$, there is a $v_S \in S(i_S)$ such that $v_R \in I(v_S)$.

$\Leftarrow$ Assume that for every $i_S \in \mathcal{L}_S$, every $i_R \in I_{\text{ins}}(i_S)$, and every $v_R \in R(i_R)$ that there is a $v_S \in S(i_S)$ such that $v_R \in I(v_S)$. From this we must show that $\{\, v_R \mid v_R \in R(\Pi v_R)\,\} \subseteq I(\{\, v_S \mid v_S \in S(\Pi v_S)\,\})$. That is, we must show that every v_R such that $v_R \in R(\Pi v_R)$ is in $I(\{\, v_S \mid v_S \in S(\Pi v_S)\,\})$; in other words, that there is a $v_S \in S(\Pi v_S)$ such that $v_R \in I(v_S)$.

Consider any $v_R \in R(\Pi v_R)$. Let $i_R = \Pi v_R$. Since I_{ins} is surjective, there is some i_S such that $i_R \in I(i_S)$. Thus from the hypothesis we conclude directly that there is $v_S \in S(i_S)$ so that $v_R \in I(v_S)$.

$\Rightarrow$ Assume that $\{\, v_R \mid v_R \in R(\Pi v_R)\,\} \subseteq I(\{\, v_S \mid v_S \in S(\Pi v_S)\,\})$ and from this show that for every $i_S \in \mathcal{L}_S$, every $i_R \in I_{\text{ins}}(i_S)$, and every $v_R \in R(i_R)$ that there is a $v_S \in S(i_S)$ such that $v_R \in I(v_S)$. That is, we know first that each v_R such that $v_R \in R(v_R)$ is in $I(\{\, v_S \mid v_S \in S(\Pi v_S)\,\})$; i.e., for each v_R such that $v_R \in R(\Pi v_R)$ there is a $v_S \in S(\Pi v_S)$ such that $v_R \in I(v_S)$. Second, when we consider any $i_S \in \text{ins}(S)$ we know from the definition of agent S that if $v_S \in S(i_S)$ then $i_S \in \Pi v_S$. Consider any $i_R \in I(i_S)$ and any $v_R \in R(i_R)$. Again we know $i_R = \Pi v_R$. Thus we conclude from the first and second facts above that there is a $v_S \in S(i_S)$ such that $v_R \in I(v_S)$.

■

Note that we needed surjectivity for only the first part of the proof[2]. Moreover, surjectivity is no severe requirement. Rather, it is merely a technical one. If it does not hold, one could always add a "bottom" input $\perp$ to $\text{ins}(S)$. Agent S could be allowed to produce, on this input, any consistent behavior, and I would map the new element $\perp$ to the entire set $\text{ins}(R)$. In fact, this could be done implicitly.[3] Formally we can express this by defining a machine $\hat{S}$ and mapping $\hat{I}$ to be the same as S and I, except that

$$\begin{aligned} \text{ins}(\hat{S}) &= \text{ins}(S) \cup \{\, \perp \,\} \\ \hat{I} &= I \setminus \perp \mapsto \text{ins}(R) \\ \hat{S}(\perp) &= S(\text{ins}(S)) \end{aligned}$$

Corollary 11 *Obedience is transitive.*

[2] which is, however, the portion of this theorem that is most useful

[3] An alternative would be to limit the inputs of R. We will find later a need for another surjectivity property in a different context, so for consistency we prefer surjectivity here as well.

Proof: We must show that if R obeys S with respect to I and S obeys T with respect to J then R obeys T with respect to $I \circ J$. This is immediate from the transitivity of the subset relation. ∎

6.1.3 Exposing and hiding internal state

Implementation is necessarily defined in terms of input-output relationships. Only through a machine's input-output behavior does it interact with its environment. But a machine's behavior cannot be concisely defined by giving only its input-output behavior. Most interesting machines take sequences of inputs and produce sequences of outputs. The outputs produced at any instant depend on the history of the machine's inputs. Although abstractly we think of sequences as being atomic, so that machine behavior is indeed defined purely as a function, in practice this is cumbersome. Real machines see one input at a time. When we build real machines, we design them so that they will remember something about their past inputs, using some hidden, or internal, state.

Again, recall the stack example from Chapter 2. The I/O behavior of the stack was useful for stating the correctness of the circuit, but it was only after internal state was exposed, in Figure 2.19 (p. 39) and Figure 2.20, that we could reason about correctness.

So that our formalism reflects practice, we now define a hiding operator that lets us talk about such a machine by mentioning only its inputs and its outputs. The definition turns out to be quite simple.

Definition 23 (Hiding Function) *A hiding function is a many-to-one machine homomorphism.*

Properties

The key properties of a hiding function are that both it and its inverse are obedience functions.[4]

Proposition 12 *Every hiding function is an obedience function.*

Proof: Note that each hiding function is also a machine homomorphism. ∎

Proposition 13 *The inverse of every hiding function is an implementation mapping.*[5]

[4] Recall from p. 113 in Chapter 5 that an obedience function is a mapping with respect to which obedience holds, and that an implementation mapping is defined similarly.

[5] Subsequently it need be only an obedience function, but the stronger result is easy.

$$\begin{tikzcd} S \arrow[r, "H_S^{-1}"] \arrow[d, "I"] & \hat{S} \arrow[d, "\hat{I}"'] \\ R & \hat{R} \arrow[l, "H_R"] \end{tikzcd}$$

Figure 6.1: State obedience.

Proof: Note that the inverse of a hiding function is also a machine homomorphism, hence obedience holds. Since the function is input-preserving, conformity is immediate. Furthermore, the inverse of a hiding function cannot violate distinction. ■

Theorem 14 (State Obedience) *Let S, R, $\hat{S}$, and $\hat{R}$ be agents. Let $\hat{R}$ obey $\hat{S}$ with respect to $\hat{I}$. Let $H_S\colon \hat{S} \to S$ and $H_R\colon \hat{R} \to R$ be hiding functions. Then R obeys S with respect to I where $I = H_R \circ \hat{I} \circ H_S^{-1}$.*

Proof: Since H_S is a hiding function, H_S^{-1} is an obedience function. The result follows by transitivity. ■

Significance

The state-obedience theorem is illustrated in the commutative diagram of Figure 6.1.

We wish to show that one agent behaves according to another. This desideratum corresponds to the left-hand side of this diagram. Suppose we cannot make our proof directly, because the agents S and R make use of internal state which is not visible in their behavior. This is the usual case for sequential systems.

Suppose that if we were to include the hidden state in our analysis, then we could show that one agent implements another. This is the right-hand side of the diagram. The state obedience theorem says that we can transfer an obedience result from the right-hand side of the diagram to the left-hand side, provided only that moving from right to left involves only the hiding of internal state.

6.2 Specialization to machines

The preceding development has been rather abstract. We have considered implementation relations between agents which we modeled as pure functions. While this is suitable for the mathematical modeling of our most abstract ideas, actual machines that exist in the real world do not receive their input sequences atomically. Instead, they get inputs one at a time, as inputs arrive. Henceforth we will consider sequential machines which receive such sequence of inputs. We continue to keep the treatment abstract where possible.

6.2.1 Obedience

We now turn to the constituents of implementation: distinction, conformity, and obedience, in the context of sequential machines.

The key property that we must establish in order to say that a circuit implements its specification is that its behavior is allowed by the specification.

We have already discussed what we called a state-implementation property, which allows us to consider hidden state of the circuit and specification, which the environment does not see.

However, our view of agents is still one of functions from inputs to outputs. An agent takes an entire input sequence as a unit, and produces an entire output sequence. In order to complete the methodology we must somehow break down such sequences into constituent elements. Again, we will adopt an abstract point of view. The most important results do not depend on the fact that we are trying to dissect sequences. Instead, they rely on fundamental properties such as associativity, homomorphism, and closure.

6.2.2 Behavior fragments

Recall that $[G]_\bullet$ denotes the closure of set G under binary associative operator $\bullet$.

Proposition 15 *If a binary associative operator $\bullet$ is defined on sets T_R and T_S, mapping $I{:}\, T_S \to T_R$ is $\bullet$-homomorphic,*[6] *and $T_R \subseteq I(T_S)$, then $[T_R]_\bullet \subseteq I([T_S]_\bullet)$.*

Proof: Let $a \in [T_R]_\bullet$. Then there exists an ordered set of $k \geq 1$ elements $a_1, a_2, \ldots, a_k$ from T_R such that $a = a_1 \bullet a_2 \bullet \cdots \bullet a_k$. Since $T_R \subseteq I(T_S)$, for each $a_i \in T_R$ there is some $b_i \in T_S$ such that $a_i \in I(b_i)$. Let b denote $b_1 \bullet b_2 \bullet \cdots \bullet b_k$. Since I is $\bullet$-homomorphic, $a \in I(b)$. Clearly $b \in [T_S]_\bullet$, so $a \in I([T_S]_\bullet)$.

Note that $b_1 \bullet b_2 \bullet \cdots \bullet b_k$ must be defined, since $I(b_1 \bullet b_2 \bullet \cdots \bullet b_k)$ exists and I is $\bullet$-homomorphic. ■

Theorem 16 (Fragment Obedience) *Let $/\!\!/$ be a binary associative operator. If $[T_R]_{/\!\!/} = \mathrm{accept}(R)$ and $[T_S]_{/\!\!/} = \mathrm{accept}(S)$, mapping I is $/\!\!/$-homomorphic, and $T_R \subseteq I(T_S)$, then R obeys S with respect to I.*

[6] i.e., homomorphic with respect to $\bullet$

Proof: This follows from the previous result and the equivalence between obedience and the containment $\mathrm{accept}(R) \subseteq I(\mathrm{accept}(S))$. ■

This is an important result. It builds a formal bridge between behaviors and transitions. When we examined the simple circuits of Chapter 2, we looked only at single transitions—or at most short sequences—of the specification machine. By using the idea of closure, we can combine as many transitions as we need when building a behavior.

The associative operator $\bullet$ can be thought of as a generalization of concatenation. We will soon apply this result to our marked-string representation, yielding an important theorem.

6.2.3 Transitions

Transitions, i.e., pairs of configurations, form a set of generators for a language, where the associative operator is a "domino" matching operator. Let A be a set called the alphabet, and let T be a set of strings of length 2 over A, called the set of transitions.

Definition 24 (Domino Concatenation) *The partial binary operator $\cdot$ is defined on strings over alphabet A. First, $\epsilon \cdot s = s$ and $s \cdot \epsilon = s$. When both strings are nonempty, $a \cdot b = tuv$ if $a = tu$ and $b = uv$ and u has length 1. Otherwise, $a \cdot b = \top$.*

In other words, $\cdot$ behaves somewhat like the rule for the game of dominoes—things can be connected only when they match.

Proposition 17 *The operator $\cdot$ is associative.*

Proof: We must show that the equality $(a \cdot b) \cdot c = a \cdot (b \cdot c)$ holds. If at least one of a, b, or c is the empty string ϵ, this is obvious. Otherwise, if b has length one, then $a \cdot b$ is defined when $a = tb$, in which case $a \cdot b = a$, and $b \cdot c$ is defined when $c = bx$, in which case $b \cdot c = c$, so if either side is defined, both are defined, and they then reduce to $a \cdot c$. But if b is longer, then let $uvw = b$ (v might be empty) where u and w have length 1. Then if $a = tu$, then $a \cdot b = tuvw$ and if $c = wx$, then the left-hand side is $tuvwx$. On the other hand, $b \cdot c = uvwx$ so the right-hand side of the equality is also $tuvwx$. ■

It is then clear that S can be the set of states, and T the transition relation, of a finite-state system; the $\cdot$-closure of T is then the set $[T]_\cdot$ of histories or computations of the system.

6.2.4 Marked strings and overlapped concatenation

In considering machines at different levels of abstraction, an important difference is of time scale. Abstractly, a computer executes sequentially a totally ordered sequence of instructions. These instructions may be implemented in terms of the cycles of a clock. The clock, in turn, may consist of multiple phases. Finally, within each clock phase particular events may have to occur in sequence. We require a model of this phenomenon. Marked strings provide such a model. Marked strings were formally defined in Chapter 3. A full treatment appears in Appendix A. Since marked strings are new, we review them here.

A marked string over the alphabet A is a string over the alphabet $A \cup \{'\}$, assuming that the prime symbol, which we call the *marker*, is not an element of A. For example, ϵ (the empty marked string) and a and abc and $a'bc'$ and $a'''b$ and $''''$ are all marked strings.

We use marked strings to relate two models with different levels of temporal abstraction. Very roughly speaking, the symbols between two markers of the detailed model will correspond to a single symbol of the abstract model. Making an analogy to music, the higher-level model might be in terms of measures, with the lower-level one in terms of notes; the markers would correspond to the bar lines in the staff. In engineering practice, sketching vertical lines in a timing diagram to delimit successive operations is also common.

Since the intended application behind our abstraction is that the markers should delimit the beginning and the end of an operation, we will be most interested in 2^+-marked strings, that is those with at least two marks. A string a is k-marked if it contains k occurrences of the marker. It is k^+-marked if it is j-marked for some $j \geq k$. We say a string is unmarked or mark-free if it is 0-marked, and that it is really marked if it is 1^+-marked. We say that a 2^+-marked string is in normal form if it is expressed as $u'v'w$ where u and w are unmarked strings.

An alternative to including markers between symbols would be to maintain a parallel string of marker bits, with a 1 indicating a marked location and a 0 indicating an unmarked one. Equivalently, this could be represented as a function from positions to bits [175]. However, this would be less general because it disallows the possibility of two adjacent markers. We prefer marker insertion to parallel marker bits anticipating future extensions. For example, the superscalar designs comprise an interesting class of circuits. These dispatch multiple instructions in a single cycle. In moving from a totally ordered instruction stream at the abstract level to a partially ordered stream at the circuit level, it might be convenient to say that some instructions can take zero time.

Overlapped concatenation

In addition to the idea of relating a fine-grained sequence to a coarser one, we must acknowledge that in a pipelined circuit one operation can begin before its predecessors have completed. By keeping this aspect of our model sufficiently general, we find that we can also use it for more mundane aspects. For example, most circuits require that the clocks be operated a certain way before an execution cycle can commence. When one operation follows another, there will be an overlap.

We model this idea with the notion of overlapped concatenation. We can illustrate the basic idea in Figure 6.2. We take two strings $u'v'w$ and $x'y'z$, where u, w, x, and z contain no marks (but v and y may contain marks). Then we align the last mark of the first string with the first mark of the last one, to produce a new string, $u'v'w /\!\!/ x'y'z$, where we insert the mark symbol as necessary. The figure shows what might be called "well-behaved" overlapped concatenation, where x does not extend past the left of u and w does not extend past the right of z. The

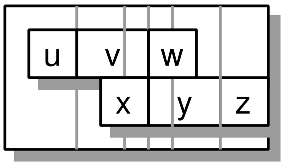


Figure 6.2: Overlapped concatenation of 2^+-marked strings

formal development of marked strings does not require such a condition on overlap. (This accounts for much of the complexity of the formal definition.)

We extend overlapped concatenation to sets, throwing out cases where conflict occurs. For example, the overlapped concatenation $\{a'bc', a'ba'\} /\!\!/ \{a'bc', a'ba'\}$ yields $\{a'ba'bc', a'ba'ba'\}$. (Here a, b, and c are symbols, not strings.)

Chapter 3 introduced the basics of a theory of marked strings. Appendix A develops the full theory. An important result is theorem 2, that $/\!\!/$ is an associative operator. Furthermore, we can define a function CL that removes the last mark from a marked string. The set of marked strings with two marks, which we call the double-marked strings and denote $A^{2'*}$ for alphabet A, is especially important. If x has two marks, we can show (proposition 5) that $\mathrm{CL}(x) /\!\!/ x = x$. That is, if we remove the last mark from a copy of x, when we form an overlapped concatenation with the original string on the right, the alignment is such that we get the original string.

Additionally, we can split strings at markers. We can define the "measurements" of a string, denoted $||x||$, as the pair giving the lengths of the portions of x before and after the first mark. We can also define a notion of "compatibility" $\approx$ between marked strings; for 2-marked strings, $x \approx y$ iff $\mathrm{CL}(x) /\!\!/ y \neq \top$. We indicate formation of a set of incompatible strings with a superior tilde; if S is a set of marked strings, $\tilde{S}$ denotes the set of incompatible strings having the same measurements.

Using this machinery, we can take maps which take states to 2-marked strings, and construct maps that take transitions to 2-marked strings, and which are homomorphic over the combining of transitions to form behaviors.

Behavior maps

Using marked strings, we can now proceed to develop a methodology capable of showing that pipelined systems implement non-pipelined specifications.

First, we define a mapping from configurations of the specification onto 2-marked strings of configurations of the realization. We call such maps configuration maps. We use this mapping to induce a mapping from strings of configurations of the specification onto marked strings of configurations of the realization. Strings of configurations are behaviors, so we call such maps behavior maps. A transition in the specification is just a string of two configurations. We can combine transitions of the specification using the domino concatenation operator defined previously (definition 24). The key result is that the behavior map induced by a configuration map is a homomorphism. This allows us to apply theorem 16 (fragment obedience). Thus, we can check containment of the image, under the behavior map, of the transitions of a specification, but conclude that the realization obeys the specification.

Definition 25 (Configuration map) *A configuration map from machine S to machine R is a $J\colon \mathrm{config}(S) \to \mathcal{P}(\mathrm{config}(R)^{2\prime *})$ such that every two elements $s, t \in \mathrm{range}(J)$ have the same measurements, $||s|| = ||t||$.*

In other words, a configuration map maps configurations of S onto 2-marked strings of $\mathrm{config}(R)$.

Definition 26 *A configuration map is* distinct *if whenever $a \neq b$, $c \in J(a)$ and $d \in J(b)$ implies $c \not\approx d$.*

Definition 27 (Behavior map) *The behavior map induced by a configuration map J is a map $I\colon \mathrm{config}(S)^* \to \mathcal{P}(\mathrm{config}(R)^{\prime *})$ given by*

$$I(x) = \mathrm{CL}(J(x))$$

where J on strings is extended as $J(ax) = J(a) /\!\!/ J(x)$.

Theorem 18 (Overlap theorem) *The behavior map I induced by a distinct configuration map J is a homomorphism between the structure $\langle \mathrm{config}(S)^*, \cdot \rangle$ and the structure $\langle \mathcal{P}(\mathrm{config}(R)^{\prime *}), /\!\!/ \rangle$.*

Proof: There are two cases to consider. Let $\hat{x}$ and $\hat{y}$ be strings over config(S). The cases are defined by whether $\hat{x} \cdot \hat{y} = \top$. If $\hat{x}$ is of the form xb and $\hat{y}$ is of the form by, where b has length 1, then the reasoning is as follows. We expand by the definition of I, then apply lemma 4. Choosing elements of $J(b)$ we can use proposition 6 in order to apply lemma 3. Finally we collapse by the definition of I.

$$\begin{aligned} I(xb \cdot by) &= \mathrm{CL}(J(x) /\!\!/ J(b)) /\!\!/ \mathrm{CL}(J(b) /\!\!/ J(y)) \\ &= J(x) /\!\!/ \mathrm{CL}(J(b)) /\!\!/ J(b) /\!\!/ \mathrm{CL}(J(y)) \\ &= J(x) /\!\!/ J(b) /\!\!/ \mathrm{CL}(J(y)) \\ &= \mathrm{CL}(J(x) /\!\!/ J(b) /\!\!/ J(y)) \end{aligned}$$

If on the other hand $\hat{x}$ is of the form xa and $\hat{y}$ is of the form by, where a and b are different and have length 1, then we can reason similarly:

$$\begin{aligned} I(xa \cdot by) &= \mathrm{CL}(J(x) /\!\!/ J(a)) /\!\!/ \mathrm{CL}(J(b) /\!\!/ J(y)) \\ &= J(x) /\!\!/ \mathrm{CL}(J(a)) /\!\!/ J(b) /\!\!/ \mathrm{CL}(J(y)) \end{aligned}$$

Since J is distinct, from proposition 5 and the subexpression $\mathrm{CL}(J(a)) /\!\!/ J(b)$ we can conclude $\mathrm{CL}(J(a)) /\!\!/ J(b) = \emptyset$. ■

This is the first central result of the thesis. It allows us to reason about transitions and conclude properties of behaviors, and it accounts for the overlapped operation of pipelines. In order to verify that a realization obeys a specification, we can provide a distinct configuration map from the specification onto the realization. Once we have done so, this theorem tells us that we can apply the fragment obedience theorem—we need only show containment with respect to transitions and we can conclude obedience. The remaining step is to show containment of transitions.

A verification methodology has taken shape. We can show that obedience holds for individual fragments, i.e., transitions, of two machines. This implies that obedience holds for the closures, i.e., behaviors of the machines. These behaviors include internal state that is not visible through the their IO behavior. But according to our state-obedience result of theorem 14, obedience holds between the two machines when they are viewed only through their IO.

The remaining step is to relate the methodology as thus far developed to our idea of what specifications should look like, which we discussed in Chapter 3.

6.3 Assertions

An assertion $N = (A, C)$ with antecedent A and consequent C represents a superset of the transition relation of M. We will denote this set by $T(N)$ (to be read "transitions of N").[7]

Definition 28 (Transitions of an assertion) *The set of transition of an assertion N is given by the equation*

$$T(N) = A \times C \cup \overline{A} \times \mathbf{U}_M$$

This reflects the intended meaning of an assertion: that 1) when the system is in a configuration within set A, it must next be in a configuration within set C, but 2) if the system starts in a configuration outside A, the assertion imposes no restriction on the configuration it will next be in. This is the meaning of a single assertion.

Let i index the set of assertions. The meaning of a set of assertions is simply the intersection of the meaning of each of its members, $\bigcap_i T(N_i)$.

Definition 29 (Images of a transition) *The set of images of a transition N_i is defined by the equation*

$$\mathrm{TI}(N_i) = \bigcup_{a \in A_i} \bigcup_{c \in C_i} \mathrm{CL}(I(a) /\!\!/ I(c)) \cup \bigcap_{a \in A_i} \mathrm{CL}\left(\tilde{I}(a) /\!\!/ \mathbf{U}\right)$$

in which the bold $\mathbf{U}$ denotes the set of all possible marked strings representing behaviors of the realization.

Our goal is to check a circuit against each assertion, and somehow draw a conclusion about the relation between the circuit and the entire specification. From the preceding material, the desired conclusion is clear: we wish to show that

$$T_R \subseteq I\left(\bigcap_i T(N_i)\right)$$

where I is a distinct behavior mapping. T_R is the behavior of realization R, and $T(N_i)$ is the set of transitions defined by the i-th assertion in a specification. This is a containment of behavior fragments, from which we can conclude containment of behaviors, hence we conclude that the circuit obeys the specification. We seek a test to imply this desideratum.

However, the check that we can actually make is, for each assertion i:

$$T_R \subseteq \mathrm{TI}(N_i)$$

where TI is the set of images of a transition, as defined above. We make this check by a form of symbolic simulation called trajectory evaluation [56]. That is to say that in essence we check two things. First, when the circuit is forced to behave

[7]The notation $\mathbf{U}_M$ was defined on page 76 to be config(M).

according to the antecedent of the assertion, it also behaves according to the consequent. Second, implicitly, when the circuit behaves contrary to the antecedent, it may do anything. That is, for behaviors which are described by marked strings incompatible with everything in the antecedent, the circuit's behavior might be described by any marked string. (We are restricting our attention to marked strings whose measurements are all the same, because of our definition of a transition map. This is simply because we are examining the circuit's behavior for only a limited period of time when we are checking assertions.)

Theorem 19 (Checking) *If the containment $T_R \subseteq \mathrm{TI}(N_i)$ holds for every assertion N_i, and I is a distinct, onto behavior mapping, then the containment*

$$T_R \subseteq I\left(\bigcap_i T(N_i)\right)$$

also holds.

Proof: We need to demonstrate that $\bigcap_i \mathrm{TI}(N_i) \subseteq I\left(\bigcap_i T(N_i)\right)$ which implies the desired containment by transitivity from the previous equation. Since the map I is distinct we can easily see that

$$\bigcap_i I(T(N_i)) \subseteq I\left(\bigcap_i T(N_i)\right)$$

so we need only to show for every i that $\mathrm{TI}(N_i) \subseteq I(T(N_i))$ When we expand this by the definitions of TI and T and I we get the new desideratum

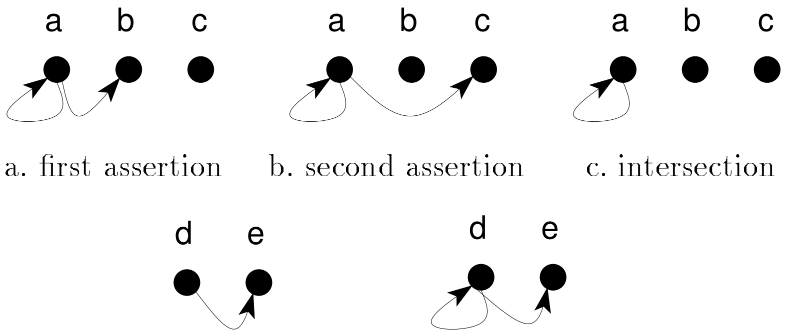
$$\begin{aligned}\bigcup_{a\in A_i} \bigcup_{c\in C_i} \mathrm{CF}(I(a) /\!\!/ I(c)) \cup \bigcap_{a\in A_i} \mathrm{CF}\left(\tilde{I}(a) /\!\!/ \mathbf{U}\right) \subseteq \\ \bigcup_{a\in A_i} \bigcup_{c\in C_i} \mathrm{CF}(I(a) /\!\!/ I(c)) \cup \bigcup_{a\in \overline{A_i}} \bigcup_{c\in \mathbf{U}_M} \mathrm{CF}(I(a) /\!\!/ I(c))\end{aligned}$$

Each side of this containment is a union, and the left-hand side of the unions on each side is the same. For the containment to hold, we need only establish containment between the right-hand side of each union. This will follow from the containment

$$\bigcap_{a\in A} \tilde{I}(a) \subseteq I\left(\overline{A}\right)$$

We can see that anything in the left-hand side must be in the right by choosing any $s \in \cap_{a\in A} \tilde{I}(a)$. We know for any $t \in A$ that $s \not\approx t$, so $s \notin I(A)$. Since I is surjective, $s \in I(\overline{A})$. ■

This is the second central result of the thesis. It tells us that we can check that the circuit behavior falls within the image of each assertion in the specification, and conclude that the circuit's transitions are contained within the image of the



a. first assertion b. second assertion c. intersection

d. realization e. mapped specification

Figure 6.3: Example of specification illustrating need for distinction.

specification's transitions. This means that the the overlap theorem holds, so the circuit obeys its specification.

The requirement that the mapping be distinct is fundamental to the direction of the mapping, as the following example illustrates.

For ease of illustration, in this example we will label states, represent transitions with diagrams, and ignore the implicit transitions not covered by antecedents. Consider a specification machine with three states a, b, and c. Let its specification be given by the two assertions $a \stackrel{\delta}{\Rightarrow} a \vee b$ and $a \stackrel{\delta}{\Rightarrow} a \vee c$. The first says that when the machine is in state a, either it can stay there, or it can move to state b. The second says that when the machine is in state a, either it can stay there, or it can move to state c. Taken together, they say that when the machine is in state a it must stay there. Figure 6.3, parts a, b, and c, illustrate the transition system defined by this specification.

Consider a realization machine with two states d and e, which from state d can only make a transition to state e. Figure 6.3d illustrates its transition system. Consider the mapping $f\colon a \mapsto d, b \mapsto e, c \mapsto e$. Under this mapping both assertions hold, yet the second machine does not implement the first.

Consider the image of the first assertion $a \stackrel{\delta}{\Rightarrow} a \vee b$ under this mapping, namely $d \stackrel{\delta}{\Rightarrow} d \vee e$. This is shown in Figure 6.3e. Clearly this assertion is satisfied by the realization machine, whose transition could be described exactly by the assertion $d \stackrel{\delta}{\Rightarrow} e$. The image of the second assertion $a \stackrel{\delta}{\Rightarrow} a \vee b$ is the same, $d \stackrel{\delta}{\Rightarrow} d \vee e$, so it is also satisfied.

Yet the specification machine (Figure 6.3c) always stays in one state, while the realization machine (Figure 6.3e) always leaves it. Clearly this realization does not implement this specification. Distinction is an important property.

6.4 Distinction and conformity

In addition to obedience, the other two properties we must establish are distinction and conformity.

6.4.1 Distinction

In order to say that one machine implements another, distinct outputs of the specification must correspond to distinct outputs of the realization. It is necessary to tell different things apart to prohibit trivial implementations.

It might be thought that this is a simple property which can be checked pointwise, and then shown inductively to hold for sequences. Unfortunately, this is not so. When successive realization outputs are concatenated, the boundary between them disappears. But the location of this boundary may be the only thing distinguishing two output sequences.

Ambiguous boundaries

A simple example illustrates this. Consider a serial adder (or any other circuit having an unsigned, bit-serial output). Assume as usual that the least-significant bit of output is presented first. Since we are reading left-to-right, for this example only we will write the least-significant bit of each word on the left. For example, 10 is one and 01 is two.

Suppose the circuit is being used on words of several different lengths.[8] Individually, outputs are not ambiguous. The bit-string 0 is a one-bit representation of the number 0. The bit-string 1 is a one-bit representation of the number 1. The bit-string 10 is a two-bit representation of the number 1. The bit-string 01 is a two-bit representation of the number 2.

However, when we concatenate outputs, ambiguity results, even if we know the length of the abstract output. The bit-string 101 is the one-bit representation of the number 1, followed by the two-bit representation of the number 2, but it is also the two-bit representation of the number 1, followed by the one-bit representation of the number 1. In each case, the abstract output is a sequence of length two; it is either 12 or 11.

However, adding marks to the input and output strings eliminates the ambiguity. In the example above, the marked bit-string $'1'01'$ could be used for the first alternative, and the marked bit-string $'10'1'$ for the second.

The marks, however, are fiction. Actual machines do not have them. The proper interpretation of the timing of a system's outputs is up to its environment, which we are not considering.

[8] Actually, most bit-serial systems are pipelined and the word size is fixed [222], so this is a somewhat contrived example.

The marks are analogous to the vertical lines one might sketch in a timing diagram to see how the input timing relates to the output timing. The vertical lines, which correspond to our marks, are useful in reasoning about the circuit. The circuit itself doesn't need this information, and the environment in which the circuit is used keeps track of timing in its own way (which we do not even have reason to consider).

Distinction is a component property of implementation because in order to say that one machine implements another, different outputs of the specification must not correspond to the same output of the realization. If were were to allow different specification outputs to correspond to the same realization output, we would allow the trivial realization, which has only one output, to implement every specification. Clearly this is undesirable, so we imposed the requirement of distinction on the output mapping in our definition of implementation.

Furthermore, we required that the state mapping be distinct in order to prove the overlap theorem, and as we demonstrated with a simple example, this was not merely a technical requirement. Thus, we must be able to guarantee that a mapping is distinct in order to use it for verification.

We will discuss this further in Chapter 7 when we discuss the semantics of a mapping language, in section 7.3; for now we note that there are two possible approaches: to check that entire mappings are distinct, or to guarantee that they are distinct from the way in which they are constructed.

6.4.2 Conformity

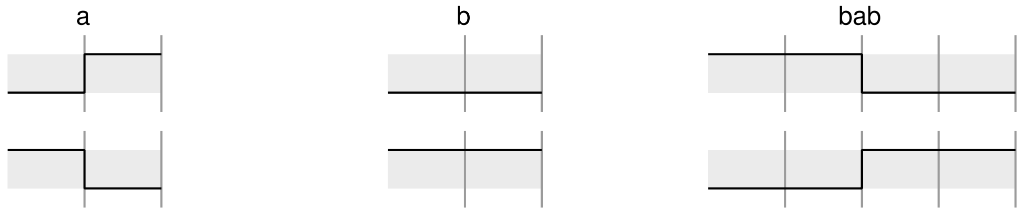
The last of the properties that we must establish in order to say that a circuit implements its specification is conformity.

Briefly, conformity is the property that every legal abstract input sequence can be applied to the circuit. Where two adjacent abstract symbols map to overlapping circuit operations, the areas of overlap in the operation must conform to each other.

Conformity is problematic, as it is a property of the input language. Since the input language could (at least conceptually) be anything, the complexity of general conformity could be quite high—even undecidable.

Such theoretical generalities should not hinder progress on some practical cases. Fortunately, in the case of a microprocessor the input language is rather simple. The alphabet is small, consisting of a few symbols such as "reset," "run," and "interrupt" and the interesting inputs—the input language—are those that contain exactly one "reset," at the beginning.

Example Conformity is not merely of theoretical concern, as the following example illustrates. Consider a system whose inputs are encoded with a NRZ code (transition signalling). We can model such a system with specification input alphabet $\{a, b\}$. For concreteness, let the realization input alphabet be $\{0, 1\}$ and



a. Map of input symbol a b. Map of input symbol b c. Can map input string bab

d. Bad map of input symbol b e. Cannot map input string bab

Figure 6.4: Conformity of input sequence under NRZ code. c. Mapping the abstract input string bab yields two possible realization inputs. d, e. With a mapping that lacks conformity, the input string bab cannot be mapped onto the circuit. Conflict is indicated by the dark grey area. Note that it is necessary to examine a sequence of at least three symbols in order to notice this problem.

an implementation mapping be $I\colon a \mapsto \{\,0'1', 1'0'\,\} : b \mapsto \{\,0'0', 1'1'\,\}$. Thus the specification input a is represented by a transition in the circuit, while the input b is represented by the absence of a transition. This situation is illustrated by Figure 6.4 parts a, b, and c.

Suppose, however, that instead of this mapping, we use a mapping $J\colon a \mapsto \{\,0'1', 1'0'\,\} : b \mapsto \{\,0'0'\,\}$. In other words, a is represented by a transition, while b is represented by the absence of a transition *at a particular logic level.* Figure 6.4 part d illustrates this situation.

Under this second mapping, the input string bab cannot be mapped onto the circuit. This can be determined from the marked string formalism. (Recall that strings are mapped by overlapped concatenation of the images of their elements, and that when we extended overlapped concatenation to sets, we "threw out" the conflict indicator $\top$.) If we attempt to compute the mapping, we find that while $J(ba) = \{\,0'0'1'\,\}$ and $J(b) = \{\,0'0'\,\}$, the mapping $J(bab) = \emptyset$. We must find a way to ensure conformity of languages and mappings more complicated than this example.

Construction and inspection of a machine that accepts the conformable inputs is a plausible approach for such a simple language. For more complicated languages, such as ones with data-dependent sequencing constraints[9], a more general model-

[9]Imagine a pipeline without interlock or bypass. The input language should be defined to

checking approach could be used in place of inspection.

Infinite conformity machine

It is easy to construct an infinite machine accepting the conformable inputs. The idea is to label states with strings. Let L denote a labeling function on states, let A be the abstract input alphabet, and let I the mapping from A onto 2-marked strings of the circuit. Start with an empty string labeling a start state, the sole element of some frontier set F. While the frontier set F is nonempty, choose some $f \in F$. For each $a \in A$ and each $s \in I(a)$, create a new state q with label $L(f) /\!\!/ s$ and transition $f \xrightarrow{a} q$ and add q to the frontier set. Then remove f from F.

Clearly this does not terminate; F is shrinking no faster than it grows. However, it does yield an easy test for conformable strings: run this machine on the string, and check that the final state label is not $\top$.

Finite conformity machine

The infinite construction is the basis for a better algorithm, by mapping the infinite machine to a finite one, using a map "rep" such that $L(\mathrm{rep}(s)) = \top$ iff $L(s) = \top$.

Note that the important thing about a state is its label, that being the information used in the ultimate decision of conformity for a string. The infinite machine used two operations on labels: the construction of new labels, and the test for equality with $\top$. Thus, the mapping of the infinite machine to a finite one should obey the equation $\mathrm{rep}(a /\!\!/ b) = \mathrm{rep}(\mathrm{rep}(a) /\!\!/ b)$, and $\mathrm{rep}(s)$ should be $\top$ iff $s = \top$. Note that the equation need only hold for $b \in I(\mathrm{inp}(S))$, not arbitrary b.

With such a mapping we can construct a machine accepting the conforming strings. Instead of creating state q with label $L(f) /\!\!/ s$, we first check for an existing state with label $\mathrm{rep}(L(f) /\!\!/ s)$ and if so we use it; otherwise we create q. Clearly the modified algorithm terminates if the range of rep is finite. Construction of a suitable rep is an exercise in marked strings.

Further work

Symbolic extensions of this are clearly possible; their details and utility would depend heavily on the symbolic model-checking to be applied to the constructed machines.

6.5 Related work

Those familiar with program verification are no doubt wondering how assertions over state machines relate to the assertions of Hoare logic of program verification

avoid pipeline hazards.

[133]. A Hoare assertion is a triple $P\{Q\}R$ where P and R are predicates on program state, and Q is a statement—possibly a compound statement—in an imperative programming language. Predicate P is called the precondition, and predicate R is called the postcondition. The meaning of a Hoare assertion is that when a program in a state within P executes statement Q, the program is then in a state within R.

The chief difference between our assertions and the Hoare-assertion notation is that our assertions describe single transitions of a system. We have no combining forms. Supposing, however, that we were to invent a notation to allow the sequential composition of transitions, we can sketch a rough correspondence between our notion of assertions and Hoare logic.

To cast an assertion (A, C) in Hoare-logic form, we have two alternatives. The most straightforward is to identify $P = A$ and $R = C$, and let Q denote the state transition—the passage of time. We can show that in such a formulation, the proof rules for Hoare logic—strengthening preconditions, weakening postconditions, and sequential composition—remain valid.

The other alternative is to identify $P = \Pi_r A$ and $R = \Pi_r C$, and let $Q = \Pi_i A$. In addition, we require that $\Pi_i C$ be the set of all possible inputs. That is, P is the portion of the assertion's antecedent that constrains initial state, R is the portion of the assertion's consequent that describes final state, and Q is the portion of the assertion's antecedent that describes inputs. We require that the consequent say nothing about inputs after the transition. Thus, for machine S we would have

$$\Pi_r A', \{\Pi A\} \, (\Pi_r C \times \mathrm{inp}(S))$$

where Π is the projection function which yields inputs.

In this formulation, the "statement" Q denotes the machine's input—the "operation" that the machine is being told to perform during the transition. While this is artificial, it maintains the segregation between command and state present in Hoare logic.

Our notation for transitions is rather cumbersome if extended to sequential composition. But suppose we do so. For example, we could write the compound assertion $(A_1; A_2; A_3,\ C_1; C_2; C_3)$ with $A_{i+1} = C_i$. We would need to extend domino concatenation in a way so as to be able to construct such structures. The ideal notation would also elide elements like C_1, C_2, since they can be recovered, and somehow hide the state portion of A_2, A_3. In such a formulation it should be possible to show that an analog of Hoare's sequential proof rule holds.

Before carrying out the construction in detail, however, we should consider the purpose of the work. The purpose of Hoare logic is to enable compositional reasoning about statements in the presence of several combining forms, such as **if** and **while**. This is analogous to reasoning about the specifications discussed here, rather than the implementation properties we are focusing on. Also, we have only a single combining form: sequential composition, rather than several. In summary, Hoare logic, while having some similarity to our notations, is not an appropriate formalism for this work—the goal and the character of the problem are actually different.

The methodology described here depends upon the explicit statement of the relationship between a realization and its specification. For systems designed by engineers, such a relationship already exists. At worst, it is in tacit form. However, for a system synthesized from its specification using the mechanisms of high-level synthesis, this relationship may not be available. Blackburn [19] has considered the problem of relating levels of design representation in a high-level synthesis system. In fact, he proposed that formal verification could be one application of his work. It would be possible to use his techniques to automatically construct mappings between simple synthesized circuits and optimized synthesized circuits, and then check their relationship by the techniques of this thesis, thereby ensuring that the applied optimizations were correct.

There has been relatively little work actually capable of showing that a pipelined circuit meets a specification having non-overlapped timing. Sometimes—when the IO behaviors of the specification and of the circuit are actually the same—simple cases can be handled by state-machine comparison algorithms, e.g., constructing the product machine and checking that it never produces an inconsistent output. Nobody has used such a technique to verify a microprocessor.

Many approaches do not attempt to encompass the possibility that the realization of a specification may have quite different timing from the original. One approach that has similar goals is that of Devadas and Keutzer [91] based on p-automata. A p-automaton is a machine with extra inputs, called metainputs, which represent nondeterminism. P-automata may be constructed from a behavioral specification, and circuits may be compared to them using standard techniques. This approach is subject to the state explosion problem.

The string-functional semantics originated by Bronstein [36] and further developed by Van Aelten [236] contains several specialized relations for describing pipelines. Bronstein's original formulation allowed for two "pipeline relations:" one called α expressed a delay relationship, and one called β expressed a "stuttering" relationship [1]. Van Aelten revised β so that it nearly subsumed α, and introduced a "parallelism" relation. Bronstein's mechanization was in logic using the Boyer-Moore prover, while Van Aelten has taken more of a state-machine approach. Bronstein verified a simple pipelined CPU, and Van Aelten and colleagues have verified a CPU and other simple pipelines [237]. So far neither technique has been used to verify a processor, but there appears to be no reason that this formalism could not be used as the basis of such a task.

Stuttering [1] is a relationship between two sequences where one may repeat occurrences of symbols. For example, the strings "abcdefg" and "aaabccc" are in a stuttering relationship, while "abcdefg" and "aaabdcc" are not. Stuttering is used to model different levels of timing by "speeding up" the more abstract machine to let it make transitions at the same rate as the more concrete machine, while requiring that these new or stuttering transitions always be from a state to itself. One conceptual difficulty with stuttering is that it may add state to the more abstract machine—knowing the machine's present state and input is insufficient for predicting the transition, not knowing whether it will be a stuttering transition. This can be modeled with nondeterminism, but this converts a deterministic specification into a nondeterministic one, and may also require imposing auxiliary conditions to ensure progress (i.e., to disallow the case where the system henceforth makes only stuttering transitions).

Srivas and various colleagues have used theorem provers which were developed for reasoning about functional languages to verify two pipelined processors [213, 227], using a stuttering relationship. Functions that manipulate (infinite) sequences can be written using (lazy) functional languages, making such provers suitable tools.

It is significant that all of the work capable of addressing this—including the present thesis, which tries to take a state-machine-based view of systems—ultimately at some point describes relations between sequences. At the granularity of a single clock cycle, there is a great difference between a pipelined machine and a non-pipelined version. Only by considering multiple cycles can the similarity be seen.

6.6 Chapter summary

Implementation of one machine by another can be viewed as set containment: of I/O behaviors, but also when internal state is exposed. When machines differ in their level of abstraction, a nontrivial relation between their behaviors becomes necessary. When this relation is expressed as a nondeterministic mapping, implementation becomes containment within an image under this mapping. When behaviors form semigroups and the mapping is homomorphic, containment of one set in the image of another will follow from a similar containment involving only generators of the semigroup. Thus, to show that a realization implements the behaviors of its specification, requires only to show that it implements the transitions of its specification.

Digital systems employ pipelining, the overlap of successive operations. Thus, each "operation" is actually a sequence. Short sequences of system operation can be represented by strings. Overlapped sequences can be analyzed by augmenting the alphabet with a distinguished symbol, the *alignment marker*, and defining an associative *overlapped concatenation*.

Combining these two ideas lets us generate verification conditions by mapping abstract transitions to sequences of circuit operation. An assertion is a symbolic representation of a set of transitions.

Chapter 7

Applying the methodology

The theory discussed thus far is rather abstract. Details must be supplied if we are to apply the theory to a real example. This chapter supplies these details.

First, it develops a notion of decomposition, which is needed to verify a processor realization against a computer specification. Then it discusses issues regarding the representation of sets of marked strings in a verifier, and the facilities needed to make a verifier usable.

7.1 Decomposition

This section is a digression from the methodology. It discusses the verification of decomposed systems. Decomposition is an important pragmatic concern. If P denotes a processor and M denotes a memory, their composition PM is a computer. The behavior of the components P and M determines the behavior of the composition. We already saw a simple example of decomposition in section 2.3 when we considered a 3-bit stack built from a 2-bit stack and a 1-bit cell. In the present notation, the 2-bit stack is M and the 1-bit cell is P. As we saw in this example in Chapter 2, we cannot simply say that decomposition can be treated at the specification level. A computer may perform many memory operations in executing one instruction, so although the entire instruction can be considered to be a single transition of the entire system, the treatment of the memory system must consider the temporal aspect of the sequence of memory operations.

7.1.1 Compositions

Two agents can be composed if they can be connected using projection functions. Suppose we have agents M and P. We will construct a composite PM. The choice of letters suggests that P might be a processor and M a memory system; PM would be a computer. This is not a requirement. For example, P might be a

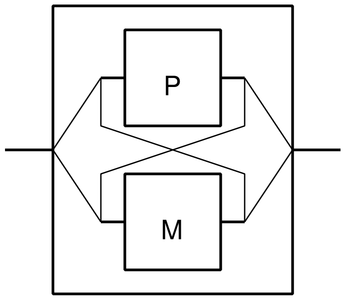


Figure 7.1: Wiring diagram for machine composition. Two component machines, wired together as shown, form a composite machine. Outputs of either machine may be connected to inputs of the other. Inputs and outputs may also be connected externally.

data path and M another circuit such as a controller.

To connect P and M, we would wire some outputs of P to inputs of M, and some outputs of M to inputs of P. In order to connect outputs of P to inputs of M, some projection of $\text{ins}(M)$ must be a projection of $\text{beh}(P)$. This projection denotes the outputs of P that are connected as inputs to M. Similarly, some projection of $\text{ins}(P)$ must be a projection of $\text{beh}(M)$. It denotes the outputs of M that are connected as inputs to P. Figure 7.1 illustrates the connection of two machines. Inputs are on the left and outputs are on the right.

Let int denote the function that takes an input or output of a component machine P or M and projects it onto the portion connected to the other component machine. This function represents internal connections within the composite. We can form a composite of any two machines for which such an int exists.

Let ext denote the function that takes an input of a component machine, and projects it onto the portion that does not get connected. This function represents external inputs that the composite receives from its environment.

In this case, the set $\text{beh}(PM)$ will be $\text{beh}(P) \times \text{beh}(M)$. The set $\text{ins}(PM)$ will be $\text{ext}(\text{ins}(P)) \times \text{ext}(\text{ins}(M))$.

We can let Π_P denote a projection function that takes $\text{ins}(PM)$ to $\text{ext}(\text{ins}(P))$ and similarly takes $\text{beh}(PM)$ back to $\text{beh}(P)$. Similarly, Π_M can take $\text{ins}(PM)$ back to $\text{ext}(\text{ins}(M))$ and $\text{beh}(PM)$ to $\text{beh}(M)$.

We can also let Π_{IBP} denote a projection function that takes $\text{beh}(PM)$ to the internally connected portion of $\text{beh}(P)$. Similarly, Π_{IBM} can take $\text{beh}(PM)$ to the internally connected portion of $\text{beh}(M)$.

The commutative diagram in Figure 7.2 illustrates these connections.

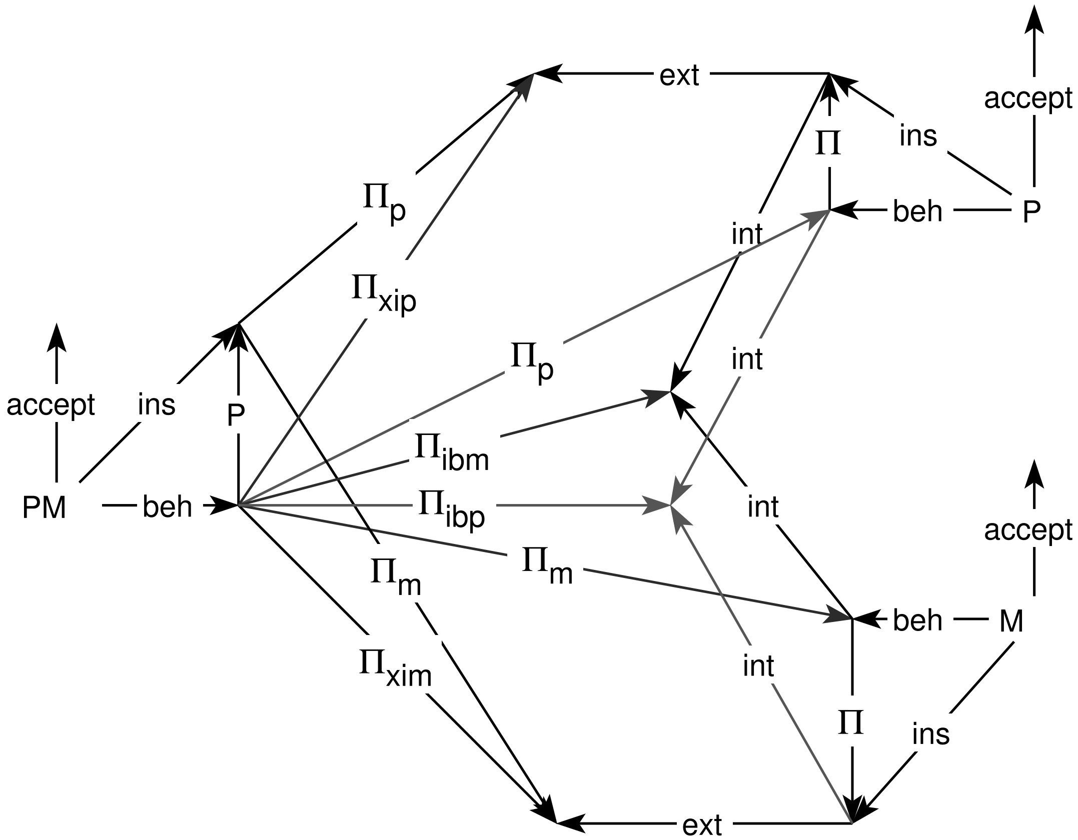


Figure 7.2: Composition of two machines. This commutative diagram shows the large number of equivalences that hold when two machines are connected. It illustrates one of the difficulties of discussing composition: there are many possible notations for almost every object.

Behavior of compositions

We have shown how to connect two machines to form a composite, and examined the resulting structure. We now look at the resulting behavior, in terms of acceptance. A behavior is allowed by the composite if it is allowed by both components. Thus, the connected portion of the input that a component sees must be equal to the connected portion of the result that the complementary component produces—just as expected from the wiring diagram of Figure 7.1.

Define the relation $\mathrm{agree}(v_p, v_m)$ to hold when the two equalities $\mathrm{int}(v_P) = \mathrm{int}(\Pi v_M)$ and $\mathrm{int}(v_M) = \mathrm{int}(\Pi v_P)$ both hold. This captures the notion that the connected portions of the component behaviors agree. Thus, the set $\mathrm{accept}(PM)$ can be written as $(\mathrm{accept}(P) \times \mathrm{accept}(M)) \cap \mathrm{agree}$. If we treat the notation $\mathrm{agree}(v_P, v_M)$ as the characteristic predicate of the relation, we can also write the set $\mathrm{accept}(PM)$ as $\{\,(v_P, v_M) \in \mathrm{accept}(P) \times \mathrm{accept}(M) \mid \mathrm{agree}(v_P, v_M)\,\}$.

7.1.2 Behavior of decompositions

Given a composition, we might wish to reason about a component. Our motivation is that we find it easier to describe computers (which have instructions with precisely defined semantics) and memory systems (which perform read, write, and storage functions) than to describe processors. Ultimately we would like to show that a processor connected to a memory implements a computer, but avoid the need to explicitly specify all the details of the processor itself.

Definition 30 (Decomposition function) *The decomposition function, with respect to M, is the function $\frac{1}{M}$ defined by the following equation.*

$$\frac{1}{M}(V_C) = \{\, v_P \mid \forall v_M \in \mathrm{accept}(M) \centerdot \mathrm{agree}(v_P, v_M) \Rightarrow (v_P, v_M) \in V_C \}$$

Recall that the notation $\centerdot$ was defined back in Section 3.1.

Theorem 20 *If* $\mathrm{accept}(P) \subseteq \frac{1}{M}(S)$ *then* $\mathrm{accept}(PM) \subseteq S$.

Proof: We know that $\mathrm{accept}(PM) = \{\,(v_P, v_M) \in \mathrm{accept}(P) \times \mathrm{accept}(M) \mid \mathrm{agree}(v_P, v_M)\,\}$ and we assume that $\mathrm{accept}(\mathrm{P}) \subseteq \frac{1}{M}(S)$. Thus $\mathrm{accept}(PM) \subseteq \{\,(v_P, v_M) \in \frac{1}{M}(S) \times \mathrm{accept}(M) \mid \mathrm{agree}(v_P, v_M)\,\}$. The right-hand side is equivalent to $\{\,(v_P, v_M) \mid v_P \in \frac{1}{M}(S) \wedge v_M \in \mathrm{accept}(M) \wedge \mathrm{agree}(v_P, v_M)\,\}$. Expanding the definition of $\frac{1}{M}$ we see that this is $\{\,(v_P, v_M) \mid v_M \in \mathrm{accept}(M) \wedge \mathrm{agree}(v_P, v_M) \wedge \forall \hat{v}_M \in \mathrm{accept}(M) \centerdot \mathrm{agree}(v_p, \hat{v}_m) \Rightarrow (v_p, \hat{v}_M) \in S\,\}$. Since $v_M \in \mathrm{accept}(M)$ the condition of the implication applies, so we can deduce that the set is equivalent to $\{\,(v_P, v_M) \mid v_M \in \mathrm{accept}(M) \wedge \mathrm{agree}(v_P, v_M) \wedge (v_P, v_M) \in S\,\}$ which is obviously a subset of S. ∎

This result allows us to deduce containment for the composite PM by checking containment of a component P, using the function $\frac{1}{M}$.

7.1.3 Checking decomposed systems

Instead of checking containment directly, we compose the decomposition function $\frac{1}{M}$ (p. 148) with the mapping function when we check fragment containment.

While this seems acceptable, one might wonder whether a more elegant result is possible. In particular, can the decomposition function $\frac{1}{M}$ be made $/\!/$-homomorphic? If so, then since function composition preserves homomorphism, the entire mapping would be homomorphic.

A simple observation from the formalism proves that this is not possible. The function $\frac{1}{M}$ is similar to a projection function, so it will not be injective (i.e., one-to-one), and thus will not be homomorphic with respect to the marked string join operations $\sqcup_p$ and $\sqcup_s$, which appear in the definition of $/\!/$.

A more compelling argument, however, is that if the mapping were homomorphic, by using it we could state that a processor implements the specification of a computer—eliminating the memory entirely! Clearly, if we could deduce such a false statement something would be amiss.

Although $\frac{1}{M}$ is not homomorphic so we cannot obtain an obedience result directly for a decomposed system, we can obtain a result indirectly. We do this by applying theorem 20. It states that containment of the component behavior in the image, under the decomposition function, of the system specification, implies containment of the entire system's behavior. Thus we can check the component and proceed as if we had checked the original containment directly.

7.1.4 Applying decomposition

The key to applying decomposition is the decomposition function of Definition 30. Given a set of computer behaviors, we must find the set of consistent processor behaviors by "factoring out" the behavior of the memory system. Each aspect of the computer behavior will be due to either the processor or the memory. Thus, the first step in decomposition is identifying the aspects of the computer behaviors due to the memory. These aspects then can be replaced by the their interaction with the processor.

These interactions will not necessarily be unique. For example, a specification that two memory locations will be read does not imply either sequencing of the "read" operations. In the specification of Hector in Appendix B, a "hint" was added to each statement of the contents of a memory location. This hint was used to determine the timing of the associated memory operation.

The scope of this work precludes full treatment of decomposition. This area is ripe for future work.

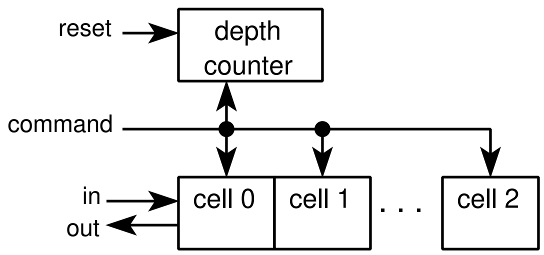


Figure 7.3: Stack, with "dummy" depth counter

7.1.5 Other applications

Decomposition can be applied in other contexts besides breaking computers into processors and memory systems. For example, it can be used to verify systems where the specification is not in reduced form, and it can be used to verify systems that require an adaptive reset.

Non-reduced specification

Recall the stack circuit which we have been using as an example, and the stack specification. The stack circuit maintains state for each location in the stack (i.e., the top of the stack in the first cell, the next location in the next cell, etc.). However, it does not maintain any indication of the current depth of the stack—i.e., the number of elements that it is presently holding. Instead, when this stack circuit performs an operation, it makes every data transfer that would be necessary in the worst case (a full stack). If no indication of whether the stack is full or empty is needed, this circuit is a fine implementation of a stack—it avoids maintaining unnecessary state.

However, the stack specification we have been studying maintains a depth counter. For this stack circuit, the depth counter is an unnecessary state component. (There are other stack circuit implementations, such as a stationary-data RAM-and-pointer implementation [15], which need to maintain this counter.) This poses a problem for verification: the Checking Theorem (page 135) requires that the state mapping be distinct. Yet the state mapping must map the abstract specification state—which includes a depth component—onto the concrete realization state—which does not include it. Such a mapping cannot be distinct, since it maps a larger set onto a smaller one.

Fortunately, we can instead map the specification state onto the state of an augmented circuit, as shown in Figure 7.3, which does include a depth counter, and then apply decomposition to factor out the depth counter. By doing this, we verify that the stack with a depth counter implements the specification, but since

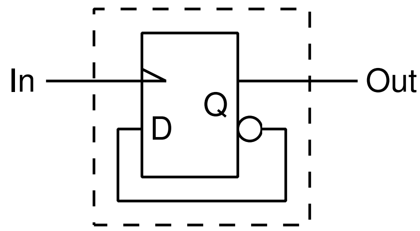


Figure 7.4: Circuit requiring adaptive reset. To clear the latch, it must be toggled if it is not already clear.

the depth counter has no outputs (the dummy cannot speak), omitting it from our actual circuit cannot possibly affect the circuit's behavior. Hence the stack without a depth counter implements our specification.

It is instructive to consider why this approach remains valid despite the requirement that, in general, the mapping must be distinct. The states of the stack without a depth counter can be thought of as representing equivalence classes of states of the stack with a depth counter. In contrast, in our example illustrating the necessity of distinction (Figure 6.3), the state space lacked such structure.

It is interesting to note that a similar addition of a depth counter would be necessary in order to verify this stack circuit using abstraction techniques, i.e., mappings in the opposite direction from ours, such as Bose and Fisher [27]. In such approaches, circuit state is represented symbolically, and an abstraction function is defined to map circuit state onto abstract system state. Obviously it would be necessary to maintain the depth as part of the circuit state in order to abstract from it. Thus, it would be necessary to actually construct a model of this dummy circuit in order to verify the actual stack. In contrast, our approach does not require actual construction of the dummy.

Adaptive reset

We can also use decomposition to deal with circuits which have an adaptive reset (or homing sequence) rather than an ordinary (or "oblivious") reset signal. A system has an adaptive reset if the procedure for resetting it varies, depending on its current state. For example, to reset a stack which signals whether it is empty, we could perform 0 to k "pop" operations, where k is the depth of the stack, where the precise number of operations is determined by stopping when the stack becomes empty. A much simpler example is a 1-bit counter (toggle circuit), which can be reset to 0 by applying a count input if it already stores a 1, and doing noting if it already stores a 0. Figure 7.4 shows such a circuit built from an edge-triggered D-latch.

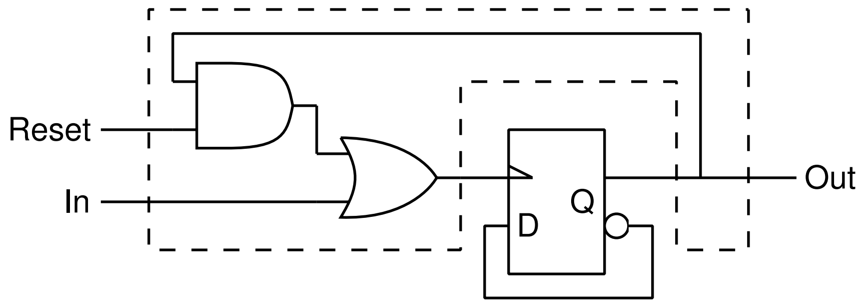


Figure 7.5: Circuit composed with reset circuit. Saying that a circuit requires an adaptive reset is actually saying that its environment will behave in a certain way.

Such a circuit cannot be verified by our methodology alone, for we cannot simply map the abstract input “reset” into the set of all possible circuit reset sequences. The circuit cannot be reset by using any arbitrary reset sequence. It must be reset by choosing the proper reset sequence. What we would like is a mapping that somehow chooses the proper sequence by peeking at the circuit's state. However, such a function would not be an input mapping—it would be something else, since it has a dependency on the circuit's state.

The problem is that we do not want simply to specify a mapping—we also want to describe some behavior—how the environment in which the circuit will be used is going to act. It is this composition—the circuit plus part of its environment—that we wish to specify.

Figure 7.5 shows a 1-bit counter with an adaptive reset circuit. We can verify the counter by decomposing the system and factoring out the reset circuit, namely, the part enclosed in the dashed line.

7.2 Representational issues

The interesting checks that can be made of a specification—obedience (containment), conformity, and distinction—all require the representation of sets of marked strings of the realization. The representation of sets of marked strings can be built on a representation of sets of states by adding a temporal aspect. A set of marked strings of states can be represented as one or more marked string of sets of states. Usually, one marked string suffices.

7.2.1 Representation of state subspaces

We now develop a representation for sets of states. During this development we will evaluate the representation with respect to several properties: expressiveness, compactness, and the ease of applying operators, checking membership, and checking the subset relation.

The notion of representation is a familiar, fundamental idea. One set can represent another if there is a bijection between the two.

Partial-order representation

Our first candidate is a partially ordered representation. It is based on the standard bit-vector representation, extended to a partially-ordered ternary domain. Let the set of all states be $S = \{ s_1, s_2, \ldots, s_n \}$ where $n = 2^k$. Each of these states an be represented by an element of $\mathcal{B}^k$ where $\mathcal{B} = \{ 0, 1 \}$. In other words, we can represent individual states by using bit vectors, as usual.

By introducing a partial order on the bits we can represent sets of states. More precisely, we can represent some of the subsets of S by using elements of $\mathcal{T}^k$ where $\mathcal{T} = \{ 0, 1, X \}$ as follows. Define a reflexive relation $\sqsubseteq$ as $X \sqsubseteq 0$ and $X \sqsubseteq 1$. Pointwise, this induces a partial order on $\mathcal{T}^k$. Then $t \in \mathcal{T}^k$ represents $\{ s \in S \mid t \sqsubseteq s \}$. (We know that this set can be defined because $\mathcal{B}^k \subseteq \mathcal{T}^k$.)

Such a representation cannot represent all possible sets of states. (There are 2^{2^k} sets but only 3^k representatives.) It cannot even represent one particularly important set: the empty set. This deficiency can be remedied with an augmented version. If we let T denote $\mathcal{T}^k$, we can use as our set of representatives the set T^+, defined to be $T \cup \{ \top \}$ where we have defined $t \sqsubseteq \top$ to hold for every t.

Such a partially-ordered representation alone is not highly expressive. With it, we can only represent certain sets. If we think of states as bit strings from $\mathcal{B}^k$, we can represent only convex sets. If we think of the state space as a space, we can only represent subcubes of the space. An augmented version can represent the empty set, but there are still many sets that cannot be represented. Thus, this representation has a significant shortcoming.

However, it also has several attractive properties. First, it is quite compact. Each (representable) subset can be represented in about the same space as is needed to represent a single state. Second, if we wish to find the image of a set under some operator defined on states, we can often find this image easily—provided we need a conservative answer rather than an exact one. If the operator is expressed as a vector of Boolean functions, we need only a conservative ternary extension of the operator. Applying the ternary extension to a ternary vector (which represents a set) will yield a new ternary vector representing another set. This new set is not necessarily the exact image of the original set, but it will contain the desired image.

Third, it is easy to check the subset relation between two sets represented in this way: simply check the ordering relation, pointwise, on the representations. Since singletons and elements have identical representations, this also yields a fast set-membership test.

Since the expressiveness of this representation is lacking, we continue to examine alternatives.

Image representation

Another way to represent a set is as the codomain of a function. (Coudert [84] was the first to recognize this as an efficient method of analyzing sequential systems using BDDs.) If we augment the state set S with an element $\top$ to yield $S^+ = S \cup \{ \top \}$, then we can represent any subset of S as the codomain of a *total* function. A function $f \colon D \to S^+$ for any domain D, which is not necessarily a surjection onto S^+, represents its codomain $U \subseteq S^+$. If we let D be the set of valuations for a set of variables V, then $f(V)$ is a symbolic expression. Furthermore, $f(V)$ represents a subset of S, namely $\{s \mid \exists d \in D \centerdot s = f(d) \}$. (It will soon be useful to note that the logical notation $\lambda u P$ is closely related to the more familiar set-builder notation $\{ u \mid P \}$.)

Representing a set as an image (in particular, as a symbolic expression) is more expressive than the partially-ordered representation. Provided that we augment the set S to yield S^+, we can represent any set, including $\emptyset$. The compactness of this representation depends on the technique used to represent functions or expressions. It is easy to apply operators, provided that they are given as manipulation of symbolic expressions. In fact, this forward image computation is precisely what most people call "symbolic simulation." However, it is not straightforward to check membership or subset relations.

Combined representations

A combined representation is more useful, for it builds on the strengths of its ancestors. If we combine the first two representations, we obtain a symbolic partially-ordered representation. Thus, we represent each subset of S with some $f \colon D \to T^+$. Then the symbolic representation $f(V)$ represents the set $\{ s \mid \exists d \in D \centerdot s \sqsupseteq f(d) \}$. If we wish, we can write this set as a logical predicate $\lambda s(\exists V \centerdot s \sqsupseteq f(V))$. It is often convenient to abbreviate this as $f(V)$, where the ordering test, the quantifier, and the lambda-binding are implicit.

A symbolic, partially-ordered representation inherits the advantages of its ancestors. It is expressive, since it can represent every subset. It is also potentially compact: sets which can be represented purely in the partially-ordered representation are represented as constant functions. It is easy to apply operators, provided that they can be formulated as symbolic versions of ternary extensions. Finally, we can check membership by evaluating the logical predicate given above. This predicate involves only the ordering check, symbolic application of f, and quantification.

However, there is no straightforward check for the subset relation. There does exist a check which is sound. Given sets $e(W)$ and $f(V)$, we can evaluate the predicate $\forall w \exists v (e(w) \sqsupseteq f(v))$. Provided only that $\sqsubseteq$ is a transitive relation, if this predicate is true, then so is the predicate $\forall x \exists w (x \sqsupseteq e(w)) \supset \exists v (x \sqsupseteq f(v))$, which expresses the subset relation.[1]

Finally, a symbolic, partially-ordered representation has one additional advantage. So far, we have let the quantifier in the logical predicate be implicit. If, instead, we make this quantifier explicit, then we allow the possibility of including in the representation variables which are free (i.e., not quantified). This will allow us to reason about symbolic functions (of the free variables) whose values are sets of states. This will become important when we consider the checking algorithm, trajectory evaluation.

For these reasons, we will use a symbolic, partially-ordered representation for state subspaces. The fundamental objects that we will manipulate will then be vectors of ternary-valued functions of Boolean variables. These objects represent either state subspaces (if all variables are bound by existential quantifiers) or symbolic functions whose values are state subspaces (if there are also free variables).

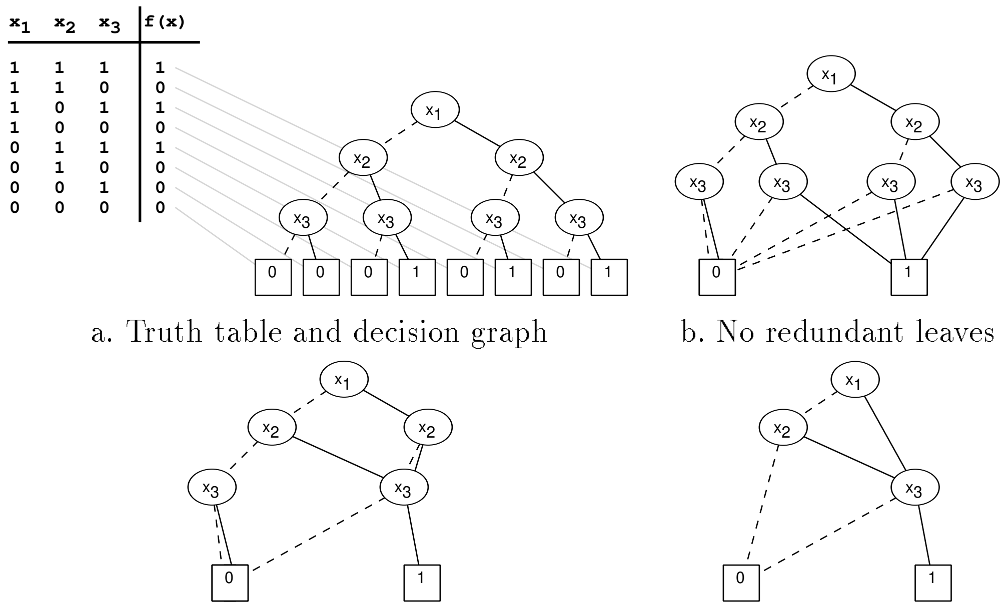
Sparse representation One final algorithmic detail is worth acknowledging. The representation we have chosen is based on vectors of symbolic ternary functions. The nature of this representation combined with the state sets we will be manipulating will result in most sets being represented by vectors which contain many instances of the function $\mathbf{X}$ (i.e., the constant symbolic function whose value is always the ternary X). It will be advantageous to let vector elements have this value implicitly, and note the actual value only when it differs. Thus, we will use a sparse vector representation, where $\mathbf{X}$ is the default value.

7.2.2 BDDs: binary-decision diagrams

Since the state representation we have adopted is based on representing functions, we need a representation for functions. A BDD (binary-decision diagram, or more properly a reduced, ordered binary-decision diagram) is a representation of a Boolean function. A few properties of BDDs are useful in understanding the implementation of the verification methodology.

A BDD is a directed acyclic decision graph obeying certain constraints, which has been *reduced*, i.e., all redundancy has been removed. A BDD can be con-

[1] We have not been able to determine whether the structure of $\sqsubseteq$ implies that this check is actually complete.



a. Truth table and decision graph b. No redundant leaves

c. No redundant nodes d. No redundant tests

Figure 7.6: Construction of a BDD from a truth table. a. Correspondence between truth table and decision tree. Just as in a truth table, the variables in a BDD appear in a fixed order. b. Redundant leaves eliminated. c. Redundant interior nodes eliminated. d. Redundant tests eliminated, yielding final, reduced graph.

structed from a truth table by representing the truth table as a decision tree, and then reducing it, i.e., removing all redundancy from the tree.

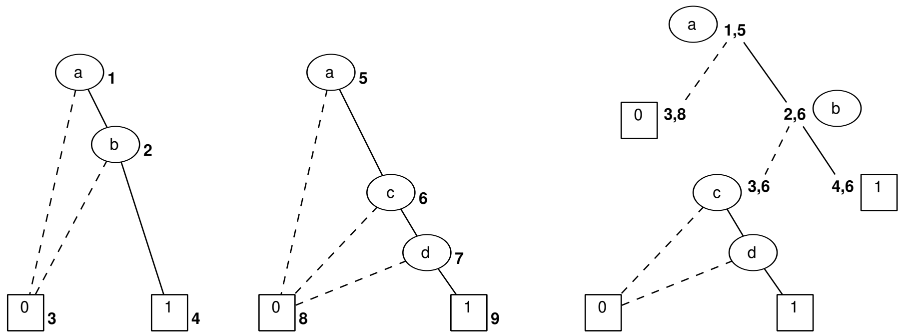
Each vertex of the graph represents a distinct function, and every Boolean function can be represented as a BDD. Since for k variables there are 2^k distinct Boolean functions, this immediately implies that a BDD may have size exponential in the number of variables.

BDDs are often a good representation for Boolean functions arising in practice.

An illustration of the construction of a BDD from a truth table is shown in Figure 7.6. The figure provides a good illustration of the properties that often lead to the compactness of a BDD. In practice BDDs are constructed directly, rather than from truth tables.

BDDs are not always compact. For certain functions such as integer multiplication [50] their size will be exponential in the number of Boolean variables. However, BDDs are often good for functions that actually arise. For any fixed variable order, BDDs are a canonical form.

The graph structure of a BDD reflects the Shannon expansion of the function that it represents. Thus, recursive algorithms that manipulate Boolean functions



a. First operand b. Second operand c. Recursive AND operation

Figure 7.7: Example of a the Boolean AND operation applied to two BDDs. The operation recursively visits pairs of vertices in the operands, beginning with their roots, and recurring on the "highest" child. Vertices are numbered to illustrate the structure of the recursion on the Shannon expansion. The result is not always in reduced form (although in this case it is).

via their expansions can be written as graph algorithms that traverse BDDs. For example, the calculation of the Boolean AND of two functions is illustrated in Figure 7.7. Some algorithms that construct BDDs do not yield reduced graphs directly, but they can be followed by a postprocessing reduction step.

Two important enhancements to the basic BDD improve its effectiveness. First, if all BDDs within a system are represented as a single shared graph, BDDs are a strong canonical form: testing Boolean equivalence then requires only unit time. Figure 7.8 illustrates a small shared BDD graph which represents the operands and the result of the operation that was illustrated in Figure 7.7. Such a BDD implementation is an ideal representation for many convergent algorithms, from fixed-point calculations to event-driven simulation, that require frequent testing for convergence.

The second important enhancement adds an edge attribute to the graph. Ordinary edges continue to have their old meaning, while negated edges indicate that the vertex upon which the edge is incident should be treated as representing the Boolean negation of the function that it normally represents. Although negation edges can reduce the size of a BDD by at most a factor of two, this can be a significant advantage in a system that often maintains both a function and its negation—for example, when encoding a ternary-valued function by a pair of Boolean functions.

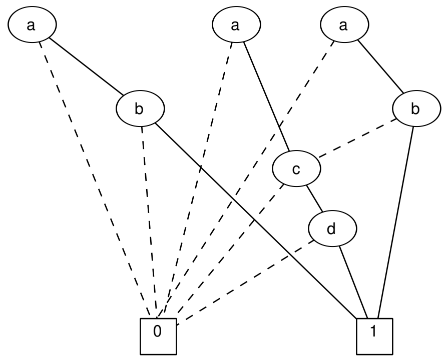


Figure 7.8: Shared BDD structure, showing the operands and result of the operation illustrated in the previous figure.

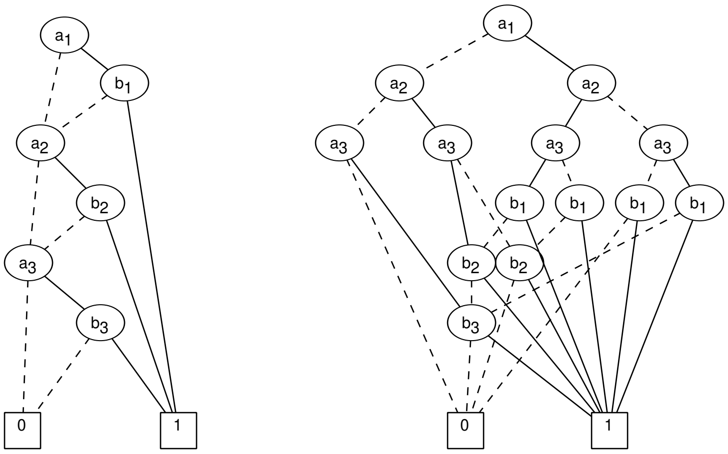
Variable ordering

The order in which the Boolean variables appear in a BDD sometimes has an extremely strong effect on the size of the BDD. For example, Figure 7.9 shows one Boolean function represented with two different variable ordering. In applications that represent Boolean functions constructed from words of bits, the best variable orderings are those which are interleaved, as in Figure 7.9a. We will make additional remarks on variable ordering later.

Ternary encoding

There is one inconsistency between the symbolic partially-ordered state representation we discussed at the beginning of this section, and the BDD representation just discussed: BDDs represent symbolic binary functions, but we need to represent ternary functions. We will encode each ternary value as a pair of bits. If we encode the ternary value 1 with the pair 10, the ternary value 0 with the pair 01, and the ternary value X with the pair 11. A ternary-valued function will then be encoded as a pair of binary-valued functions. For example, the binary-valued function $f(V)$, considered as a ternary-valued function, will be encoded as $f(V)\,\overline{f(V)}$.

Thus, we have finally arrived at a representation for state sets. A state set will be represented by a sparse vectors. The vector elements will be symbolic ternary functions of Boolean variables. The ternary functions will actually be encoded as pairs of binary-decision diagrams. (If we must also represent the empty set, we need only slightly more machinery, e.g., one additional BDD indicating whether or not the set is empty.)



a. Good variable ordering   b. Bad ordering

Figure 7.9: Good and bad BDD variable orders for an OR-of-AND function. The function is 1 if both bits in any corresponding pair from two 3-bit words are both 1. a. Interleaved order. Corresponding bits of different words are adjacent in the order. b. Separate order. All bits of the first word appear in the order before any bits of the second word.

7.2.3 Symbolic indexing

It is clear that the preceding representation can be used to represent sets of states. The representation for some sets of states is almost obvious. For example, the empty set is represented by a symbolic function whose value is the constant $\top$, and the set of all states is represented by a vector whose value is the constant $XX \ldots X$. The set of states where the first state variable is 0 would be represented by the constant vector $0X \ldots X$. When we reason symbolically, the set of states where the first state variable is a would simply be the vector $aX \ldots X$.

If we consider the lower-level representation of the ternary values as pairs of bits, the representations are the vectors $11\ 11 \ldots 11$, $01\ 11 \ldots 11$, and $a\overline{a}\ 11 \ldots 11$, respectively.

It is instructive to examine this representation for more complex symbolic predicates. For this it helps to have a concrete example. Suppose that the system under consideration contains 4 bits of state, represented by the vector R, with the least-significant bit at the right. The universal predicate is represented by the vector of eight 1's, 11 11 11 11. The statement $R[0] = u$ is represented by the vector 11 11 11 $u\overline{u}$. In terms of the underlying binary decision diagram representation for Boolean functions, this can be illustrated as in Figure 7.10.

The statement $R[a] = u$, however, is not quite so straightforward. The first

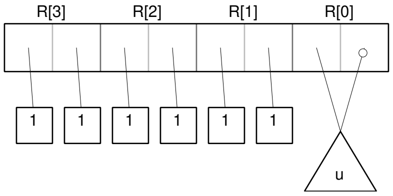


Figure 7.10: Representation of the predicate $R[0] = u$, which contains constant indexing. Unaddressed locations have the value pair 11 (the encoding for ternary X). The addressed location (location 0) has the symbolic value u. The triangle represents a BDD whose precise structure we are not interested in. The bubble indicates a negation edge.

question to ask is what it really means. Although it is clear what is meant to say that a particular location given by a constant index has some value, it is not obvious what is meant to say that a general location given by a symbolic index has the same value. Reflection reveals, however, that what is intended is a conjunction of conditional predicates. Thus, $R[a] = u$ actually refers to $\bigwedge_i(\textbf{if } a = i \textbf{ then } R[i] = u)$ where i ranges over the locations in array R, and i is a particular constant within each instance of the conditional expression. In other words,

$$(R[a] = u) \;\equiv\; \begin{cases} R[0] = u, & \text{if } a = 0 \\ R[1] = u, & \text{if } a = 1 \\ R[2] = u, & \text{if } a = 2 \\ R[3] = u, & \text{if } a = 3 \end{cases}$$

It is rather cumbersome to write out the Boolean functions of the encoding of such a predicate. However, they can be represented clearly in the diagram of Figure 7.11 if we duplicate the terminal 1-vertex to avoid graphical clutter.

Variable-ordering aspects

Examining the illustration of symbolic indexing leads to a very important point: a distinction must be made between control variables and data variables. Control variables, such as the variables encoding a in the examples, should appear early in the variable order. Symbolic indexing effectively constructs a decoder for the values of these variables. Data variables, such as u, should appear late in the order. Note that in the BDD representation for the symbolic indexing expression $R[a] = u$, the data variable u appeared only once. If the data variable u had instead appeared at the top of the ordering, it would have occurred multiple times. This is of little consequence in this particular illustration, because the data value u is a

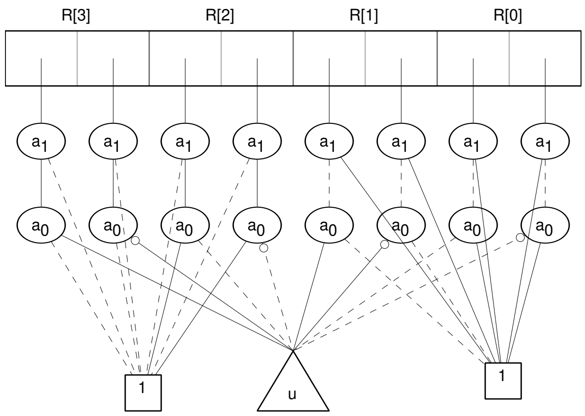


Figure 7.11: Representation of the predicate $R[a] = u$, which contains symbolic indexing. Unlike the representation in Figure 7.10, here the addressed location is given symbolically. The upper part of the BDD representation consists of a decoder structure that identifies the proper location. Unaddressed locations have the ternary X value, and the addressed location has symbolic value u. (The triangle represents a BDD whose precise structure we are not interested in.)

simple expression. However, if it had been a large, complicated one, the difference in space would have been significant.

7.3 Elements of a mapping language

We have previously discussed how to prove implementation. This required a mapping that took configurations of the specification and mapped them onto 2-marked strings of configurations of the circuits. Here we discuss the requirements for expressing such mappings from abstract machines to concrete ones. Fortunately, we already have most of the machinery necessary for such a mapping language: our specification language SMAL, from Chapter 3. A mapping language needs only a few additions to SMAL, to map abstract state onto circuit state.

7.3.1 Requirements of a mapping language

The first need of a mapping language is to identify circuit state. For this we introduce a third kind of variable, which we will call a node name. A node name (i.e., the textual element in the mapping language) denotes either a circuit node (i.e., an element of the set N_R in a switch-level circuit) or a vector or two-dimensional arrays of circuit nodes.

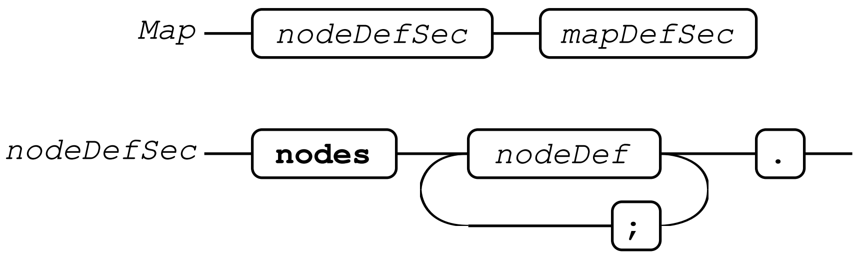


Figure 7.12: Highest level of syntax of a mapping language. The `nodeDefSec` is the node-definition section. The `mapDefSec` is the map-definition section. A `nodeDef` is a node definition.

The next need is to talk about sequences of circuit operation. For this we introduce a next-time temporal operator. However, the next-time operator alone requires frequent nesting, which is cumbersome in practice, so instead we use a parameterized form where the parameter denotes the number of iterations, i.e., the level of nesting.

Our formalism for representing sequences consists of marked strings, and for that we must introduce marks. Marks can be introduced implicitly by having next-time and previous-time operators, and introducing positive, integer durations for operations. Then the "start" mark is implicitly at time 0, and the "next" mark is implicitly at the time when the duration has elapsed. Finally, the mapping language must allow definitions, to be instantiated whenever the matching abstract primitives occur in a specification. It is with these definitions that the mappings are actually defined.

Adding these simple extensions to SMAL allows it to function as a mapping language. We simply need the syntax and semantics of the extensions.

7.3.2 Syntax of CAMP: a circuit assertion mapping language for pipelines

This section defines the syntax of a mapping language, and should be interpreted as an extension of the SMAL language syntax defined in Chapter 3.

A mapping consists of a node definition section followed by a map definition section. A node definition section consists of a series of node definitions, as shown in Figure 7.12. Each of the node definitions, as shown in Figure 7.13, contains a new identifier which is the name being defined. It also contains a string structure—either a string, or a vector of strings, or a rectangular array of strings. The string structure names the circuit nodes that the node definition is to denote. In this way,

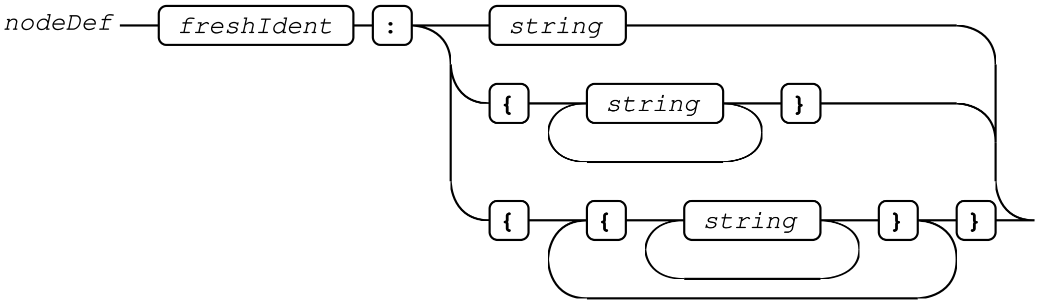


Figure 7.13: Syntax of node definitions. A `freshIdent` is a "fresh" identifier, i.e., one not already in use.

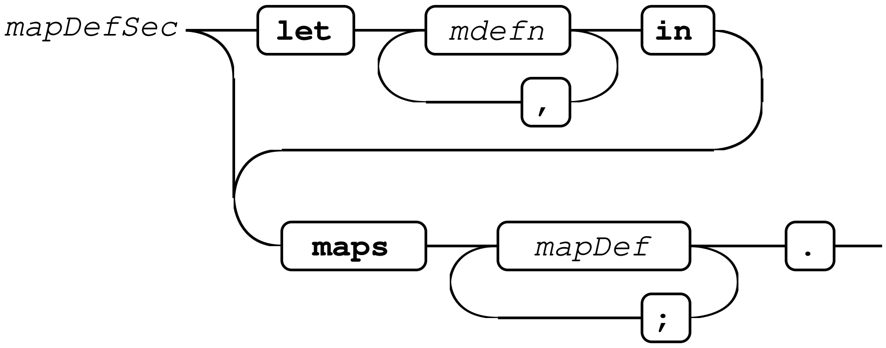


Figure 7.14: Syntax of mapping definition section. A `mdefn` is an auxiliary definition. A `mapDef` is a map definition.

the set of identifiers used in the specification language, and the lexical structure of this language, can be decoupled from details of the CAD system with which the circuit is defined.[2]

A map definition section consists of a series of map definitions, as shown in Figure 7.14. Optionally, a set of auxiliary definitions may be defined and used in the map definition section. An auxiliary definition may be either an auxiliary definition allowed in SMAL, or one which takes parameters denoting nodes, as shown in Figure 7.15.

[2]It would be tedious and error-prone to actually write this portion of the mapping for a large array of circuit nodes, such as a register file. It is intended that this portion would be mechanically generated by some other program. This delegates the problem of string manipulation to some other language, instead of incorporating string manipulation into CAMP.

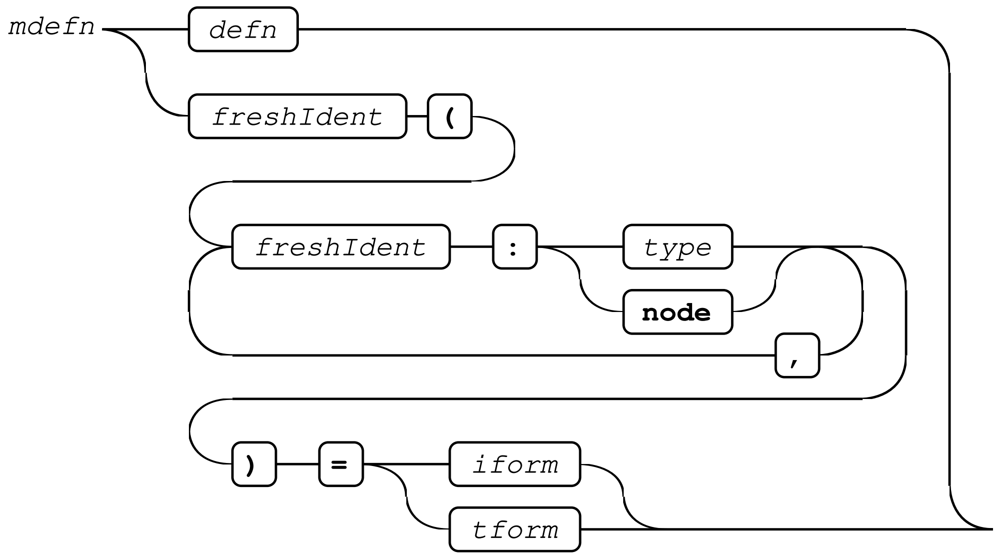


Figure 7.15: Syntax of auxiliary definitions. A `defn` is a definition, from SMAL. An `iform` is an instantaneous formula. A `tform` is a temporal formula.

Figure 7.16 gives the syntax of map definitions. Each map definition may also have its own auxiliary definitions. A map definition must be given for each system variable defined in the SMAL specification. A map definition consists of a header and a body. The body is simply a temporal formula. The header takes one of two forms, depending on whether the system variable being mapped is a scalar or an array. Each syntax is similar to the syntax for using the respective kind of system variable when constructing a primitive formula in SMAL. Thus, the header for the mapping of a scalar system variable binds one parameter, while the header for the mapping of an array binds two parameters: an index parameter and a value parameter.

A temporal formula is constructed from instantaneous formulas. We present instantaneous formulas first, in Figure 7.17. An instantaneous formula can take several forms. There are two primitive forms. It can assert that a scalar node variable has a value. Alternatively, it can assert that an indexed location within a array node variable has a value.

There are also several combining forms: conjunction, disjunction, the instantiation of an instantaneous formula abbreviation that was previously defined using a local definition, an existentially quantified instantaneous formula, or a case analysis.

Instantaneous formulas are used to construct temporal formulas, which also can take several forms, as shown in Figure 7.18. They can be created by combin-

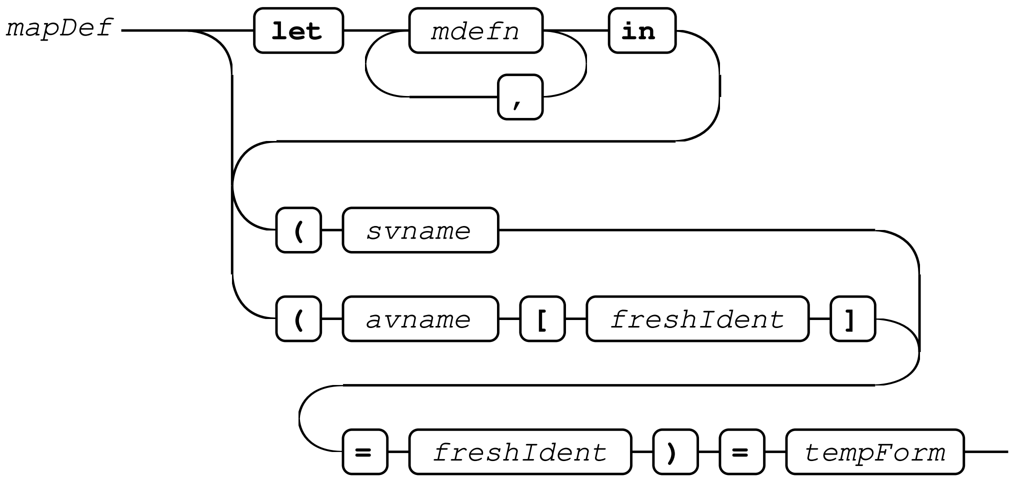


Figure 7.16: Syntax of map definitions. A `svname` is a scalar variable name. An `avname` is an array variable name. A `tempForm` is a temporal formula.

ing other temporal formulas using conjunction or disjunction, by instantiating a temporal abbreviation, or by an existential quantification. The primitive way of creating a temporal formula is to assign a time (or range of times) to an instantaneous formula. Establishing the duration of an operation by introducing a marker also creates a temporal formula. Finally, temporal formulas can be constructed by case analysis.

Example Figure 7.19 gives an example of a mapping expressed in this notation. The abstract latch specification is in terms of an abstract operation determined by the state variable "op," an abstract input "D," and an abstract state value "Q." The mapping expresses the signal values on circuit nodes, over time, that correspond to values of each of these abstract state variables. Note that the choice of operation dictates the duration of the operation, i.e., a load operation takes longer than a hold operation. (This is not a requirement of this particular circuit, but it illustrates a capability of the methodology.)

7.3.3 Semantics of the mapping language CAMP

We can define the semantics of the mapping language such that a program in the language denotes a mapping from the states (or configurations) of a specification machine onto sets of marked strings of circuit states. For brevity, here we give the semantics of only an essential subset of the mapping language.

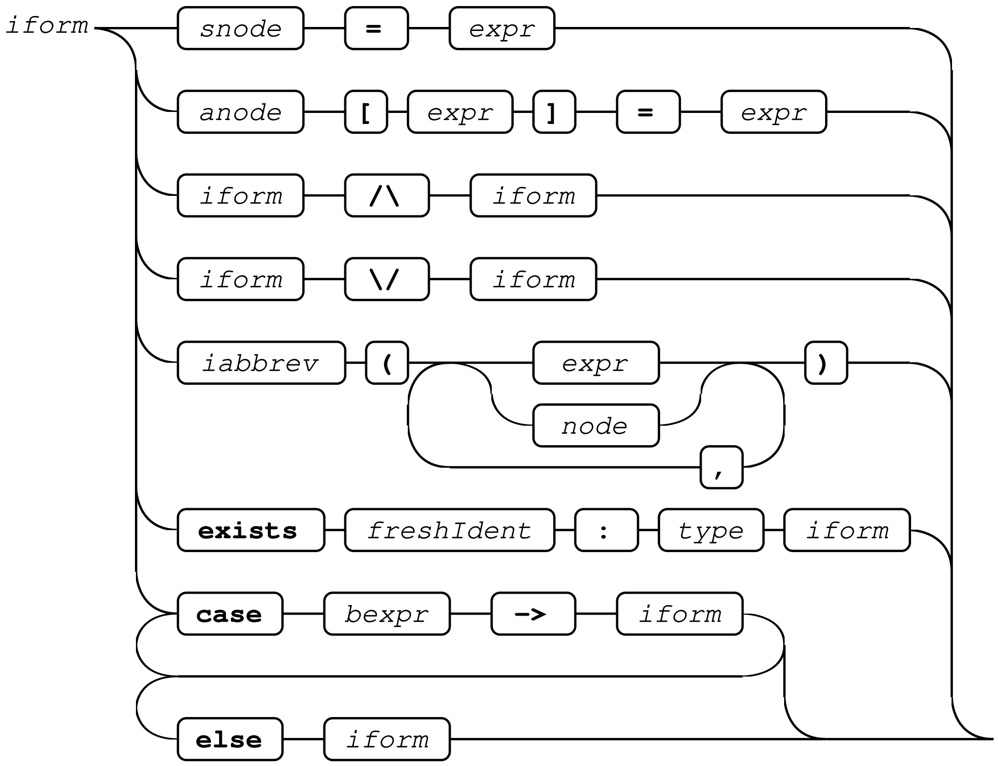


Figure 7.17: Syntax of instantaneous formulas. A `snode` is a scalar node name. A `expr` is an expression. An `anode` is an array node name. An `iform` is an instantaneous formula. An `iabbrev` is an instantaneous abbreviation, i.e., one defined as in Figure 7.15. A `bexpr` is a Boolean expression.

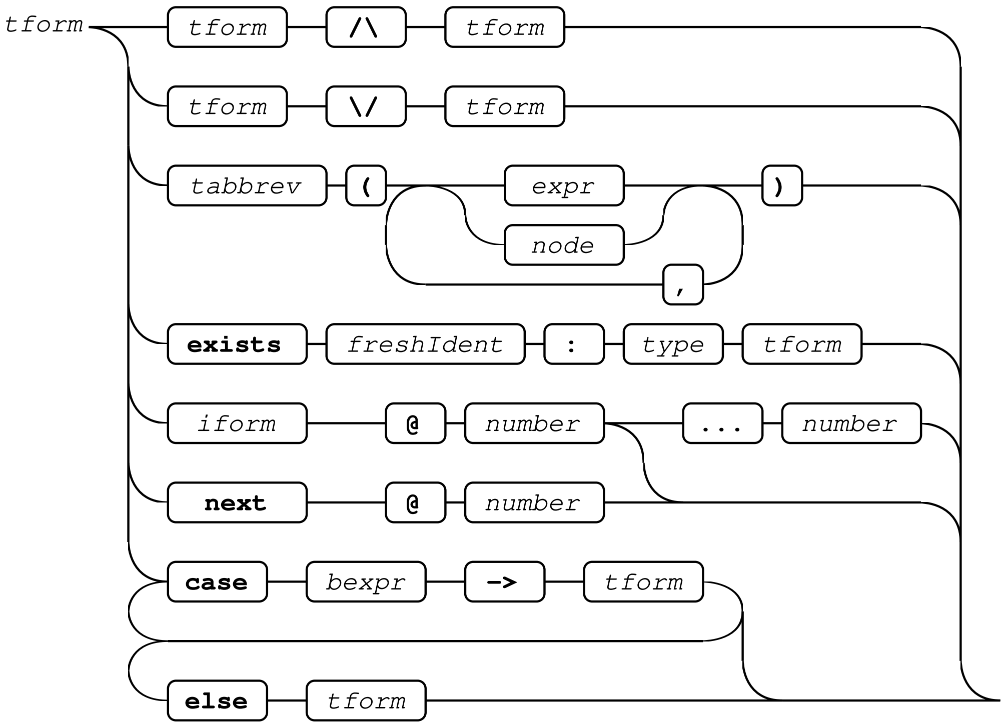


Figure 7.18: Syntax of temporal formulas. A `tabbrev` is a temporal abbreviation, defined as in Figure 7.15.

```
nodes
  nL: "L";     // latch signal ("clock")
  nD: "D";     // input
  nS: "S";     // storage capacitor
  nQ: "Q" .    // output

maps
  ( op = o )
  =  case o = load ->
       nL = 0 @ -1...0 /\
       nL = 1 @  1...2 /\
       nL = 0 @  3...4 /\
       next @ 4
     else ->
       nL = 0 @ -1...2 /\
       next @ 2
   ;
   ( D = b )
   = nD = b @ 2...3
   ;
   ( Q = b )
   = (nS = b /\ nQ = b) @ 0
   .
```

Figure 7.19: Mapping of the latch specification of Figure 2.8 onto the circuit of Figure 2.1, using the timing of Figure 2.2.

Notation We will represent the set of nodes in the circuit by a set W of wires (for convenience, they can be strings, though their only essential character is that they be distinct elements). We will also use Γ to denote the set $\{0,1\}^{|W|}$, the set of possible *circuit* states or configurations. The set Θ will denote the set of functions $\{\theta : W \rightarrow \{0,1\}\}$ which assign values to the wires. Note that Γ and Θ are isomorphic. That is, a circuit state assigns a bit to every wire. As in the definition of SMAL, E will represent an environment. Also as in SMAL, ψ will represent a valuation of the specification's system variables—i.e., a specification configuration or specification state. An environment E will map the name of a node aggregate to either a vector of wires or an array of wires.

Node definitions A string structure denotes either a vector or array of strings. (In turn, the strings name circuit nodes.) An aggregate of circuit nodes is assigned a name by a "node definition." The semantics of a node definition are given by the following equation.

$$[[name \;:\; strStruc]](E, W) = (E\colon name \mapsto [[strStruc]], W \cup \mathrm{elts}([[strStruc]]))$$

Here the function "elts" produces a set of the elements of a string structure. In other words, this equation says that a node definition augments the current environment[3] by mapping the new name to the structure it is to represent, and it augments the set of wires with the new wires.

A sequence of node definitions works as expected: the last node definition is evaluated in the environment given by evaluating the earlier ones first.

$$[[nodeDefs \;;\; nodeDef]](E, W) = [[nodeDef]]([[nodeDefs]](E, W))$$

The entire node definition section is evaluated relative to the current environment. It evaluates the node definitions, beginning with the empty set of wires.

$$[[\texttt{node}\;\; nodeDefs]](E) = [[nodeDefs]](E, \emptyset)$$

Temporal formulas A temporal formula denotes a set of marked strings over circuit state, and is evaluated relative to an environment, a set of wires, and a parameter assignment ϕ. (A parameter assignment is analogous to a case assignment in the semantics of the assertion language, for which we also used the letter ϕ.) Temporal formulas are constructed by two means. First, a temporal formula can be constructed by assigning an integer (a time) to an instantaneous formula.

$$[[iform@t]](E, w, \phi) = \begin{cases} \{r \in \Gamma'^{t} \mid r = {}'s \text{ and } s_t \in [[iform]](E, W, \phi)\}, & \text{if } t \geq 0 \\ \{r \in \Gamma'^{|t|} \mid r = s' \text{ and } s_0 \in [[iform]](E, W, \phi)\}, & \text{if } t < 0 \end{cases}$$

[3]CAMP mappings are intended to follow SMAL specifications. The initial environment E is the one "left over" from the semantics of the SMAL specification.

Here s is some string in Γ^t, and s_i denotes the i-th symbol in s. That is, such a temporal formula restricts the allowable configuration only at a particular time. Furthermore $\Gamma = \{0,1\}^{|W|}$ and Γ'^n is the set of marked strings $\{ r \in \Gamma'^* \mid |r| = n \}$

Second, a temporal formula which denotes a marker can be constructed.

$$[[\texttt{next@}t]](E, W, \phi) = \{ r \in \Gamma^{1/t} \mid \exists s \in \Gamma^t \,\mathbf{.}\, r = s' \}$$

Here Γ^t is the set of strings of length t, and $\Gamma^{1't}$ is the set of 1-marked strings of length t (discounting the marker). This form is used to assign a duration to an operation. Note that this is defined only for $t \geq 0$. (An operation can't have a negative duration.)

Temporal formulas can be combined, for example by conjunction, when both formulas represent sets of 2-marked strings or both represent sets of 1-marked strings.[4]

$$\begin{aligned}[[tform_1/\backslash tform_2]](E, W, \phi) = & \,\mathrm{ext}^{\Gamma}_{||[[tform_1]](E,W,\phi)||}([[tform_2]](E, W, \phi)) \\ & \bigcap \mathrm{ext}^{\Gamma}_{||[[tform_2]](E,W,\phi)||}([[tform_1]](E, W, \phi))\end{aligned}$$

Here "ext" is a function which extends a set of marked strings (see definition 13, p. 59). The notation $||x||$ refers to the measurements of marked string x (see definitions 11 and 13 for details).

Case restriction can also be applied to temporal formulas.

$$[[\texttt{case}\ bexpr\ -> \ tform]] = \begin{cases} \mathrm{ext}^{\Gamma}_{||[[tform]](E,W,\phi)||}(\ '\ ), & \text{if } [[bexpr]](\phi) = 0 \\ [[tform]](E, w, \phi), & \text{if } [[bexpr]](\phi) = 1 \end{cases}$$

Instantaneous formulas The basic constituent of of a temporal formula is the instantaneous formula, which denotes a set of circuit configurations. Recall that there are two kinds of node aggregates: vectors and arrays. A primitive instantaneous formula contains a node aggregate (i.e., a vector or array of wires) and one or two expressions. The meaning of an instantaneous formula involving a vector of wires is defined to be the set of circuit configurations in which the values on the nodes in the vector encoding the integer denoted by the expression.

$$\begin{aligned}&[[wvect = expr]](E, W, \phi) \\ &\quad = \left\{ \theta\colon W \to B \mid \left(\sum_{j<|E(wvect)|} 2^j \cdot \theta(\Pi_j(E(wvect))) \right) = [[expr]](\phi) \right\}\end{aligned}$$

Here Π_j denotes a projection function which extracts the j-th wire from the vector.

[4] When one formula represents a set of 1-marked strings and the other represents a set of 2-marked strings, the set intersection is replaced by overlapped concatenation (extended to sets), with the 1-marked strings on the left.

The meaning of an instantaneous formula involving an array of wires is defined similarly.

$$[[warray[index] = val]](E, W, \phi)$$
$$= \left\{ \left(\sum_{j<E(warray)} 2^j \cdot \theta(\Pi_{i,[[index]](\phi)}(E(warray))) \right) = [[val]](\phi) \right\}$$

Here the projection function $\Pi_{j,k}$ selects the element in column j and row k of the array.

Case restriction can also be applied to instantaneous formulas.

$$[[\texttt{case}\ bexpr\ ->\ \ iform]](E, W, \phi) = \begin{cases} \Gamma, & \text{if } [[bexpr]](\phi) = 0 \\ [[iform]](E, W, \phi), & \text{if } [[bexpr]](\phi) = 1 \end{cases}$$

Mappings Mappings map configurations of the specification to sets of marked strings of the circuit. We have already defined temporal formulas so that they denote sets of marked strings of the circuit. We define the mappings in terms of the system variables of the specification. Recall from the semantics of the specification language that there are two types of system variables: scalars, and arrays. Recall also that a configuration of the specification is represented as an assignment ψ to the system variables of that specification.

We will define the meaning of a mapping involving a scalar system variable relative to an environment E and a set of wires W. We let it denote the function that maps a specification state ψ to the set of marked strings given by the denotation of the temporal formula, evaluated in an environment including an assignment ϕ which assigns, to the parameter, the value of the state variable according to the specification state ψ. The formalism is actually easier to read:

$$[[(svname = param) = tform]](E, W) =$$
$$\lambda\psi\,([[tform]](E, W, \phi\colon param \mapsto \psi(svname))$$

A mapping involving a system variable that denotes an array is slightly more complicated. Recall that an array is a vector of scalars. Thus, a mapping involving an array must in some way involve all of those scalars. We will define the meaning of such a mapping to be a function that maps a specification state ψ to the set of marked strings given by intersecting the set given for each of the scalars i. The set given by each scalar i is given by evaluating the meaning of the temporal formula with a particular assignment. It assigns, to the parameter, the value given by selecting that scalar out of the array. It also assigns, to the index, the value of i.

$$[[(avname[index] = param) = tform]](E, W)$$
$$= \lambda\psi \left(\bigcap_{i \in |E(avname)|} [[tform]]\,(E, W, \phi\colon param \mapsto \Pi_i\psi(avname)\colon index \mapsto i) \right)$$

The meaning of a sequence of mappings is defined to be the mapping that produces the intersection of the sets that the individual maps produce.

$$[[maps\ ;\ map]](E,W) = \lambda\psi\,(([[maps]](E,W)(\psi)) \cap ([[map]](E,W)(\psi)))$$

The entire mappings section simply denotes what the sequence of maps denotes.

$$[[\texttt{maps}\ maps]](E,W) = [[maps]](E,W)$$

Thus, it denotes a function mapping specification states to 2-marked strings of circuit states.

Distinction of mappings As discussed earlier, mappings must be distinct. We have already observed that there are two possible approaches to establish distinction: to check that entire mappings are distinct, or to guarantee that they are distinct by their construction.

Distinction is not a property of great concern, because in practice most mappings are distinct. This is reflected in the mapping language. Simple mappings (those that simply assign values, unchanged, to circuit nodes) are distinct. Mappings involving injective functions over domain values are distinct. The conjunction of two distinct mapping that refer to disjoint sets of circuit nodes is distinct. Symbolic indexing, since it can be expanded into a series of such conjunctions, is distinct. There are only a few forms, such as case analysis, that lead to mappings that are not distinct. Thus it might be possible to show that mappings are distinct by checking only a few parts of the mappings.

On the other hand, it might be more straightforward to check distinction of entire mappings. Let I be the mapping we wish to check i, and a_1 and a_2 be disjoint sets of variables representing two abstract states. Given symbolic representations of circuit state, such as those we have discussed, we could form a symbolic Boolean function representing the equivalence $I(a_1) \approx I(a_2)$. If this function is equal to the Boolean function representing the equivalence $a_1 \equiv a_2$, then the mapping I is distinct.

Developing and evaluating ways of checking or otherwise ensuring distinction of mappings seem like a straightforward direction for future work.

We can now express specifications and mappings. Once we fill the remaining requirement—for a tool implementing the methodology—we can apply the methodology to verify circuits.

7.4 Trajectory evaluation

The previous sections described a representation for sets of marked strings, and discussed how to map abstract specifications onto such representations. As might be suspected, the representation of state sets as symbolic partially-ordered vectors was not an arbitrary choice. Instead, this representation was chosen to enable us to check assertions using trajectory evaluation.

It is not our intention to present the comprehensive foundations of trajectory evaluation. However, some understanding of the checking algorithm will be useful in evaluating the forthcoming case study.

Symbolic ternary trajectory evaluation, or simply trajectory evaluation, is an algorithm for checking a symbolic, operational model against an implicative assertion in a restricted next-time temporal logic [56, 212]. The restricted form of the logic is an important factor in the checking algorithm. Thus, it is first necessary to understanding this logic. As in any logic, we will have syntactic elements including combining forms, primitives, and variables to inhabit the primitives. Statements in the logic have abstract syntax given by the first column of the following table.

F (syntax)	$\models F$ (validity)	a_F (defining value)	OK_F (validity function)
$N = 0$	node N has value 0	$XX \ldots 0 \ldots X$	1
$N = 1$	node N has value 1	$XX \ldots 1 \ldots X$	1
$b \to F$	$b = 0$ or $\models F$	$\begin{cases} a_F, & \text{if } b \\ X, \ldots X, & \text{otw.} \end{cases}$	$\overline{b} \vee OK_F$
$F_1 \wedge F_2$	$\models F_1$ and $\models F_2$	$a_{F_1} \sqcup a_{F_2}$	$OK_{F_1} \wedge OK_{F_2} \wedge (a_F \neq \top)$
$\mathbf{X}F$	if $\models F$ after next state transition	$XX \ldots X; a_F$	OK_F
$F_A \Rightarrow F_C$	–	–	–

In these expressions, N denotes a node of the circuit, b is any Boolean expression over an implicit set of Boolean variables, and F (possibly subscripted) is any formula. The first five forms define formulas. The last form defines an assertion. An assertion is a special object.

The basic element of the specification logic is the primitive formula (the first two lines of the table). A primitive formula simply says that some specified state variable (i.e., a circuit node) has a particular specified value (either 1 or 0). For example, if a circuit has a wire called "in," the formula `in=0` is a primitive formula. In terms of our representation for sets of states, which appear in our table as defining values, a primitive formula is represented by a vector which contains the ternary X value in all locations save one: the location corresponding to the node that occurs in the formula. This location contains the value that occurs in the formula.

More interesting formulas are constructed from primitive formulas using three combining forms: case restriction, conjunction, and the temporal operator. The first of the combining forms is case restriction. Case restriction simply says that a formula need hold only in certain cases. A case is a valuation for the Boolean variables in the formula, which are called the case variables. Note that case restriction causes the validity function OK_F to tend toward 1, and introduces a case analysis into the defining value.[5]

Conjunction has the expected meaning. The representation of conjunction is calculated as follows. Recall that the representation for a set of states is defined in terms of a partial order. The representation of a conjunction is computed by taking the pointwise join[6], with respect to the partial order, of the representation of the conjuncts. The validity function reflects whether the result of the join is $\top$, the representation of the empty set.

The temporal operator expresses the passage of time. A formula f is a statement about the present time—the present state of a system. A formula involving the temporal operator, $\mathbf{X}f$, is a statement about the future—the next state. Repeatedly applying the temporal operator allows the statement of conditions about times further into the future. Combining repeated temporal operators and conjunction allows the statement of conditions about sequences of states. For example, the statement $f_0 \wedge \mathbf{X}(f_1 \wedge (\mathbf{X}f_2))$ holds if f_0 holds presently, and f_1 will hold at the next state in the future, and f_2 will hold in the state following that.

Formulas without the temporal operator are instantaneous formulas. Each instantaneous formula has a single defining value which is a symbolic function. A defining value uses the representation defined in section 7.2 to compactly represent the set of circuit states satisfying an instantaneous formula.

Any formula can be written in normal form, where it consists of a series of instantaneous formulas: one for the present state, another for the next state, and so on until the maximum nesting of the temporal operator is exhausted. Thus, a formula such as $f_0 \wedge \mathbf{X}(f_1 \wedge (\mathbf{X}f_2))$ can be represented as a sequence of defining values. a_0, a_1, a_2. They in turn represent a sequence of sets of states. By a shift of perspective, this can be thought of as a set of sequences of states, or a set of paths in the state space of the system.

Trajectory evaluation is a way of checking an assertion. An assertion is a pair of temporal formulas: an antecedent A and a consequent C. (These were written F_A and F_C respectively for consistency within the table above.) Each of the formulas expresses a set of paths through the state space of a system. An assertion states that if the system follows a path in the antecedent, then it follows a path in the consequent.

The trajectory evaluation algorithm makes use of symbolic simulation and the sequence of defining values that corresponds to a formula in normal form. The normal form of the formula, in the context of an assertion, is

$$A_0 \wedge (\mathbf{X}A_1 \wedge (\ldots \wedge A_n)\ldots) \quad \Rightarrow C_0 \wedge (\mathbf{X}C_1 \wedge (\ldots \wedge C_n)\ldots)$$

[5] Of course, this case analysis is represented symbolically.

[6] Recall that in order to represent the empty set, we augmented the set with an element $\top$. This ensures that the join is always defined.

The trajectory evaluator performs a symbolic simulation of the circuit, initializing all simulated system state to the ternary X value—the representation of all possible binary states. It iterates through the instants of time $i = 0, 1, \ldots, n$ corresponding to the assertion. At each instant i, it applies the instant's antecedent's defining value, a_{A_i} as a constraint on the symbolic simulation state, by using the join operation. Meanwhile, it checks that the state remains in accord with the instant's consequent's defining value, a_{C_i}, by testing the partial order. As the evaluator proceeds, it maintains a Boolean function OK that indicates the cases (i.e., assignments to the Boolean variables) in which the circuit's behavior agrees with that specified by the assertion.

This Boolean function is constructed by maintaining three other functions: OK_A, OK_T, and OK_C. As the verifier performs symbolic simulation and attempts to apply the constraints determined by the assertion's antecedent, it may find that it cannot apply the constraints because they are inconsistent with circuit state.

For example, if some node is driven to the value 0 because of circuit operation, and the antecedent asserts that the node has the symbolic value v, the constraint can be applied only in the case that the variable v is assigned the value 0. The function OK_T records such cases—the cases in which the constraints actually can be applied. Those cases in which the constraints cannot be applied are termed antecedent failure cases. The function OK_A records the conjunction of the validity functions OK_{A_i} over all instants i. The function OK_C records the cases in which the checks determined by the assertion's consequent are actually met. The final Boolean function OK is found from the equation $OK = \overline{(OK_A \wedge OK_T)} \vee OK_C$. That is, the final Boolean function represents an implication. It is the ultimate result of trajectory evaluation: if it is the constant function **1** then the circuit satisfies the assertion for all cases.

In addition, as the verifier proceeds through the assertion, whenever it finds a case where the antecedent cannot be applied (such as when v has the value 1 in our example), it performs an implicit case restriction, so that it simulates only the valid cases. This is because, when the antecedent cannot be applied (i.e., in the antecedent failure cases), the function OK_T will have the value 0, i.e., the function OK must have the value 1. The circuit need not be simulated any longer for such cases—the result of the trajectory evaluation procedure has already been determined. Consequently, an implicit, global case restriction is performed, using OK_A as the guard.

As we will see, this automatic global case restriction has important implications for the efficiency of the trajectory evaluation process.

It also has important implications for the reliability of verification. When the antecedent fails, the assertion succeeds. This is fine if the condition that led to the antecedent failure does not correspond to expected circuit operation. For example, if an asssertion's antecedent states $R[a] = u \wedge R[b] = v$ the antecedent will fail in the case $a = b$ but $u \neq v$. This is fine, since it reflects a condition that would never actually arise: one register holding two different values simultaneously. This antecedent failure is simply an artifact of the way that the specification is stated.

On the other hand, suppose that a node s that is supposed to store state has actually been connected to ground. An assertion's antecedent of the form $s = v$ would then fail in the case $v = 1$. Since the antecedent fails, the assertion succeeds. The statement "if the circuit stores a one, then..." is vacuously true because the circuit never stores a one on the grounded node.

In order to avoid this problem, we can require that the antecedent never fail in the circuit when it would not fail at the abstract level. This allows antecedent failure of the first type while avoiding that of the second.

It is easy to see that it is always possible to avoid antecedent failure. One need only take the Boolean negation of the failing conditions and apply it as a case restriction. If adopt this as a convention, we can avoid antecedent failure entirely, at the cost of clouding the specification with more case restriction.

7.4.1 Efficiency, global case restriction, and antecedent failure

A global case restriction is one which is applied to an entire assertion (to both the antecedent and the consequent). Global case restriction can be thought of as limiting the values that the case variables in the specification are allowed to take on. In contrast, the effect of global case restriction on the simulated circuit state can be seen from the defining value for case restriction in the table from the last section. In the cases that are "thrown out" by the case restriction, the defining value is X—the undefined or universal system state—and in the remaining cases, the defining value is unchanged. This distinction is maintained symbolically. Thus, the representation for a state involving a case restriction must "test" the Boolean variables in the set of support[7] for this guard, for every bit of state in the system that is not X.

Since these Boolean variables appear in every state bit in the circuit, they have an important effect on the efficiency of the symbolic simulation during trajectory evaluation. First, if there are many such variables, and every state bit must test them, the BDDs can "blow up," i.e., require too much space. Second, if they are at the top of the ordering, every symbolic manipulation must traverse the vertices that test them. This will slow the simulation down.

Unfortunately, antecedent failure interacts particularly poorly with symbolic indexing. Consider the example of symbolic indexing given earlier, $R[a] = u$. Suppose that the register file R is a component of a system which can perform

[7]The set of support for a Boolean function $f(V)$ over a set of variables V is the smallest subset of V whose valuation must be known in order to determine the value of f for any arbitrary valuation. In other words, it consists of the variables that f actually depends upon.

operations on two register values. An assertion describing such an operation will be of the form

$$R[a] = u \wedge R[b] = v \stackrel{\delta}{\Rightarrow} R[c] = f(u, v)$$

where a and b are source addresses, u and v are operand values, c is a destination address, and $f(a, b)$ is the result of the operation. Such an assertion will cause antecedent failure for the case where $a = b$ but $u \neq v$. All of the variables in a, b, u, and v will participate in the antecedent failure. Consequently, all of these variables will appear in practically every bit of state in the system.

The variables u and v can be eliminated from the antecedent failure by rewriting the assertion using case restriction, as follows.

$$R[a] = u \wedge ((a \neq b) \rightarrow (R[b] = v)) \stackrel{\delta}{\Rightarrow} R[c] = f\left(u, \begin{cases} v, & \text{if } a \neq b \\ u, & \text{otw.} \end{cases}\right)$$

This eliminates the antecedent failure, at the cost of introducing a case restriction involving a and b. This has the disadvantage of drawing efficiency issues into the high-level specification—precisely the place where we would like to be as abstract as possible—but at least it does help reduce the size of the BDDs.

However, the remaining variables, those for a and b, do appear in the case restriction. Because they are addresses used in symbolic indexing, they should ideally appear at the top of the variable ordering. But because they appear in the case restriction, they should ideally appear at the bottom of the variable ordering. Of course, they cannot occur twice, so one or the other must be chosen. In practice, they work better at the top of the ordering.

7.5 Verification tool

Having discussed the trajectory evaluation algorithm, it is appropriate to make some remarks on a tool implementing the algorithm and verification methodology.

7.5.1 Usability

A verification tool, or verifier, must encompass several essential features. First, it must correctly implement trajectory evaluation. Second, it must provide a symbolic notation for writing specifications and mappings, and a way to apply the mappings to the specification to yield simulation patterns.

The original prototype of the methodology obtained each of the essential features from a separate program. The original implementation of trajectory evaluation was written as a small set of modifications to the Cosmos symbolic simulator. This allowed the simulator to read formulas in the primitive temporal language and check circuits against them.

The original implementation of the specification notation and mappings was written in an object-oriented Lisp dialect called T [219]. This program took high-level specifications and applied mappings to them, expanding them into the primitive form accepted by the trajectory evaluator. It was quickly discovered that the this approach was not ideal. It could be workable when given a good specification and a correct circuit to check. However, when the circuit and the specification failed to agree, debugging was difficult. It was particularly cumbersome to relate a failure of trajectory evaluation back to the corresponding part of the original specification. This made it evident that attention to debugging would be needed to apply the methodology to large circuits.

A revised verifier addressed this issue. It was constructed as a single program, based on a combination of the Cosmos symbolic simulator and a Scheme [8] implementation. The implementation of the verifier was kept flexible and extensible, reusing existing code from the original prototype when possible, and using Scheme representations unless efficiency dictated otherwise.

The specification language was embedded in Scheme. The runtime binding of Scheme made it easy to modify specifications when debugging them. In contrast, the static binding of a traditional compiled language, or an interpreted language like ML [199], would have been less convenient.

Although much of the underlying verification engine, comprising trajectory evaluation and symbolic simulation, is compiled, the user interface interpreted, for easy extensibility. The basic design of its user interface is an interactive postfix language. It is oriented toward the immediate execution of commands, with a limited but useful iteration facility.

The result of running the verifier is an indication, for each assertion, of whether the circuit implements the state transition specified by the assertion. If the circuit fails to meet the specification, the checker provides the Boolean function *OK* indicating the failing conditions.

This Boolean function is the starting point for debugging. The first step of debugging is to find a falsifying case, using a verifier command provided for this purpose. A falsifying case is an assignment to the Boolean variables under which *OK* is 0. The falsifying case can be used as a restriction on symbolic system state. This simplifies the appearance of symbolic values in the simulated circuit. If further simplification is desired, the falsifying assignment can be augmented by assigning additional values to additional variables.

Once a failing assignment that gives a satisfactorily simple appearance to system state has been found, the structure of the circuit can be explored and the values on nodes examined until the cause of the failure is determined. Failures, of course, may be caused by specification errors or circuit errors, or they may be false negatives due to the conservatism of the simulation model. The stack-based com-

[8] Scheme is a dialect of Lisp.

mand language makes it easy to traverse the circuit, and mark interesting areas by pushing them onto the stack, so that they can be later revisited and explored more fully.

Alternatively, Boolean or vector expressions in the specification language can be evaluated under the current restriction, to explore the specification.

Much of the implementation of the verifier is coded in Scheme. So as to retain easy access to the facilities of the Cosmos simulation system, the Scheme system used is the `Scheme->C` system [12], an implementation that adheres to the Scheme programming language whenever possible, but compiles into C code which is then processed by the operating system's native compiler. This requires a few deviations from the Scheme standard with respect to tail recursion, but the language is otherwise complete Scheme. This allowed a portable implementation of a complex symbolic computation system, and compatibility with the existing Cosmos switch-level simulation system.

The advantages of doing programming in a language that provides an interpretive execution environment, particularly for exploratory research programming, are well known. Another advantage of programming in a dialect of Lisp is automatic garbage collection. Since `Scheme->C` generates C code, it employs a garbage collector that allows ambiguous roots. In practice this worked well.

Wentworth [242] discusses some of the drawbacks of such a garbage collector. The chief one is of inadvertent capture: if the collector scans a pointer which appears to point to a large data structure in the heap, but which is actually a value instead of a pointer, the large data structure must be retained in memory.

This capture phenomenon was not observed in the verifier. The verifier is structured so that various portions of the Cosmos simulator are called as subroutines, but these routines very seldom make calls to Scheme code. The garbage collector will only be invoked while Scheme code is executing. Thus, the garbage collector is likely to be invoked when the runtime stack contains only Scheme data, so that most bit patterns that look like pointers into the heap will in fact be pointers into the heap.[9]

7.5.2 Visualization aids

Writing the text of a high-level SMAL-style specification is relatively straightforward. Writing the CAMP-style mapping is more difficult, due to the need to express temporal behavior. Mappings can be most easily written by first sketching out timing diagrams and then writing the formal notation. The verifier provides commands for drawing timing diagrams from mapped assertions. Visually comparing diagrams generated by the verifier against the original sketches can be helpful

[9]Moreover, the MIPS C compiler on the DECstation computer we used for our experiments has a strong register allocator, so the values held in registers are likely to be useful current values, rather than bit patterns left over from long ago.

in locating inconsistencies. Several of the figures in the case study section, for example Figure 9.3, are direct illustrations of this visualization facility.

In addition, the verifier can be instructed to display a timing diagram as it performs the trajectory evaluation, one unit step at a time. If a restriction has been established in order to debug a circuit and specification, the diagram displayed during trajectory evaluation is simplified according to the restriction. Such diagrams can be quite useful during debugging. They illustrate many aspects of circuit operation. For example, Figure 9.8 shows how, during an ALU operation, the less-significant bits of the result bus resolve to binary values before the more-significant bits do.

7.5.3 BDDs and efficiency

A useful visual representation for a BDD is its *profile*. A profile is a histogram indicating the number of BDD nodes that test each Boolean variable.

The verifier draws profiles in X windows as histograms with horizontal bars centered horizontally, yielding symmetrical displays somewhat like the ink blots of a Rorschach test. If a Boolean function is represented efficiently by a BDD, its profile will be thin—often it will look like a vertical chain of beads. This typically indicates that at each bit position, each Boolean variable is tested only a few times. In contrast, a poor ordering yields a BDD with a fat profile ("neckties," "birds," or "bats"). This would indicate that many nodes tested the variables in the fat area.

The ability to view profiles of BDDs is helpful in tuning the performance of verification. Suppose that arbitrary binary values must be asserted onto some control lines (e.g., a microcode address register) early during a sequence, and then at some later point a particular set of binary value must be applied to those lines (e.g., a particular microcode entry point). Should the Boolean variables representing the arbitrary Boolean value appear at the top or the bottom of the variable order? Suppose they are assigned to the beginning (top) of the order. After verifying some aspect of system operation, the BDD space requirements would be rather high. Examining profiles of BDDs that should be dissimilar would reveal that they share a common element at the top of the ordering: every BDD in the system must test the Boolean variables mentioned earlier, to see if they correspond to a microstate whose successor is the state represented by the binary values of the microcode entry point. In contrast, placing the variables at the end (bottom) of the order results in two terminal sub-DAGs. If the variables were not present, the other BDDs would terminate in 0 or 1. With them present, the other BDDs instead terminate in one or the other of the sub-DAGs.

Instead of introducing a test on the Boolean variable for every BDD in the system, most of the BDDs in the system will share a single test.

For example, the change in ordering discussed above was motivated by a quick glance at a few profiles while verifying a "store flags" instruction of the Hector microprocessor. It reduced the number of BDD nodes by a factor of 6, and the total BDD storage requirements, BDD cache sizes, and CPU time required by a factor of 4. As explained earlier in section 7.4, the verifier performs best when the Boolean variables associated with antecedent failure appear at the bottom of the variable order.

7.6 Related work

Bryant's survey of binary-decision diagrams [54] is recent and comprehensive, so we will not repeat such discussion here.

Coudert and colleagues [84] recognized the utility of representing sets by means of codomains of functions using BDDs, and developed algorithms for manipulating such representations within algorithms to compare state machines.

Symbolic indexing was introduced in [15] and has been generalized by Hu and colleagues [135]. An initial formulation of trajectory evaluation [57] has been used previously to verify stacks, memories, and simple pipelines including a data path and an accumulator [47]. Both pipelines bypassed the read-after-write hazard. A more complete formulation is in progress [212]. The present work is the first application of trajectory evaluation to a large pre-existing circuit.

7.7 Chapter summary

This chapter has discussed several issues related to the application of the methodology. First, it discussed decomposition, a way by which a computer specification could be written but a processor verified. This was accomplished by defining a decomposition function parameterized by the memory system, and showing that if the processor's behavior was allowed by the result of applying the decomposition to the specification, then the behavior of a computer constructed from the processor and a memory system would be allowed by the specification.

Next the chapter turned to representational issues. It discussed how to represent sets circuit states using functions onto partially ordered domains. Then it showed how this could be used to represent sets of marked strings of circuit configurations. After discussing how to represent marked strings, the chapter discussed how to express them, by defining the abstract syntax and essential semantics of a language for mapping assertions onto circuits.

Finally, it turned to the implementation of the verification itself. It outlined the trajectory evaluation algorithm and discussed some practical concerns in implementing a useful verifier.

We have now discussed a methodology for verification and some of the issues regarding its implementation. It is now time to apply the methodology to verify a circuit.

Part III

Case study

Chapter 8

Hector specification

This chapter begins the final part of the thesis, a case study of a microprocessor called Hector. It first discusses the traditional way to describe a processor. Then it describes the Hector microprocessor, informally. Finally, it discusses the formal description of Hector found in Appendix B.

8.1 Traditional specification of an instruction set

Traditionally, the instruction set of a microprocessor is specified in a programmer's reference manual.

8.1.1 A typical programmer's reference manual

The manual for the Motorola 68000 family [189] is representative; the discussion below is based on it. A programmer's reference manual begins with an introduction categorizing the instruction set by instruction category, describing the operands of the instructions, and explaining the notation that appears in later chapters. Next comes a series of chapters describing instructions from several categories. Finally, a series of charts or tables summarize the instruction formats, give the instruction set as a whole, and provide cross-references.

The introduction sketches instruction format, describes the constituents of instructions (such as register specifiers and addressing modes), and divides the instruction set into categories. A table of notation gives descriptive names for mnemonic abbreviations (e.g., ISP: interrupt stack pointer). This list may not even be alphabetized. The intended function of each of the condition codes is given by a paragraph of text. Each instruction category is given in tabular form, with a very brief (one-line) description of the instruction's effect. The introduction is also likely to give examples of any "unconventional" instructions such as multiprocessor instructions, or instructions with remarkable side effects.[1]

The chapters containing instruction descriptions describe each instruction on its own page or set of pages, using a standard format. This usually includes the instruction's assembler mnemonic name and syntax, a one-line summary of its operation, the kinds of operands allowed, the effects of the instruction on the condition codes, and a textual description of the instruction's effect. For example, the ADD instruction of the 68000 family is described by the text:

> Adds the source operand to the destination operand using binary addition, and stores the result in the destination location. The size of the operation may be specified as byte, word, or long. The mode of the instruction indicates which operand is the source and which is the destination as well as the operand size.

Much remains tacit and is simply not specified in such a description. For example, the meaning of the "predecrement" addressing mode is not described. Even information which is critical to the definition of an instruction must sometimes be assumed by the reader (and then tested by writing and running small test programs).

For example, the "move from CCR" instruction takes the processor's condition code register (i.e., flags) and places them in a data word. However, the bit positions within this word are not defined. The user must assume that the "move to CCR" instruction is defined in a consistent way. In its description the bit positions are defined—but only because it is a convention in this manual to give the value that every condition code receives. Thus, it is *impossible* to write a correct nontrivial interrupt service routine without making assumptions.[2] This omission is not particularly compelling, but it occurs in a manual for a popular commercial product, in use for well over a decade.

It is no wonder that bugs in processor designs persist through the shipment of products. With an ambiguous specification, the ambiguous cases cannot be checked. Semantic games such as "the result is undefined" are of no help. They are little help to an engineer designing a system, but completely useless to the end-user of a system containing an embedded microcontroller.

On the other hand, vague specifications do have one (quite dubious) advantage: they are less likely to contain errors. As the maxim goes, "it is better to remain silent and be thought a fool than to speak and remove all doubt." In order to be able to provide reliable, precise specifications, there must be a precise way of checking that they are correct. This is the role of formal verification. In this

[1] For example, the 68000 family's "no-op" instruction actually flushes the integer pipeline.

[2] An interrupt service routine must leave the condition codes unchanged. Any nontrivial routine must accomplish this by saving and restoring them. Yet the effect of the instruction that is supposed to save the condition codes is not defined!

chapter we will describe the Hector microprocessor in a formal way, by writing a formal specification.

The need for checkable, precise processor specifications is apparent.

8.1.2 Instruction-set simulators

As the preceding discussion illustrated, a programmer's manual is a terrible foundation on which to build a processor. Fortunately, processors are not designed by first writing a manual describing an instruction set, and then implementing the instruction set from that vague specification.

The "specification" actually used in designing a system is usually a kind of simulator: a program, written in either a general-purpose programming language or a so-called *hardware description language* (HDL). An HDL simulator is at least a concrete standard. But such a simulator describes much more than an instruction set. It requires a very detailed description of how to implement the instruction set. After all, the compiler or interpreter for the HDL must actually be able to implement it. Thus, such a description does not actually simply specify an instruction set. Instead, it specifies a mechanism by which to execute the instruction set.

This has several drawbacks. For example, the simulation program is likely to contain much unnecessary sequencing, if the HDL is not a parallel language. Even if the HDL is a parallel language, the HDL interpreter may attempt to simulate this parallelism by nondeterministic sequencing, but this leaves no guarantee that observed behavior was specified behavior rather than an artifact of some particular choice that the simulator made. Even when the behavior observed from a simulator is unambiguously implied by the text of the simulation program, it still may be either an intended or an unintended effect of the simulation. The designer's tacit distinction between intended and unintended behavior is not apparent even by examining the text of the simulation program itself. Concealing the simulation program from customers is then important. It helps to prevent them from relying on unintended behavior which may be changed in future versions of the design. Moreover, simulators are kept secret because (since they are actually implementations) they contain implementation details. Any company would be foolish to reveal a solid piece of its design to its competitors.[3]

A specification structured as a set of assertions lacks these disadvantages. It lacks implementation details. Thus there is no need to conceal it from competitors. It can be structured so as to specify only intended behavior, and leave unintended behavior unspecified. Thus there is no need to conceal it from customers.

The remainder of this chapter will describe the Hector microprocessor, first in this traditional way, and then in a formal, assertional style (referring to Ap-

[3]Thus, companies exist whose business is simply to supply simulation models, since manufacturers themselves won't supply them. Of course, without access to the design itself, such an enterprise is even more likely to supply a model whose behavior differs from the actual system.

pendix B). This will show some advantage to specifying the instruction set as assertions. The real advantage, however, will be in the possibility of formal verification, which will be carried out in the following, penultimate chapter, Chapter 9.

8.2 Introduction to Hector

The Hector microprocessor is a 16-bit CISC fabricated in 1985. Its 2-address architecture is similar to the PDP-11, but with more (16) registers and fewer addressing modes. System state is held in the register file and a few condition code bits. The implementation is microcoded; at the microcode level it is slightly pipelined, but at the instruction set level it is not.[4] The bus interface is similar to the Motorola 6800. In addition to a reset line, there is a wait line, DMA, prioritized interrupts, and a single-step facility. Hector has no cache and does not support virtual memory.

The Hector processor architecture consists of a 16-bit ALU with condition codes, and a file of sixteen 16-bit registers, which include the program counter and the stack pointer. Other specialized values such as interrupt vectors are also kept in the register file. Altogether, 7 of the registers are completely general-purpose, and the other 9 sometimes have special use. Hector is a 2-address machine, similar to the PDP-11, but register addresses occupy 4 bits and addressing modes occupy only 2 bits. Figure 8.1 shows the architecture.

There are four addressing modes: register, indirect, indirect with post-increment, and indexed. Figure 8.2 shows their encoding.

Additional useful addressing modes can be synthesized since the stack pointer and program counter are among the addressable registers. Indirect off of the PC with post-increment yields immediate addressing. Clearing a register before indexing off of it yields absolute addressing. Indexing off of the stack pointer yields access to parameters of procedures. Indexing off of the program counter gives relative addressing.

However, this also results in several peculiar addressing modes. For example, Hector has a "push" instruction; applying it on the program counter overwrites the next instruction to be executed.

The instruction set includes data movement, clear, binary and unary arithmetic and logical operations, comparison, a test-and-branch, conditional branch and call, and a software interrupt instruction. Additionally, a push instruction treats any register as a stack pointer, and there are instructions to swap bytes within a word, and to load and store the condition codes. Separate instructions also set and clear the carry and interrupt-enable bits. Finally, there are instructions for searching and exchanging arrays of structured data. These instructions were added as an

[4]Actually, there is a very slight degree of pipelining: the processor senses the state of the interrupt lines as it completes execution of each instruction.

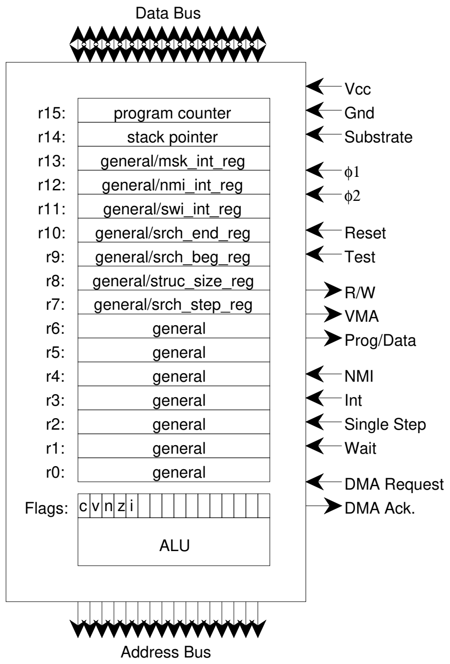


Figure 8.1: I/O pins and registers of the Hector microprocessor

Code	Src Mode (SM)	Dst Mode (DM)	Branch Mode (BM)
00	Register	Register	Register
01	Indirect	Indirect	Absolute
10	Indirect++	Indirect++	Relative
11	Indexed	Indexed	Indexed

Figure 8.2: Operand addressing modes of Hector

afterthought by Hector's designers. Figure 8.3 shows the instruction set and its encoding.

Op	SM	DM	Src	Dst	Instruction	Flags
0000	SM	DM	Src	Dst	Add Src,Dst	CVNZ
0001	SM	DM	Src	Dst	Addc Src,Dst	CVNZ
0010	SM	DM	Src	Dst	Sub Src,Dst	CVNZ
0011	SM	DM	Src	Dst	And Src,Dst	NZ
0100	SM	DM	Src	Dst	Subc Src,Dst	CVNZ
0101	SM	DM	Src	Dst	Or Src,Dst	NZ
0110	SM	DM	Src	Dst	Xor Src,Dst	NZ
1000	SM	00	Src	Src	Not Src	NZ
1000	SM	01	Src	Src	Neg Src	CVNZ
1000	SM	10	Src	Src	Inc Src	CVNZ
1000	SM	11	Src	Src	Dec Src	CVNZ
1001	SM	00	Src	Src	Shl Src	C NZ
1001	SM	01	Src	Src	Rol Src	C NZ
1001	SM	10	Src	Src	Shr Src	C NZ
1001	SM	11	Src	Src	Ror Src	C NZ
1010	SM	DM	Src	Dst	Cmp Src,Dst	CVNZ
1011	SM	DM	Src	Dst	Btst Src,Dst	NZ
1100	BM	00	Src	CC	Bra Src,CC	
1100	BM	01	Src	CC	Jsr Src,CC	
1100	SM	10	Src	Src	Swap Src	NZ
1100	SM	11	Src	Src	Clr Src	
1101	SM	DM	Src	Dst	Move Src,Dst	
1110	SM	00	Src	Src	Test Src	C
1110	DM	01	Dst	Dst	Stf Dst	
1110	SM	10	Src	Src	Ldf Src	CVNZI
1110	SM	11	Src	Dst	Push Src,Dst	
1111	00	00	—	—	Sec	C
1111	00	01	—	—	Clc	C
1111	00	10	—	—	Sei	I
1111	00	11	—	—	Cli	I
1111	01	00	—	—	Rti	
1111	01	01	—	—	Swi	
1111	01	10	Src	Dst	Exch Src,Dst	
1111	01	11	Src	CC	Srch Src,CC	CVNZ

Figure 8.3: Instruction set of Hector

An instruction-level simulator written in stylized C by Hector's designers details the instruction set in a formal notation. This served as a basis for the formal specification of the instruction set (Section 8.3.4). Systems built with the Hector microprocessor have been used in undergraduate labs at North Carolina State University [179]. Finally, the designers of Hector know that it contains logic bugs [178], though details of their nature have been deliberately withheld. Thus, an immediate test of the methodology is that I should detect these known bugs when attempting to verify the processor.

Hector seemed particularly suitable as a case study for verification for several reasons. It had not been designed with formal verification in mind. It was real, in the sense that working parts existed. It was large enough to exercise tools, without

```
clrnx()
{
        odr = alu(ZERO,odr,odr);
        t1 = mread(reg[15]);
        mar = alu(ADD,t1,reg[src]);
        memwrite(mar,odr);   reg[15] = alu(INCSRC,reg[15],reg[15]);
        cycles += 4L;
}
```

Figure 8.4: Fragment of the Hector simulator, implementing the "clear" instruction for indexed addressing. (The first word of instruction fetch has already been simulated when this fragment is reached.) First, the output data register is cleared to 0. Then a base address is fetched from the program. Next, an effective address is calculated. Finally, the 0 is written to memory, and the program counter is advanced past the instruction word containing the base address.

having all the complexity of a modern microprocessor. It was a microcoded design, was not heavily pipelined, and had no cache or cache controller. Finally, I knew one of Hector's designers well enough to expect to obtain the design if I asked.

Hector exhibits the basic features of any processor. It fetches instructions and operands from memory, executes instructions, stores results, and responds to several interrupt and control lines.

8.2.1 The Hector instruction-level simulator

In addition to laying out the chip, the designers of Hector wrote an instruction-level simulator, using the C programming language in a stylized way. This simulator was intended to provide a faithful model of the microprocessor design. This program consists of code to control the simulation, functions to implement the ALU operations, and a dispatcher which calls the proper instruction execution subroutine. There is such a subroutine for each instruction type. Each of these subroutines contains one line of code for each processor execution cycle needed for the instruction being modeled. Figure 8.4 shows a typical example, for the "clear" instruction with indexed addressing.

This instruction clears a memory location whose address is given by adding an index (stored in a register) to a base address (stored as part of the instruction). Its execution is as follows. In the first clock cycle following instruction fetch and decode, the ALU generates a zero word and places it in the output data register. In the next cycle, the memory location addressed by register 15 (the program counter) is read into a temporary register, yielding the base address. This is added to the contents of the source register to calculate the effective address. Finally, the zero that was calculated in the first cycle is written to memory, and the program counter is simultaneously incremented to advance beyond the base address in the instruction stream.

This instruction-level simulator can serve as the basis of a specification for Hector. It cannot serve as the specification itself. As discussed in Section 8.1.2, it is too low-level. While this does not prohibit the simulator from being used as a specification, it lacks the clarity of a more abstract specification. Many unnecessary sequencing constraints and internal details of the implementation appear in the simulator. Ideally, such details should not appear in the specification. Furthermore, the C-based description language lacks a formal semantics.

Oakley [194] considered the problem of deriving abstract specifications from simulations written in an HDL called ISP. Much of his effort was devoted to determining which of the paths through the control-flow graph of the HDL simulation could be analyzed symbolically, and which needed case analysis. While it is possible that his techniques could be applied to a form of the Hector instruction-level simulator, they are not directly applicable.

Although Hector's microcode was available, it was not suitable as a specification. The tools for which it had been developed were not available. Since it was processed by a highly configurable program, there was not even good definition of its format. Moreover, microcode is even more low-level than the HDL simulation, and the semantics of its language are even more loosely defined.

Thus, a suitable formal specification for Hector had to be developed from the existing descriptions of the processor. The instruction-level simulator was the most suitable basis with which to begin. Functions computed by instructions, and flag values, could be translated rather directly from the simulator. The overall structure of the simulator, too, could serve as a guide to the structure of the specification. Since in the simulator each instruction class and combination of addressing modes was described with its own simulation procedure, the formal specification contained an assertion for each. Developing these assertions required the translation of the action of each instruction-level simulation statement into a statement of effect. When statements did not interact, this was straightforward, but when they did, it required careful consideration of the sequence of actions.

Excerpts of the resulting formal specification appear in Appendix B.

8.3 Formal specification of Hector

This section discusses portions of the specification of Hector. The text of the specification itself appears in Appendix B. This discussion runs parallel to the appendix. Readers who are only somewhat interested in processor specification should read through this section. Those interested in delving into the details of

Hector might wish to study it more carefully, by making frequent references to the specification itself. Those tempted to skip this section might still want to compare the discussion of the formal specification of the indexed "clear" instruction (see "Clearing an array element" on p. 199) to the HDL description of Figure 8.4 discussed above.

8.3.1 Types and system variables

The specification begins by defining several data types. The first few types reflect the structure of the instruction encoding. Opcodes are divided into major and minor parts, which are 4-bit and 2-bit fields, respectively. The test performed by a branch is encoded by a 4-bit field. Addressing modes are encoded by 2-bit fields.

Hector has a 16-bit address space, so an address is a 16-bit quantity. (Address arithmetic sometimes requires computing a 17-bit value.) Registers are identified by 4-bit fields. Machine words are 16 bits wide.

Finally, the abstract control, i.e., the input that the processor receives from its external environment, takes one of several discrete values. The processor can be reset, it can be interrupted, or it can be run. Different instructions can run for different lengths of time, and this is reflected by several different "run" inputs.

The system's state space and input space are defined by a set of system variables, which follow the type definitions. The memory is an array of words, indexed by memory addresses. The register file is an array of words, indexed by register addresses. The condition codes are bits. The control input takes on the possible discrete values enumerated in the control type definition. One bit of abstract state information represents some circuit invariant information (such as the fact that some registers have definite logical values, rather than intermediate voltages). Finally, a flag indicates whether or not an interrupt is pending. These definitions take only a few lines.

8.3.2 Constants

About two pages of constant definitions follows the definition of the system variables. Placing them here allows them to follow the type definitions and precede most other definitions.

The first set of constants reflects the special function of several of Hector's registers. The program counter, stack pointer, and other special-purpose registers are members of the register file. Special-purpose registers include those holding the addresses of the interrupt-service routines for the maskable, nonmaskable, and software interrupts (as well as registers used to define arrays and records of structured data, for some specialized instructions which operate on them). Finally, the register with address 0 is sometimes read as a side-effect of other operations. This makes it necessary to refer to this register's value frequently in the specification.

The next set of constants defines the instruction encoding. This section is organized by major op-code. The binary arithmetic and Boolean operations, such as addition, subtraction, and bitwise logical "and," each have their own major op-code. Unary operations require fewer bits to encode their addressing modes, so these operations share two major codes and are distinguished by minor codes. Comparisons and tests have their own op-codes.

Branch and jump (i.e., call) instructions share a major op-code, and are distinguished by minor sub-codes. The condition code flags tested by the instruction are encoded by another instruction field. The conditional tests include a test which always succeeds, for unconditional transfers of control. Two more instructions, which swap bytes or clear words, also share this major op-code. Data movement has its own op-code. Two more major op-codes are used for several miscellaneous instructions, distinguished by sub-fields.

Finally, addressing modes are encodes in a 2-bit field. There are five addressing modes, but relative addressing is used only for branches, and post-increment is never used for branches. Register, indirect, and indexed modes round out the instruction set.

These constant definitions are made with the instruction set in mind, but they do not define the instruction set. They only give symbolic names to strings of bits, and these names are free from any meaning other than the bit string. It is only after the symbolic names are used in assertions and functions defining the actual operations that the instruction set will actually have been defined.

8.3.3 Auxiliary functions

Part of the task of specification is to define the functions that various instructions compute, or otherwise use. These definitions occupy about three pages.

The first few functions are associated with the binary-operation instructions. The functions and conditions codes computed by these instructions are given by separate definitions. Each of these definitions consists of a large case statement which selects one of several functions. The function selected depends on the value of the major op-code, which appears as a parameter to the function. Condition code values also appear as parameters, since some functions, such as add-with-carry, depend on condition code values.

The next few functions are associated with the unary-operation instructions. They are similar to those for the binary operations, except that there is only one operand, and there are more cases to consider due to the instruction count.

The last of the auxiliary functions takes a test code and and the condition-code values, and determines whether the test succeeds or fails.

Together, these functions reveal more about the functions computed by the instruction set. However, they say nothing about many important aspects, such as how operands are found, or what is done with the results. Such details are finally given in the set of assertions that define the instruction set.

8.3.4 Assertions

The assertions in a specification define the transition relation of a state machine. The interesting state transitions of a processor consist mainly of its instruction set, plus a few special operations such as interrupts and initialization. The following discussion runs strictly parallel with Appendix B.

Initialization

Before a processor begins to execute instructions, it must be initialized in some way. Initialization could be accomplished by designing the circuit so that, when power is applied, it starts in a particular, well-defined state. More often, however, applying power does not initialize a processor. Instead initialization occurs in response to some external signal.[5]

Initialization is specified by an assertion similar to[6] the following:

$$\begin{aligned} \text{control} = \text{reset} \stackrel{\delta}{\Rightarrow} \quad & \text{invariant} = 0 \\ & \wedge R[\text{PC}] = 0 \\ & \wedge R[\text{SP}] = 0 \\ & \wedge R[\text{INT}] = 4 \\ & \wedge R[\text{NMI}] = 2 \end{aligned}$$

It states that if the processor is given its reset signal, it will then enter a state where several conditions hold. First, after it is reset, the processor will be ready to execute instructions or respond to interrupts. This is reflected by the invariant condition.[7]

In addition to establishing invariant conditions, initialization of the processor also resets several registers to known values. These include the program counter, the stack pointer, and the pointers to the service routines for maskable and non-maskable interrupts. Finally, after the processor is initialized, no interrupt is pending.

[5] Of course, in a real computer this signal is often generated by the application of power.

[6] The actual syntax accepted by the prototype verifier appears in the appendix.

[7] Actually, in this specification the condition that is assumed by succeeding assertions is stronger than the condition actually proved by this assertion. The discrepancy is due to electrical effects that are not captured by the switch-level model. Additional assumptions are actually also manifest in the mapping function, to be discussed shortly (section 8.3.5). Obviously, such effects must be accounted for by some other means. It is an advantage of the approach to verification developed here that such assumptions are highlighted.

Interrupt

Initialization is a very important operation that any system must perform, but it is not usually a particularly interesting one to examine. This is because initialization necessarily does not depend on initial state. No matter what happens to a system, we should be able to reset it. On the other hand, response to other external stimuli is more complicated and more interesting, for the response to the stimulus depends on the state of the system when the stimulus arrives.

An interrupt is one such external stimulus. When the processor is interrupted, it should save certain component of its internal state in memory, and then begin executing an interrupt service routine.

The assertion describing the operation of Hector's non-maskable interrupt begins by declaring several case variables. Recall from Chapter 3 that case variables are used to define the various cases in which an assertion applies. These declarations include variables such as s, which will represent the initial value of the stack pointer. After the declarations come local definitions of the value of the stack pointer after it has been decremented once or twice, and of the value of a flag word, which consists of the condition code bits inserted into the proper field of a word. The body of the assertion follows the declarations and local definitions.

The antecedent of the assertion is similar to the formula:

$$
\begin{aligned}
&\text{control} = \text{nmi} \wedge \text{invariant} = 1 \wedge M[l] = d \\
&\wedge (r \neq \text{NMI} \wedge r \neq \text{SP} \wedge r \neq \text{PC}) \rightarrow R[r] = w \\
&\wedge R[\text{NMI}] = n \\
&\wedge R[\text{SP}] = s \\
&\wedge R[\text{PC}] = p \\
&\wedge \text{cyCC} = \text{cy} \wedge \text{ovCC} = \text{ov} \wedge \text{ngCC} = \text{ng} \wedge \text{zeCC} = \text{ze} \wedge \text{intCC} = \text{int} \\
&\wedge (r \neq 0) \rightarrow \exists w \,\blacksquare\, R[0] = w
\end{aligned}
$$

It describes the conditions in which a non-maskable interrupt occurs. A non-maskable interrupt occurs when the abstract input is "nmi." Any arbitrary memory location l holds some arbitrary data word d. Any arbitrary register r (other than the special registers: namely, the register NMI which holds the address of the interrupt service routine, the stack pointer register SP, or the program counter register PC) holds some arbitrary word w. The special registers hold values; n is the address of the interrupt service routine, s is the value of the stack pointer, and p is the program counter. The condition codes have arbitrary values. Register 0 also holds some arbitrary[8] value w, if register 0 was not the arbitrary register r selected above. (If register 0 was selected, then we have already stated that it has a value, namely w.)

[8]The existential quantifier serves to bind the variable w so that this w is distinct from the one representing the contents of register number r.

The consequent of the assertion is similar to the formula

$$
\begin{aligned}
&\text{invariant} = 0\\
&\wedge (l \neq s = 1 \wedge l \neq s - 2) \rightarrow M[l] = d\\
&\wedge (r \neq \text{SP}) \rightarrow R[r] = w\\
&\wedge M[s-1]\langle 4:0\rangle = \text{int}\,\text{ze}\,\text{ng}\,\text{ov}\,\text{cy}\\
&\wedge M[s-2] = p\\
&\wedge \text{cyCC} = \text{cy} \wedge \text{ovCC} = \text{ov} \wedge \text{ngCC} = \text{ng} \wedge \text{zeCC} = \text{ze} \wedge \text{intCC} = 1\\
&\wedge R[\text{NMI}] = n\\
&\wedge R[\text{SP}] = s - 2\\
&\wedge R[\text{PC}] = n
\end{aligned}
$$

It describes the conditions that follow receipt of a non-maskable interrupt. After an interrupt is received, it will not be pending, and memory will be unchanged, except for two locations on the stack, which will hold the previous condition codes and program counter. Most of the condition codes will be unchanged, but the "interrupt" flag will be asserted. The register holding the interrupt service routine address will be unchanged, and the program counter will also have this value. Finally, the stack pointer will have been updated, since two values will have been pushed onto the stack.[9] Since Hector does not allow instructions to be interrupted, i.e., interrupts are sensed only between instructions, this assertion captures all possible conditions in which an interrupt could occur.

Instruction set

The instruction set describes the normal operation of the microprocessor. In normal operation, the processor fetches instructions from memory and executes them. This can be specified by giving an assertion for each instruction. However, the specification can be made more concise by parameterizing assertions. Thus, rather than have an assertion for each binary operation and each combination of addressing modes, we can specify a single assertion for each combination of addressing modes. It describes all of the binary operations available in the instruction set.

[9]The careful reader, comparing this discussion to the text of the specification, will have by now noticed that references to values stored in memory are given in the specification with an extra parameter, a small integer. Strictly speaking, this should not be present. It is a "hint," used by the program that applies the processor-memory decomposition, to establish the clock cycle on which a memory operation takes place. To be entirely strict, assertions should be mapped so that they allow *any* sequence of memory operations, provided that they yield the desired effect. For example, the order in which locations are read from memory does not matter. However, checking for all possible orders would be expensive. For Hector, it is easy to identify the specific order that actually is used, by examining the instruction level simulator. The generated assertion is then specific to the particular sequencing that assumed, and would fail (a "false negative") if it the processor attempted to perform a different sequence of memory operations.

Two-operand register-to-register instructions The simplest of the addressing modes is register addressing. Using register addressing for the source operand means that an operand data value will be read from a register of the register file. Using register addressing for the destination operand means that another operand data value will be read from a register. Since Hector is a two-address machine, the result will also be written to that register.

The assertion describing the register-to-register instructions begins, as the interrupt assertion did, by declaring its case variables. Next, there is a case restriction, whose scope is the remainder of the entire assertion. This indicates that the assertion is to apply only when the major op-code field is one of the binary operations, namely add or subtract (possibly with carry), or Boolean "and," "or," or exclusive-or. These comprise all of the binary operations available in the Hector instruction set. The last part before the body of the assertion gives several local definitions. First, a value p' represents the incremented value of the program counter. Second, a value u' represents the source operand, which will either be some arbitrary value u (if the source register is not the program counter) or the value p' (if the source is the program counter). Finally, a similar value v' represents the destination operand.

The antecedent of the assertion describes the conditions in which a register-to-register binary operation will be performed. The external inputs must be "normal." Any arbitrary memory location l holds a data word d. The program counter contains some value p, and the memory location p contains the encoded instruction, consisting of the op-code, source and destination mode specifiers that indicate register addressing, and source and destination register addresses. The condition codes contain arbitrary values. Any arbitrary register r (except the program counter, or source or destination register) will contain a data word w. If the source register is not the program counter, it will contain some operand value u. (Note from the local definitions that if the source is not the program counter, then $u = u'$.) If the destination is not the program counter or the source, then it will contain some operand value v. Finally, register 0 will contain an arbitrary value w (unless the source or destination is register 0, in which case it has already been stated that the register contains some value).

The consequent describes the result of the register-to-register binary operation. If the arbitrary register r was neither the program counter, the source, nor the destination, then it will contain its original value. If the source was neither the program counter nor the destination, then it also will contain its original value. If the program counter was not the destination, the program counter will have been incremented. The destination will contain the result of the operation computed by the instruction. The condition codes will have been updated according to the operation. Finally, the memory location l will be undisturbed.

Several observations are possible from this assertion. The first is that it is more complicated than might be expected. Numerous case restrictions are needed to qualify various statements. While some of these (those in the antecedent that qualify statements about register values) could be eliminated at the cost of efficiency, others (those in the local definitions, or qualifying the final value of the program counter) cannot. Such case analysis is inherently a part of the processor's execution of the instruction set. Hector fetches the first word of each instruction, and increments the program counter, before it decodes the instruction and loads operands. Thus, if the program counter is used by the execution phase of an instruction, this phase will see the incremented value rather than the original. Similarly, if the destination register is the program counter, the execution phase will over-write the incremented value.

Accurately describing, in a declarative notation, the imperative, sequential nature of the instruction execution in this micro-coded machine, exposes the complexity of its instruction set. This becomes even more apparent when more complex addressing modes are used, such as those with side effects, as the following paragraphs illustrate.

Clearing a register or memory location The addressing modes of the Hector instruction set can be illustrated with the "clear" instruction. This instruction clears a destination value, that is, it sets a word to 0. Depending on the addressing mode, it will clear a register, a memory location given by a register (possibly post-incrementing the register as a side effect), or an array element—that is, a memory location computed by adding an index taken from a register to a base address taken from the instruction stream.

Clearing a register Clearing a register is rather straightforward. This action occurs when the processor is running, the program counter points to the appropriate instruction, and the other registers have values.[10]

After a "clear register" instruction has been executed, the program counter will have been advanced (unless the program counter was being cleared) and the addressed register will contain a zero.

Clearing a memory location addressed by a register Clearing a memory location addressed indirectly by a register is similar, but more complicated due to the effective-address calculation. The assertion for this instruction declares several values, including a value, b, to possibly represent the address of the location to be cleared. A value b' is defined locally, which represents the address of the location that will actually be cleared: if the register being used is the program

[10]Because of the the way the Hector ALU is implemented, it is necessary that the register to be cleared hold a definite value. The register file in Hector is composed of static cells, so this is ensured by electrical effects, but these are not captured by the switch-level model. Thus we must include them in the assumptions of the assertion, i.e., its antecedent.

counter, b' will not equal b; instead, it will be p', the incremented value of the original program counter.

After the memory location has been cleared, the memory location b' will contain zeros, and the program counter will have been incremented. Other locations and registers will be unchanged.

The definition of the value b', which is essentially an effective address, underscores the earlier point about the formal verification of processors: that complicated instruction sets require complicated specifications. Using the program counter as the indirect register for clearing a memory location is a useless operation. Its effect is to overwrite the next instruction in memory with a word of zeros. This word will then be executed as the next instruction!

The same effect could be achieved more easily by writing the word of zeros as the instruction in the first place. There is no need for the processor to implement this case of this instruction. However, given that the processor does implement this case, there is a need to *consider* this case in the specification.

That is not to say that there is a need to *specify* the behavior in this case. It would suffice to indicate explicitly that, in this case, it does not matter what the processor does. This could be accomplished by enclosing the assertion within a case restriction. What is necessary, though, is to explicitly indicate that this is a special case, because it is actually a special case to the operation of the processor. In order to formally verify a system, the formal specification must accurately reflect what the system actually does.

Clearing a memory location and incrementing a register The "clear" instruction with postincrement addressing is essentially the same as with indirect addressing, except that after the instruction has been executed, the indirect register will have been incremented. However, it is necessary to express the fact that if the indirect register is the program counter, then it also will have been advanced, i.e., its final value will be **two** greater than its original value.

Clearing an array element Clearing the memory location given by adding a base address from the instruction stream to an index from a register is somewhat similar. The variable declarations include a value to be used for the base address, and a value j which may be used as an index. A locally defined value j' is the actual index, which will be the incremented value of the program counter, if the program counter is selected as the index register.

Before this instruction is executed, the processor must be running, the program counter must point to the instruction, the memory location following the instruction must contain the base address, and the condition codes and registers must contain the appropriate values. After execution, unaddressed memory locations and registers must not have changed, the selected memory word must be zero, and the program counter must have been incremented by two—once for the instruction word, and once for the base address.

The various assertions capture the various operations of the processor. Assertions describe the initialization and interrupt response as well as the individual instructions. The assertion describing each instruction is complicated because the instruction itself is. Statements that are tacit in informal descriptions (such as that unaddressed state does not change, and advancement of the program counter) must be made explicit. Special cases must be identified.

The text of the indexed "clear" in the formal specification is obviously less concise than the text of the simulation, Figure 8.4. Some of the difference can be explained by differences in notation (lisp requires (too many) parentheses), the presence of explicit declarations in the specification, and the fact that the code in the figure is only a fragment of the simulation, whereas the specification includes the entire instruction execution including instruction fetch. Nonetheless, the assertional form is more complicated.

This may at first seem like a drawback of formal specification compared to informal techniques, but in fact it can be an advantage. Making tacit information explicit, and ensuring that the conditions in which they apply are properly qualified, results in an accurate description of the instruction set. Such formal descriptions are more precise than informal ones, and as the examples in Appendix B show, while they are not equally concise, they are not unduly burdensome. It takes little more notation to make a careful statement than to make an informal one, but we should expect to say more in order to express more.

From the given instructions, we can estimate that a complete description of the instruction set would occupy about 100k bytes, or 50 pages—roughly the same as would the description of a similar instruction set in a traditional programmer's reference manual.

8.3.5 I/O mappings

A specification of an abstraction of a system, alone, is incomplete, because it says nothing about the actual system's behavior. The definitions of how the inputs and outputs of the abstraction are encoded as inputs and outputs of the circuit is a necessary part of the specification.

When viewed as a state machine, a computer is actually a rather unusual system, because it has few inputs and outputs compared to the amount of its internal state. Furthermore, computers typically used "memory-mapped I/O." That is, most inputs and outputs of a computer system are actually implemented so that the processor accesses them as it does memory locations, rather than as specialized signals connected directly to the processor. When the processor reads a memory-mapped I/O location, it actually senses the value of some input. When it writes to a memory-mapped I/O location, it actually affects some output.

For such systems, the mapping of the memory state is effectively both a state mapping and an I/O mapping, depending on the location being addressed. For those abstract memory locations that are actually memory locations, it is a state mapping, but for those "memory locations" that are actually memory-mapped I/O, it is an input mapping or an output mapping.

Either of two approaches can be taken in order to deal with memory-mapped I/O. The approach taken here is to just treat memory-mapped I/O as memory. Alternatively, a system that included memory-mapped I/O could be verified by including the memory-mapped I/O circuit, rather than decomposing the system at the processor-memory boundary.

Since we are treating memory-mapped I/O as state, Hector has no outputs which appear in the abstract specification. Thus, we conclude that the inputs that the processor receives from the external world complete the specification. These we modeled abstractly with the "control" system variable, which was of the enumerated control type that we also defined. This is the only abstract system variable which is an input.

To complete the specification we must map this abstract variable onto the inputs of the Hector microprocessor circuit.

The bus interface for Hector is similar to that of Motorola's early 6800 microprocessor. In addition to a reset line, there is a wait line, DMA request and acknowledgment, prioritized interrupts, and a single-step facility. A signal distinguishes memory references for instruction fetch, allowing separate instruction and data address spaces.

The mapping of the control input appears toward the end of the mappings for Hector (in section B.4). It makes use of the definition of several circuit nodes within the Hector chip, which appear earlier in the mapping.

Most of the complexity in the mapping is temporal. Consequently, the specification of the mapping first defines a series of names for specific intervals within clock cycles. The clock cycle itself is then defined by specifying the signal values on the clock inputs for each interval within the cycle.[11] The mapping for the control input makes use of the clock definition itself, as well as the names for intervals within clock cycles. It should be emphasized, however, that the clock is otherwise not special. The definition of the clock is certainly central to the timing of the system, but the clock inputs themselves are completely ordinary circuit inputs.

The mapping for the control input consists of a case analysis with a separate definition for each possible abstract input value.

An input value of "reset" represents processor initialization. Figure 8.5 shows

[11] This definition also specifies some "model weakening," which will be discussed in Chapter 9.

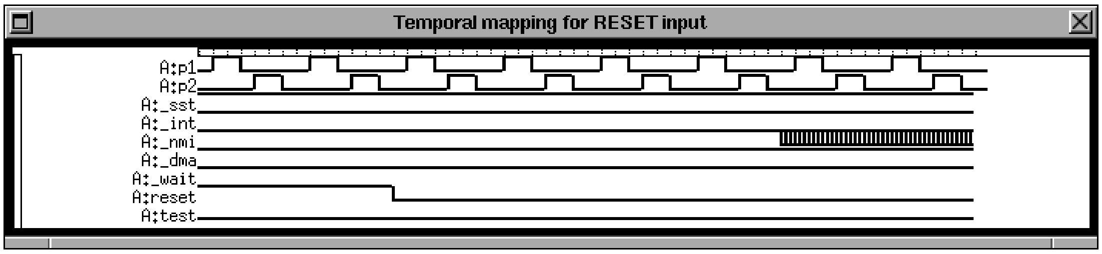


Figure 8.5: Temporal mapping for resetting the Hector microprocessor. The traces represent signal values asserted onto the processor's control input pins. The two traces at the top are the two-phase non-overlapping clock. The reset signal trace is second from bottom. The reset signal is asserted for two clock cycles, then is withdrawn so that the processor is allowed sufficiently many cycles to go through its reset sequence.

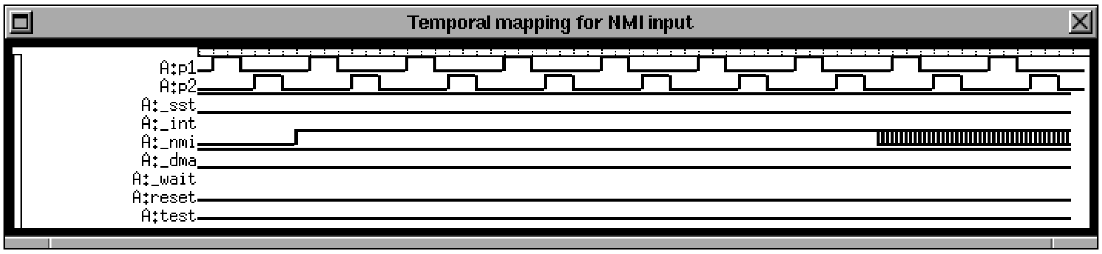


Figure 8.6: Temporal mapping for interrupting the Hector microprocessor. The non-maskable interrupt signal is briefly asserted, and the processor is allowed sufficient time to go through its interrupt sequence.

the mapped image of the reset operation as a timing diagram.[12] In order to reset the processor, its clocks must be sequenced through 8 cycles of operation. For the first two cycles, the reset input must be asserted, and for the remainder it must be inactive. The nominal beginning of the reset operation occurs when the first clock phase rises of the figure. The nominal end occurs shortly after the last clock phase falls. The interrupt signal may be either asserted or inactive as the processor completes its reset sequence.

An input value of "nmi" represents a non-maskable interrupt. Figure 8.6 shows the interrupt operation. In order to interrupt the processor, its clocks must be

[12]This and the following diagrams are snapshots of a display produced by the verifier from the text from Appendix B.

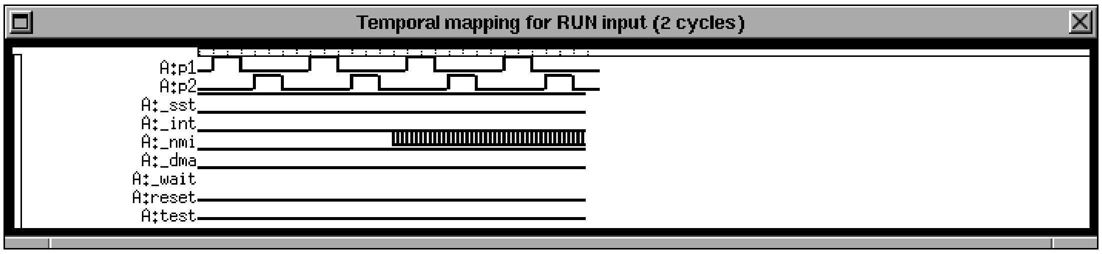


Figure 8.7: Temporal mapping for normal execution of Hector instructions, for a 2-cycle instruction. More than 2 clock cycles must be allowed to occur, because the clock must be running before the processor will operate.

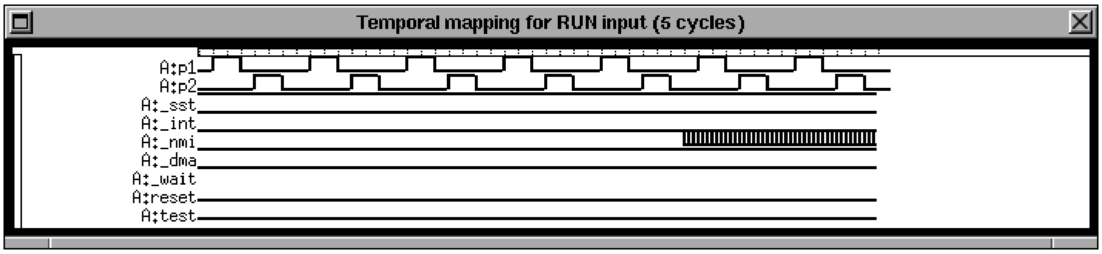


Figure 8.8: Temporal mapping for a longer instruction.

running, and the interrupt signal (which is active when low) must be briefly applied. The processor will then go through its interrupt-response sequence. The nominal beginning of the interrupt operation occurs at the third rising edge of the first clock phase. The nominal end occurs slightly after the last clock phase ends.

Input values of "run2," "run3," etc., represent the normal inputs to the processor, which allow it to execute instructions, for a particular number of clock cycles. Figures 8.7 and 8.8 show normal operation, for a short (single-cycle) instruction and a longer instruction. A single cycle instruction actually requires two cycles: the first one to fetch the instruction, and the second to execute it. The nominal beginning of the operation occurs at the third rising edge of the first clock phase. The nominal ending occurs slightly after the last clock phase ends. As the instruction is being executed, the interrupt signal may be either asserted or inactive.

This concludes the formal specification of Hector. We have described the processor's state, and its instruction set, and its I/O. Any system fulfilling this specification can legitimately be called a Hector microprocessor. We will soon show that the fabricated chip actually is a Hector microprocessor.

8.4 Related work

8.4.1 Hector

Miller and his students [179] designed the Hector microprocessor as a pedagogical example. The reasoning behind the design of Hector's micro*architecture* is described in an unpublished manuscript by its designers. Fernald and colleagues [103] described CMOS implementation of a low-power microprocessor-based system suitable for implanted telemetry from animals. Its processor core is based on Hector. Though the reliability requirements for implantation in animals are less stringent than those for human hosts,[13] it is still desirable that such hardware be correctly designed.

8.4.2 Processor specification

Work that includes processor verification is discussed in Chapter 9.

Leonard [159] thoroughly surveys the specification of computer architectures. He includes several works on verification of the same; the coverage of other topics related to verification, i.e., formal foundations and automata verification, is a bit uneven.

Bowen specified the instruction set of the Motorola 6800, an early commercial 8-bit microprocessor, using the **Z** notation [30, 31].

Boyer and Yu [32] specified most of the user mode instructions of the Motorola 68020, a commercial 32-bit microprocessor, using the Boyer-Moore proof system. Yu [254] used this specification to prove correct the compiled object code of several standard subroutine library functions, and in doing so detected several bugs.

8.5 Chapter summary

This chapter began by discussing existing specifications such as data books and HDL simulators, in order to point out the need for better specifications. Then it turned to a microprocessor called Hector, describing it informally, and with an HDL. Finally, it discussed the formal specification of Hector. Portions of the text of the specification itself appear in Appendix B.

[13] As was noted in the introduction, this has been anticipated by microprocessor manufacturers.

Chapter 9

Hector verification

This chapter completes the case study by discussing the verification of Hector.

Figure 9.1 lists the aspects of Hector's operation that were verified. Initialization was verified.[1] Response to the non-maskable interrupt signal was also verified. Finally, execution of over two dozen combinations instructions and addressing modes were verified. They included several ALU instructions, branch instructions, processor status instructions, and all addressing modes.

Not all of Hector was verified.[2] The distinction between program and data memory was ignored. Response to the maskable interrupt signal was not verified. Several other "abnormal" conditions were not considered as well, including single-step, test,[3] the wait input, and DMA.

Hector contains several instructions which can operate on large arrays of data, and which are implemented by loops in the microcode. None of these "looping" instructions were considered. They are beyond the power of trajectory evaluation, which considers only sequences of circuit operation that have fixed length. Conceptually, however, these instructions do fall within the methodology: they do not loop forever, so they could be checked for each possible number of iterations (i.e., 0 through the size of the memory).[4]

The first aspect of processor operation to verify is initialization. Initialization must work in order for any other operations to be possible. Moreover, during initialization, the processor has no interaction with the memory system. This makes

[1] This is the most important operation of the processor! Every processor which cannot be initialized is useless. One in which there are only (known!) errors in the execution of some instructions might still be useful if the bad instructions are avoided.

[2] All of the instructions listed in Figure 9.1 were verified. The verification was conducted as the supporting theory was developed, and the specification was refined. Not all were verified against the final form of the specification.

[3] Formal verification of test mechanisms has received little attention from researchers, yet it is crucial that test structures actually work.

[4] However, this would be impractical. It would be more practical to extend the theory to allow some sort of bounded inductive reasoning.

Instr.	Addr. Mode	Instr.	Addr. Mode	Instr.	Addr. Mode
initialization		NOT	reg.	ADD	reg., reg.
NMI		INC	reg.	ADDC	reg., reg.
CLR	reg.	DEC	reg.	SUB	reg., reg.
CLR	ind.	SHL	reg.	SUBC	reg., reg.
CLR	inc.	ROL	reg.	AND	reg., reg.
CLR	index	SHR	reg.	OR	reg., reg.
BCS	reg.	ROR	reg.	XOR	reg., reg.
BPL	ind.	SWAP	reg.	BTST	reg., reg.
BGE	rel.	LDF	reg.		
BVC	reg.	STF	reg.		

Figure 9.1: Verified instructions and operations of Hector. These were verified by symbolic simulation of a switch level circuit extracted from the layout. They include all of the operations specified in Appendix B and discussed in Chapter 8.

it possible to begin verification before even considering the memory system. However, before the processor could be verified, it had to be modeled for verification.

9.1 Modeling of Hector

The verification of Hector consisted of three steps. The first was constructing a usable switch-level simulation of the Hector chip. The second was identifying the correspondence between the simulation and the specification. The last was actually checking the correspondence between the specification and the simulation by using the verifier. (Interspersed among these tasks were the construction of a verifier tool and the formulation of the methodology itself.) Of course, debugging of the specification and the circuit occurred at all stages, whenever errors were made manifest.

T. K. Miller made a description of the Hector design available by FTP over the Internet. This included:

- the CIF[5] description that had been used by MOSIS to fabricate the chips[6]
- C source code for an assembler and an instruction simulator

[5]CIF, the Caltech Intermediate Format, is a simple language for representing the geometry of chip designs [174].

[6]The CIF file contained 36850 lines and 813k bytes.

- a small amount of additional rudimentary documentation, including:
  - a pinout for the packaged chip
  - a draft paper on the microcontroller architecture
  - microcode source (though not the microassembler!)
  - manual pages for the assembler and simulator.

9.2 Preparation of Hector

In preparing a simulation model Hector design, CAD tool issues came to the forefront. Such issues yield little deep insight into verification, but they have pragmatic significance. Before verification could commence, the layout had to be represented correctly, and the switch-level simulation had to be made to work.

The initial representation of the layout was a CIF file. Reading this description into Magic [209], a layout editor developed at UC Berkeley, eventually revealed that Magic made an undocumented assumption about CIF, which were not satisfied by the Hector description. This manifested itself only when attempting to simulate the design. Minor modifications to the hierarchical structure of the layout (but not to the actual layout itself) proved necessary.

9.2.1 Simulation

Once the layout was correctly represented within Magic, the circuit extractor produced a representation that could be simulated. A few modeling problems remained, such as structures that yielded X values during simulation, due to node sizing or transistor strengths. Most of these were easily corrected by adjusting Cosmos parameters. Some transistors were marked as zero-delay in order to reduce the number of simulation steps required to simulate each phase of operation using Cosmos [14].

Other than these modeling changes, one significant change to the layout was necessary: drivers were added to the bidirectional busses. As reviewed in Chapter 4, the switch-level model analyzes each transistor group separately. Recall that a transistor group is a set of transistors which share charge because their sources and drains are interconnected. Thus, a transistor group that contains a bidirectional bus necessarily includes all of the transistors that can source or sink current onto the bus. So as to be able to treat the bus as truly being bidirectional, it was necessary to add input enabling transistors, which effectively determined whether an external signal was being applied to the bus, as shown in Figure 9.2. When the enable signal is low, the bus can function as an output, and its logic value is present on the output pin. When the enable signal is high, however, the bus functions as an input, and the input logic value is driven onto the bus. Dealing

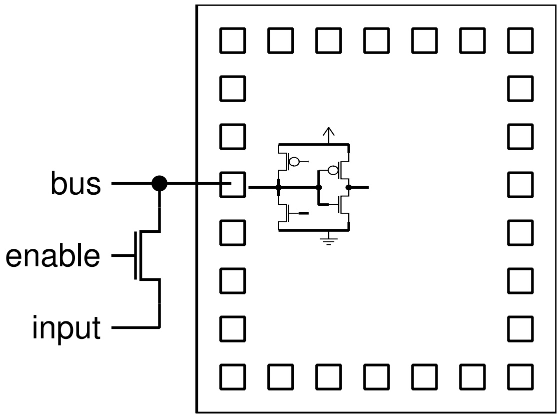


Figure 9.2: Bus driver modification to Hector layout. In order to properly model bidirectional busses at the switch level, input driver transistors were added to Hector's bidirectional pins, such as the one shown. When the "enable" signal is low, the bus functions as an output, but when it is high, a value can be asserted on the labelled input.

with the bidirectional busses in this way allowed all the modeling to be done at the switch level, without extensions. In contrast, attempting to add a bidirectional bus element to the switch-level model would have raised questions of the correctness of the extended model.[7]

In addition to circuit modifications, one tool improvement was necessary. The first step in analyzing a transistor circuit in Cosmos is to run the program "sim2ntk." It converts a *transistor* circuit to a *switch-level* circuit by assigning discrete sizes and strengths to nodes and transistors. Originally, sim2ntk would not automatically assign sizes within a transistor group in which some sizes had been manually annotated. Since the ALU consisted of many multiplexors, it contained some extremely large transistor groups. During verification it became necessary to adjust sizes of some of their nodes, but due to number of nodes in the group, manually assigning all their sizes would have been prohibitive. A simple modification to sim2ntk corrected this deficiency.

Some additional modifications were made to Cosmos and the verifier for convenience or efficiency, but no additional functional modifications to the conventional CAD tools proved necessary.

After addressing these tool issues and deficiencies, circuit extraction and analysis with Cosmos produced a working switch-level simulation model of Hector. This simulation model served as the realization to be verified against the Hector

[7] It would have also allowed an objection that the work no longer used an existing circuit model, and it would have been more work, yet not contributed to the purpose of this research.

specification of Chapter 8 and Appendix B.

9.3 Verification of Hector

The actual verification of Hector commenced after producing a working switch-level simulation. The necessary ingredients of the verification methodology, in addition to the simulation model, are a high-level specification (as discussed in Chapter 8), and state mappings. Since development of the methodology proceeded in parallel with verification of Hector, the initial form of the specification was not as well-structured as final result, but the key distinction, between the high-level specification and the mappings, was always present. The discussion below is primarily chronological.

9.3.1 Initialization

As mentioned before, the first thing to verify is initialization. Verifying initialization required some reverse-engineering. Initial values of registers were not specified in the paper describing Hector, but they could be obtained from microcode. They could also have been obtained from the simulation itself.[8]

State mapping

The abstract assertion describing initialization of the Hector microprocessor makes several statements about the contents of the registers once the processor is initialized. It was necessary to identify the bits of the register file in order to map statements about the abstract register file contents into statements about charges on nodes in the circuit. This required some reverse-engineering of the layout (following traces from the pinout to the busses, to determine the bit ordering), as well as some experiments with the simulator (to determine the word ordering).

It was not necessary at this point to identify all of the state within Hector that would ultimately be needed. Initialization says nothing about the condition codes. Moreover, since initialization does not access memory, it was not yet necessary to consider processor-memory decomposition.

It is important to emphasize that it was *never* necessary to identify several important elements of the processor state, such as the instruction register, the memory address register, and memory data register. Though these are crucial components in the implementation of the instruction set, they are not visible to the programmer. Consequently, they do not appear in the specification of the

[8]It might at first seem to be "cheating" to adjust the specification so that it fits the realization. However, the end result of the process remains a specification to which the circuit bears a formal relation, regardless of how the specification is derived.

instruction semantics, and there is no need to identify them. Every detail of their operation is dealt with in an entirely automatic way by the simulation model of the processor.

Verification

The first version of the specification of initialization asserted binary values directly on internal busses during certain times. This amounted to assuming that valid logic levels, rather than intermediate voltages, were present on the busses. There is not a good basis on which to make such an assumption. However, the assumption was easy to make, and it allowed gaining some additional familiarity with Hector. In addition, there was a rather weak reason to make this assumption: the values on theses busses would have come from the registers, where there was closed positive feedback.

The verification was later refined so as to assert binary values in registers rather than on busses. There is a good electrical basis for assuming that a static register with closed positive feedback will hold a binary value. The patterns generated this way had essentially the same form as the final version, but the specification, at this point, was not structured into assertions plus mappings.

Problems

In first attempting to verify initialization, there was some glitching X oscillation in the ALU, which was eliminated by adjusting sizes. Making the nano-ROM outputs visible aided the identification of the problem.

It was also found that bit 1 of several registers took on the X value instead of the desired logic level, 1. Assuming that an arbitrary binary value was held in one of the registers finally fixed this.[9] In general, destination registers must have binary values in order for the ALU to operate. The selected source and destination registers always drive their values onto the corresponding busses on each cycle. If either contains X values, the X values reach the ALU, where they are applied to the select lines of multiplexors, hence they propagate to the result.

Then it was found that the wrong nodes had been identified for the microcode pointer: its outputs had been identified, but its storage nodes were the ones that should have been identified.

Generally, during verification, problems manifest themselves as ternary X values appearing on some node. (Wrong binary values are less common, and usually much easier to deal with.) It took about a week to go from a working simulation to a verification of initialization, albeit not in a well structured framework.

[9] This register was the destination of an ALU operation during one of the cycles of initialization, and in the word being stored, only the affected bit was 1.

9.3.2 Instruction set

After verifying initialization, I turned to the instruction set. The strategy in conducting verification of instructions was to start with particular instances, using constants rather than symbolic values. This provides some simple validation without the expense of a full symbolic analysis. It was often a simple matter to repeat verification later with symbolic values after getting constant values to work.[10]

State mapping

Execution of an instruction is more interesting than initialization for two reasons. First, it involves more antecedent state. That is, the system can be initialized from any state, but it can only execute instructions from its initialized, running states. This is specified in the antecedents of the assertions that describe the instruction set. Thus, mapping antecedent state becomes important when verifying instructions.

Figure 9.3 illustrates the antecedent as well as the consequent mappings. The mapping for the register state is performed by symbolic indexing. The key observation of this figure is that the antecedent state and the consequent state are mapped from the abstract specification onto the circuit in the same way. The only difference is the timing. The antecedent refers to the state before the instruction is executed, while the consequent refers to the state after the instruction has been executed. Note that although the microcode pointer is mapped "earlier" than the register and flag values, this is done consistently in both the antecedent and the consequent.

The second property of execution of an instruction that makes it more interesting is that it involves interaction with the memory system. Even if the instruction itself does not access memory, the instruction must be fetched from memory.

Thus, verifying an instruction requires confronting processor-memory decomposition. The bus timing for memory operations was determined by examining the timing diagrams in the paper describing Hector [179], and confirmed with simulation. Figure 9.4 illustrates the resulting timing for one particular instruction.

Clearing a register

The first instruction attempted was "CLR R0." This instruction clears register 0, and was a good choice of a first instruction because it did not deal with the condition codes or fetch an operand. On the other hand, it did write a result, so there would be similar instructions using more complex addressing modes that could be verified next.

[10]But not always. For performance reasons, symbolic versions had to be carefully crafted when there was the possibility of antecedent failure. See the discussion of performance in section 9.4.1.

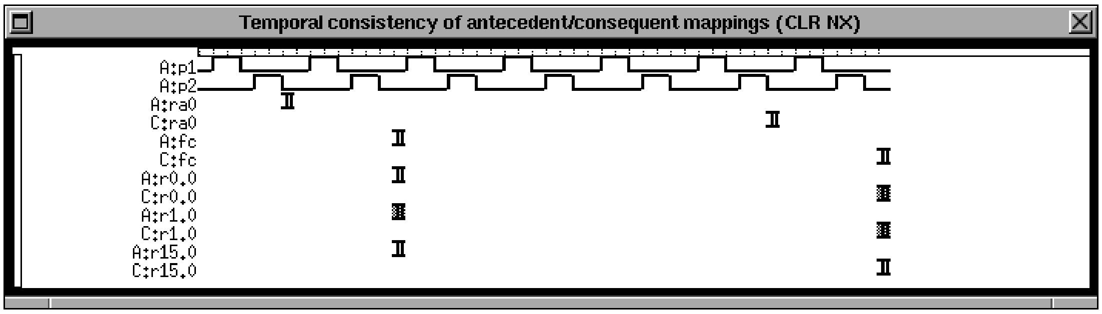


Figure 9.3: Consistency of antecedent and consequent mappings for the "clear" instruction using indexed addressing. Lines beginning with `A:` indicate values asserted by the antecedent. Those beginning with `C:` indicate values checked by the consequent. Nodes `p1` and `p2` are the clocks, `ra0` is the least-significant bit of the microcode pointer, `fc` is the carry flag, and `r0.0`, etc. are least-significant bits of several registers. The grey background for `r1.0` indicates that register 1 receives a binary value in only some of the cases, and is X in others.

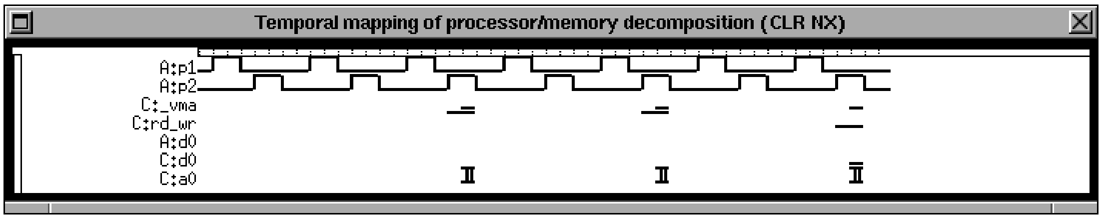


Figure 9.4: Temporal mapping of memory values for the "clear" instruction using indexed addressing. The processor reads memory twice (once to fetch the first instruction word, and once to fetch the base address, and writes once (to clear the indicated array location). During the "read" memory operations, the consequent checks that the control and address values are correct, while the antecedent provides the data value. During the "write" operations, the consequent checks the control, address, and data values.

Problems Initially, not even instruction fetch could be verified. It was determined that simulation of two cycles was needed, starting from X values throughout the simulated system state, before the simulated processor could fetch an instruction. The first of these cycles precharged various structures throughout the microprocessor, and the second initialized the micro-machine. However, the initial formulation of the "CLR R0" instruction still failed. Two bits of the microcode pointer were being set to X when the other bits made their second transition. Tracing through the circuit revealed that an edge-triggered flip-flop used to latch the non-maskable interrupt signal was the source of the X value. It was now apparent that an indication of whether an interrupt was pending—which had not been included in the original abstract specification machine's state—should be included. Re-running the initialization assertion and examining the value to which this latch was initialized gave the proper value for this latch for instruction execution. After making these two changes—allowing two clock cycles before instruction fetch in the mapping, and including the state of this latch in the antecedent of the assertion—the "CLR R0" instruction was verified.

Clearing memory

The next instruction attempted was "CLR (R0)." This instruction clears a word of memory whose address is in register 0. It differs from the previous instruction in that it writes to memory. Attempting to verify this instruction revealed an error in the specification of the timing of the $\overline{\mathrm{vma}}$ signal during memory writes. After fixing this and verifying the instruction, the instruction "CLR (R0++)" was attempted. This is a similar instruction but it increments register 0 after clearing the memory location. When this instruction first tried, it failed at first because the specification used the wrong opcode, so this was corrected.

Finally, the "CLR I(R0)" instruction was attempted. At first it could not be verified because, during the fetch of the second instruction word, data was not asserted on bus for long enough.[11] The problem manifested itself with ternary X values on the address bus during the cycle following the actual problem. After correcting the timing, the instruction was verified.

At this point, all of Hector's operand addressing modes had been verified to some extent, though not in their full generality. Thus far, the problems that had been encountered fell into three categories. The first were simple mistakes such as transcription errors, e.g., the opcode error in the "CLR (R0++)" instruction. The second were timing mistakes, e.g., the $\overline{\mathrm{vma}}$ signal for memory write operations, and the memory read timing. Finally, there was a more significant specification error:

[11] This instruction was the first to read from memory into the data path. The other instructions each occupied a single word of memory. The fetch of the first word of an instruction reads from memory into the instruction register, not the data path. The two types of read operations had slightly different timing constraints.

the "NMI latch" error. This was due to a deficiency in the abstract specification, which failed to consider whether or not an interrupt was pending. This is an easy thing to forget, since interrupts are often considered only tacitly when thinking about normal instructions. However, operation of the actual system depends on interrupt signals. In order to verify a system, it is necessary to specify what it actually does, not what we might think it does.

Debugging

Identifying the source of errors during verification seems straightforward in retrospect, when the causes of the errors can be simply stated. However, locating these errors was the most tedious part of the verification of the microprocessor. There were several reasons for this. First, without schematics or layout plots, it was difficult to even know what circuitry surrounded the node exhibiting the error. In order to see what the circuitry was, it was necessary to sketch portions of the processor schematic by exploring the network within the simulator.[12]

Second, there is often considerable activity between the time that an error occurs and the time that it is noticed.[13] Thus, in order to find the cause of an unwanted X value on one node, it was often necessary to trace through the circuit, find that the X value was coming from some other node, and then re-run the verification, monitoring the newly discovered node, to find the time that it in turn had been set to X. Sometimes the process had to be repeated several times.

Third, understanding the state of a symbolic simulator is difficult. This is because a symbolic simulator does not represent a single state for the system being modeled. Instead, it represents many states, one for each valuation of the symbolic variables. Understanding even a simple Boolean function of three variables requires a moment of thought. The requisite effort is increased if the function is expressed in some automatically generated form (e.g., as an ordered sum of products, or as a BDD) rather than an expression designed by its writer for exposition. Understanding a large set of even more complicated functions, and the structure of a circuit, and the relation between the two—at the same time—is all but impossible. Thus, when errors are detected by symbolic simulation, a strategy different from symbolic simulation is required to analyze them. The first step is to select a valuation for the symbolic variables, one which manifests the error. Although in principle any such valuation will do, simpler valuations—such as those in which most of the variables take the value 0—are often easier to understand. Given such a valuation, examining the symbolic simulation state under this valuation becomes tractable, for the state values become 0, 1, and X rather than complex functions.

[12]A useful stack-based browser was programmed and included in the verifier's user interface. This eliminated some of the tedium of typing long node names, but it was still necessary to sketch and annotate fragments of transistor network as they were explored.

[13]This is a good general rule of debugging which bears repeating.

Finally, identifying the source of errors can be difficult because of the slow speed of Boolean manipulation caused by the sheer size of BDDs, in some cases. Particularly when a circuit is not behaving as expected, it may be computing functions for which the BDD variable ordering chosen by the user is not a good ordering. In a few extreme cases while verifying Hector, the only sign that something was amiss was simply that the BDDs were growing so much larger than expected that the simulation was slowed to a near-halt. When this occurred, it was possible to interrupt the simulation and examine the profiles of the BDDs representing values on internal nodes, in order to select an improved variable ordering.

Thus, debugging the specification was tedious for several reasons. Nonetheless, as more instructions were verified and the design became more familiar, the process became faster.

Specifications at this time lacked a good structure, and they consisted of trajectories written directly in a simple temporal language, embedded in Scheme.

More instructions

Since the addressing modes appeared to be working, other different aspects of the processor were explored, first by verifying the "BCS R0" instruction. This instruction transfers control to the location held in register 0, if the carry flag is set.

Initially, it failed in the case when the flag was set (the taken-branch case). The problem manifested itself as X values in the program counter. The cause was determined to be X values in register 16.[14] Register 16 was used as a temporary destination during the execution of the branch. However, the selected source and destination registers always drive their values onto the corresponding busses on each cycle, so the X values reach the ALU, where they propagate to the result, due to the conservatism of the switch-level model. Specifying that arbitrary binary values were held in this register[15] allowed the "BCS R0" instruction to be verified.

The next instruction verified was the "LDF R0" instruction. This required the identification of the condition code flag bits within the circuit.

Completing the state mapping It was easy to find the condition codes in the layout, but it would have been tedious to attempt to distinguish them by tracing through the layout. It was much easier to examine the operation of the "load flags" instruction in order to identify each of the particular flag bits.

[14] Although Hector's programmer's model (Figure 8.1) has only 16 registers, numbered 0–15, as does the abstract specification of Hector, the actual chip has two additional registers used by the microcode as temporary locations.

[15] Since this register does not appear in the abstract model of the processor, this had to be specified as a portion of the mapping. This can be seen in the definition of the mapping for `invariant`, in Appendix B.

```
(define (binCy op v u cy ov ng ze)
  ...
  (?: (// (vec-== op sub) (vec-== op cmp))
        (vec-< (cons 0 u) (cons 0 v)) ; unsigned test
   (?: (vec-== op subc)
       (vec-< (cons 0 (cons 0 u))
              (cons
               0 (vec-add v '(0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 ,cy))))
    ...)...)
```

Figure 9.5: Specification of the carry condition for the "SUBC" instruction, from Appendix B. The carry flag will be set if the subtrahend (`u`) is less than the minuend (which is the sum of `v` and the "old" carry `cy`). The comparison is *unsigned*. The specification language is geared toward signed two's-complement representation. In order to specify the unsigned comparison, it is necessary to extend the 16-bit quantities to *18* bits: once because the minuend may require 17 bits (if the addition of the carry causes overflow), and a second time in order to perform the unsigned comparison using the signed test `vec-<`. In contrast, the SUB and CMP instructions need only one bit of sign extension.

Verification After verifying this instruction, the companion instruction "STF R0" was tried. At first, it failed, and it was necessary to adjust node sizes in order to verify it.

Next, the instruction "ADD R8,R4" was attempted. At first, it failed because it did not specify that there was a binary value in register 0, which is read as a side-effect when the program counter is incremented. After correcting this problem, a specification error was found: the source and destination registers had been interchanged.

Verification of other instructions proceeded similarly. Typically, only minor errors would be found, but determining their causes was time-consuming. Verification of some instructions proceeded more smoothly. For example, the "NOT Rn" instruction was the first unary instruction attempted. Verifying it required locating and correcting specification errors, but most of the other unary operations for register addressing were then straightforward.

The "SUBC" instruction is an instructive example because it was difficult to verify. This instruction's use of the condition codes, in particular the carry flag, is quite subtle to specify correctly. Figure 9.5 illustrates this.

9.3.3 Interrupts

Hector responds to interrupts between instructions. That is, instructions themselves are not interruptible. However, this does not mean that Hector's interrupt mechanism is so simple as might be expected. Hector's microcode senses the interrupt signal as it jumps to the top of the interpreter loop, using a multi-way branch. This saves one cycle for each instruction, compared to a naive approach which senses the interrupt signal after finishing one instruction and before commencing another. One possible invariant that can be used in verifying Hector's instruction execution is "between instructions the microcode pointer reaches the top of the main loop." This is correct in the absence of interrupts.

In the presence of interrupts, the proper invariant is more complicated. It states that, one cycle before the instruction boundary,[16] the microcode is at a location that jumps to the top of the loop using a multi-way branch.

According to the microcode listing, there are 49 such locations. However, because the microcode assembler used in implementing Hector re-orders instructions, there is no obvious correspondence between the locations in the microcode listing and microcode addresses in the Hector chip. Identifying such locations in the chip was therefore somewhat tedious. As shown in the specification in Appendix B, only a subset of those locations was actually identified, since only a subset of the instruction set was actually verified.

The strategy followed in identifying each of these locations was to let them be discovered by simulated instruction execution during verification. The first step was to try to verify an instruction whose "ending microcode address" was not known. This of course would fail, but it was possible to note the failing microcode address, at a time preceding the end of instruction execution by one clock cycle. this new address was then incorporated into the invariant. After all such addresses were identified, it was necessary to repeat the verification of all instructions, to be sure that they all could *start* from any such address.

An alternative strategy would be to move the sensing of an interrupt into the meaning of an instruction. Then the specification of each instruction would state that execution begins at the "top of the loop" and ends either at the same point, or somewhere else (the start of the interrupt response microcode), depending on the state of the external interrupt input. Thus, the invariant would not be mapped to microcode address. Instead, the "pending interrupt" element of specification state would be mapped to microcode address. It might be easier to verify the entire instruction set this way, though it would require developing these slightly different state mappings.

---

[16]i.e., one cycle before the instruction begins (in the antecedent), or one cycle before it ends (in the consequent)

9.4 Observations

A number of observations can be made from the case study. Section 9.4.1 shows that, while performance was adequate for the Hector microprocessor, it is not yet useful for verifying a multi-million-transistor microprocessor at the switch level today. Even to reach this current level of performance, some interesting steps were required. Section 9.4.2 discusses Hector bugs. No real, significant bugs were found in Hector, although a minor difficulty was revealed. Section 9.4.3 describes the assumptions made during verification. Section 9.4.4 discusses several of the difficulties encountered.

9.4.1 Performance

Figure 9.6 shows the performance of the verifier for several instructions and operations. This is a thesis on methodology, not algorithms. Consequently, the table is intended as an indication of the magnitude of the numbers involved, and not for a detailed analysis of a factors contributing to the verifier's performance. As the table indicates, checking an assertion is not a fast process.

Checking each assertion involves a significant amount of work. Consider the "clear" instruction with indexed addressing. Referring back to the timing diagrams of Figures 9.3 and 9.4 is instructive. In order to verify this instruction, 7 cycles of system operation must be simulated. Simulation of the first two cycles is necessary because of precharging. Then instruction fetch is simulated. Only then does the actual verification of instruction execution commence, and it requires 4 additional cycles: the clearing of an internal register, a memory access, an effective-address calculation, and a final memory access.

One of the factors that contributes to the time required to verify an instruction is particularly worth discussion: analysis of charge sharing.[17]

Figure 9.7 illustrates the charge-sharing problem with multiplexors. The problem can be illustrated by examining a single path through the multiplexor, as shown in the figure. Suppose that initially the two transistors at the ends of the path are on, but the one in the middle is off. Charge representing the input value a and the output value b will be present on the respective internal nodes. If the two transistors at the ends are then turned off, these charges will be retained. If the middle transistor is now turned on, the two internal nodes will share charge. If values a and b represent the same logic level, both nodes will retain this level. However, if a and b represent different logic levels, charge sharing will result in both nodes taking on an intermediate voltage—the ternary X value.

[17]Designs which conserve power by reducing unwanted transitions in inactive functional units (by holding their inputs constant) may be easier to analyze because this charge-sharing phenomenon would be reduced.

Instr.	Addr Mode	Time (s)	BDD size final	BDD size max
clr	reg.	518		240000
clr	ind.	341		31000
clr	inc.	380		
clr	indexed	853		159000
clr	reg.	559		241000
clr	indexed	819		156000
clr	indexed	611	6930	
add	reg.,reg.	1711		
xor	reg.,reg.	647	22500	86000
sub	reg.,reg.	1090		122000
subc	reg.,reg.	1068		62000
add	reg.,reg.	644		103000
or	reg.,reg.	741		53000
xor	reg.,reg.	893		67000
clr	ind.	534	23000	45000
initialization		303	496	2376
nmi		790	4783	15445
initialization		369	256	2051

Figure 9.6: Performance of verifier on several assertions. Performance varied with the exact form of the assertion, with BDD variable ordering, with variation of the tuning parameters of Cosmos, and with model weakening. Time is measured in user CPU seconds on a DECstation 5000/200 with 32 MB memory (25 MHz R3000 CPU, 19.9 SPECmark) under the Mach 2.6 operating system.

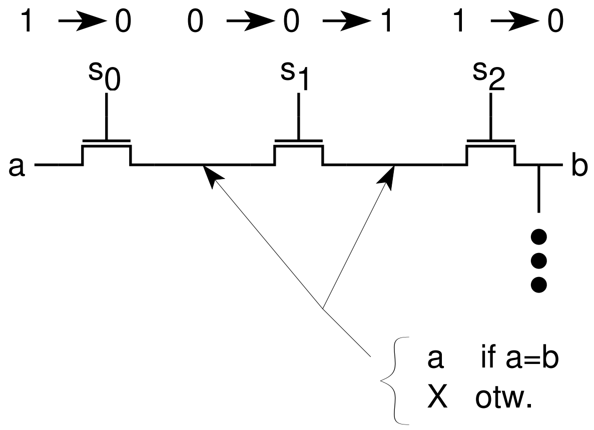


Figure 9.7: Charge sharing in a multiplexor path.

Symbolic simulation of such a circuit when a and b are complex Boolean expressions thus implicitly requires formation of a Boolean expression indicating the cases in which the values differ. This process will occur at every pair of internal multiplexor nodes like those in the figure.

It is important to realize that at no time does the entire path shown in the figure conduct. Thus, this analysis can potentially occur between the output value on a multiplexor and any of its possible input values. Moreover, (assuming that the multiplexors actually operate as multiplexors and not as charge-coupled devices) this path, and consequently its analysis, contributes nothing to the operation of the circuit. Such widespread, detailed analysis, involving unrelated BDDs, is expensive and reduces performance of the verifier.

The charge sharing that causes this performance problem is due to stored charge. In order to reduce the problem it is necessary to eliminate either the charge storage or its analysis. Since the charge storage is a physical phenomenon in the circuit, eliminating its analysis is the only choice. This can be accomplished by treating the internal nodes as if they cannot store charge, allowing them to participate in charging paths, but not to act as capacitors. In Cosmos, such nodes are represented as having a size of 0. Forcing Cosmos to treat the internal multiplexor nodes as if they cannot store charge prevents the charge sharing analysis, isolating the separate values a and b. Note that this is a conservative approximation.

However, not all unwanted analysis takes place in multiplexors. The Hector chip contains only a single ALU. During the execution of an instruction, the ALU is used for several things. First, when the instruction is fetched, the ALU is used to increment the program counter. As operands are fetched, the ALU is used to perform effective-address calculations. Then the ALU is used to compute the

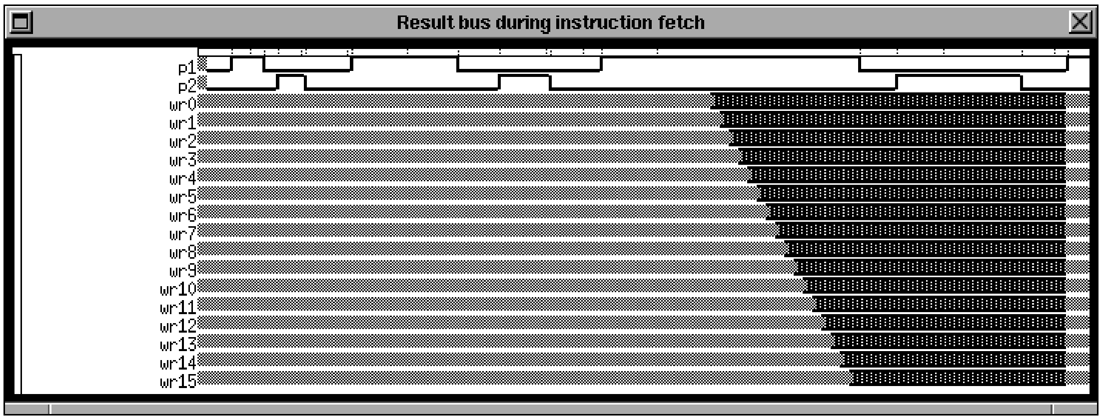


Figure 9.8: Values on the result bus during verification of instruction fetch (illustrated for the "clear" instruction using indexed addressing). Each unit step of simulation is shown. The result bus stabilizes beginning with the least-significant bit (the diagonal "marching" portion). After the result bus has stabilized and its value has been latched elsewhere, the simulation model is weakened by returning the ALU state, to the ternary X, which makes the result bus also take on this value (at the right-hand edge of the figure).

result. Thus, during execution of one instruction the ALU operates on a sequence of unrelated values.

As the ALU performs computation, signals propagate through the carry chain. To simulate this requires numerous evaluations of the excitation functions that describe the ALU operation. Values left-over from previous computations participate in the operation. This too can lead to the calculation of large Boolean functions merely to represent transient values that have no lasting effect.

Isolating successive ALU computations from each other is not as easy as isolating inputs of multiplexors from their outputs, because the ALU depends on charge storage in order to function. It is necessary to modify the circuit model in order to accomplish this isolation. By temporarily resetting circuit nodes to the ternary X value at certain times, we effectively cause them to "forget" their values. Computation in Hector's ALU occurs during the $\phi 1$ clock phase. By weakening the ALU state at the leading edge of this clock, we can isolate successive ALU operations, reducing the size of BDDs. Figure 9.8 illustrates this weakening. It was necessary to weaken the ALU state in order to verify instructions which involved complex addressing modes. Data inputs and nanoROM inputs both were weakened.

In summary, performance of the verifier in modeling the Hector processor is usable but not ideal. This is largely due to the low level at which the processor is

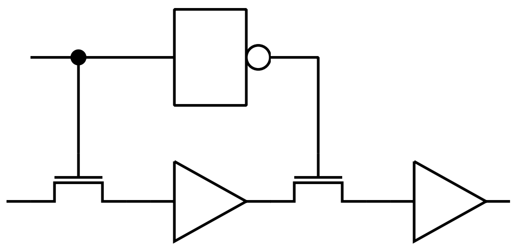


Figure 9.9: Race condition detected by model weakening.

modeled. Judicious modifications to the circuit model were necessary in order to achieve the current performance level.

9.4.2 Hector bugs

The partial verification that was actually completed did not detect any logic errors.

There is one logic error in Hector known to its designers, which is related to cleanup when returning from an interrupt [177]. However, the verification did not examine this aspect of the system's operation.

One small potential problem, a race condition on the memory address bus, as shown in Figure 9.9, was detected. However, it was not detected directly by the actual verification, because of the simple timing model in Cosmos. Instead, it became apparent due to model weakening. When the internal address bus was weakened during a portion of the clock cycle during which its value should not have mattered, the value latched in the memory address register became X. The problem is that the same clock edge, the rising edge of ϕ_1 is used to gate a value off of the "destination" bus leaving the register file, and also to latch this same, buffered value at the output of a multiplexor. In the instructions actually verified, this race is active only when the value being driven onto the address bus is actually also the value already on the address bus. Thus, it does not matter, in this case, which side wins the race. In either outcome the address bus will not change. However, it is possible that the microcode would drive the system in such a way that this error would cause unreliable operation. Only by verifying the entire instruction set would we be able to determine whether this was actually a bug.

Figure 9.9 illustrates this problem. In the figure, the race is obvious, but in the chip the error is much more subtle. The data is gated out in the data side of the chip, while the latching signal is generated in the controller, on the other side of the design, and it is not obvious that there is a sensitized path through the inverting logic. Thus, the race can be classified as the violation of bus timing discipline. This is a common class of design errors.

No serious problems were detected in the Hector chip design. This is not particularly surprising, since it is known to work.

9.4.3 Assumptions

It is important in understanding any work to grasp the assumptions that underlie it. As has been described, it was not possible to simply take the existing layout, and without further effort use it as the realization to be verified. In addition to the physical CAD difficulty with the layout representation and Magic, several electrical assumptions were made. Most of these were ordinary assumptions that must be made in the course of switch-level modeling. For example, the sizes and strengths of some nodes and transistors had to be changed from the values our simple-minded translator had assigned to them. As a further example, the addition of input driver transistors to the bidirectional I/O pins has already been addressed.

One additional assumption that has been hinted at (p. 210) should be made explicit. Typically, in switch-level simulation, the ternary X value is seen as "bad." That is, X values are eliminated—for example, by putting the circuit through its initialization sequence—in order to model binary operation. In contrast, for verification the ternary X value must be seen as "good" because of its power in covering many cases of circuit operation with a single pattern. However, stating that a node holds a ternary X value is a weaker statement than stating that the node hold an arbitrary binary value. In many cases, this difference is inconsequential, but in order to verify Hector it is necessary to state that the values in several of the registers are binary.

The registers in Hector store values in closed feedback loops having positive gain. Figure 9.10 shows one bit of such a register. Given sufficient time, even a metastable value stored in such a loop will resolve to a binary value, as random noise will eventually cause it to leave the metastable point. Thus, the assumption that binary values are stored in the registers is sound.[18] Assumptions such as this should be confirmed with a circuit simulator such as Spice.

This binary-value assumption can be phrased as an existential quantification, stating that "there exists" some binary value v such that the register cell stores the value v. When such a quantifier appears in an assertion, it is treated the same as any other form in an assertion—according to the semantics of the specification language. Since an assertion is implicative, the antecedent is effectively negated. By De Morgan's law, an existential quantifier in the antecedent can be replaced by a universal quantifier over the entire assertion, which then, by the convention of mathematics, can be tacitly omitted. Thus, to assume that a register stores some

[18] If we were to be completely formal, every assumption would create a corresponding proof obligation. This would in turn entail that we have a formal model in which we could express and prove the resolution of binary values. While this would be possible [155] it is beyond the scope of this work.

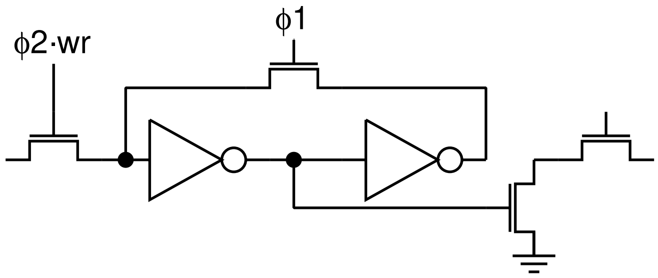


Figure 9.10: Cell from Hector's register file. During the first clock phase, the feedback loop is closed. Provided that the phase has an appreciable duty cycle and that the inverters are properly sized, the cell will store a binary value. If an intermediate voltage is applied, it will resolve to a binary value after sufficiently many clock cycles. (Alternatively, holding the $\phi 1$ clock input high for a sufficiently long period could be used to guarantee that the registers are initialized to binary values.)

arbitrary binary value is ultimately expressed by a "new" variable (i.e., one which does not appear elsewhere).

In the context of initialization it is not possible to check this binary-register assumption within the confines of the switch-level model. In the context of instruction execution, however, a check is possible. In fact, if the assumption is expressed as a part of the mapping of an invariant to be assumed and maintained by all instructions, it will be automatically included by the consistency of antecedent and consequent mappings imposed by the methodology, unless we make special effort to avoid it. The existentially quantified statement that the register cell contains some value will appear in the consequent as well. This existential quantifier, however, remains existential if its scope is enlarged from the consequent to include the entire assertion. Within the quantifier, the check imposed by this binary-values statement will fail for some cases (those where the quantified variable v is assigned a value different from the value actually on the node) and pass for others (those where the values agree). Applying the quantification by using the smoothing operator on the final Boolean function will eliminate the failing cases provided that the passing cases exist, and yield an ultimate indication that the circuit is correct.

However, as was just shown, this check is not without expense. Fortunately, we can argue that we need not make it. Since the binary values are already being assumed by one assertion (namely, the initialization assertion), we can consider this statement to be an assumption of *all* of the assertions, and eliminate the check. Pragmatically, this can be done by assuming in the antecedent a stronger form of the invariant (i.e., one containing the binary-values statement) but checking a weaker form (one without this statement). In terms of the specification language and the abstract machine that it conceptually defines, an additional binary state variable is introduced. This splits the state space of the system into two halves, and in one of them the binary values are indeed present. We assume that the system is always in this half, but we do not check this. All this is reflected in the specification in Appendix B.

Categorizing assumptions

The assumptions that were made during verification of Hector can be placed into 3 categories. The first consists of annoyances, such as the changes made to the layout in order to make it acceptable to Magic. While any sort of change *could* compromise the validity of verification, these changes were entirely straightforward

The second kind of assumptions are those made in modeling the circuit at the switch level: sizes, strengths, and drivers for bidirectional pins. If they are violated, the validity of the verification is called into question. Here a strength of our model is revealed, for we needed only a few assumptions, which are standard in modeling circuits at this level. In contrast, some "switch-level" models used in approaches to verification based on theorem-proving make many more assumptions, e.g., that the circuit is well-formed according to some criterion. We believe that the way to prove designs correct is to make as few such assumptions as is practical.

The final kind of assumptions are those that transcend the switch-level model, such as that the registers store binary values rather than indeterminate voltages. Here we satisfied ourselves that this was indeed true by our experience in electrical engineering, and the relative simplicity of these small structures. To actually formally guarantee that these assumptions hold would require a deeper electrical analysis. Such an analysis could be carried out by some other means if necessary. Again, the presence of such details in our model keeps it much closer to the actual hardware than are the models used by many others.

9.4.4 Difficulties

It is worthwhile to categorize the difficulties encountered in verifying Hector, in order to provide guidance for future efforts and tools. The difficulties fell into two categories: working with Hector, and the interplay of efficiency and the form of the specification.

Working with Hector

The documentation obtained with Hector lacked many details of the internal operation of the circuit. Moreover, the representation of the circuit—as a layout—was inconvenient for hand analysis, and the internal representation of the circuit in the simulation tools—as an abstract graph—was also unpleasant to work with.[19] This made two parts of the verification process particularly tedious. The first was the development of the state mappings, and the second was debugging.

Locating nodes for state mappings Locating relevant circuit nodes—pins, bits in the register file, and condition codes—was hampered by the CAD tools tending not to name internal nodes. Some of this difficulty was due to particular procedures followed. One problem was that the layout was represented by a CIF file. Though this format preserved cell names, it did not preserve node names. Moreover, regular arrays in the designers' original layout were not preserved in the CIF format. Instead, they were represented as collections of individually placed cells, given in an indeterminate order. The strategy adopted was to label nodes as their functions became known. Pins were trivially identified, given the chip pinout. Internal nodes were more difficult.

By comparing structures in the layout with a block diagram from the Hector paper [179], and interpreting cell names, it was possible to identify the locations of many major structures such as busses, the register file, the microcode ROMs, and the ALU. The basic cell of the register file was fairly small, and a schematic of its circuit appeared in the Hector paper, so it was easy to find and label the node within the cell on which data was actually stored. However, the organization of the register file was more problematic. It was easy to distinguish the least-significant from the most-significant bit, and locate bits in the register file geometrically, but the cell names, and hence the fully qualified node names, bore little relationship to geometry. Here it was necessary to enumerate some of this structure by hand, by following the layout.

The register file was an easily identifiable, regular layout of an essentially simple structure. The condition codes, in contrast, consisted of random logic intermixed with random wiring, made more confusing because the control signals from the microcode to the ALU were routed through this area. Thus, locating the storage nodes within the condition code block required some careful study and signal tracing of the layout. Ultimately, the individual condition code bits were identified, but not distinguished, by examining the layout. Simulation was used to distinguish the bits from one another.

The microcode pointer was relatively easy to identify in the layout, for it was a small but regular structure.

Documentation and debugging Verifying a microprocessor is conceptually straightforward, although there are many details to attend, and (as mentioned earlier) finding the appropriate storage nodes in a switch level circuit can be tedious. Surprisingly little documentation is needed to actually verify a correct

[19] It did have the advantage of automated traversal.

circuit against a correct specification.

However, such a situation seldom arises. Circuits and specifications are not correct. If they were, verification would not be an interesting problem. When the specification and the realization disagree, debugging commences. Not having documentation can then be a significant impediment. In particular, exploring a large switch-level circuit without aid of schematic diagrams is slow.

The typical case is the diagnosis of an unexpected X value on a node. It is first necessary to determine the time at which the node actually receives the X value (or, the time at which it should have received a binary value, if it is always X). Once this time is determined, the simulation can be repeated and stopped at this point. Tracing through the circuit commences.

A value can be propagated onto a node only through conducting transistors. Thus, tracing can omit those transistors known to be off. All other transistors must be included. Multiplexors are particularly time-consuming.[20] The large number of possible paths through a multiplexor composed of switches requires exploration of many cases during such a low-level analysis. In every case during debugging, the unwanted X values were propagated through multiplexors, rather than being generated within them by charge sharing. Thus, abstracting from the multiplexor structure to its function might have been an effective way to speed this tracing. Unfortunately, since the modular structure of the design was not preserved during the switch-level analysis, this was not possible. Moreover, since the generation of X values remains a possibility, analysis in the general case must include the internal structure of multiplexors.

Providing even a slight additional amount of structure to the circuit, either in the form of schematics, or by preserving the cell structure of the layout, would have eased the debugging problem by decreasing the amount of reverse-engineering needed to recover this structure in order to understand the circuit whenever a failure occurred.

Efficiency and specification form

Another difficult task was maintaining a balance between four things: between abstraction, clarity, and correctness in the specification, and efficiency in the verification. Consider the specification of an operation which uses two values, respectively u and v, from two registers, a and b. The fragment $R[a] = u \wedge R[b] = v$ will appear in the antecedent of an assertion in the specification, in some form. Such a statement cannot hold for all cases (i.e., valuations of the case variables a, b, u, and v). In particular, when $a = b$ then the statement fails when $u \neq v$. When such a statement appears in the antecedent, this failure is known as antecedent failure.

[20]This makes it unfortunate that they are also the principal implementation technique used in Hector.

Moreover, since the condition of whether or not the antecedent fails depends on the data values (i.e., the valuation for u and v), this is a *data-dependent* antecedent failure. Antecedent failure is equivalent to an automatically generated case restriction. As discussed in Chapter 7, data-dependent case restriction is expensive.

Address-dependent[21] case restriction is less expensive.[22] For example, the similar specification fragment (if $a \neq b \rightarrow (R[a] = u \wedge R[b] = v)$) has address-dependent case restriction. Unfortunately, this particular example is not equivalent to the first statement above: the case where $a = b$, i.e., where both values are fetched from the same register, is not covered at all. If the specification fragment is rewritten as $R[a] = u \wedge (\text{if } a \neq b \rightarrow R[b] = v)$, on the other hand, then this case will be covered.

Thus, an efficiency issue becomes reflected in the abstract specification itself. This clutters the specification and makes it seem less abstract. By introducing additional notation, it also introduces additional chance for errors.

On the other hand, the explicit identification of case restriction can be an important advantage. When case restriction is introduced automatically through antecedent failure, the set of cases that remain uncovered is not indicated explicitly in the specification. The information is tacitly present, but without analysis its very existence may not even be acknowledged.

It is easy to analyze antecedent failure during verification of an assertion by trajectory evaluation. In fact, the maintenance of a BDD that indicates the antecedent success conditions (the complement of the antecedent failure conditions) is a side-effect of the trajectory evaluation algorithm. However, this BDD is not particularly easy to understand.

Making the case analysis explicit in the specification avoids this problem. Since this strategy is more efficient, and also makes the results of trajectory evaluation easier to interpret, it was adopted in verifying Hector. On the other hand, it does tend to clutter the specification, as Appendix B shows.

9.5 Related work

Several microprocessors have been formally verified by other researchers. Except for MTI, which was designed to study built-in self-test techniques, they have been specialized designs created to evaluate verification techniques. Designs created

[21] Assuming that there are fewer address bits than data bits.

[22] One might speculate that Coudert's "restrict" or generalized cofactor might be useful in approaching this problem. It will not make up all the difference. With explicit antecedent failure on a small previously verified [56] pipelined data path circuit, using the generalized cofactor actually reduced performance by 7–10% compared to a specification using explicit antecedent failure.

for verification allow focus on various specific issues, but not necessarily on the way the issues interact. The verification of Hector is unique in that it was a pre-existing design, its initial description was produced entirely with conventional CAD techniques, an an accurate low-level model was used in verification, but, nonetheless, it was specified at a fairly high level. Using a real design in our verification meant that the description was purely structural. This immediately raised low-level issues such as timing and the absence of an initial state. Also, even though Hector cannot truly be called a pipelined machine, there was a slight degree of overlapped operation, and this made it necessary to develop a theory capable of describing pipelining.

The other verified microprocessors include FM8501 [137, 86] and related designs [136, 239], simplified versions of Cayuga [227] and Lilith [213], Tamarack [150], Viper [77], SECD [119, 120], and MTI [79].

9.5.1 FM8501

Hunt [137] verified a microprocessor called FM8501 using Boyer-Moore logic. He designed FM8501 himself because no suitable formal description of existing machines was available. This description was taken down to the level of recursive functions describing state update. Thus, its implementation was at much the same level of abstraction as the Hector instruction simulator used as a basis for specification in the present thesis. The notion of "verification" that Hunt used was formal proof of equivalence of two formal descriptions.

The basic strategy was to model hardware devices by recursive functions, and then show that they to possess desired properties. Hunt used the Boyer-Moore logic rather than an existing HDL because of the reasoning system associated with Boyer-Moore. Boyer-Moore logic is quantifier-free first-order logic with equality, expressed in a Lisp-like notation. New objects are added by providing axioms to define them inductively. The axioms must satisfy certain principles to ensure that only total recursive functions are defined.

Hunt modeled sequential operation over time by using recursive functions taking one "clock," argument plus arguments representing state (i.e., registers). These recursed at each clock tick, providing their recursive invocations with arguments describing state modification, until the clock was exhausted. These functions were proven equivalent to specification functions, which maintained less state and operated on different time scales.

The formal description of FM8501 was given as a recursive definition of the state machine. It made use of an oracle to model the interaction between the processor and its memory system. The formal specification was a recursive function having fewer parameters (i.e., fewer state details). The correctness proof abstracted the hardware state to yield only the components of interest (those mentioned in the specification.)

The definition of the oracle is somewhat interesting, because although only the existence of the oracle was actually important, the oracle actually had to be constructed, because the Boyer-Moore logic lacks the existential quantifier. [23]

Hunt's original work sparked much subsequent research. Hunt has continued to work on successors to FM8501, gradually reducing the distance between his designs and real systems [136, 239]. Crocker re-verified FM8501 using an entirely different system, SDVS, his state delta verification system [86].

Work related to Hunt's has also been done in the area of "totally verified systems" where many aspects of system operation, from compilers and operating systems to hardware, are verified within a comprehensive framework—in this case, Boyer-Moore [17, 188, 187].

Hunt identified the contribution of his work as having verified two descriptions of a design, one at the gate level. He identified better characterization of clocks, particularly low vs. high level, and better characterization of external devices and separation of the external device specification from the processor specification, as particular needs to be addressed by future work. This work did not consider pipelining, which as we have observed in section 1.5.2 (p. 17) would be difficult in the Boyer-Moore approach.

9.5.2 Cayuga

Srivas and Bickford [227] verified a pipelined microprocessor called mini-Cayuga using a functional-language verification system called Clio. Their specification was written in a lazy functional programming language [235] called Caliban[24] for which a theorem prover exists. Lazy languages are convenient for modeling infinite sequences, and functional languages are more amenable to formal reasoning than are imperative languages. This makes it possible to consider pipelining. The Clio prover is based on rewrite rules.

The verification of mini-Cayuga is interesting because its controller uses a rather realistic clocking scheme (four-phase non-overlapping), and it is implemented with a three-stage pipeline.[25] The Caliban language includes an element $\perp$ which was used to represent unknown values during transitions.

The correctness criterion is given in terms of an abstraction mapping, which maps low-level states to abstract states. An abstract "step" of system operation is identified with each transition into a high-level states which is the image of some low-level state in which an invariant holds. For mini-Cayuga, this invariant held in each state in which an instruction had just completed execution. Since the

[23] Proof of an existential quantification in an automated system typically requires that the user provide the prover with many "hints" which effectively amount to a constructive definition, even in a logic which allows quantifiers.

[24] after Turner's similar language Miranda and Shakespeare's *Tempest*

[25] The pipeline consists of fetch, compute, and write-back stages.

processor was pipelined, the statement of the invariant had to mention the values in the processor's pipe registers. This notion of is similar to Abadi and Lamport's notion of "stuttering" [1]. It is helpful to realize that in such a situation, not all the state of the system is contained in the formalized notion of what a state is[26]. That is, not all high-level states that are purportedly the same can actually be identified as identical, since there is some implicit information as to how long the system will remain in such a state before making a transition.

The verification of mini-Cayuga assumed that the processor started in a known, power-up state. The processor was modeled at a register-transfer level (i.e., the interaction of components was verified, but their individual correctness was not considered). Nonetheless, this work should be taken as a considerable achievement. It demonstrated the verification of pipelining while using a model of system timing that was accurate to individual clock phases.

Sekar and Srivas [213] used similar techniques in a different proof system, called SBL, to verify a simplified version of Wirth's LILITH processor. Their model included a simplified form of instruction prefetch.

This work is of particular interest because they processor-memory interaction was modeled using a nondeterministic function. This they called an "asynchronous" memory, meaning that the time between the issue of a request to the memory and the response from the memory system could vary arbitrarily. They modeled this by giving the elapsed time as the value of a "randomization function."[27]

9.5.3 Tamarack

Joyce's thesis [150] consists of the verification of a microprocessor called Tamarack but also the development of a more generic framework with which to construct such proofs. Tamarack is a version of Gordon's computer. It has been described at a variety of levels, and is not pipelined. At the outset Joyce recognizes that existing well-conceived abstractions should have a significant role in applying formal methods.

Joyce observed that the primary role of formalization is "to support thoughtful human participation in the proof process." This has often been the observation of people working with formal methods [109, 125, 127, 246].

Joyce was perhaps the first to verify formally a fabricated chip [145]. However, this chip had reset problems. (Though it is probably obvious, it bears repeating that if part of a system has been formalized, the parts not formalized are a likely source of problem.)

---

[26]Victor Yodaiken pointed this out to me, in a slightly different context.

[27]This required that they adjust their prover since the randomization function was not referentially transparent (in other words, it does not obey standard substitution rules). Each occurrence of the randomization function was taken to be different.

Joyce also experimented with a silicon compiler, translating the specification into the GENESIL language, but encountered timing problems; specifically, relating the two-phase clock of the silicon compiler to the single-phase clock of his formal specification.

Windley [245] worked on making processor proofs using HOL more tractable by carefully choosing intermediate levels of "generic interpreters." By reducing the amount of detail that any one proof must contend with, he was able to reduce dramatically the number of lemmas required to complete a proof. Such work bears out Davie's observation [89] that though the obvious levels at which to describe a processor are very high or very low (i.e., the two levels used in the current thesis), there are actually a wide range of levels available.

9.5.4 Viper

Viper (Verifiable Integrated Processor for Enhanced Reliability) is a commercially available device that has been advertised as having been formally verified. It is not pipelined. The verification was carried out at several levels. Cohn [78] reports in detail on the verification of the "second level" of the Viper microprocessor. Verification of Viper consisted of defining in the HOL logic functional expressions for a block model, analyzing the block model using this functional representation, deducing results at higher levels of abstraction for each instruction type, and comparing the two levels by relating state, and by relating conditions at the high level to block-level conditions. The third task was not completed because of HOL's lack of support for bit strings, and Cohn's unfamiliarity with processor architecture. The achievement of this verification effort was essentially a symbolic execution of a register-transfer level model, and Cohn concluded that lacked analysis "at levels at which problems seem likeliest to occur." This verification is most remarkable for sheer size of the effort, rather than the techniques used; the proof considered 122 major state paths.

Cohn made several conclusions and remarks, speculating that the original plan for multilevel verification was not ideal. The second level proof had to consider sequencings that ought to have been needed in only the higher level, due to the way that the results of each state transition composed with the results of its successors.

Cohn observes that for large efforts, it would be useful for a proof system to have at least two facilities that HOL lacks (and notes that providing them is a research problem). The first is a facility for describing the abstract structure of proofs (the ML code to generate a particular proof is too concrete). The second is a way to trace the "material dependence" of theorems, so as to obviate unnecessary repeated proof when only a small part of some lemma is changed.

Critiques

Since Viper has been commercially promoted as being a verified microprocessor, it has been subject to a number of critiques. Cohn [77] has pointed out that there will always be two gaps in any verification effort—one between the highest level of formalization and the designer's actual intentions, and the other between the lowest level of formalization and the actual, physical world. Other observations were that at every level of abstraction, there will be some things ignored, and that extra-logical factors play a large, possibly overriding, role in the reliability of entire systems.

Cohn also made two observations to guide future work: that common models and languages for communication between designers, verifiers, and fabricators, form a prerequisite for widespread adoption of formal verification, and that concise and clear abstract representations will lead to increased confidence in correctness of specifications.

Brock and Hunt [33] also critiqued the verification of Viper. They noted that despite the use of a powerful logic-based environment, the specification of Viper was still far removed from high level abstractions, e.g., no formal relation was shown between the `ADD32` function and mathematical addition. They were also critical of the lowest level of verification, which was accomplished by "intelligent exhaustive simulation" [203] and was not related formally to the higher levels.

9.5.5 SECD machine

Graham [119, 120] has verified an implementation of Landin's SECD machine, an architecture designed to execute a functional programming language (a functional subset of Scheme). The specification was in terms of Lisp S-expressions—a much higher level than the bit level of the actual hardware. The low-level formalization modeled the chip's 2-phase non-overlapping clock. Temporal abstraction was performed by the techniques of Melham [175]. Since the SECD architecture implements several high-level operations, the chip implementation is microcoded, and the implementation of some instructions uses loops in the microcode. Proofs were carried out in the HOL system, and the proof obligations included termination of these loops.

The lowest level formalized was the level of cells of the circuit design. Switch-level simulations were carried out for all of these library cells, except the CMOS exclusive-OR gate (it is difficult to simulate [233]). The actual fabricated CMOS chips were found to have wiring errors in shift registers and also the exclusive-OR gate.

Graham found in conducting the verification that many instructions were over-specified. For example, the order in which two memory-read operations occurred might be specified when the order did not matter. He also found that careful design of proof dependencies was needed so that revising fundamental definitions did not entail more re-proof than necessary.

9.5.6 Other processors

Collavizza [79] presents a parts of a semantic specification of the programmer's view ("level 1") of a microprocessor (the MTI processor [23]), as an example of efforts to develop a comprehensive methodology of specification at this level. Then instructions are implemented in terms of microprograms ("level 2"), and correctness is defined in terms equivalent to a commutative diagram. All objects are treated as bit vectors, avoiding the need to convert between integers and their binary representations. An instruction buffer is defined as part of the microprogram machine state. For pipelined systems, additional variables are added to denote pipeline registers. The intent is to decompose verification into a set of smaller, hence easier, verifications which can then be composed. Provided the description of the microcoded implementation is sufficiently precise, the proof that the higher level was equivalent was found to be relatively easy. Partial proofs were done with the Boyer-Moore prover and the OBJ3 term-rewriting prover. The lowest level modeled was essentially the microcode.

Corella [81] has developed a technique for proving the equivalence of microprocessor controllers. It is based on an extension of the state-machine comparison technique of Supowit and Friedman [232], and constructs an explicit state graph. Each vertex of the graph represents a set of states determined by a set of constrains, or equations, associated with the vertex. The program operates by breadth-first search from a set of roots representing initial states, and the state graph is expanded only when new nodes are not subsumed by existing ones. This procedure can produce false negatives, and the algorithm is not guaranteed to terminate, because little or no reasoning about data values is performed. The example considered was the controller from Tamarack-3.

9.6 Chapter summary

This chapter reported on the verification of a substantial subset of the operations performed by the Hector microprocessor. Verifying a real microprocessor, is, as expected, an ambitious project. Using a pre-existing design had the advantage that the design, unquestionably, was not simplified so that it could serve as an example for verification. It had the usual disadvantage of real systems: tools often break in unexpected ways, and simple expedient fixes may be necessary.

Although *verification* of the circuit against an accurate specification was fairly straightforward despite a lack of circuit documentation, *debugging* the initial specification in order to reach an accurate specification was not. Errors manifest themselves as unexpected binary values or ternary X's on circuit nodes. In order to evaluate the cause of these values, it is necessary to understand the surrounding circuit. Without relevant documentation, reverse engineering is necessary.

An unexpected result of the case study was the realization that, at the switch level, symbolic simulation of short sequences of system operation (e.g., simple instructions) is relatively inexpensive, but simulation of longer sequences becomes more expensive. This is due to charge storage between successive operations of functional units. During symbolic simulation this charge sharing must be analyzed in detail for *all* cases, although it contributes nothing to the function of the circuit. Selectively weakening the simulation model is a possible approach for dealing with this problem, but more work is required to evaluate the useful scope of its application.

Despite the problems, a substantial subset of Hector's operation was verified. Not all aspects of processor operation were considered. However, a varied set of instructions was verified. Initialization and interrupt response were also verified. Verifying the entire processor would allow the claim "Hector is a verified microprocessor," but it would add little additional insight.

Part IV

Postliminaries

Chapter 10

Conclusion

We have presented a methodology and applied it to an extensive example.

10.1 Summary

We started by noting that processor correctness was an interesting problem. Then we examined two simple examples—a latch and a stack—to get an idea of what it meant for a circuit realization to be correct with respect to a specification, and how we might go about expressing and proving this. We saw that though these circuits were very simple, we needed to consider details such as clocks and the timing details of the overlapped operation of pipelined circuits. We also sketched the decomposition of the stack into components, for separate verification. This introduction also tacitly established our approach of formalizing existing practice when possible.

After this introduction, we turned to the methodology itself. First we established some mathematical background, including a concept called marked strings, in order to express overlapped operation. Then we started at the specification level. We defined the appropriate model of computation. We did this first in a very abstract way, letting an agent be simply an entity with inputs and outputs. Then we made the model more concrete—specializing it to Moore machines—and developed a specification technique based on a language of assertions. We introduced this language and its semantics with a subset, and gave several examples including our latch, a textbook finite-state machine, a RAM, and our stack.

Next we turned to the realization level. We discussed the general needs of a model of realizations, including conservatism and monotonicity. We also gave particular details of one particular model, the switch-level circuit model, and showed that a switch-level simulation defined a Moore machine.

Having established specification and realization levels, we turned to implementation, the relation between them. Since the two levels differed, we required a means to relate them. We defined implementation as a relation between general agents, and illustrated it with a general example. Implementation is a relationship between input-output behaviors, which for sequential systems (Moore machines) are input-output sequences.

The goal of verification is to prove implementation, and we next discussed how to do this. We related implementation to containment of sets, and discussed how to expose the internal or hidden state of a system. Then we described how to establish containment of entire behaviors by looking only at transitions. We did this by using a general mathematical idea, the containment of one set of semigroup generators within the image of another, under a homomorphism. We showed that we could use marked strings to develop the appropriate homomorphism. Then we showed that we could establish the proper containment of transitions by checking assertions that had been mapped onto marked strings of the circuit. Along the way we discussed two auxiliary properties, conformity and distinction.

Having built up a theory, we next described how it could be applied. We briefly discussed the verification of decomposed systems, which was necessary for verifying processors against specifications of computers. We then mentioned additional applications. We also considered how to represent sets of marked strings of circuit configurations in a way that was efficient and compatible with our symbolic simulator. We defined a language for mapping assertions onto circuits, and gave its semantics. Then we discussed the checking algorithm itself, and some aspects of a verification tool.

In the final part of the thesis, we considered a case study. After discussing existing informal techniques for describing processors, we described the behavior of a pre-existing 16-bit microcoded nMOS microprocessor called Hector, and specified it formally. Finally, we extracted a switch-level circuit from the layout used in fabricating the Hector chips, and verified a diverse set of operations of the processor, including initialization, interrupt response, and several instructions.

10.1.1 Objects in the methodology

Figure 10.1 illustrates the main objects in the methodology, and the properties tested of them. The diagram reflects the distinction between the specification, the realization, and the proof that the latter implements the former.

Figure 10.2 provides more context, and includes more tests that we did not consider. This figure is also divided into the specification, the realization, and the proof. Our model of realizations was a standard model: the switch-level circuit model. Our model of specifications, however, was less standard. In an abstract sense, illustrated at the top of the diagram, our specifications were descriptions, in an assertional specification language, of nondeterministic Moore machines. Such a machine takes, as input a string from an input language, which is defined over some input alphabet. It produces a string over an output alphabet. In addition, it oper-

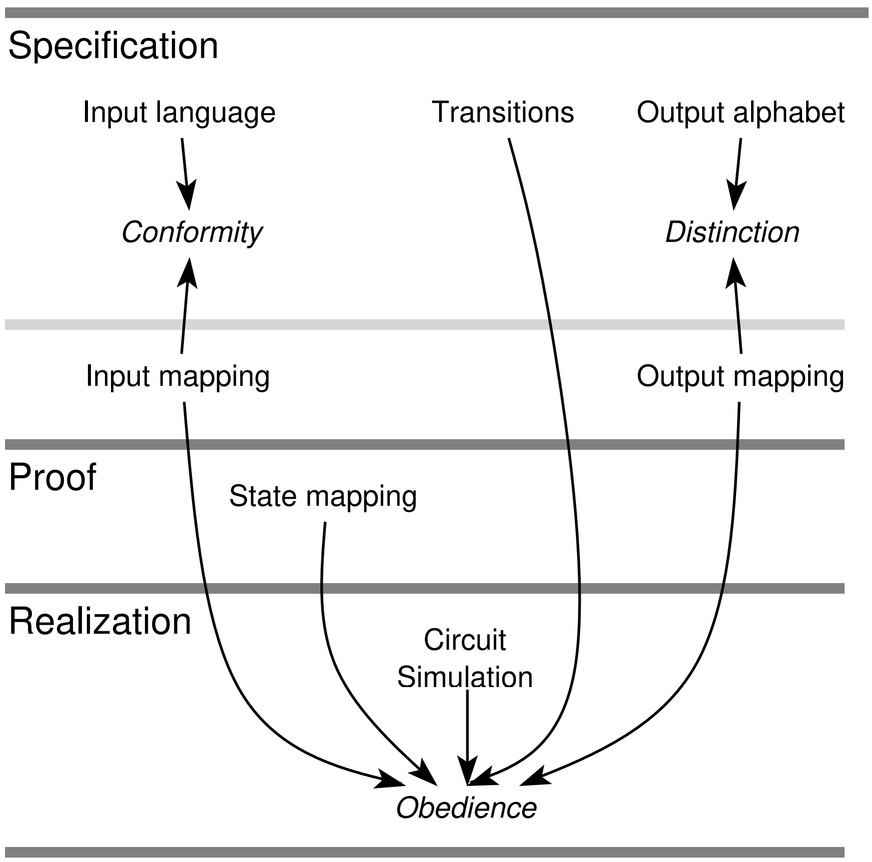


Figure 10.1: Objects in the methodology, and the tests on them. Objects are shown in Roman type, and tests in italics.

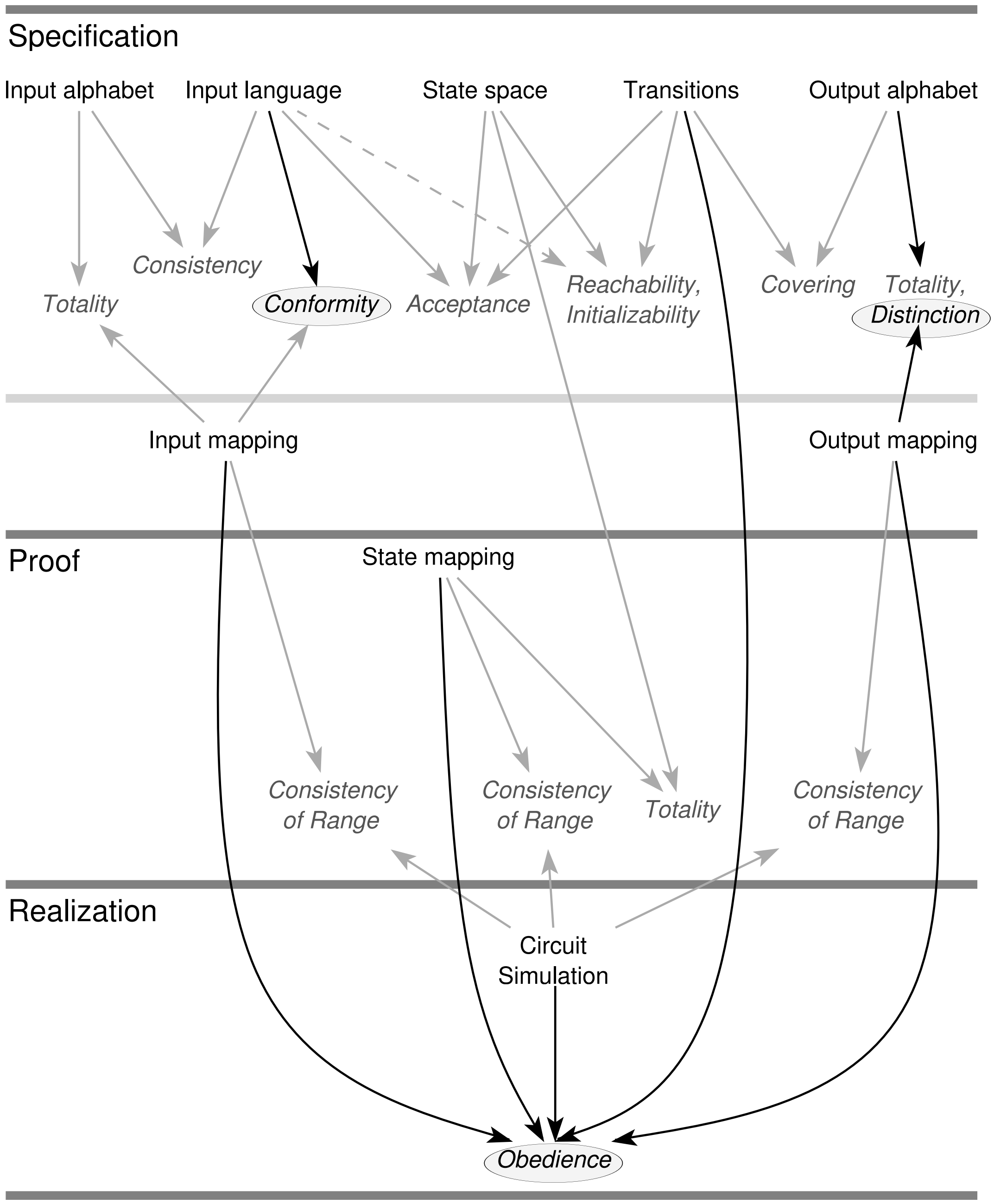


Figure 10.2: Possible objects of the methodology.

ates over a certain internal state space. As it receives inputs, it produces outputs and moves among the states in the space according to its transition relation.

This is rather standard, but two usual elements are omitted from our model: start states and accepting or final states. The omission of final states is actually quite common when modeling systems that are intended to "run forever" rather than compute a final result then halt. The omission of start states is less common. It reflects the reality of circuits whose state, when power is first applied, cannot be predicted. The correct way to verify circuits that must be initialized before they are used is not only to verify that they can be used, but also that they can be initialized. (Omitting both start and end states also lends a certain simplicity.)

Given only a description of a specification machine—the top part of the diagram—a number of consistency checks are conceptually possible. We might check that the input language is indeed a language over the input alphabet. We might check that the machine actually has some defined behavior on each string in the input language, or that all the states in the state space are actually reachable via strings in the input language. We might also check that the machine can be initialized, that is, that from an arbitrary state the machine can be driven into a known state. We could also check whether the machine could actually produce all the outputs in the alleged output alphabet.

Some of these properties—such as consistency—are rather simple, while others—such as initializability—are quite subtle, but all are specification properties. Each could be checked. Even the simple ones are likely sometimes to detect errors that people actually make when they write specifications. However, in this thesis we have concentrated on verification, rather than on specification correctness.

Another essential part of our specifications are mappings of inputs and outputs. They reflect the way in which the system's abstract inputs are encoded for the circuit, and the way in which the circuit outputs are decoded to yield abstract outputs. (The mappings that we use are set-valued, so they are actually capable of expressing arbitrary relations.) While the previous, uppermost portion of the specification was conceptually independent of the particular realization we wished to verify, mappings are usually specialized to a single circuit.

Having these mappings made more tests possible. We could have checked that the mapping functions were actually total—that is, that they actually defined the representation in circuit terms of every symbol of the input or output alphabet. We could also now check two more important properties as well. First, the *input conformity* property expresses the ability to encode each abstract input *string* in the input language.[1] We could also check output distinction, that is, that different abstract output symbols map to distinct sets at the circuit level. (Otherwise, we

[1] This does not follow automatically from totality of the input mapping, because when we consider pipelined circuits we find that two or more successive, independent, abstract input symbols might be represented by overlapping circuit input sequences. Portions intended to overlap must actually match up.

would find that a trivial circuit, where we mapped every abstract symbol to a single circuit symbol, would implement every specification.)

Below the specification in our diagram we find the proof. This consists first of an additional mapping, called the state mapping. This mapping describes the way in which circuit state—the presence or absence of electrical charge on the nodes of the circuit—encodes the abstract system state. With the addition of this mapping alone, the only additional check possible is to ensure that the mapping is indeed defined for every abstract system state.

We needed the realization itself—shown below the proof in the figure—to make the most significant test possible. Once we were given the realization, several more tests became possible. First, for each of the mappings, we could have checked that we indeed have been given a mapping onto the actual circuit. More importantly, we could now also check obedience: that the circuit actually behaves according the the specification.

The methodology developed here concentrated on the properties highlighted on this diagram. These properties are the elements that are fundamental for verification. Although all of the properties in the figure are of some interest to anyone pursuing the formal study of hardware in a broad sense, not all are essential for verification. Consistency checks on the mappings, while desirable as a tool for the early detection of inconsistencies, are straightforward. Properties of the specification are extremely important, for nobody desires to prove vacuous or nonsense statements about the systems they build. To limit the scope of this thesis, we did not consider them in detail. Nonetheless, they are important.

10.1.2 Relation of concepts

Figure 10.2, then, maps the terrain now behind us. Let us now review its exploration by illustrating the relation of a few of the concepts we developed. Figure 10.3 gives this illustration at an abstract level. The central relationship we considered was implementation. It was defined in terms of three properties: obedience, distinction, and conformity; the latter two were fairly straightforward. Obedience was more complicated.

Obedience was the property that the realization's behaviors were allowed by the specification. This followed from a surjectivity condition on the mapping of specification inputs onto realization inputs, plus containment of behaviors. If entire behaviors were expressed as the closure under a combining operator of fragments of behaviors, then containment of behaviors in turn followed from containment of the fragments, plus a homomorphic condition on the mapping. Finally, containment of the fragments followed from containment of assertions, which represent sets of fragments, provided that the mapping was distinct and surjective.

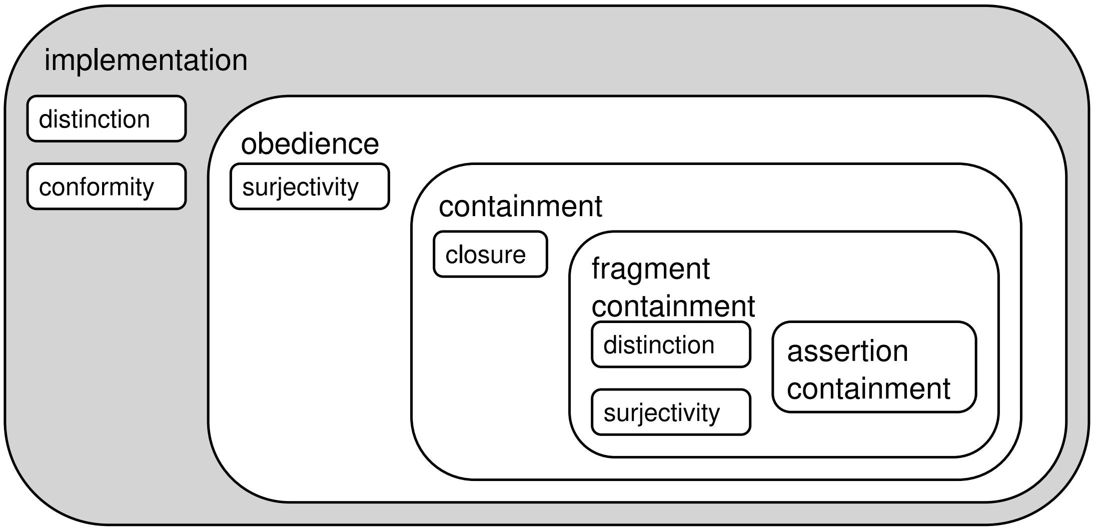


Figure 10.3: Venn diagram of concepts

10.2 Evaluation

Anyone pursuing the serious application of formal methods to real design should be cognizant of the famous objections raised by DeMillo, Lipton, and Perlis [90]. Their point was that mathematics and proofs are "organic." The validity of proofs (or lack) is determined by their acceptance (or rejection) among mathematicians—not by a decision procedure or a satisfaction relation. A proof is not its formalism.[2] A proof is only a proof when someone reads it and is convinced. Machine-generated listings of deductive steps are not proofs in this sense, because they generally remain unread.

Evaluating the work in this thesis—and indeed, any automatic verification technique—by that light is promising. The "proof" of correctness of Hector (for the operations examined) consists of the specification, its mapping onto the symbolic simulation of the microprocessor design, and this simulation itself. The tedious details of these steps are handled algorithmically. The proofs in this methodology that have any depth are the ones appearing here, in this thesis, where they can be studied and understood, not the executions of the verification program, which are to be acknowledged and filed away.

[2]In other words, the objection is that the working mathematician's "proof" is not the logician's technical term of the same name, and the two cannot be interchanged freely.

10.2.1 Hector vs. modern processors

It is worth reviewing Hector, the case study microprocessor, in order to compare it to the microprocessors whose errors were mentioned in the introduction. Though Hector is a real microprocessor and a firm step for microprocessor verification, there are some key differences between it and modern, commercial designs.

First, modern microprocessors are now two orders of magnitude larger than Hector, at least by transistor count, and the trend to larger designs continues. Performance of the verifier on Hector was barely adequate, so clearly the approach must be modified to scale to larger designs. There are two possible ways to modify the approach that has been presented here. First, it might be possible to decompose the processor into components, and verify each of them separately. However, this has the distinct drawback that it may require significant changes to the microprocessor design, if it is not already expressed in terms of a workable decomposition. A more attractive approach may be to verify at a higher level. As we observed in Chapter 9, much of the time spent during verification was required to model low-level aspects such as charge sharing. Such an analysis would not be necessary at a higher level, such as a gate level, or a detailed register-transfer level that still considered all clock phases. It is difficult to estimate the performance improvement that would occur with a higher-level simulation, but it would be quite substantial.

In terms of the scaling to larger register files and word sizes, our experience in verifying a simple pipelined data path [47] is that the scaling of time required is sub-quadratic in word size and sub-linear in number of registers, and memory requirements scale more gradually. Thus, it is quite certain that 64-bit processors can be verified.

Second, modern microprocessors are highly pipelined, and Hector is essentially not pipelined. The practical implications of pipelining on the verification of realistic circuits are not well understood. The methodology handles the overlap of pipelining in full generality, but further experiments are required. Nonetheless, our experiments on simple pipelines lead us to be optimistic.

Third, modern microprocessors handle exceptions. For example, if an instruction encounters a page fault, the instruction will be stopped mid-execution and a trap handler will be invoked. After the trap handler has corrected the source of the page fault, the interrupted instruction will be restarted—either by repeating the entire instruction, or by continuing mid-execution. Hector, in contrast, has a simple interrupt model in which interrupts are synchronized to instruction execution so that they occur only between instructions. Exceptions will complicate the specification. Furthermore, verifying exceptions will be tricky. For example, we would not want to try to verify a trap handler, since this would entail proving that the trap handler—a program—is correct. Nonetheless, an accurate description of the effects of exceptions on the hardware state—such as we could produce by verifying the generation of the exception itself—should be a useful aid to the authors of trap handlers. It seems especially important to treat exceptions well, since they are a source of real problems, e.g., the Intel 486 bus-interface bug.

In addition, modern microprocessors contain some structures, such as multipliers, whose functions cannot be compactly represented using BDDs. It may be possible to verify some aspects of such a processor's operation, those that do not make use of the multiplier. Such a verification would have to be carefully constructed. It is not sufficient merely not to verify the operation of the multiplier; it is necessary to ensure that the multiplier is never operated with symbolic data values, lest the symbolic simulator attempt to construct BDDs to represent the result of multiplication. Nonetheless, such an approach may be fruitful. It might also be possible to use this approach—not verifying the multiplier—and then also adapt a compositional approach to verify the multiplier itself separately, for example by first checking that it computes each partial product correctly, and then checking that it combines them correctly.

Finally, most modern microprocessors contain one or more on-chip primary cache memories. The best approach to verifying such processors is to decompose them at the boundary between the processor and the memory system—in other words, to verify the processor separately from the cache. Attempting to draw the boundary on the other side of the cache, so as to treat the processor plus cache as a unit, entails dealing with matters such as cache consistency protocols in addition to processor behavior.

10.2.2 Limitations

The methodology and the tool implementing the methodology suffer a number of limitations. None of them are fatal flaws, but it is important to acknowledge that they exist.

Limitations of the methodology

First, the methodology described here is most suited for verifying functional properties of data intensive systems, i.e., those whose operation can be thought of as updating data values stored as components of a large stored state, in response to a relatively small number of operations. This is reflected in our specification notation, where each operation requires a separate assertion. Systems that are not data-intensive, such as the state machine example of Chapter 3, developed in Figures 3.33 and 3.34, are cumbersome to describe using our approach.

The suitability of this methodology for functional properties is reflected in our limited notion of time, where in an assertion we consider only a state and its successor—the bare minimum we needed to verify function. There are many important temporal properties of systems that are not easily phrased as functional properties: for example, absence of deadlock.

We have concentrated in this thesis on verification methodology, rather than on correctness of specifications. Obviously, specifications must be correct in order for verification to be meaningful, but we have not fully addressed this issue here. Toward the beginning of this chapter we briefly mentioned a few of the specification properties that are easier to define. The correctness of a specification is not easy to define. In general, it is another verification problem! We have simply "pushed the problem upstairs," that is, raised the level of abstraction at which one can reason confidently about a system. However, it *is* easier to reason about an abstract specification than about a concrete circuit. One possible way to verify temporal properties of a circuit is to use the methodology described here to verify that the circuit implements a state machine, and then use other techniques such as symbolic model checking [173] to reason about this state machine.

One of the principal dangers of verification is what we have called antecedent failure. Whenever we are checking an implication, we must remember that there are two ways in which an implication can be true. First, the antecedent condition can actually imply the consequent condition. This *intentional* interpretation is what we generally think of an implication as meaning. However, formally, implications have an *extensional* meaning: they are also true if the antecedent condition is not true. Antecedent failure means that we can speak nonsense and not realize it. One strength of our approach is that we can check for antecedent failure, and structure our specifications so that antecedents never fail. While structuring a specification this way lengthens it, it improves our confidence that the specification actually makes sense.

Finally, our specifications are given in an unconventional format. We have defined a new language, based on assertions rather than imperative commands, that is quite different from most hardware description languages. Some may see this as a drawback. It is an impediment to the widespread adoption of formal verification, but it is necessary that formal techniques be founded on languages with formal semantics and, furthermore, that they be based upon simple models of computation. It is possible to derive assertions from more-conventional HDL descriptions [194], but it is not clear what difficulties would be encountered in using such assertions for verification. Moreover, for complete confidence it would be necessary to show that the procedure for deriving the assertions is correct—and this requires a formal semantics for the HDL. So lack of connection to conventional hardware description languages is a drawback, but this is more a limitation of current practice than of our methodology.

Limitations of the present tool

The present verifier, which we used in verifying Hector, has a number of limitations. These are not limitations of the methodology; they are characteristics of the tool.

In comparing Hector to more recent microprocessors, above, we have already mentioned that multipliers cannot be verified automatically with BDD-based techniques. We must await better representations for Boolean functions in order to verify multipliers automatically.

The present tool, and the CAMP language (which has not been implemented), cannot deal with unbounded wait states. For example, consider a multiprocessor with a large distributed memory system. If a data value is present in a processor's local cache, the processor can access it in a single cycle, but if the data value is not cached, the processor must wait for it to be fetched. In a large multiprocessor, there could be significant contention for cache lines and system busses, leading to a large number of possible latencies. If the latency could be bounded, it would be possible to write a specification that allowed for a latency of 1 clock cycle, or of 2, or 3, etc., up through the bound. This would be cumbersome in practice and all but impossible to verify, as the system was simulated first with 1-cycle latency, then again with 2, etc. It would be conceptually straightforward to remedy this by adding a fixed-point computation to the tool, and a corresponding construct to the language.

Some of the debugging facilities of the current tool are not ideal. In particular, if the size of the BDDs becomes extremely large so that performance becomes poor, there is no good debugging technique. It is possible in the tool to interrupt the verification and examine the BDDs, but as the BDD package is not designed to be reentrant and allow for concurrent access, this is prone to crashes, and verification cannot be continued after such an interruption. Structuring the program so as to allow reentrant use, such as by developing a BDD package allowing concurrent use by multiple threads of control, could make interactive use of this and other BDD-based systems more attractive.

Finally, the present tool is tied quite tightly to the Cosmos switch-level simulation system. It might be difficult to generalize to another simulation model, although it would be possible to make use of existing intermediate formats used by Cosmos. It would be useful to generalize to another simulation model for two reasons. The first is performance, as discussed above. The second is the time within the design cycle. The switch level is reached comparatively late during the design of a system.[3] It would be preferable to begin verification sooner, so as to detect errors earlier.

10.3 Future work

The goal of research in formal verification is to eliminate harmful design errors without compromising design goals. The work outlined in this thesis is only a step toward that goal.

[3] Especially if the circuit is extracted from layout!

Possible continuations of this work may be divided into theory, tools, and circuits. Theory is the crucial foundation. Without reliable formalisms there can be no formal verification. Tools are the crucial superstructure. Without them the theory cannot be applied. Circuits are the touchstone. The continued verification of circuits is needed to ensure that tools and techniques remain applicable to important designs.

10.3.1 Theory

Several relatively small theoretical steps would be helpful in advancing this research. To better connect it with other work, one could develop a formal relationship between the model of computation used here and other formal models such as the various process calculi (sometimes termed process algebras) [182], Kripke structures [74], or languages such as SMV [173].

Another possible connection to make with existing work is to formalize this methodology more precisely, by working within an existing automated proof system based on some foundational branch of mathematics, such as HOL. The first step here would be to develop the theory of marked strings within the HOL system.

This thesis has presented a specification language SMAL and a mapping language CAMP, but it has given a formal semantics for only a subset of each of these languages. In addition, the definition of SMAL is not complete, for it refers to a library of useful functions that has been left unspecified. Defining the semantics for the full languages and defining and writing the function library would be necessary in order to build a tool based on these languages.

The specification of the Hector microprocessor was cluttered with a number of statements that had to be specified in slightly different ways for slightly different instructions. Defining a specification language based on a non-monotonic logic [108] would allow specifications to be written more concisely by allowing defaults to be written only once. This would entail fundamental changes throughout the methodology.

Decomposition has not been fully treated in this thesis. The discussion of decomposition given here remained at the level of behaviors, and was not carried through to individual transitions. This should be done. Some way of reasoning about serial composition of specification transitions to yield "supertransitions" (possibly using combining forms similar to those of Hoare logics of programs) would be necessary in order to reason about components that operate at different time scales—a processor and memory, for example, do operate on different scales. One possibility would be to represent transitions of the specification level as marked strings instead of pairs of states. Ultimately, decomposition should be automatic, starting from an instruction set and a memory specification like the example on p. 91.

Finally, we conjecture[4] that distinction—the requirement that state mappings be distinct—is actually a stronger property than is required. It is actually sufficient that the mapping, together with the set of assertions, satisfy a similar but weaker condition, namely that for each pair of assertions $A_1 \overset{\delta}{\Rightarrow} C_1$ and $A_2 \overset{\delta}{\Rightarrow} C_2$, that if $A_1 \cap A_2 \neq \emptyset$ then $I(C_1 \cap C_2) = I(C_1) \cap I(C_2)$. It would be useful to phrase this conjecture in terms of marked strings and prove it correct. For example, this would remove the need for the dummy depth counter in verifying Mead and Conway's stack.

10.3.2 Tools

A number of tool improvements and extensions are possible. First, SMAL and CAMP, the specification and mapping languages, should be implemented. Second, conformity and distinction were not fully automated, and they must be implemented in any generally applicable and useful tool supporting this methodology.

The mapping language seems the natural place for a graphical notation based on extended timing diagrams, perhaps similar to those of Borriello [22]. Here one question is whether the diagrams or the text should be primary—should the notation consist of diagrams annotated with text, or text annotated with diagrams? Alternatively, perhaps a dual-view editor could be built, in which neither notation held primacy over the other.

Finally, generalization of the trajectory-evaluation implementation away from the switch level to higher level models is necessary for reasonable performance on large circuits. One drawback we identified in the current implementation is the low level of the switch model. Performing the symbolic simulation at this level sometimes resulted in the detailed analysis of conditions that do not affect circuit operation, such as charge sharing in isolated segments of internal paths through multiplexors, or transient charge sharing during transitions on busses. Simulating at a higher level would reduce the time required for simulation by eliminating such analysis. It would also reduce the space required by the BDDs, since the BDDs representing the effects of such unimportant actions would not be built.

As our example with Hector illustrated, to be most useful the model of a system for verification should be an accepted, existing model. The current implementation of the verifier is based on the Cosmos switch-level simulator, so it uses its model, and in particular, its set of data values.

Other simulators use other sets of data values. For example, many simulators allow a high-impedance (or "Z") value in addition to 0, 1, and X. Often the sets are larger. For example, though VHDL does not define a set of values, the recently-standardized convention among its users is to use a nine-valued model. The values are uninitialized, forcing zero, forcing one, forcing unknown, high impedance, weak

[4] We have been able to prove a similar condition for sets, but not yet for marked strings.

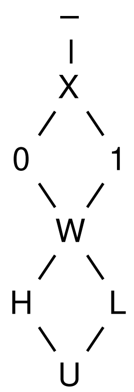


Figure 10.4: A possible partial order for the VHDL 9-valued model.

zero, weak one, weak unknown, and don't care [69, sidebar].

In order to build a verifier capable of applying symbolic simulation to such a model, it is necessary for the model to meet at least three criteria. First, the model must be symbolic. In practice, this can be achieved for any simulator based on Boolean operations over encoded system state by replacing the Boolean operations on values with the corresponding operations on BDDs. Second, the set of values must be partially ordered, e.g., as in Figure 10.4. Finally, the model must be monotonic with respect to a partial order. For the simple structure of the Cosmos partial order, monotonicity was easy to guarantee, but for more complex orders this may not be trivial.

Representations for Boolean functions that improve upon BDDs in some way are currently a very active area of research, and progress in this area might be reflected in a tool. For example, we have pointed out that antecedent failure and symbolic indexing interact poorly. Perhaps read-k-times BDDs might be useful, since the failing variables could appear at two different locations within the variable ordering.

Usability and bookkeeping facilities would be needed in a tool. For example, it is important to to realize that it is not necessary to repeat past successful checks when changes to a specification are cosmetic, or when they weaken the specification.

Finally, supporting conventional HDL's in some way will be crucial to the use of a tool by designers—even if the purpose is to capture their attention sufficiently to explain the advantages of declarative specification. The methodology presented here is based on having a declarative specification written as a set of assertions. Some may perceive this as a drawback rather than as an advantage.

There are two principal objections. First, executable specifications have received some attention recently, but a set of assertions is not easily viewed as an executable specification. Second, the language is new, not being based on an existing language.

Executable specifications have lately received some attention. A specification is a representation[5] of a designer's intention. The argument in favor of executable specifications says that specifications which can be executed are more likely to actually reflect the designer's intent. After all, the argument goes, the designer can check the specification by executing it.

This is a reasonable argument, and it is likely to be compelling in some circumstances. However, executable "specifications" demand implementation details in order to make them executable. Thus, part of the design task becomes intertwined with the specification task.

For example, consider the stack we have so frequently referred to as an example in this thesis. Suppose that it is specified in a language such as Verilog by writing code to accomplish the effects of the "push" and the "pop" operations. The "push" may be accomplished by iterating up through the stack from the bottom, moving data down. The "pop" may be accomplished by iterating down through the stack from the top, moving data up. Such a specification introduces considerable machinery—an index of iteration, and the proper sequencing of the assignment statements that move data—in order to express the data movement. Thus, such a specification is implicitly a moving-data stack. Suppose a stack circuit is implemented with an array plus a pointer to the top location. A necessary step is then to somehow winnow the effect of each operation from the implementation details of the so-called specification.

Another possible objection is that the specification language described here is new, while an existing language would suffice. However, existing hardware description languages are oriented to simulation. In order to support simulation, they include features such as timing, explicit discrete-event management, and large sets of signal values. Such things are needed in order to describe how a digital system operates. However, they are inherently imperative. Instead of specifying what a system does, specifications in such a language tell how to do what the system does. In other words, they make implementation decisions. Because existing HDL's are designed to be executed, specifications written in them will be executable. In its essence, then, the original objection in favor of executable specifications subsumes this second objection.

Nonetheless, the reality of the CAD engineer trying to adopt formal verification within an existing design methodology dictates that some attention be given to an incremental approach: one where specifications *can* be given in an existing

[5] usually a textual representation

HDL, even if the result is not optimal.[6] At the same time, a system build for such an approach should include more powerful facilities, to be used by the more adventurous users, or by others once the inadequacy of imperative specification becomes clear to them. The toughest problem in introducing formal verification to industry may well be a language problem, one which transcends most questions of methodology, models, and algorithms. To continue with existing HDL's like VHDL and Verilog requires great care, as such languages lack formal semantics. It is necessary to found formal verification on a firm basis. Introducing a tool based on some ill-conceived, ad-hoc pseudo-formal subset of an HDL would ultimately set back the acceptance of formal verification.

In an industry roundtable discussion held at ICCD [66], several researchers considered what would be necessary for practicing designers to adopt formal methods. The participants first defined formal verification as involving proof and starting from a specification. This includes synthesis approaches attempting to yield designs that are correct by construction. Fourman cautioned that correctness-by-construction claims were made about compiler optimizations despite lack of good underlying formalism. The participants agreed that engineers could and would handle the formalism once its benefits were clear and the supporting tools were usable. One immediate direction for a first step that emerged was to make hardware description languages more suitable to formal verification.

Such an approach contains several challenging problems. The extension of an HDL to include assertions should not be difficult, since assertions are simple. (Recall that in Chapter 3 we defined both their essential syntax and formal semantics in a half-dozen pages.) But the integration of formal verification with an existing HDL will be mined with pitfalls.

Not the least of the problems is the question of the language's semantics. Most hardware description languages—indeed, most programming languages—lack a formal semantics. Rare exceptions such as LDS [166] were those designed with formal verification in mind. This is not a serious impediment for practical application. Existing practice bears this out. However, formal verification requires formal specification. Formal specification requires formalism. If the "formalism" is actually lacking, the enterprise falls apart, like a chain whose last link is of clay. Though the specification language need not be spelled out entirely in foundational mathematics, some assurance is needed that its terms denote precisely defined objects. Moreover, it is not necessary to formalize all of an HDL to use it for formal specification, provided that users restrict themselves to a formal subset (with appropriate support from tools).

Assuming that a suitable semantics exists for a subset of an existing hardware description language such as Verilog or VHDL, three problems are immediately

[6] This section benefited from discussion with Alok Jain, and his specification of a stack in Verilog.

identifiable. First, there must be some way to convert imperative descriptions to assertions. This likely requires the generation or synthesis of a set of specification case variables from the specification. Since these will in turn be encoded with BDD variables, the interaction of the variables must be carefully considered, or the variable-ordering heuristics made available to the user.

Extracting the variables is only one part of the problem of converting an imperative description to declarative form. There are several possibilities. Data-flow analysis, in the style of optimizing compilers, could be used to extract the computation a system performs from its expression as a program. Alternatively, it might suffice to define a set of simple restrictions that, when obeyed by the specification, would allow a simpler analysis to suffice. Dealing with conditional execution in the HDL will be a challenge, and it is likely that a procedure such as Madre's use of "contexted variables" [166] will be necessary.

10.3.3 Circuits

In verifying additional circuits, care should be taken to ensure that the choice of circuits leads to advances. In addressing the differences between Hector and more modern processors, one place to start would be with simple pipelines that allow interrupts or exceptions. Such toy circuits would allow the development of insight into the difficult issues without confronting the incidental problems of working with real designs. It might also be useful to verify a content-addressable memory (CAM) as a first step toward verifying a cache (since cache tags are stored in CAM).

Later, more modern processors should be verified. Lou Scheffer has suggested that condition codes are an especially likely source of errors [217]. In particular, the original Berkeley RISC processor had condition-code errors, which were difficult to find, and which required that the compiler generate extra code to work around them. Demonstrating a tool capable of finding these errors would be a very convincing example of the utility of verification.

One possible processor to verify would be the "Tiny RISC." This processor was designed by Abnous and his colleagues at UC Irvine [2] as a core functional unit for a VLIW architecture, or as a control processor embedded in a larger chip. It is a 16-bit processor implemented with about 12000 transistors, which achieves 14 MIPS in its first silicon (MOSIS 2-micron CMOS n-well). It appears to be similar to conventional RISC designs, although the complexity of bypassing a VLIW dictated the elimination of the MEM pipe stage to reduce the number of possible dependencies. The small size of this processor should ease tool concerns while allowing concentration on the issues of verifying a pipelined RISC.

It might also be instructive to verify an implementation of a widely used textbook example, such as DLX [198]. While this might not constitute a direct research advance, it would serve as an example that would be accessible to a wide audience.

Finally, if the ultimate goal of verification is to attain currency with state-of-the-art design techniques, it will be necessary to deal with superpipelined and superscalar designs. Although we have given multiple-issue systems some thought in our development of marked strings, we have not given this area serious study. The approach of starting with simple examples seems useful, but so far we hare unaware of any simple superscalar designs.

10.4 Final remarks

This thesis has established that it is possible to take simulation models of pipelined systems—even detailed circuit models expressed with existing design techniques, and relate them to straightforward state machines expressed declaratively. Along the way, it has considered

- what it means for a system to implement a specification, in a general way,
- a model of computation appropriate for both high-level, abstract specifications and low-level circuit details,
- an approach to establishing implementation between a system and its specification,
- a formalism, called marked strings, for reasoning about short overlapped intervals of computation,
- the specification of an existing microprocessor's instruction set,
- a digression into the decomposition of a processor from its memory system,
- the identification and expression of correspondences between the instruction set state and the microprocessor circuit, and
- the verification of the microprocessor.

We have seen that the possibility of formal specification and verification as a routine part of the design of complex digital systems such as microprocessors holds promise. We remain far from this goal, but it is reachable. This thesis has laid a foundation for verification and framed sufficient support for an example based on an existing design. It has answered the challenge that formal methods are a mere academic exercise, by responding with a counterexample consisting of a real design of significant size, and thereby raised the standard of evaluation for hardware verification. It has also shown by its use of an existing simulation model that formal methods, at least in this context, need not represent a break with the past, but the improving evolution of existing practice.

Bibliography

[1] Martín Abadi and Leslie Lamport. The existence of refinement mappings. *Proceedings of Third Annual Symposium on Logic in Computer Science* (Edinburgh, 5–8 July 1988), pages 165–75. IEEE Computer Society Press, 1988.

[2] Arthur Abnous, Christopher Christensen, Jeffrey Gray, John Lenell, Andrew Naylor, and Nader Bagherzadeh. Design and implementation of the 'Tiny RISC' microprocessor. *Microprocessors and Microsystems*, **16**(4):187–93, 1992.

[3] Filip Van Aelten and Jonathan Allen. Efficient verification of VLSI circuits based on syntax and denotational semantics. *Applied Formal Methods for VLSI* (Leuven, Belgium), pages 188–97, Luc M. Claesen, editor. North-Holland, Amsterdam, 1989.

[4] Gregory R. Andrews and Fred B. Schneider. Concepts and notations for concurrent programming. *ACM Computing Surveys*, **15**(1):3–43, March 1983.

[5] Peter B. Andrews. *An Introduction to Mathematical Logic and Type Theory: to Truth through Proof.* Academic Press, London and New York, 1986.

[6] C. M. Angelo, D. Verkest, L. Claesen, and H. De Man. A synopsis on the comparison of HOL and Boyer-Moore for formal hardware verification. *Correct Hardware Design Methodologies: Proceedings of Advanced Research Workshop* (Turin, Italy, 12–14 June 1991), pages 421–6. North-Holland, Amsterdam, 1992.

[7] Bill Arnold. Scarcity, bugs plague 68040; Motorola says ramp up is underway. *EDN*, 7 March 1991. Abstracted in [80].

[8] Larry M. Augustin, Benoit A. Gennart, Youm Huh, David C. Luckham, and Alec G. Stanculescu. Verification of VHDL designs using VAL. *25th Design Automation Conference*, pages 48–53, June 1988.

[9] Cyrus Bamji and Jonathan Allen. GRASP: a grammar-based schematic parser. *26th Design Automation Conference*, pages 448–53, 1989.

[10] Harry G. Barrow. VERIFY: a program for proving correctness of digital hardware designs. *Artificial Inteligence*, **24**:437–91, 1984.

[11] Harry G. Barrow. Proving the correctness of digital hardware designs. *VLSI Design*, **V**(7):64–77, July 1984.

[12] Joel F. Bartlett. `Scheme->C`: a portable Scheme-to-C compiler. Research Report number 89/1. Digital Equipment Corporation Western Research Laboratory, Palo Alto, CA, January 1989.

[13] D. A. Basin, G. M. Brown, and M. E. Leeser. Formally verified synthesis of combinational CMOS circuits. *Integration, the VLSI Journal*, **11**(3):235–50, June 1991. Citation obtained from Inspec.

[14] Derek Beatty, Karl Brace, Randal E. Bryant, Kyeongsoon Cho, and Lawrence Huang. *User's guide to COSMOS: a compiled simulator for MOS circuits.* Computer Science Department, Carnegie-Mellon University, Pittsburgh, PA, October 1987.

[15] Derek L. Beatty, Randal E. Bryant, and Carl-Johan H. Seger. Synchronous circuit verification by symbolic simulation: an illustration. *Advanced Research in VLSI: Proceedings of the 6th MIT Conference*, pages 98–112. MIT Press, March 1990.

[16] William R. Bevier. Kit and the short stack. *Journal of Automated Reasoning*, **5**:519–30, 1989.

[17] William R. Bevier, Warren A. Hunt Jr., J. Strother Moore, and William D. Young. An approach to systems verification. *Journal of Automated Reasoning*, **5**:411–28, 1989.

[18] Jean Paul Billon and Jean Christophe Madre. Original concepts of Priam, an industrial tool for efficient formal verification of combinational circuits. *The Fusion of Hardware Design and Verification* (Glasgow, 4–6 July 1988), pages 487–501, G. Milne, editor. Elsevier Science Publishers, 1988.

[19] Robert L. Blackburn. *Relating Design Representations in an Automated IC Design System.* PhD thesis, published as Technical report CMUCAD–88–45. Carnegie-Mellon University, Pittsburgh, PA, October 1988.

[20] Gregor V. Bochmann. Hardware specification with temporal logic: an example. *IEEE Transactions on Computers*, **C–31**(3):223:231, March 1982.

[21] I. Bolsens, W. De Rammelaere, L. Claesen, and H. De Man. Electrical debugging of synchronous MOS VLSI circuits exploiting analysis of the intended logic behavior. *26th Design Automation Conference*, pages 513–18, 1989.

[22] Gaetano Borriello. *A New Interface Specification Methodology and Its Application to Transducer Synthesis.* PhD thesis, published as Technical report UCB/CSD 88/430. University of California at Berkeley, May 1988.

[23] D. Borrione, P. Camurati, J. L. Paillet, and P. Prinetto. A functional approach to formal hardware verification: the MTI experience. *International Conference on Computer Design*, pages 592–5, 1988.

[24] D. D. Borrione, L. V. Pierre, and A. M. Salem. Formal verification of VHDL descriptions in the Prevail environment. *IEEE Design and Test of Computers*, **9**(2):42–56, June 1992.

[25] Dominique Borrione, Laurence Pierre, and Ashraf Salem. PREVAIL: a proof environment for VHDL descriptions. *Correct Hardware Design Methodologies: Proceedings of Advanced Research Workshop* (Turin, Italy, 12–14 June 1991), pages 163–86. North-Holland, Amsterdam, 1992.

[26] Soumitra Bose and Allan L. Fisher. Automatic verification of synchronous circuits using symbolic logic simulation and temporal logic. *Applied Formal Methods for VLSI* (Leuven, Belgium), pages 759–64, 1989.

[27] Soumitra Bose and Allan L. Fisher. Verifying pipelined hardware using symbolic logic simulation. *International Conference on Computer Design*, 1989.

[28] R. T. Boute. Declarative languages—still a long way to go. *Computer Hardware Description Languages and their Applications*, pages 165–92. Elsevier Science Publishers, April 1991.

[29] Raymond T. Boute. Current work on the semantics of digital systems. *Formal Aspects of VLSI Design*, pages 99–112, G. Milne and P. A. Subrahmanyam, editors. Elsevier Science Publishers, 1986.

[30] Jonathan Bowen. The formal specification of a microprocessor instruction set. Technical report, technical monograph PRG-60. Oxford University, Computing Laboratory, January 1986.

[31] Jonathan Bowen. The formal specification of a microprocessor instruction set. technical monograph PRG–60. Oxford University Computing Laboratory, January 1987.

[32] Robert S. Boyer and Yuan Yu. A Formal Specification of Some User Mode Instructions for the Motorola 68020. Technical report TR–92–04. Department Computer Science, University Texas at Austin, February 1992.

[33] B. Brock and W. A. Hunt, Jr. Report on the formal specification and partial verification of the VIPER microprocessor. *COMPASS '90: Proceedings of Fifth Annual Conference on Computer Assurance, Systems Integrity, Software Safety and Process Security* (Gaithersburg, MD, 24–27 June 1991), pages 91–8. IEEE, June 1991.

[34] Bishop C. Brock, Warren A. Hunt Jr, and William D. Young. An introduction to a formally defined hardware description language. Technical report 76. Computational Logic, Incorporated, Austin, Texas, April 1992.

[35] A. Bronstein and C. L. Talcott. Formal verification of synchronious circuits based on string-functional semantics: the 7 Paillet circuits in Boyer-Moore. *Automatic Verification Methods for Finite State Systems: International Workshop* (Grenoble, 12–14 June 1989), number 407 in Lecture Notes in Computer Science, Joseph Sifakis, editor. Springer Verlag, Berlin, 1989.

[36] Alexandre Bronstein. *MLP: String-Functional Semantics and Boyer-Moore Mechanization for the Formal Verification of Synchronous Circuits*. PhD thesis, published as Technical report STAN–CS–89–1293. Department of Computer Science, Stanford University, December 1989.

[37] Alexandre Bronstein and Carolyn L. Talcott. String-functional semantics for formal verification of synchronous circuits. Technical report STAN–CS–88–1210. Department of Computer Science, Stanford University, June 1988.

[38] Carolyn Van Brussel. Chip shipments stalled by Intel testing problems. *Computing Canada*, 26 September 1991. Abstracted in [80].

[39] Randal E. Bryant. *A Switch-Level Simulation Model for Integrated Logic Circuits*. PhD thesis. MIT, Cambridge, MA, 1981.

[40] Randal E. Bryant. Symbolic verification of MOS circuits. *Chapel Hill Conference VLSI*, page 419:438, 1985.

[41] Randal E. Bryant. Symbolic verification of MOS circuits. Technical report CMU–CS–85–120. Computer Science Department, Carnegie-Mellon University, Pittsburgh, PA, 1985. A preliminary version was presented as [40].

[42] Randal E. Bryant. Can a simulator verify a circuit? *Formal Aspects of VLSI Design*, pages 125–6, G. J.Milne and P. A. Subrahmanyam, editors. North-Holland, Amsterdam, 1986.

[43] Randal E. Bryant. Boolean analysis of MOS circuits. *IEEE Transactions on Computer-Aided Design*, **CAD–6**(4):634–49, 1987.

[44] Randal E. Bryant. A survey of switch-level algorithms. *IEEE Design and Test of Computers*, **4**(4):26–40, 1987.

[45] Randal E. Bryant. Verifying a static RAM design by logic simulation. *Advanced Research in VLSI: Proceedings of 5th MIT Conference*, page 335:349, 1988.

[46] Randal E. Bryant. Verification of synchronous circuits by symbolic logic simulation. *Hardware Specification, Verification, and Synthesis: Mathematical Aspects*, number 408 in Lecture Notes in Computer Science, Miriam Leeser and Geoffrey Brown, editors. Springer Verlag, Berlin, 1989.

[47] Randal E. Bryant. A methodology for hardware verification based on logic simulation. *Journal of the ACM*, April 1991.

[48] Randal E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, **C-35**(8):677–91, August 1986.

[49] Randal E. Bryant. A switch-level model and simulator for MOS digital systems. *IEEE Transactions on Computers*, **C–33**(2):160–77, February 1984.

[50] Randal E. Bryant. On the complexity of VLSI implementations and graph representations of Boolean functions with application to integer multiplication. *IEEE Transactions on Computers*, **40**(2):205–13, February 1991.

[51] Randal E. Bryant. Algorithmic aspects of symbolic switch network analysis. *IEEE Transactions on Computer-Aided Design*, **CAD–6**(4):618–33, July 1987.

[52] Randal E. Bryant. A methodology for hardware verification based on logic simulation. Technical report CMU–CS–87–128. Computer Science Department, Carnegie-Mellon University, Pittsburgh, PA, June 1987. Published as [39].

[53] Randal E. Bryant. An algorithm for MOS logic simulation. *Lambda Magazine*, **1**(4):22–30, September 1980.

[54] Randal E. Bryant. Symbolic Boolean manipulation with ordered binary-decision diagrams. *ACM Computing Surveys*, **24**(3):293–318, September 1992.

[55] Randal E. Bryant, Derek Beatty, Karl Brace, Kyeongsoon Cho, and Thomas Sheffler. COSMOS: a compiled simulator for MOS circuits. *24th Design Automation Conference*. Also reprinted in [192], 1987.

[56] Randal E. Bryant, Derek L. Beatty, and Carl-Johan H. Seger. Formal hardware verification by symbolic ternary trajectory evaluation. *28th Design Automation Conference*, 1991.

[57] Randal E. Bryant and Carl-Johan Seger. Formal verification of digital circuits using symbolic ternary system models. Technical report CMU–CS–90–131. SCS., Carnegie-Mellon University, Pittsburgh, PA, 1990. Also available as [58].

[58] Randal E. Bryant and Carl-Johan Seger. Formal verification of digital circuits using symbolic ternary system models. *Computer-Aided Verification* (Rutgers NJ.), June 1990.

[59] J. A. Brzozowski and C. J. Seger. A unified framework for race analysis of asynchronous networks. *Journal of the ACM*, **36**(1):20–45, January 1989.

[60] Janusz A. Brzozowski and Michael Yoeli. *Digital Networks*, Series in Automatic Computation. Prentice-Hall, Englewood Cliffs, NJ, 1976.

[61] Jerry R. Burch. *Trace Algebra for Automatic Verification of Real-Time Concurrent Systems*. PhD thesis, published as Technical report CMU–CS–92–179. Department of Computer Science, Carnegie-Mellon University, Pittsburgh, PA, August 1992.

[62] Steven Burke. Intel delivers on 486; IBM first to market; Power Platform in short supply. *PC Week*, 9 October 1989. Abstracted in [80].

[63] A talk with Intel: designers of Intel's 386, i486, and future microprocessors talk about what lies ahead in CPU design. *Byte*, April 1991. Abstracted in [80].

[64] Albert John Camilleri. Simulation as an aid to verification using the HOL theorem prover. *Design Methodologies for VLSI and Computer Architecture: Proceedings of IFIP TC10 Working Conference* (Pisa, Italy, 19–21 September 1988), pages 148–68, D. A. Edwards, editor. North-Holland, Amsterdam, 1988.

[65] Albert John Camilleri. Simulation as an aid to verification using the HOL theorem prover. Technical report 150. University of Cambridge Computer Laboratory, October 1988.

[66] Paolo Camurati, Shiu-Kai Chin, Michael Fourman, Carl Pixley, Paolo Prinetto, and Atsushi Takahara. Formal verification: is it practical for real-world design? *IEEE Design and Test of Computers*, **6**(6):50:58, December 1989. Roundtable discussion held at ICCD.

[67] Paolo Camurati and Paolo Prinetto. Formal verification of hardware correctness: an introduction. *Computer Hardware Description Languages and their Applications: Proceedings of the IFIP WG 10.2 Eighth International Conference* (Amsterdam, 27–29 April 1987), pages 225–47, M. R. Barbacci and C. J.Koomen, editors. North-Holland, Amsterdam, 1987.

[68] Paolo Camurati and Paolo Prinetto. Formal verification of hardware correctness: introduction and survey of current research. *IEEE Computer*, **21**(7):8–19, July 1988.

[69] Michael Carroll. VHDL—panacea or hype? *IEEE Spectrum*, pages 34–7, June 1993.

[70] Intel finds bug in 50MHz 80486, suspends ships for a week or so. *Computergram International*, 28 August 1991. Cited in [80].

[71] Shiu-Kai Chin. Verified functions for generating signed-binary arithmetic hardware. *IEEE Transactions on Computer-Aided Design*, **11**(12):1529–58, December 1992.

[72] L. Claesen, D. Borrione, H. Eveking, G. Milne, J. L. Paillet, and P. Prinetto. CHARME: towards formal design and verification for provably correct VLSI hardware. *Correct Hardware Design Methodologies: Proceedings of Advanced Research Workshop* (Turin, Italy, 12–14 June 1991), pages 3–25. North-Holland, Amsterdam, 1992.

[73] E. M. Clarke, J. R. Burch, K. L. McMillan, and David L. Dill. Sequential circuit verification using symbolic model checking. *27th Design Automation Conference*, June 1990.

[74] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, **8**(2):244–63, April 1986.

[75] E. M. Clarke and O. Grümberg. Research on automatic verification of finite-state concurrent systems. *Annual Review of Computer Science*, pages 269–90, 1987.

[76] Edmund Clarke. Applications of Hoare logic to circuit simulation, October 1989. Unpublished notes.

[77] Avra Cohn. The notion of proof in hardware verification. *Journal of Automated Reasoning*, **5**:127–39. Kluwer Academic Publishers, Boston, 1989.

[78] Avra Cohn. Correctness properties of the Viper block model: the second level. Technical report 134. University of Cambridge Computer Laboratory, May 1988.

[79] Hélène Collavizza. Functional semantics of microprocessors at the microprogram level and correspondence with the machine instruction level. *Proceedings of the European Design Automation Conference* (Glasgow, 12–15 March 1990), pages 52–6. IEEE Computer Society Press, 1990.

[80] Computer Database. Information Access Company, Foster City, CA, 1993. Online database, accessible through CMU library.

[81] Francisco Corella. Automated verification of behavioral equivalence for microprocessors. Research report RC 17751 (#78056). IBM Research Division, 27 February 1992.

[82] W. E. Cory. Symbolic simulation for functional verification with ADLIB and SDL. *18th Design Automation Conference*, 1981.

[83] Warren E. Cory. *Verification of Hardware Design Correctness: Symbolic Execution Techniques and Criteria for Consistency.* PhD thesis, published as Technical report 83–241. Computer Systems Laboratory, Stanford University, June 1983.

[84] Oliver Coudert, Christian Berthet, and Jean Christophe Madre. Verification of sequential machines using Boolean functional vectors. *IMEC-IFIP International Workshop on Applied Formal Methods for Correct VLSI Design* (Leuven, Belgium), pages 111–28, 1989.

[85] Olivier Coudert, Jean Christophe Madre, and Christian Berthet. Verifying Temporal Properities of Sequential Machines Without Building their State Diagrams. *Computer-Aided Verification* (Rutgers, NJ), June 1990.

[86] Steven D. Crocker, Eve Cohen, Sue Landauer, and Hilarie Orman. Reverification of a microprocessor. *Proceedings of 1988 IEEE Symposium Security and Privacy*, pages 166–76. IEEE, April 1988.

[87] Luis Damas and Robin Milner. Principle type-schemes for functional programs. *Proceedings of 9th ACM Annual Symposium on Principles of Programming Languages*, pages 207–12, 1982.

[88] John A. Darringer. The application of program verification techniques to hardware verification. *Design Automation Conference*, 1979. Also reprinted in [192].

[89] B. S. Davie. Hardware description languages: some recent developments. Technical report CSR–198–86. University of Edinburgh, April 1986.

[90] Richard A. DeMillo, Richard J. Lipton, and Alan J. Perlis. Social processes and proofs of theorems as programs. *Communications of the ACM*, **22**(5):271–80, May 1979.

[91] S. Devadas and K. Keutzer. An automata-theoretic approach to behavioral equivalence. *Integration, the VLSI Journal*, **12**(2):109–29, December 1991. Citation from INSPEC.

[92] Inderpreet Singh Dhingra. *Formalising an integrated circuit design style in higher order logic*. PhD thesis, published as Technical report 151. University of Cambridge, Computer Laboratory, 1988.

[93] David L. Dill. *Trace Theory for Automatic Hierarchical Verification of Speed-Independent Circuits*. PhD thesis, published as Technical report CMU–CS–88–119. Carnegie-Mellon University, Pittsburgh, PA, February 1988.

[94] R. Kent Dybvig. *The Scheme Programming Language*. Prentice-Hall, Englewood Cliffs, NJ, 1987.

[95] M. Yoeli (editor). *Formal Verification of Hardware Design*. IEEE Computer Society, Los Alamitos, CA, 1991.

[96] Michael Slater (editor). 68040 performance and errata update. *Microprocessor Report*, **5**(11):4–5, 12 June 1991.

[97] Michael Slater (editor). Microprocessor Report, 1989. Various issues.

[98] Michael Slater (editor). Microprocessor Report, 1990. Various issues.

[99] Intel delays debut of P5 processor. *Electronic News*, 27 July 1992. Abstracted in [80].

[100] Rolf Ernst and Jayaram Bhasker. Simulation-based verification for high-level synthesis. *IEEE Design and Test of Computers*, page 14:20, March 1991.

[101] Hans Eveking. Axiomatizing hardware description languages. *International Journal of Computer Aided VLSI Design*, **2**:263–80, 1990.

[102] Hans Eveking. Experience in designing formally verifiable HDL's. *Computer Hardware Description Languages and their Applications*, pages 321–34. Elsevier Science Publishers, 1991.

[103] Kenneth W. Fernald, Todd A. Cook, Thomas K. Miller III, and John J. Paulos. A microprocessor-based implantable telemetry system. *IEEE Computer*, **24**(3):23–30, March 1991.

[104] Lawrence M. Fisher. Intel is putting new chip on hold. *New York Times*, 28 August 1991. Abstracted in [80].

[105] Lawrence Flon and Jayadev Misra. A unified approach to the specification and verification of abstract data types. Technical report TR–80. Computer Science Department, University of Southern California, 1978.

[106] Michael J. Foster. Syntax-directed verification of circuit function. *VLSI Systems and Computations*, pages 203–12, Kung, Sproul, and Steele, editors. Computer Science Press, 1981.

[107] Masahiro Fujita, Hisanori Fujisawa, and Yusuke Matsunaga. Variable ordering algorithms for ordered binary decision diagrams and their evaluation. *IEEE Transactions on Computer-Aided Design*, **12**(1):6–12, January 1993.

[108] Michael R. Genesereth and Nils J. Nilsson. *Logical Foundations of Artificial Intelligence*. Morgan Kaufmann Publishers, 1987.

[109] Susan L. Gerhart. Application of formal methods: developing virtuoso software. *IEEE Software*, pages 7–11, September 1990.

[110] Brian Gillooly and Wendy Goldman Rohm. New chip expected; Intel's problems began when Compaq discovered two significant bugs in the chip. *Computer Reseller News*, 30 October 1989. Abstracted in [80].

[111] Caryn Gillooly. Intel chip woes may delay new superserver models. *Network World*, 2 September 1991. Abstracted in [80].

[112] Seymour Ginsburg. *An Introduction to Mathematical Machine Theory*. Addison-Wesley, Reading, Mass. and London, 1962.

[113] Abraham Ginzburg and Michael Yoeli. Products of automata and the problem of covering. *Transaction of the American Mathematical Society*, **116**(4):253–66, April 1965.

[114] Joseph A. Goguen. OBJ as a theorem prover. Technical report SRI–CSL–88–4. Computer Science Laboratory, SRI International, April 1988.

[115] Michael J. C. Gordon. Why higher-order logic is a good formalism for specifying and verifying hardware. *Formal Aspects of VLSI Design*, pages 153–77, G. J.Milne and P. A. Subrahmanyam, editors. Elsevier Scientific Publishers, 1986.

[116] Michael J. C. Gordon. HOL: a proof generating system for higher-order logic. *VLSI Specification, Verification and Synthesis* (Calgary, Canada), pages 73–128, Graham Birtwistle and P. A. Subrahamanyam, editors. Kluwer Academic Publishers, Boston, 1987.

[117] Mike Gordon. Proving a Computer Correct. Technical report 42. University of Cambridge Computer Laboratory, 1983. Year of publication uncertain.

[118] Mike Gordon. HOL: a Machine Oriented Formulation of Higher Order Logic. Technical report 68. University of Cambridge Computer Laboratory, May 1985.

[119] Brian Graham and Graham Birtwistle. Formalizing the design of an SECD chip. *Proceedings of of the Mathematical Sciences Institute Workshop on Hardware Specification, Verification, and Synthesis: Mathematical Aspects*. Cornell University, July 1989.

[120] Brian T. Graham. *The SECD Microprocessor: a Verification Case Study*. Kluwer Academic Publishers, Norwell, MA, 1992.

[121] Torbjörn Granlund and Richard Kenner. Eliminating branches using a superoptimizer and the GNU C compiler. *Proceedings of ACM SIGPLAN '92 Conference on Programming Language Design and Implementation*, page 341:352, June 1992.

[122] Evan O. Grossman. Intel will replace buggy 486s for free. *PC Week*, 4 December 1989. Abstracted in [80].

[123] A. Gupta. Formal hardware verification methods: a survey. *Formal Methods in System Design*, **1**(2/3):151–238, October 1992. Revised version of [124].

[124] Aarti Gupta. Formal Hardware Verification Methods: A Survey. Technical report CMU–CS–91–193. Department of Computer Science, Carnegie-Mellon University, Pittsburgh, PA, October 1991. Also published as [123].

[125] John Guttag, Jim Horning, and Jeannette Wing. Some notes on putting formal specifications to productive use. *Science of Computer Programming*, **2**:53–68, 1982.

[126] Gary D. Hachtel and Reily M. Jacoby. Verification algorithms for VLSI synthesis. *IEEECAD.*, **7**(5):616–40, May 1988.

[127] Anthony Hall. Seven myths of formal methods. *IEEE Software*, pages 11–19, September 1990.

[128] Larry Lewis Hanes. *Logic Design Verification Using Static Analysis.* PhD thesis, published as Coordinated Science Laboratory computer systems group report CSG-15. University of Illinois, Urbana, May 1983.

[129] F. K. Hanna and N. Daeche. Specification and verification using higher-order logic: a case study. *Formal Aspects of VLSI Design*, pages 179–213, G. J.Milne and P. A. Subrahmanyam, editors. Elsevier Science Publishers, 1986.

[130] F. K. Hanna, N. Daeche, and M. Longley. Veritas+: a specification language based on type theory. *Hardware Specification, Verification, and Synthesis: Mathematical Sciences Institute Workshop Proceedings* (Ithaca, NY, 5–7 July 1989), pages 358–79. Springer Verlag, Berlin, 1990.

[131] John P. Hayes. A unified switching theory with applications to VLSI design. *Proceedings of of the IEEE*, **70**(10):1140–51, October 1982.

[132] Frederick J. Hill and Gerald R. Peterson. *Introduction to Switching Theory and Logical Design*, 3rd edition. John Wiley, New York, 1981.

[133] C. A. R. Hoare. Proof of correctness of data representations. *Acta Informatica*, **1**:271–81, 1972.

[134] John E. Hopcroft and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation*, Addison-Wesley series in computer science. Addison-Wesley, Reading, Mass. and London, 1979.

[135] Alan J. Hu, David L. Dill, Andreas J. Drexler, and C. Han Yang. Higher-level specification and verification with BDDs. *Computer-Aided Verification: Fourth International Workshop* (Montreal, 29 June–1 July 1992), pages 82–95. Springer Verlag, Berlin, 1992.

[136] Warren A. Hunt, Jr. Microprocessor design verification. *Journal of Automated Reasoning*, **5**:429–60, 1989.

[137] Warren A. Hunt, Jr. *FM8501: a Verified Microprocessor.* PhD thesis. University of Texas, Austin, December 1985.

[138] Diana Hwang. Token-ring skirmish: reliability of Falcon 16M-bit chip debated. *Computer Reseller News*, 24 September 1990. Abstracted in [80].

[139] Seung Ho Hwang and A. Richard Newton. An efficient verifier for finite state machines. *IEEE Transactions on Computer-Aided Design*, **10**(3):326–34, March 1991.

[140] 486 bugs derail PC vendors' plans; systems with corrected version of Intel chip unlikely before 1990. *InfoWorld*, 30 October 1989. Abstracted in [80].

[141] Prabhat Jain and Ganesh Gopalakrishnan. Some techniques for efficient symbolic simulation-based verification. *International Conference on Computer Design*, 1992.

[142] Prabhat Jain, Prabhakar Kudva, and Ganesh Gopalakrishnan. Towards a verification technique for large synchronous circuits. *Computer-Aided Verification*, 1992.

[143] Stuart J. Johnston. 486-based languages join wait for bug-delayed chip. *InfoWorld*, 27 November 1989. Abstracted in [80].

[144] Jeff Joyce, Graham Birtwistle, and Mike Gordon. Proving a computer correct in higher order logic. Technical report 100. University of Cambridge Computer Laboratory, December 1986.

[145] Jeffrey J. Joyce. Formal verification and implementation of a microprocessor. *VLSI Specification, Verification and Synthesis* (Calgary, Canada), pages 129–57, Graham Birtwistle and P. A. Subrahamanyam, editors. Kluwer Academic Publishers, Boston, 1987.

[146] Jeffrey J. Joyce. Using higher-order logic to specify computer hardware and architecture. *Design Methodologies for VLSI and Computer Architecture: Proceedings of IFIP TC10 Working Conference* (Pisa, Italy, 19–21 September 1988), pages 129–45, D. A. Edwards, editor. North-Holland, Amsterdam, 1988.

[147] Jeffrey J. Joyce. More reasons why higher-order logic is a good formalism for specifying and verifying hardware. *Proceedings of of the ACM/SIGDA International Workshop in Formal Methods in VLSI Design*, January 1991.

[148] Jeffrey J. Joyce. More reasons why higher-order logic is a good formalism for specifying and verifying hardware. Technical report 90–35. Department of Computer Science, University of British Columbia, November 1990.

[149] Jeffrey J. Joyce and Carl-Johan H. Seger. Linging BDD-based symbolic evaluation to interactive theorem-proving. *30th Design Automation Conference* (Dallas, TX, 14–18 June 1993), pages 469–74, June 1993.

[150] Jeffrey John Joyce. *Multi Level Verification of Microprocessor-Based Systems*. PhD thesis, published as Technical report 195. University of Cambridge, Computer Laboratory, May 1990.

[151] Timothy Kam and P. A. Subrahmanyam. Comparing layouts with HDL models: a formal verification technique. *International Conference on Computer Design*, pages 588–91, 1992.

[152] Pagan Kennedy. R&D push allows Intel to ride high in laptop market: investment in research yields new technology. *PC Week*, 11 November 1991. Abstracted in [80].

[153] Steven Kovsky. Intel says it has pinpointed, fixed defect in 80486 chip. *Digital Review*, 6 November 1989. Abstracted in [80].

[154] R. P. Kurshan. Analysis of discrete event coordination. *REX Workshop* (Mook, Netherlands, May 29–June 2, 1989), pages 414–53, J.W. de Bakker, W.-P. de Roever, and G. Rozenberg, editors. Springer Verlag, Berlin, 1990.

[155] R. P. Kurshan and K. L. McMillan. Analysis of digital circuits through symbolic reduction. *IEEE Transactions on Computer-Aided Design*, **10**(11):1356–71, November 1991.

[156] A. K. Kutti. On a graphical representation of the operating regime of circuits. In Edward F. Moore, editor, *Sequential Machines: Selected Papers*, pages 228–35. Addison-Wesley, Reading, Mass. and London, 1964. Trans. by E. F. Moore of "O graficheskom izobrazhenii rabochego rezhima skhem," *Trudy Leningradskoĭ Ėksperimental'noĭ Ėlektrotekhnicheskoĭ Laboratorii*, Vol. 8 (1928), pp. 11–18.

[157] Leslie Lamport. An axiomatic semantics of concurrent programming languages. *Proceedings of the NATO Advanced Study Institute on Logics and Models of Concurrent Systems* (Colle-sur-Loup, France, 8–19 October 1984). Published as Krzysztof R. Apt, editor, *NATO ASI series, Series F, Computer and System Sciences*, **13**:77–144. Springer Verlag, Berlin, 1985.

[158] M. Langevin and E. Cerny. Verification of processor-like circuits. *Correct Hardware Design Methodologies: Proceedings of Advanced Research Workshop* (Turin, Italy, 12–14 June 1991), pages 141–62. North-Holland, Amsterdam, 1992.

[159] Timothy E. Leonard. Specification of computer architectures: a survey and annontated bibliography. Technical report 188. University of Cambridge Computer Laboratory, January 1990.

[160] Beth Levy. Microcode verification using SDVS—the method and a case study. *17th Anuual Microprogramming Workshop* (New Orleans, LA, 30 October–2 November 1984). Published as *ACM Sigmicro Newsletter*, **15**(4):234–45. IEEE Computer Society Press, December 1984.

[161] Paul Loewenstein. Reasoning about state machines in higher–order logic. *Workshop on Hardware Specification, Verification, and Synthesis: Mathematical Aspects*, LNCS., M. Leeser and G. Brown, editors. Springer Verlag, Berlin, 1989.

[162] Paul Loewenstein and David L. Dill. Verification of multiprocessor cache protocol using refinement relations and higher-order logic. *Computer-Aided Verification* (Rutgers, NJ), June 1990.

[163] David Long. *Model Checking, Abstraction, and Compositional Verification.* PhD thesis. Department of Computer Science, Carnegie-Mellon University, Pittsburgh, PA, 1993.

[164] Benny Lorenzo. Compaq: no escaping Wall Street's wrath. *Computer Reseller News*, 13 November 1989. Abstracted in [80].

[165] Nancy A. Lynch and Mark R. Tuttle. An introduction to input-output automata. Technical report MIT/LCS/TM-373. Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA, November 1988.

[166] Jean-Christophe Madre and Jean-Paul Billon. Proving circuit correctness using formal comparison between expected and extracted behavior. *25th Design Automation Conference*, pages 205–10, 1988.

[167] Zohar Manna. *Mathematical Theory of Computation*, Computer Science Series. McGraw-Hill, New York, 1974.

[168] Leo Marcus, Stephen D. Crocker, and Jaisook R. Landauer. SDVS: a system for verifying microcode correctness. *17th Annual Microprogramming Workshop* (New Orleans, LA, 30 October–2 November 1984). Published as *ACM Sigmicro Newsletter*, **15**(4):246–55, December 1984.

[169] John Markoff. I.B.M. to base a computer on powerful i486 Intel chip. *The New York Times*, 20 December 1989. Abstracted in [80].

[170] John Markoff. Top-of-line Intel chip is flawed; the design error will be fixed. *The New York Times*, 27 October 1989. Abstracted in [80].

[171] Vance McCarthy. Intel halts production of 50MHz 486 chips; PC makers advised to delay shipments. *PC Week*, 26 August 1991. Abstracted in [80].

[172] Michael C. McFarland. Formal verification of sequential hardware: a tutorial. *IEEE Transactions on Computer-Aided Design*, **12**(5):633–54, May 1993.

[173] Kenneth L. McMillan. *Symbolic Model Checking: An approach to the state explosion problem.* PhD thesis, published as Technical report CMU–CS–92–131. School of Computer Science, Carnegie-Mellon University, Pittsburgh, PA, May 1992.

[174] Carver A. Mead and Lynn A. Conway. *Introduction to VLSI Systems*. Addison-Wesley, Reading, Mass. and London, 1980.

[175] Thomas F. Melham. Abstraction mechanisms for hardware verification. *VLSI Specification, Verification and Synthesis* (Calgary, Canada), pages 267–91, Graham Birtwistle and P. A. Subrahamanyam, editors. Kluwer Academic Publishers, Boston, 1988.

[176] Thomas Frederick Melham. *Formalizing Abstraction Mechanisms for Hardware Verification in Higher Order Logic*. PhD thesis, published as Technical report 201. University of Cambridge, Computer Laboratory, August 1990.

[177] T. K. Miller, III. Personal communication, 18 June 1993. Telephone conversation regarding Hector.

[178] T. K. Miller, III. Personal communication, July 1989. Telephone conversation.

[179] T. K. Miller III, Bharat L. Bhuva, Russell L. Barnes, Jyy-Chiang Duh, Hsing-Bang Lin, and David E. Van den Bout. The HECTOR microprocessor. *International Conference on Computer Design*, pages 406–11, 1986.

[180] George J. Milne. The formal description and verification of hardware timing. *IEEE Transactions on Computers*, **40**(7):811–26, July 1991.

[181] Robin Milner. *A Calculus of Communicating Systems*, volume 92 of Lecture Notes in Computer Science. Springer Verlag, Berlin, 1980.

[182] Robin Milner. *Communication and Concurrency*, Prentice Hall international series in computer science. Prentice-Hall, Englewood Cliffs, NJ, 1989.

[183] Robin Milner. Elements of interaction. *Communications of the ACM*, **36**(1):78–89, January 1993.

[184] MIPS Co. R4000 bug list. *Microprocessor Report*, page 16, 30 October 1991.

[185] Edward F. Moore. Gedanken-experiments on sequential machines. In Shannon and McCarthy, editors, *Automata Studies*, volume 34 of Annals of Mathematical Studies, pages 129–53. Princeton University Press, 1956.

[186] Edward F. Moore. Bibliographic comments on sequential machines. In *Sequential Machines: Selected Papers*, pages 236–44. Addison-Wesley, Reading, Mass. and London, 1964.

[187] J. Strother Moore. A mechanically verified language implementation. *Journal of Automated Reasoning*, **5**:461–92, 1989.

[188] J. Strother Moore. System verification. *Journal of Automated Reasoning*, **5**:409–10, 1989.

[189] Motorola, Incorporated. *M68000 Family Programmer's Reference Manual*, 1989.

[190] Jannis Moutafis. Inmos to deliver Transputer: T9000 silicon had been delayed due to design flaw. *Electronic Engineering Times*, 13 July 1992. Abstracted in [80].

[191] Amar Mukherjee. *Introduction to nMOS and CMOS VLSI Systems Design*. Prentice-Hall, Englewood Cliffs, NJ, 1986.

[192] A. Richard Newton and Bryan T. Preas (Editors). *25 Years of Electronic Design Automation: A compendium of papers from the Design Automation Conference*. Association for Computing Machinery, 1988.

[193] Intel device has defect. *New York Times*, 30 Jan 1990. Abstracted in [80].

[194] John D. Oakley. *Symbolic Excution of Formal Machine Descriptions*. PhD thesis. Carnegie-Mellon University, Pittsburgh, PA, April 1979.

[195] Andrew Ould. Motorola confirms chip delays; revisions begin on 68040 batch. *PC Week*, 24 September 1990. Abstracted in [80].

[196] David Park. Concurrency and automata on infinite sequences. *Theoretical Computer Science: 5th GI-Conference* (Karlsruhe, 23–25 March 1981). Published as Peter Denssen, editor, *Lecture Notes in Computer Science*, **104**:167–83. Springer Verlag, Berlin, 1981.

[197] David A. Patterson. STRUM: structured microprogram development system for correct firmware. *IEEE Transactions on Computers*, **C–25**(10):974–85, October 1976.

[198] David A. Patterson and John L. Hennessy. *Computer Architecture: a Quantitative Approach*. Morgan Kaufmann Publishers, 1990.

[199] Laurence C. Paulson. *ML for the Working Programmer*. Cambridge University Press, 1991.

[200] Intel debugs 486 chip. *PC User*, 20 December 1989. Summarized in [80].

[201] Vijay Pitchumani and Edward P. Stabler. An inductive assertion method for register transfer level design verification. *IEEE Transactions on Computers*, **C-32**(12):1073–80, December 1983.

[202] Carl Pixley. Personal communication, 31 March 1993. Remarks on industry verification needs.

[203] Clive H. Pygott. NODEN-HDL: an engineering approach to hardware verification. *The Fusion of Hardware Design and Verification* (Glasgow, 4–6 July 1988), pages 211–29. Elsevier Science Publishers, 1988.

[204] J. Rees and W. Clinger (editors). The revised^3 report on the algorithmic language Scheme. *SIGPLAN Notices*, **21**(2), 1986.

[205] D. S. Reeves and M. J. Irwin. Fast methods for switch-level verification of MOS circuits. *IEEECAD.*, **CAD–6**(5):766–79, September 1987.

[206] Robert Ristelhueber. Intel snag on 486 hits IBM, Compaq. *Electronic News*, 30 October 1989. Abstracted in [80].

[207] J. Paul Roth. VLSI verification and correction. In Tosiyasu L.Kunii, editor, *VLSI Engineering: Beyond Software Engineering*, pages 174–6. Springer Verlag, Berlin, 1984.

[208] J. Paul Roth. Hardware verification. *IEEE Transactions on Computers*, **C–26**(12):1292–4, December 1977.

[209] Walter S. Scott, Robert N. Mayo, Gordon Hamachi, and John K. Ousterhout (editors). 1986 VLSI tools: still more works by the original artists. Technical report UCB/CSD 86/272. Computer Science Division, University of California, Berkeley, December 1985.

[210] Carl-Johan Seger. A bounded delay race model. *International Conference on Computer-Aided Design*, pages 130–3, 1989.

[211] Carl-Johan Seger and Randal E. Bryant. Modeling of circuit delays in symbolic simulation. *Applied Formal Methods for VLSI* (Leuven, Belgium), pages 625–39, 1989.

[212] Carl-Johan H. Seger and Randal E. Bryant. DRAFT: Formal verification by symbolic evaluation of partially-ordered trajectories. *Formal Methods in System Design*, 1993. Submitted for publication; manuscript obtained from authors.

[213] R. C. Sekar and M. K. Srivas. Formal verification of a microprocessor using equational techniques. *Current Trends in Hardware Verification and Automated Theorem Proving* (Banff, Canada, 12–18 June 1988), G. Birtwistle and P. A. Subrahmanyam, editors. Springer Verlag, Berlin, 1989.

[214] Andrew M. Seybold. Morality and the computer industry. *Andrew Seybold's Outlook on Professional Computing*, November 1989. Abstracted in [80].

[215] Mary Lee Shalvoy and Kristen Hedlund. Intel: 486 is flawed; concession's timing causes market anxiety. *Computer Reseller News*, 30 October 1989. Abstracted in [80].

[216] Eben Shapiro. Shipment of new chip has begun at Motorola. *The New York Times*, 27 November 1990. Abstracted in [80].

[217] Lou Sheffer. Personal communication, 17 May 1993. Question session following presentation.

[218] Intel resumes 486 shipments. *Electronic News*, 4 December 1989. Abstracted in [80].

[219] Stephen Slade. *The T Programming Language: a Dialect of Lisp*. Prentice-Hall, Englewood Cliffs, NJ, 1987.

[220] Michael Slater. Swatting 386 bugs. *ESD: The Electronic System Design Magazine*, September 1989. Abstracted in [80].

[221] Rebecca Smith. Intel to delay rollout of next-generation microprocessor: introduction of P5 chip put off until '93. *San Jose Mercury News*, 23 July 1992. Abstracted in [80].

[222] Stewart G. Smith and Peter B. Denyer. *Serial-Data Computation*. Kluwer Academic Publishers, Boston, 1988.

[223] Tracey Snell. BUG bites EISA PCs: suppliers of EISA machines are suffering from bug in 80486 chip. *PC User*, 28 February 1990. Abstracted in [80].

[224] Lisa L. Spiegelman. Intel halts shipments of 50MHz 486 processor. *Computer Reseller News*, 26 August 1991. Abstracted in [80].

[225] Lisa L. Spiegelman. PC makers ignore 486 bugs, ready 33-MHz 486s. *PC Week*, 5 February 1990. Summarized in [80].

[226] Nagendra C. E. Srinivas and Vishwanie D. Agrawal. Formal verification of digital circuits using hybrid simulation. *IEEE Circuits and Devices*, **4**(1):19–27, 1988.

[227] Mandayam Srivas and Mark Bickford. Formal verification of a pipelined microprocessor. *IEEE Software*, **7**(5):52–64, September 1990.

[228] V. Stavridou, H. Barringer, and D. A. Edwards. Formal specification and verification of hardware: a comparative case study. *25th Design Automation Conference*, pages 197–204, 1988.

[229] Victoria Stavridou, Howard Barringer, and Doug Edwards. Formal specification and verification of hardware: a comparative case study. Technical report UMCS–87–11-1. Department of Computer Science, University of Manchester, 1987.

[230] Elizabeth A. Stein. 040 production up to speed. *Computer Design*, 14 January 1991. Abstracted in [80].

[231] Ben G. Streetman. *Solid State Electronic Devices*, Prentice-Hall series in solid state physical electronics, 2nd edition. Prentice-Hall, Englewood Cliffs, NJ, 1980.

[232] Kenneth J. Supowit and Steven J. Friedman. A new method for verifying sequential circuits. *23rd Design Automation Conference*, pages 200–5, 1986.

[233] Christer Svensson and Robert Tjärnström. Switch-level simulation and the pass transistor exor gate. *IEEE Transactions Computer-Aided Design*, **7**(9):994–7, September 1988.

[234] H. Touati, H. Savoj, B. Lin, R. Brayton, and A. Sangiovanni-Vincentelli. Implicit state enumeration of finite state machines using BDD's. *Proceedings of International Conference on Computer-Aided Design*, pages 130–3, November 1990.

[235] D. A. Turner. A new implementation technique for applicative languages. *Software—Practice and Experience*, **9**:31–49, 1979.

[236] Filip Van Aelten. *Automatic Procedures for the Behavioral Verification of Digital Designs*. PhD thesis. Massachusetts Institute of Technology, Cambridge, MA, May 1992.

[237] Filip Van Aelten, Jonathan Allen, and Srinivas Devadas. Verification of relations between synchronous machines. *IEEE Transactions Computer-Aided Design*, 1993. Accepted for publication.

[238] T. J. Wagner. *Hardware verification*. PhD thesis. Stanford University, 1977.

[239] Jr Warren A. Hunt and Bishop C. Brock. A formal HDL and its use in the FM9001 verification. Technical report 79. Computational Logic, Incorporated, Austin, Texas, July 1992.

[240] Daniel Weise. Constraints, abstraction, and verification. *Hardware specification, verification, and synthesis: mathematical aspects*, number 408 in Lecture Notes in Computer Science, Miriam Leeser and Geoffrey Brown, editors. Springer Verlag, Berlin, 1989.

[241] Daniel Weise. Multilevel verification of MOS circuits. *IEEE Transactions on Computer-Aided Design*, **9**(4):341–51, April 1990.

[242] E. P. Wentworth. Pitfalls of conservative garbage collection. *Software—Practice and Experience*, **20**(7):719–27, July 1990.

[243] Neil H. E. Weste and Kamran Eshraghian. *Principles of CMOS VLSI Design: A Systems Perspective*, VLSI Systems series. Addison-Wesley, Reading, Mass. and London, 1985.

[244] Phillip J. Windley. *The Formal Verification of Generic Interpreters*. PhD thesis. University of California, Davis, Division of Computer Science, June 1990. Citation obtained from author.

[245] Phillip J. Windley. A hierarchical methodology for the verification of microprogrammed microprocessors. *IEEE Computer Society Symposium on Research in Security and Privacy* (Oakland, CA, 7–9 May 1990), pages 345–57, May 1990.

[246] Jeanette M. Wing. A specifier's introduction to formal methods. *IEEE Computer*, **23**(9):8–24, September 1990.

[247] David Winkel and Franlkin Prosser. *The Art of Digital Design*. Prentice-Hall, Englewood Cliffs, NJ, 1980.

[248] Loring Wirbel. Intel swats bug; users of 486 get Halloween scare. *Electronic Engineering Times*, 30 October 1989. Abstracted in [80].

[249] Intel Corp. ties flaw in a new 486 chip to inadequate tests. *Wall Street Journal*, 28 August 1991. Abstracted in [80].

[250] Another 'bug' in Intel chip may delay few shipments. *Wall Street Journal*, pages B, 10:4, 30 January 1990. Abstracted in [80].

[251] Stephen Kreider Yoder. Intel delays debut of P-5 chip to refine production process and eliminate bugs. *Wall Street Journal*, 23 July 1992. Abstracted in [80].

[252] Stephen Kreider Yoder. Chip by Intel contains flaw in calculating; 'bugs' could stall makers of certain computers; most users unaffected. *Wall Street Journal*, 27 October 1989. Abstracted in [80].

[253] Stephen Kreider Yoder. Chip delay at Motorola sparks ire. *Wall Street Journal*, 7 September 1990. Abstracted in [80].

[254] Yuan Yu. *Automated Proofs of Object Code for a Widely Used Microprocessor*. PhD thesis. University of Texas at Austin, December 1992.

[255] William Zachmann. Watch out for those first Intel 486-based machines. *PC Week*, **6**(11), 6 November 1989. Abstracted in [80].

[256] Pamela Zave. Seminar, 2 March 1992. Programming Systems seminar at CMU.

[257] C. Zhou and C. A. R. Hoare. A model for synchronous switching circuits and its theory of correctness. *Formal Methods in System Design*, **1**(1):7–28, July 1992. Citation from Inspec.

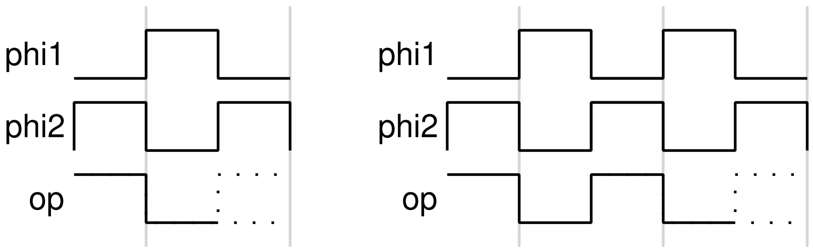
[258] Han Zuidweg. Verification by abstraction and bisimulation. *Automatic Verification Methods for Finite State Systems* (Grenoble, June 1989). Published as Joseph Sifakis, editor, *Lecture Notes in Computer Science*, **407**:105–16. Springer Verlag, Berlin, 1990.

Appendix A

A theory of marked strings

Here we define a notion of marked strings suitable for reasoning about overlapping state sequences, such as in a pipelined system. The development is rather formal. It makes use of no more than elementary lattice theory, such as that found in Chapter 3 of Brzozowski and Yoeli's text [60]. We begin with a short review of the reason for developing the formalism.

A.1 Motivation



a. Timing diagram b. Two overlapped copies

Figure A.1: Overlap of timing diagram

The basic reason to define marked strings is to provide a formal model analogous to the controlled, aligned overlapping of two timing diagrams. Consider as an example the idealized timing diagram shown in Figure A.1a. Three signals are constrained during the first two intervals of time, but during the final, third interval of time, only the first two signals are constrained. Suppose that this timing diagram represents an operation that some circuit performs. Two light gray vertical lines indicate the beginning and ending of this operation. Notice that although we want to say that the operation begins at the first gray line, some the signals have already been determined during the interval leading up to this instant. We wish to be able to determine the representation for the operation performed twice in succession.

Figure A.1b represents the repetition of the operation—the first diagram twice in succession. The vertical lines show the times when the first operation begins, when the first ends and the second begins, and when the second ends. We will formalize the construction of this diagram from copies of the previous one. Of course, if we were to combine still more copies, we should get a larger diagram, but the final result should not depend on the order in which we put together the smaller pieces.

Suppose that the letter a represents the combination of values[1] $(0, 1, 1)$, that the letter b represents $(1, 0, 0)$, and the letter c represents $(0, 1, 0)$. Furthermore, suppose that the prime symbol, $'$, represents the gray marker line. Then the first diagram can be represented by the set of strings $\{ a'bc', a'ba' \}$ and the second can be represented by $\{ a'ba'bc', a'ba'ba' \}$. The following sections develop a theory that allows us to do this, and to find the representation of the second from two copies of the representation of the first, using an operation we call "overlapped concatenation."

A.2 Basic definitions

Let A be an alphabet. Define A'^* to be the set of marked strings over A, where a marked string is a string over $A' = A \cup \{ \prime \}$ and the symbol $\prime$ (to be read "mark") does not appear in alphabet A. The symbol ϵ denotes the empty string. Note that $\epsilon \in A'^*$.

We will need an error indicator, which we denote by $\top$. The functions we define will be strict with respect to $\top$. Thus we will consider functions over the universe $A'^* \cup \{ \top \}$. Without further mention we will abuse notation and write A'^* when strictly speaking we mean $A'^* \cup \{ \top \}$. We express the usual concatenation by adjacency, but we let it be strict on $\top$. That is, $x\top = \top = \top x$ whenever $x \in A'^*$.

Definition 2 (p. 53) **(Marked strings)** *The set A'^* of marked strings over alphabet A is defined inductively by the equations:*

$$
\begin{aligned}
\epsilon &\in A'^* \\
\top &\in A'^* \\
\prime x &\in A'^* \quad \textit{for } x \in A'^* \\
ax &\in A'^* \quad \textit{for } a \in A \textit{ and } x \in A'^*
\end{aligned}
$$

For example, ϵ (the empty string) and a and abc and $a'bc'$ and $a'''b$ and $''''$ are all marked strings, and $\top$ (the error indicator) is also a marked string.

[1] on the nodes phi1, phi2, and op respectively

The formal definition provides a schema of structural induction to use in subsequent proofs. Most of our proofs involving marked strings use this schema, by either ordinary or complete induction.

In the following definitions, statements, and proofs, letters from the beginning of the (English) alphabet (i.e., a, b) denote distinct symbols from A, and letters from the end of the alphabet (i.e., x, y, z) denote arbitrary marked strings already known to be in A'^*.

We will need prefix and suffix-related properties of marked strings. There is a strong symmetry between these sets of properties. We use this by giving full details of prefix properties and letting suffix properties follow by symmetry according to the following definition. The discussions accompanying the formal mathematics in this section generally omit mention of the symmetric case.

Definition 3 (p. 54) **(Reversal)** *If $\hat{x}$ is a marked string, its reversal $\hat{x}^R$ is given inductively by the equations:*

$$\begin{aligned} \epsilon^R &= \epsilon \\ \top^R &= \top \\ ({\prime}x)^R &= \left(x^R\right){\prime} \\ (ax)^R &= x^R a \end{aligned}$$

For example, $(a'bc')^R = {'cb'a}$.

A.3 Ordering and lattice properties

Definition 4 (p. 54) **(Marked prefix and suffix orders)** *The relation $\sqsubseteq_p$ is given by induction.*

$$\begin{aligned} \epsilon &\sqsubseteq_p x \\ x &\sqsubseteq_p \top \\ ax &\sqsubseteq_p ay \quad \textit{whenever } x \sqsubseteq_p y \\ {\prime}x &\sqsubseteq_p {\prime}y \quad \textit{whenever } x \sqsubseteq_p y \\ x &\sqsubseteq_p {\prime}y \quad \textit{whenever } x \sqsubseteq_p y \end{aligned}$$

The relation $x \sqsubseteq_s y$ is defined to hold exactly when $x^R \sqsubseteq_p y^R$ does.

For example, the relations $\epsilon \sqsubseteq_p a$ and $a \sqsubseteq_p ab$ and $ab \sqsubseteq_p a'b$ all hold.

The relation $\sqsubseteq_p$ is similar to the familiar prefix ordering of strings. The innovation here is the addition of the marker. Intuitively, inserting markers anywhere in a string produces a larger one. This relation does also have the expected prefix property.

Proposition 21 *If x is a prefix of y, that is, $y = xz$ for some z, then $x \sqsubseteq_p y$. If x is a suffix of y, that is, $y = zx$ for some z, then $x \sqsubseteq_s y$.*

Most proofs in this appendix are by complete structural induction, where we show a property for $\hat{x}$ from the inductive hypothesis that it holds for any x having fewer symbols. These inductions are generally given in tables, where each case occupies a line. The column "pf. by" justifies the case; "insp." indicates that it can be seen by inspection without the inductive hypothesis; "ind." indicates that the inductive hypothesis was used. In each we show only one result; the other follows by symmetry.

Proof: The prefix result can be seen by structural induction, according to the lines of the following table. The third line follows because concatenation is strict with respect to $\top$.

$\hat{x}$	$\hat{x} \sqsubseteq_p \hat{x}y$	pf. by
ϵ	$\epsilon \sqsubseteq_p y$	insp.
$\top$	$\top \sqsubseteq_p \top y$	strict
$\prime x$	$\prime x \sqsubseteq_p \prime xy$	ind.
ax	$ax \sqsubseteq_p axy$	ind.

■

Lemma 22 *The relations $\sqsubseteq_p$ and $\sqsubseteq_s$ are reflexive, antisymmetric, and transitive.*

Proof: By structural induction. Reflexivity and anti-symmetry can be seen according to the lines of the following tables, respectively.

$\hat{x} \sqsubseteq_p \hat{x}$	pf. by
$\epsilon \sqsubseteq_p \epsilon$	def.
$\top \sqsubseteq_p \top$	def.
$\prime x \sqsubseteq_p \prime x$	ind.
$ax \sqsubseteq_p ax$	ind.

$\hat{x}$	$\hat{y}$	$\hat{x} \overset{?}{\sqsubseteq_p} \hat{y}$	$\hat{y} \overset{?}{\sqsubseteq_p} \hat{x}$	pf. by
ϵ	y	true	if $y = \epsilon$	insp.
x	$\top$	true	if $x = \top$	insp.
ax	ay	if $x \sqsubseteq_p y$	if $x \sqsubseteq_p y$	ind.
$\prime x$	$\prime y$	if $x \sqsubseteq_p y$	if $x \sqsubseteq_p y$	ind.
x	$\prime y$	if $x \sqsubseteq_p y$	if $x = \prime y$	insp.

Transitivity can be seen from the following table.

$\hat{x}$	$\hat{y}$	$\hat{y}$	$\hat{z}$	assumptions	pf. by
ϵ	y	ϵ	z	$y = \epsilon$	insp.
ϵ	y	y	$\top$		insp.
ϵ	y	ay	az	n/a	
ϵ	y	$'y$	$'z$	n/a	
ϵ	y	y	$'z$		insp.
x	$\top$	ϵ	z	n/a	
x	$\top$	y	$\top$	$y = \top$	insp.
x	$\top$	ay	az	n/a	
x	$\top$	$'y$	$'z$	n/a	
x	$\top$	y	$'z$	$y = \top \sqsubseteq_p z$:n/a	
ax	ay	ϵ	z	n/a	
ax	ay	y	$\top$	n/a	
ax	ay	ay	az	$x \sqsubseteq_p y$,$y \sqsubseteq_p z$	ind.,def. $\sqsubseteq_p$
ax	ay	$'y$	$'z$	n/a	
ax	ay	y	$'z$	n/a	
$'x$	$'y$	ϵ	z	n/a	
$'x$	$'y$	y	$\top$	n/a	
$'x$	$'y$	ay	az	$a = '$	next
$'x$	$'y$	$'y$	$'z$	$x \sqsubseteq_p y$,$y \sqsubseteq_p z$	ind.,def.
$'x$	$'y$	y	$'z$	n/a	
x	$'y$	ϵ	z	n/a	
x	$'y$	y	$\top$	n/a	
x	$'y$	ay	az	$a = '$	next
x	$'y$	$'y$	$'z$	$x \sqsubseteq_p y$,$y \sqsubseteq_p z$	ind.,def.
x	$'y$	y	$'z$	n/a	

■

Theorem 23 *The structures* $\langle A'^*, \sqsubseteq_p\rangle$ *and* $\langle A'^*, \sqsubseteq_s\rangle$ *are posets.*

Proof: Immediate from Lemma 22 and the definition of a poset. ■

Definition 5 (p. 55) **(Marked prefix and suffix joins)** *The binary operator* $\sqcup_p$

is given inductively.

$$x \sqcup_p \top = \top$$
$$\top \sqcup_p x = \top$$
$$\epsilon \sqcup_p x = x$$
$$x \sqcup_p \epsilon = x$$
$$'x \sqcup_p 'y = '(x \sqcup_p y)$$
$$'x \sqcup_p y = '(x \sqcup_p y)$$
$$x \sqcup_p 'y = '(x \sqcup_p y)$$
$$ax \sqcup_p ay = a(x \sqcup_p y)$$
$$ax \sqcup_p by = \top$$

The operator $x \sqcup_s y$ *is defined as* $(x^R \sqcup_p y^R)^R$.

Observe that we have covered the entire set A'^*.

For example, $a'b \sqcup_p a'b = a'b$ while $aa'b \sqcup_p a'b = \top$. On the other hand, $a'bb \sqcup_p = a'b = a'bb$. Finally, $'abb' \sqcup_p a'b' = 'a'b'b'$.

The operator $\sqcup_p$ is similar to the least upper bound on prefix ordering. Again, the innovation is the marker. In the ordinary prefix ordering, the least upper bound of two strings is the longer of the two, if the shorter is its prefix, and is $\top$ otherwise. Adding markers to the ordering requires inserting of the markers from both of the operand strings. In fact, the same correspondence between joins and associated orders holds with the markers included.

We can also define the prefix meet of marked strings.

Lemma 24 *The ordering* $\hat{x} \sqsubseteq_p \hat{y}$ *holds precisely when* $\hat{x} \sqcup_p \hat{y}$ *is equal to* $\hat{y}$*. The ordering* $\hat{x} \sqsubseteq_s \hat{y}$ *holds precisely when* $\hat{x} \sqcup_s \hat{y}$ *is equal to* $\hat{y}$*.*

Proof: By structural induction as in the following table, using the first applicable line.

$\hat{x}$	$\hat{y}$	$\hat{x} \sqcup_p \hat{y}$	$\hat{x} \overset{?}{\sqsubseteq}_p \hat{y}$	$\hat{x} \sqcup_p \hat{y} \overset{?}{=} \hat{y}$
ϵ	y	y	true	true
x	ϵ	x	iff $x = \epsilon$	iff $x = \epsilon$
$\top$	y	$\top$	iff $y = \top$	iff $y = \top$
x	$\top$	$\top$	true	true
$'x$	$'y$	$'(x \sqcup_p y)$	iff $x \sqsubseteq_p y$	iff $x \sqcup_p y = y$
$'x$	y	$'(x \sqcup_p y)$	false	false
x	$'y$	$'(x \sqcup_p y)$	iff $x \sqsubseteq_p y$	iff $x \sqcup_p y = y$
ax	ay	$a(x \sqcup_p y)$	iff $x \sqsubseteq_p y$	iff $x \sqcup_p y = y$
ax	by	$\top$	false	false

■

We can define meet and prove an analogous meet-order result.

Definition 31 (Marked prefix and suffix meets) *The binary operator $\sqcap_p$ is given inductively.*

$$\begin{aligned}
\epsilon \sqcap_p y &= \epsilon \\
x \sqcap_p \epsilon &= \epsilon \\
\top \sqcap_p y &= y \\
x \sqcap_p \top &= x \\
{}'x \sqcap_p {}'y &= {}'(x \sqcap_p y) \\
{}'x \sqcap_p y &= x \sqcap_p y \\
x \sqcap_p {}'y &= x \sqcap_p y \\
ax \sqcap_p ay &= a(x \sqcap_p y) \\
ax \sqcap_p by &= \epsilon
\end{aligned}$$

The operator $x \sqcap_s y$ is defined as $(x^R \sqcap_p y^R)^R$.

Lemma 25 *The ordering $\hat{x} \sqsubseteq_p \hat{y}$ holds precisely when $\hat{x} \sqcap_p \hat{y}$ is equal to $\hat{x}$. The ordering $\hat{x} \sqsubseteq_s \hat{y}$ holds precisely when $\hat{x} \sqcap_s \hat{y}$ is equal to $\hat{x}$.*

Proof: By structural induction as in the following table, using the first applicable line.

$\hat{x}$	$\hat{y}$	$\hat{x} \sqcap_p \hat{y}$	$\hat{x} \overset{?}{\sqsubseteq_p} \hat{y}$	$\hat{x} \sqcap_p \hat{y} \overset{?}{=} \hat{x}$
ϵ	y	ϵ	true	true
x	ϵ	ϵ	if $x = \epsilon$	if $x = \epsilon$
$\top$	y	y	if $y = \top$	if $y = \top$
x	$\top$	x	true	true
${}'x$	${}'y$	${}'(x \sqcap_p y)$	if $x \sqsubseteq_p y$	if $x = x \sqcap_p y$
${}'x$	y	$x \sqcap_p y$	false	false
x	${}'y$	$x \sqcap_p y$	if $x \sqsubseteq_p y$	if $x = x \sqcap_p y$
ax	ay	$a(x \sqcap_p y)$	if $x \sqsubseteq_p y$	if $x = x \sqcap_p y$
ax	by	ϵ	false	false

■

Lemma 26 *The structures $\langle A'^*, \sqsubseteq_p \rangle$ and $\langle A'^*, \sqsubseteq_s \rangle$ are lattice ordered sets.*

Proof: From theorem 23 (posets) and lemmas 24 and 25 (relating lub/glb to meet/join). ■

Theorem 1 (p. 55) *The structures $\langle A'^*, \sqcup_p, \sqcap_p \rangle$ and $\langle A'^*, \sqcup_s, \sqcap_s \rangle$ are lattices.*

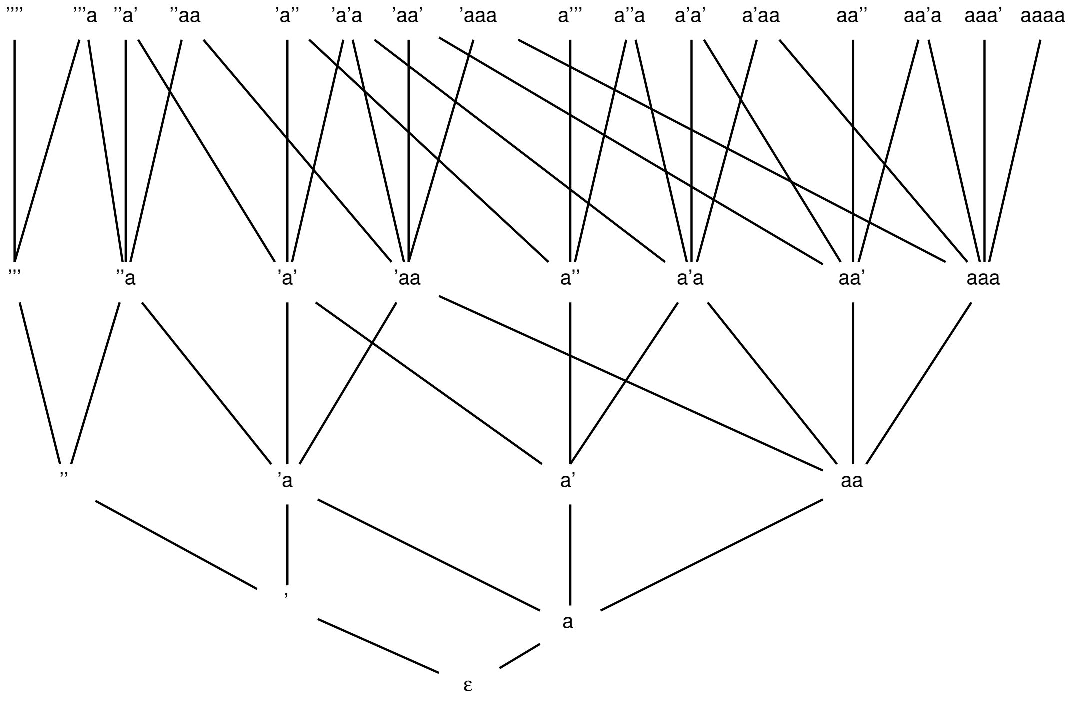


Figure A.2: Hasse diagram for the beginning of the marked prefix ordering for a one-symbol alphabet

Proof: By lemmas 24, 25, and 26, and some elementary lattice theory. ■

Corollary 27 *Idempotent, commutative, associative, and absorptive laws hold in the structure* $\langle A'^*, \sqcup_p, \sqcap_p \rangle$ *and the structure* $\langle A'^*, \sqcup_p, \sqcap_p \rangle$.

The visually-oriented reader may gain some appreciation for the ordering from the Hasse diagrams of Figures A.2 and A.3.

A.4 An overlapped concatenation operator

Definition 32 (Clean length of marked string) *The clean length of a marked*

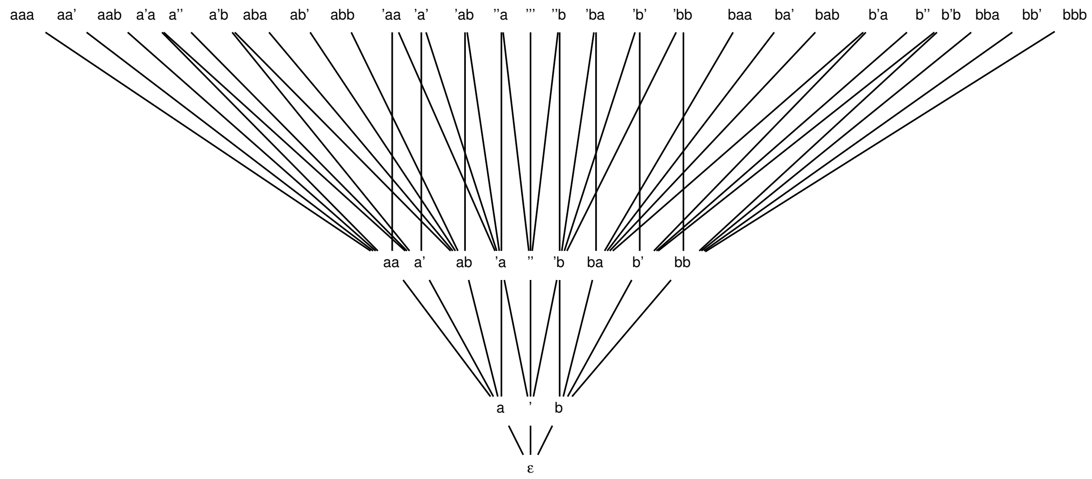


Figure A.3: Hasse diagram for the beginning of the marked prefix ordering for a two-symbol alphabet

string, which is its length discounting any marks, is defined inductively:

$$|\epsilon| = 0$$
$$|\top| = \infty$$
$$|{\prime}x| = |x|$$
$$|ax| = 1 + |x|$$

For example, $|a| = 1$ and $|a'| = 1$ and $|a'bc'| = 3$.

We can easily see that length behaves as expected with respect to concatenation.

Proposition 28 *If x and y are marked strings, $|xy| = |x| + |y|$.*

Proof:

$\hat{x}$	$\hat{y}$	$\lvert\hat{x}\rvert$	$\lvert\hat{y}\rvert$	$\lvert\hat{x}\hat{y}\rvert$	pf. by
ϵ	ϵ	0	0	0	insp.
ϵ	y	0	$\lvert y\rvert$	$\lvert y\rvert$	insp.
x	ϵ	$\lvert x\rvert$	0	$\lvert x\rvert$	insp.
$\top$	$\top$	∞	∞	∞	insp.
$\top$	y	∞	$\lvert y\rvert$	∞	insp.
x	$\top$	$\lvert x\rvert$	∞	∞	insp.
${\prime}x$	y	$\lvert x\rvert$	$\lvert y\rvert$	$\lvert xy\rvert$	ind.
x	${\prime}y$	$\lvert x\rvert$	$\lvert y\rvert$	$\lvert xy\rvert$	ind.
ax	y	$1 + \lvert x\rvert$	$\lvert y\rvert$	$1 + \lvert xy\rvert$	ind.
x	by	$\lvert x\rvert$	$1 + \lvert y\rvert$	$1 + \lvert xy\rvert$	ind.

■

Definition 33 (Clean and really marked) *A marked string is* really marked *if it is $\top$, or if it contains at least one occurrence of the ′ marker. Otherwise it is* clean.

For example, abc is clean, and $a'bc'$ is really marked.

Proposition 29 *If x and y are clean and $x \sqcup_p y \neq \top$ then $x \sqcup_p y$ is clean. If x and y are clean and $x \sqcup_s y \neq \top$ then $x \sqcup_s y$ is clean.*

Proof: By structural induction according to the table.

$\hat{x}$	$\hat{y}$	$\hat{x} \sqcup_p \hat{y}$	pf. by
ϵ	y	y	hyp.
x	ϵ	x	hyp.
$\top$	y		n/a
x	$\top$		n/a
$'x$	$'y$		n/a
$'x$	y		n/a
x	$'y$		n/a
ax	ay	$a(x \sqcup_p y)$	ind.
ax	by	$\top$	n/a

■

We can break marked strings into convenient pieces at the first or last marker.

Definition 6 (p. 55) **(Clean prefix and suffix)** *If $\hat{x}$ is a marked string, its clean prefix (or "first" part) $\mathrm{F}(\hat{x})$ is given inductively:*

$$\begin{aligned}
\mathrm{F}(\epsilon) &= \epsilon \\
\mathrm{F}(\top) &= \top \\
\mathrm{F}(ax) &= a\,\mathrm{F}(x) \\
\mathrm{F}('x) &= \epsilon
\end{aligned}$$

If $\hat{x}$ is a marked string, its clean suffix (or "last" part) $\mathrm{L}(\hat{x})$ is defined to be $(\mathrm{F}(\hat{x}^R))^R$.

Definition 7 (p. 56) **(Marked suffix and prefix)** *If $\hat{x}$ is a marked string, its marked suffix (the "rest" left after removing the "first" part and the demarcating* *marker)* $\mathrm{FR}(\hat{x})$ *is given inductively:*

$$
\begin{aligned}
\mathrm{FR}(\epsilon) &= \epsilon \\
\mathrm{FR}(\top) &= \top \\
\mathrm{FR}(\prime x) &= x \\
\mathrm{FR}(ax) &= \mathrm{FR}(x)
\end{aligned}
$$

If $\hat{x}$ *is a marked string, its marked prefix (the "rest" left after removing the "last" part and marker)* $\mathrm{LR}(\hat{x})$ *is defined to be* $(\mathrm{FR}(\hat{x}^R))^R$*.*

For example, the equations $\mathrm{F}(a'b'c) = a$ and $\mathrm{L}(a'b'c) = c$ and $\mathrm{FR}(a'b'c) = b'c$ and $\mathrm{LR}(a'b'c) = a'b$ all hold.

Observe however that F and FR are not isotone over[2] $\sqsubseteq_p$, and neither are L and LR over $\sqsubseteq_s$. For example, $a \sqsubseteq_p \prime a$ but $\mathrm{F}(a) = a \not\sqsubseteq_p \epsilon = \mathrm{F}(\prime a)$.

Proposition 30 *If* x *is clean, then* $\mathrm{FR}(x) = \epsilon$ *and* $\mathrm{LR}(x) = \epsilon$*.*

Proof: Induction, noting that the case $\prime x$ does not occur. ■

Proposition 31 $\mathrm{F}(xy) = \begin{cases} \mathrm{F}(x) & \text{if } x \text{ marked and } y \neq \top \\ x\,\mathrm{F}(y) & \text{if } x \text{ clean or } y = \top \end{cases}$
and $\mathrm{L}(xy) = \begin{cases} \mathrm{L}(y) & \text{if } y \text{ marked and } x \neq \top \\ \mathrm{L}(x)y & \text{if } y \text{ clean or } x = \top \end{cases}$

For example, $\mathrm{F}(a'bc') = a$ and $\mathrm{L}(a'bc') = \epsilon$ and $\mathrm{FR}(a'bc') = bc'$ and $\mathrm{LR}(a'bc') = a'bc$. For clean strings, $\mathrm{F}(abc) = abc$ and $\mathrm{FR}(abc) = \epsilon$.
Proof: By induction on x according to the cases in the following table.

$\hat{x}$	$\hat{x}y$	$\mathrm{F}(\hat{x}y)$	pf. by
ϵ	y	$\mathrm{F}(y)$	insp.
$\top$	$\top$	$\top$	insp.
$\prime x$	$\prime xy$	ϵ	insp.
ax	axy	$a\,\mathrm{F}(xy)$	ind.

■

Corollary 32 *If* x *is not* $\top$ *then* $\mathrm{F}(x)$ *is clean and* $\mathrm{L}(x)$ *is clean.*

Proposition 33 *If* x *is not* $\top$ *then* $\mathrm{F}(x\prime y) = \mathrm{F}(x)$*, and if* y *is not* $\top$ *then* $\mathrm{L}(x\prime y) = \mathrm{L}(y)$*.*

[2] i.e., do not respect the order

Proof: By applying proposition 31 and the definition of F. ■

Proposition 34 $\mathrm{FR}(xy) = \begin{cases} \mathrm{FR}(y) & \textit{if } x \textit{ is clean} \\ \mathrm{FR}(x)y & \textit{if not} \end{cases}$
and $\mathrm{LR}(xy) = \begin{cases} \mathrm{LR}(x) & \textit{if } y \textit{ is clean} \\ x\,\mathrm{LR}(y) & \textit{if not.} \end{cases}$

Proof: By induction on x according to the following table. The first two lines give the clean case; the last three, the marked.

$\hat{x}$	$\hat{x}y$	$\mathrm{FR}(\hat{x}y)$	pf. by
ϵ	y	$\mathrm{FR}(y)$	insp.
ax	axy	$\mathrm{FR}(xy) = \mathrm{FR}(y)$	insp.
$\top$	$\top y$	$\mathrm{FR}(\top)$	insp.
$\prime x$	$\prime xy$	$xy = \mathrm{FR}(\prime x)y$	ind.
ax	axy	$\mathrm{FR}(xy) = \mathrm{FR}(x)y$	ind.

■

Proposition 35 *If x is really marked, then the equations* $\mathrm{F}(x) = \mathrm{F}(\mathrm{LR}(x))$ *and* $\mathrm{L}(x) = \mathrm{L}(\mathrm{FR}(x))$ *both hold.*

Proof: By induction on x according to the cases in the following table. Recall that by hypothesis x is really marked.

$\hat{x}$	$\mathrm{F}(\hat{x})$	$\hat{x}^R$	$\mathrm{FR}\left(\hat{x}^R\right)$	$\mathrm{LR}(\hat{x})$	pf. by
ϵ	ϵ	ϵ	ϵ	ϵ	insp.
$\top$	$\top$	$\top$	$\top$	$\top$	insp.
$\prime x$	ϵ	$x^R\prime$	$\mathrm{FR}\left(x^R\right)\prime$	$\prime\,\mathrm{LR}(x)$	insp.
ax	$a\,\mathrm{F}(x)$	$x^R a$	$\mathrm{FR}\left(x^R\right)a$	$a\,\mathrm{LR}(x)$	ind.

■

Proposition 36 *If x is really marked, then* $\mathrm{F}(x) \sqsubseteq_p \mathrm{LR}(x)$ *and* $\mathrm{L}(x) \sqsubseteq_s \mathrm{FR}(x)$.

Proof: By structural induction according to the table in the proof of proposition 35, except that the third line is justified inductively. ■

Proposition 37 *The functions* FR *and* LR *commute. That is,* $\mathrm{FR}(\mathrm{LR}(x)) = \mathrm{LR}(\mathrm{FR}(x))$.

For example, $\mathrm{FR}(\mathrm{LR}(a'bc')) = \mathrm{FR}(a'bc) = bc$ and $\mathrm{LR}(\mathrm{FR}(a'bc')) = \mathrm{LR}(bc') = bc$.

Proof: First we expand LR by its definition, then use structural induction. The expansion is $\mathrm{FR}(\mathrm{FR}^R(x^R)) = \mathrm{FR}^R(\mathrm{FR}^R(x))$. The ϵ and $\top$ cases are immediate. The inductive cases go as follows.

$$
\begin{aligned}
\mathrm{FR}\left(\mathrm{FR}^R\left((\prime x)^R\right)\right) &= \mathrm{FR}\left(\mathrm{FR}^R\left(x^R \prime\right)\right) \\
&= \mathrm{FR}\left(\left(\mathrm{FR}\left(x^R \prime\right)\right)^R\right) \\
&= \begin{cases}
\mathrm{FR}(\epsilon) = \epsilon & \text{if } x \text{ clean} \\
\mathrm{FR}\left(\left(\mathrm{FR}\left(x^R\right)\prime\right)^R\right) & \text{if not} \\
\quad = \mathrm{FR}\left(\prime\,\mathrm{FR}^R\left(x^R\right)\right) & \\
\quad = \mathrm{FR}^R\left(x^R\right) &
\end{cases} \\
&= \mathrm{FR}^R\left(\mathrm{FR}^R(\prime x)\right) \\
\mathrm{FR}\left(\mathrm{FR}^R\left((ax)^R\right)\right) &= \mathrm{FR}\left(\mathrm{FR}^R\left(x^R a\right)\right) \\
&= \begin{cases}
\epsilon & \text{if } x \text{ clean} \\
\mathrm{FR}\left(\left(\mathrm{FR}\left(x^R\right) a\right)^R\right) & \text{if not} \\
\quad = \mathrm{FR}\left(a\,\mathrm{FR}^R\left(x^R\right)\right) & \\
\quad = \mathrm{FR}\left(\mathrm{FR}^R\left(x^R\right)\right) &
\end{cases} \\
&= \mathrm{FR}^R\left(\mathrm{FR}^R(x)\right) \quad \text{ind.} \\
&= \mathrm{FR}^R\left(\mathrm{FR}^R(ax)\right)
\end{aligned}
$$

■

Definition 34 (Middle) *The function* $\mathrm{M}(x) = \mathrm{FR}(\mathrm{LR}(x))$ *extracts the middle part of a string.*

For example, $\mathrm{M}(a'bc') = bc$.

Proposition 38 *If* $x = t\prime u\prime v$ *and* t *and* v *are clean then* $\mathrm{M}(x) = u$.

Proof: By proposition 34 and the definition of FR we can see that $\mathrm{FR}(x) = \mathrm{FR}(t{\prime}u{\prime}v) = \mathrm{FR}({\prime}u{\prime}v) = u{\prime}v$. By the same proposition and the definition of LR we see that $\mathrm{LR}(u{\prime}v) = \mathrm{LR}(u{\prime}) = u$. The result follows from the definition of M and proposition 37. ■

Proposition 39 $\mathrm{F}(x) \sqsubseteq_p x$ *and* $\mathrm{L}(x) \sqsubseteq_s x$*.*

Proof: By structural induction according to the following table.

$\hat{x}$	$\mathrm{F}(\hat{x})$	pf. by
ϵ	ϵ	insp.
$\top$	$\top$	insp.
${\prime}x$	ϵ	insp.
ax	$a\,\mathrm{F}(x)$	ind.

■

Proposition 40 $\mathrm{FR}(x) \sqsubseteq_s x$ *and* $\mathrm{LR}(x) \sqsubseteq_s x$*.*

Proof: If x is clean the result is immediate since $\mathrm{FR}(x) = \epsilon$ by proposition 30. If x is really marked, then $\mathrm{FR}(x) \sqsubseteq_s x$ and $\mathrm{LR}(x) \sqsubseteq_s x$ by structural induction according to the following table. (The last two lines follow from proposition 21.)

$\hat{x}$	$\hat{x}^R$	$\mathrm{FR}(\hat{x})$	$\mathrm{FR}^R(\hat{x})$	pf. by
ϵ	ϵ	ϵ	ϵ	insp.
$\top$	$\top$	$\top$	$\top$	insp.
${\prime}x$	$x^R{\prime}$	x	x^R	prop.
ax	$x^R a$	$\mathrm{FR}(x)$	$\mathrm{FR}^R(x)$	ind., prop.

■

Definition 8 (p. 56) **(Overlapped concatenation)** *The overlapped concatenation of marked strings x and y, written $x /\!\!/ y$, is defined to be the marked string* $(\mathrm{LR}(x) \sqcup_s \mathrm{F}(y)){\prime}(\mathrm{L}(x) \sqcup_p \mathrm{FR}(y))$*.*

When there is no conflict, overlapped concatenation yields a real marked string. For example,

$$\begin{aligned} a'ba' /\!\!/ a'bc' &= (\mathrm{LR}(a'ba') \sqcup_s \mathrm{F}(a'bc'))' \, (\mathrm{L}(a'ba') \sqcup_p \mathrm{FR}(a'bc')) \\ &= (a'ba \sqcup_s a)'(\epsilon \sqcup_p bc') \\ &= a'ba'bc \end{aligned}$$

When there is conflict, it yields $\top$, which can be thought of as an error indicator. For example, expanding $a'ba' /\!\!/ a'bc'$ from the definition yields $(a'bc \sqcup_s a) = \top$ so (since concatenation is strict) $a'ba' /\!\!/ a'bc' = \top$.

A.5 Properties of overlapped concatenation

We wish to show that $/\!\!/$ is an associative operator. Doing so requires the introduction of some machinery.

Lemma 41 *If y is clean then* $\mathrm{F}(x) \sqsubseteq_s \mathrm{F}(x \sqcup_s y)$.

Proof: By structural induction (via a schema symmetric to the one we have been using) according to the following table, making use of the definition of $\sqsubseteq_s$.

$\hat{x}$	$\hat{y}$	$\mathrm{F}(\hat{x})$		$\hat{x} \sqcup_p \hat{y}$	$\mathrm{F}(\hat{x} \sqcup_p \hat{y})$		pf. by
x	ϵ	$\mathrm{F}(x)$		x	$\mathrm{F}(x)$		insp.
$\top$	y			$\top$	$\top$		insp.
xa	ya	$\begin{cases} xa & x \text{ clean} \\ \mathrm{F}(x) & \text{otw.} \end{cases}$		$(x \sqcup_p y)a$	$\begin{cases} F(x \sqcup_p y)a & x \text{ clean} \\ F(x \sqcup_p y) & \text{otw.} \end{cases}$		ind., def. ind.
$x\prime$	y	$\begin{cases} x & x \text{ clean} \\ \mathrm{F}(x) & \text{otw.} \end{cases}$		$(x \sqcup_p y)\prime$	$\begin{cases} x \sqcup_p y & x \text{ clean} \\ F(x \sqcup_p y) & \text{otw.} \end{cases}$		def. ind.
xa	yb	?		$\top$	$\top$		insp.

■

Proposition 42 *If x is really marked then* $\mathrm{F}(x) \sqsubseteq_s \mathrm{F}(x /\!\!/ y)$ *and* $\mathrm{L}(y) \sqsubseteq_p \mathrm{L}(x /\!\!/ y)$.

Proof: By definition, $x /\!\!/ y$ is $(\mathrm{LR}(x) \sqcup_s \mathrm{F}(y))\prime(\mathrm{L}(y) \sqcup_p \mathrm{FR}(y))$ so by proposition 33, $\mathrm{F}(x /\!\!/ y) = \mathrm{F}(\mathrm{LR}(x) \sqcup_s \mathrm{F}(y))$. By corollary 32, $\mathrm{F}(y)$ is clean. So by lemma 41, $\mathrm{F}(\mathrm{LR}(x)) \sqsubseteq_s \mathrm{F}(\mathrm{LR}(x) \sqcup_s \mathrm{F}(y))$. Since x is really marked, by proposition 35, $\mathrm{F}(x) = \mathrm{F}(\mathrm{LR}(x))$. The result follows by substitution. ■

Proposition 43 *If x has at least 2 marks then the equalities* $\mathrm{F}(x)\prime\,\mathrm{M}(x) = \mathrm{LR}(x)$ *and* $\mathrm{M}(x)\prime\,\mathrm{L}(x) = \mathrm{FR}(x)$ *both hold.*

Proof: By induction on x according to the table. In the last line, $\mathrm{LR}(\hat{x})$ follows from proposition 34, and the penultimate line also follows from proposition 34.

$\hat{x}$	$\mathrm{F}(\hat{x})$	$\mathrm{FR}(\hat{x})$	$\mathrm{M}(\hat{x})$	$\mathrm{LR}(\hat{x})$	$\mathrm{F}(\hat{x})\prime\,\mathrm{M}(\hat{x})$	pf. by
ϵ						n/a
$\prime\epsilon$						n/a
$a\epsilon$						n/a
$\top$	$\top$	$\top$	$\top$	$\top$	$\top$	insp.
$\prime\top = \top$						same
$a\top = \top$						same
$\prime y\prime$	ϵ	$y\prime$	y	$\prime y$	$\prime y$	insp.
$ay\prime$	$a\,\mathrm{F}(y\prime)$	$\mathrm{FR}(y\prime)$	$\mathrm{M}(y\prime)$	$a\,\mathrm{LR}(y\prime)$	$a\,\mathrm{F}(y\prime)\prime\,\mathrm{M}(y\prime)$	ind.
$\prime yb$	ϵ	yb	$\mathrm{LR}(y)$	$\mathrm{LR}(\prime y)$	$\prime\,\mathrm{LR}(y)$	prop.
ayb	$a\,\mathrm{F}(yb)$	$\mathrm{FR}(yb)$	$\mathrm{M}(yb)$	$a\,\mathrm{LR}(yb)$	$a\,\mathrm{F}(yb)\prime\,\mathrm{M}(yb)$	ind.

■

Proposition 44 *$x(y \sqcup_p z) = xy \sqcup_p xz$ and $(x \sqcup_s y)z = xz \sqcup_s yz$.*

Proof: By induction on x according to the following table.

$\hat{x}$	$\hat{x}y$	$\hat{x}z$	$\hat{x}y \sqcup_p \hat{x}z$	$\hat{x}(y \sqcup_p z)$	pf. by
ϵ	y	z	$y \sqcup_p z$	$y \sqcup_p z$	insp.
$\top$	$\top$	$\top$	$\top$	$\top$	insp.
$'x$	$'xy$	$'xz$	$'(xy \sqcup_p xz)$	$'x(y \sqcup_p z)$	ind.
ax	axy	axz	$a(xy \sqcup_p xz)$	$ax(y \sqcup_p z)$	ind.

■

Proposition 45 *If $|w| = |x|$ then $(w \sqcup_p x)(y \sqcup_p z) = wy \sqcup_p xz$ and $(y \sqcup_s z)(w \sqcup_s x) = yw \sqcup_s zx$.*

Definition 35 *The operator $\downarrow_p$ (to be read "drop the first") is defined inductively:*

$$
\begin{aligned}
x \downarrow_p 0 &= x \\
\epsilon \downarrow_p k &= \epsilon \\
\top \downarrow_p k &= \top \\
'x \downarrow_p k &= x \downarrow_p k \\
ax \downarrow_p k &= x \downarrow_p k - 1
\end{aligned}
$$

The operator $\downarrow_s$ (to be read "drop the last") is defined by $x \downarrow_s k = (x^R \downarrow_p k)^R$.

Proposition 46 *If x is clean then $x \downarrow_p k$ and $x \downarrow_s k$ are also clean.*

Definition 36 *The operator $\uparrow_p$ (to be read "keep at most the first") is defined inductively:*

$$
\begin{aligned}
\epsilon \uparrow_p k &= \epsilon \\
\top \uparrow_p k &= \top \\
x \uparrow_p 0 &= \epsilon \\
'x \uparrow_p k &= {}' \left(x \uparrow_p k\right) \\
ax \uparrow_p k &= a \left(x \uparrow_p k - 1\right)
\end{aligned}
$$

The operator $\uparrow_s$ ("keep at most the last") is defined by $x \uparrow_s k = (x^R \uparrow_p k)^R$.

The operators $\downarrow$ and $\uparrow$ bind more tightly than $\sqcup$ but less tightly than concatenation.

Proposition 47 *If* $x \neq \top$ *then* $|x \uparrow_p k| \leq k$ *and* $|x \uparrow_s k| \leq k$.

Proof: By structural induction, according to the following table.

$\hat{x}$	$\hat{x} \uparrow_p k$	$\left\|\hat{x} \uparrow_p k\right\|$	pf. by
ϵ	ϵ	0	insp.
$\top$	$\top$	∞	n/a
$'x$	$'\left(x \uparrow_p k\right)$	$\left\|x \uparrow_p k\right\|$	ind. hyp.
ax	$a\left(x \uparrow_p k\right)$	$1 + \left\|x \uparrow_p k\right\|$	ind. hyp.

■

Proposition 48 *If* $|x| \leq |y|$ *and* x *is clean then* $x \sqcup y = \top$ *or* $x \sqcup y = y$ *where* $\sqcup$ *is either* $\sqcup_p$ *or* $\sqcup_s$.

Proof: By structural induction according to the table.

$\hat{x}$	$\hat{y}$	$\hat{x \sqcup y}$	pf. by
ϵ	y		n/a
x	ϵ		n/a
$\top$	y	$\top$	insp.
x	$\top$	$\top$	insp.
$'x$	$'y$		n/a
$'x$	y	$'(x \sqcup y)$	ind.
x	$'y$		n/a
ax	ay	$a(x \sqcup y)$	ind.
ax	by	$\top$	insp.
ϵ	ϵ	ϵ	insp.

■

Proposition 49 *If* x *and* z *are clean then* $(x \uparrow_s |y| \sqcup_s y) \sqcup_p z \uparrow_p |y| = x \uparrow_s |y| \sqcup_s (y \sqcup_p z \uparrow_p |y|)$.

Proof: By applying proposition 48. ■

Proposition 50 *When* z *is clean,* $xy \sqcup_s z = (x \sqcup_s z \downarrow_s |y|)(y \sqcup_s z \uparrow_s |y|)$. *When* z *is clean,* $yx \sqcup_p z = (y \sqcup_p z \uparrow_p |y|)(x \sqcup_p z \downarrow_p |y|)$.

Proof: By structural induction on y and z, according to the table.

$\hat{y}$	$\hat{z}$	$x\hat{y}$	$x\hat{y} \sqcup_s \hat{z}$	$\hat{z} \downarrow_s \lvert\hat{y}\rvert$	$(x \sqcup_s \hat{z} \downarrow_s \lvert\hat{y}\rvert)$	$\hat{z} \uparrow_s \lvert\hat{y}\rvert$	$(\hat{y} \sqcup_s \hat{z} \uparrow_s \lvert\hat{y}\rvert)$	pf. by
ϵ	z	x	$(x \sqcup_s z)$	z	$x \sqcup_s z$	ϵ	ϵ	insp.
y	ϵ	xy	xy	ϵ	x	ϵ	y	insp.
$\top$	z	$\top$	$\top$	ϵ	x	z	$\top$	insp.
y	$\top$	xy	$\top$	$\top$	$\top$	$\top$	$\top$	insp.
y'	z'							n/a
y'	z	xy'	$(xy \sqcup_s z)'$	$z \downarrow_s \lvert y\rvert$	$(x \sqcup_s z \downarrow_s \lvert y\rvert)$	$z \uparrow_s \lvert y\rvert$	$(y \sqcup_s z \uparrow_s \lvert y\rvert)'$	ind.
y	z'							n/a
ya	za	xya	$(xy \sqcup_s z)\,a$	$z \downarrow_s \lvert y\rvert$	$(x \sqcup_s z \downarrow_s \lvert y\rvert)$	$(z \uparrow_s \lvert y\rvert)\,a$	$(y \sqcup_s z \uparrow_s \lvert y\rvert)\,a$	ind.
ya	zb	xyb	$\top$	$z \downarrow_s \lvert y\rvert$	$(x \sqcup_s z \downarrow_s \lvert y\rvert)$	$(z \uparrow_s \lvert y\rvert)\,b$	$\top$	insp.

■

Lemma 51

$$
\begin{aligned}
\mathrm{F}(x /\!\!/ y) &= \mathrm{F}(x) \sqcup_s \mathrm{F}(y) \downarrow_s |\mathrm{M}(x)| \\
\mathrm{M}(x /\!\!/ y) &= (\mathrm{M}(x) \sqcup_s \mathrm{F}(y) \uparrow_s |\mathrm{M}(x)|)'\,(\mathrm{M}(y) \sqcup_p \mathrm{L}(x) \uparrow_p |\mathrm{M}(y)|) \\
\mathrm{L}(x /\!\!/ y) &= \mathrm{L}(y) \sqcup_p \mathrm{L}(x) \downarrow_p |\mathrm{M}(y)| \\
\mathrm{F}(y /\!\!/ z) &= \mathrm{F}(y) \sqcup_s \mathrm{F}(z) \downarrow_s |\mathrm{M}(y)| \\
\mathrm{M}(y /\!\!/ z) &= (\mathrm{M}(y) \sqcup_s \mathrm{F}(z) \uparrow_s |\mathrm{M}(y)|)'\,(\mathrm{M}(z) \sqcup_p \mathrm{L}(y) \uparrow_p |\mathrm{M}(z)|) \\
\mathrm{L}(y /\!\!/ z) &= \mathrm{L}(z) \sqcup_p \mathrm{L}(y) \downarrow_p |\mathrm{M}(z)| \\
\mathrm{F}((x /\!\!/ y) /\!\!/ z) &= \mathrm{F}(x /\!\!/ y) \sqcup_s \mathrm{F}(z) \downarrow_s |\mathrm{M}(x /\!\!/ y)| \\
\mathrm{M}((x /\!\!/ y) /\!\!/ z) &= (\mathrm{M}(x /\!\!/ y) \sqcup_s \mathrm{F}(z) \uparrow_s |\mathrm{M}(x /\!\!/ y)|)'\,(\mathrm{M}(z) \sqcup_p \mathrm{L}(x /\!\!/ y) \uparrow_p |\mathrm{M}(z)|) \\
\mathrm{L}((x /\!\!/ y) /\!\!/ z) &= \mathrm{L}(z) \sqcup_p \mathrm{L}(x /\!\!/ y) \downarrow_p |\mathrm{M}(z)| \\
\mathrm{F}(x /\!\!/ (y /\!\!/ z)) &= \mathrm{F}(x) \sqcup_s \mathrm{F}(y /\!\!/ z) \downarrow_s |\mathrm{M}(x)| \\
\mathrm{M}(x /\!\!/ (y /\!\!/ z)) &= (\mathrm{M}(x) \sqcup_s \mathrm{F}(y /\!\!/ z) \uparrow_s |\mathrm{M}(x)|)'\,(\mathrm{M}(y /\!\!/ z) \sqcup_p \mathrm{L}(x) \uparrow_p |\mathrm{M}(y /\!\!/ z)|) \\
\mathrm{L}(x /\!\!/ (y /\!\!/ z)) &= \mathrm{L}(y /\!\!/ z) \sqcup_p \mathrm{L}(x) \downarrow_p |\mathrm{M}(y /\!\!/ z)|
\end{aligned}
$$

Proof: The first three lines follow from definition 8, proposition 43, proposition 31, and proposition 50. The remainder follow by substitution and the previous lines.

■

Lemma 52 *The marked strings* $\mathrm{M}(x /\!\!/ y)$ *and* $\mathrm{M}(x)'\,\mathrm{M}(y)$ *are equal.*

Proof: From the second line of lemma 51. ■

Lemma 53 $\mathrm{F}((x /\!\!/ y) /\!\!/ z) = \mathrm{F}(x /\!\!/ (y /\!\!/ z))$ *and* $\mathrm{L}((x /\!\!/ y) /\!\!/ z) = \mathrm{L}(x /\!\!/ (y /\!\!/ z))$*.*

Proof: The reasoning for F is as follows; that for L is by symmetry.

$$
\begin{aligned}
\mathrm{F}((x /\!\!/ y) /\!\!/ z) &= \mathrm{F}(x) \sqcup_s \mathrm{F}(y) \downarrow_s |\mathrm{M}(x)| \sqcup_s \mathrm{F}(z) \downarrow_s |\mathrm{M}(x /\!\!/ y)| \\
\mathrm{F}(x /\!\!/ (y /\!\!/ z)) &= \mathrm{F}(x) \sqcup_s (\mathrm{F}(y) \sqcup_s \mathrm{F}(z) \downarrow_s |\mathrm{M}(y)|) \downarrow_s |\mathrm{M}(x)| \\
&= \mathrm{F}(x) \sqcup_s \mathrm{F}(y) \downarrow_s |\mathrm{M}(x)| \sqcup_s \mathrm{F}(z) \downarrow_s (|\mathrm{M}(x)| + |\mathrm{M}(y)|)
\end{aligned}
$$

■

Proposition 54 *If $j \geq k$ then $x \uparrow_p j \uparrow_p k = x \uparrow_p k$ and $x \uparrow_s j \uparrow_s k = x \uparrow_s k$.*

Proof: By induction on k and the structure of x. There are five cases; the last two are inductive. Letting $i = j - k$, the cases are:

$$\begin{aligned}
x \uparrow_s i \uparrow_s 0 = \epsilon &= x \uparrow_s 0 \\
\epsilon \uparrow_s i + k \uparrow_s k &= \epsilon \uparrow_s k \\
\top \uparrow_s i + k \uparrow_s k &= \top \uparrow_s k \\
xa \uparrow_s j + k + 1 \uparrow_s k + 1 &= (x \uparrow_s j + k)\, a \uparrow_s k + 1 = (x \uparrow_s j + k \uparrow_s k)\, a \\
x' \uparrow_s j + k \uparrow_s k &= (x \uparrow_s j + k)' \uparrow_s k = (x \uparrow_s j + k \uparrow_s k)'
\end{aligned}$$

■

Proposition 55 *If x is clean then $x \uparrow_p j + k \downarrow_p k = x \downarrow_p k \uparrow_p j$ and $x \uparrow_p j + k \downarrow_p k = x \downarrow_p k \uparrow_p j$.*

Proof: By induction on x and k.

$$\begin{aligned}
\epsilon \uparrow_p j + k \downarrow_p k &= \epsilon = \epsilon \downarrow_p k \uparrow_p j \\
x \uparrow_p j + 0 \downarrow_p 0 &= x \uparrow_p j = x \downarrow_p 0 \uparrow_p j \\
ax \uparrow_p j + k + 1 \downarrow_p k + 1 &= a\,(x \uparrow_p j + k) \downarrow_p k + 1 \\
&= x \uparrow_p j + k \downarrow_p k \\
&= x \downarrow_p k \uparrow_p j \\
&= ax \downarrow_p k + 1 \downarrow_p j
\end{aligned}$$

■

Proposition 56 *If $x \sqcup_p \neq \top$ then $(x \sqcup_p y) \uparrow_p k = x \uparrow_p k \sqcup_p y \uparrow_p k$. If $x \sqcup_s \neq \top$ then $(x \sqcup_s y) \uparrow_s k = x \uparrow_s k \sqcup_s y \uparrow_s k$.*

Proof: By induction on k and the structure of x and y, according to the table:

$\hat{x}$	$\hat{y}$	$\hat{k}$	$\hat{x} \sqcup_p \hat{y}$	$\left(\hat{x} \sqcup_p \hat{y}\right) \uparrow_p \hat{k}$	$\hat{x} \uparrow_p \hat{k}$	$\hat{y} \uparrow_p \hat{k}$	$\hat{x} \uparrow_p \hat{k} \sqcup_p \hat{y} \uparrow_p \hat{k}$	pf. by
ϵ	y	k	y	$y \uparrow_p k$	ϵ	$y \uparrow_p k$	$y \uparrow_p k$	insp.
x	ϵ	k	x	$x \uparrow_p k$	$x \uparrow_p k$	ϵ	$x \uparrow_p k$	insp.
$\top$	y	k						n/a
x	$\top$	k						n/a
x	y	0	$x \sqcup_p y$	ϵ	ϵ	ϵ	ϵ	insp.
$'x$	y	k	$'\left(x \sqcup_p y\right)$	$'\left(\left(x \sqcup_p y\right) \uparrow_p k\right)$	$'\left(x \uparrow_p k\right)$	$y \uparrow_p k$	$'\left(x \uparrow_p k \sqcup_p y \uparrow_p k\right)$	ind.
x	$'y$	k	$'\left(x \sqcup_p y\right)$	$'\left(\left(x \sqcup_p y\right) \uparrow_p k\right)$	$x \uparrow_p k$	$'\left(y \uparrow_p k\right)$	$'\left(x \uparrow_p k \sqcup_p y \uparrow_p k\right)$	ind.
$'x$	$'y$	k	$'\left(x \sqcup_p y\right)$	$'\left(\left(x \sqcup_p y\right) \uparrow_p k\right)$	$'\left(x \uparrow_p k\right)$	$'\left(y \uparrow_p k\right)$	$'\left(x \uparrow_p k \sqcup_p y \uparrow_p k\right)$	ind.
ax	ay	$k+1$	$a\left(x \sqcup_p y\right)$	$a\left(\left(x \sqcup_p y\right) \uparrow_p k\right)$	$a\left(x \uparrow_p k\right)$	$a\left(y \uparrow_p k\right)$	$a\left(x \uparrow_p k \sqcup_p y \uparrow_p k\right)$	ind.
ax	by	k	$\top$					n/a

■

Lemma 57 $\mathrm{M}((x /\!\!/ y) /\!\!/ z) = \mathrm{M}(x /\!\!/ (y /\!\!/ z))$.

Proof: We begin by expanding by an equation of lemma 51, and first consider the portion identified as A. We expand this by proposition 50, yielding two parts. The second part, C, can be simplified by proposition 54. By proposition 49, the order of the $\sqcup$ operations in this expression is unimportant. The first part, B, can be simplified by proposition 55. The last part of the original expansion, D, can be expanded by proposition 56. The desideratum then follows from the symmetry of C and the symmetry between B and D.

$$\begin{aligned}
\mathrm{M}((x /\!\!/ y) /\!\!/ z) &= \underbrace{(\mathrm{M}(x /\!\!/ y) \sqcup_s \mathrm{F}(z) \uparrow_s |\mathrm{M}(x /\!\!/ y)|)'}_{A} \underbrace{\Big(\mathrm{M}(z) \sqcup_p \mathrm{L}(x /\!\!/ y) \uparrow_p |\mathrm{M}(z)|\Big)}_{D} \\
A &= \Big((\mathrm{M}(x) \sqcup_s \mathrm{F}(y) \uparrow_s |\mathrm{M}(x)|)' \Big(\mathrm{M}(y) \sqcup_p \mathrm{L}(x) \uparrow_p |\mathrm{M}(y)|\Big)\Big) \\
&\quad \sqcup_s \mathrm{F}(z) \uparrow_s |\mathrm{M}(x)| + |\mathrm{M}(y)| \\
&= \overbrace{\mathrm{M}(x) \sqcup_s \mathrm{F}(y) \uparrow_s |\mathrm{M}(x)| \sqcup_s (\mathrm{F}(z) \uparrow_s (|\mathrm{M}(x)| + |\mathrm{M}(y)|) \downarrow_s |\mathrm{M}(y)|)}^{B} \\
&\quad \underbrace{{}' \Big(\mathrm{M}(y) \sqcup_p \mathrm{L}(x) \uparrow_p |\mathrm{M}(y)|\Big) \sqcup_s \mathrm{F}(z) \uparrow_s (|\mathrm{M}(x)| + |\mathrm{M}(y)|) \uparrow_s |\mathrm{M}(y)|}_{C} \\
C &= \mathrm{M}(y) \sqcup_p \mathrm{L}(x) \uparrow_p |\mathrm{M}(y)| \sqcup_s \mathrm{F}(z) \uparrow_s |\mathrm{M}(y)| \\
B &= \mathrm{M}(x) \sqcup_s \mathrm{F}(y) \uparrow_s |\mathrm{M}(x)| \sqcup_s (\mathrm{F}(z) \downarrow_s |\mathrm{M}(y)| \uparrow_s |\mathrm{M}(x)|) \\
D &= \mathrm{M}(z) \sqcup_p \Big(\mathrm{L}(y) \sqcup_p \mathrm{L}(x) \downarrow_p |\mathrm{M}(y)|\Big) \uparrow_p |\mathrm{M}(z)| \\
&= \mathrm{M}(z) \sqcup_p \mathrm{L}(y) \uparrow_p |\mathrm{M}(z)| \sqcup_p \Big(\mathrm{L}(x) \downarrow_p |\mathrm{M}(y)| \uparrow_p |\mathrm{M}(z)|\Big)
\end{aligned}$$

■

Theorem 2 (p. 57) *The operator $/\!\!/$ is associative.*

Proof: By noting with the aid of lemmas 53 and 57 that:

$$\begin{aligned}
x /\!\!/ (y /\!\!/ z) &= \mathrm{F}(x /\!\!/ (y /\!\!/ z))' \, \mathrm{M}((x /\!\!/ (y /\!\!/ z))' \, \mathrm{L}(x /\!\!/ (y /\!\!/ z)) \\
&= \mathrm{F}((x /\!\!/ y) /\!\!/ z)' \, \mathrm{M}((x /\!\!/ y) /\!\!/ z)' \, \mathrm{L}((x /\!\!/ y) /\!\!/ z) \\
&= (x /\!\!/ y) /\!\!/ z
\end{aligned}$$

■

A.6 Additional properties and definitions

We need slightly more machinery in order to construct mappings that are homomorphic over $/\!\!/$.

Definition 9 (p. 57) *The function* CL *is defined by the equation* $\mathrm{CL}(x) = \mathrm{LR}(x)\,\mathrm{L}(x)$.

Lemma 4 (p. 58) *If y has 2 or more marks, then* $x /\!\!/ \mathrm{CL}(y) = \mathrm{CL}(x /\!\!/ y)$.

Proof: Expanding by the definition, we desire to show that $x /\!\!/ \mathrm{LR}(y)\,\mathrm{L}(y) = \mathrm{LR}(x /\!\!/ y)\,\mathrm{L}(x /\!\!/ y)$. Noting that if x has at least two marks, $\mathrm{LR}(x)$ has at least one mark, we establish two useful identities. The first makes use of proposition 34 and definition 34. The second makes use of propositions 31 and 35. Then we expand the left-hand side by the definition of $/\!\!/$ (definition 8). We simplify by our identities then expand the portion after the mark using proposition 50.

$$
\begin{aligned}
\mathrm{FR}(\mathrm{LR}(y)\mathrm{L}(y)) &= \mathrm{FR}(\mathrm{LR}(y))\mathrm{L}(y) = \mathrm{M}(y)\mathrm{L}(y) \\
\mathrm{F}(\mathrm{LR}(y)\mathrm{L}(y)) &= \mathrm{F}(\mathrm{LR}(y)) = \mathrm{F}(y) \\
x /\!\!/ \mathrm{LR}(y)\mathrm{L}(y) &= (\mathrm{LR}(x) \sqcup_s \mathrm{F}(\mathrm{LR}(y)\mathrm{L}(y)))'\,(L(x) \sqcup_p \mathrm{FR}(\mathrm{LR}(y)\mathrm{L}(y))) \\
&= (\mathrm{LR}(x) \sqcup_s \mathrm{F}(y))'\,(\mathrm{L}(x) \sqcup_p \mathrm{M}(y)\mathrm{L}(y))
\end{aligned}
$$

$$
= \underbrace{(\mathrm{LR}(x) \sqcup_s \mathrm{F}(y))'\left(\mathrm{M}(y) \sqcup_p \mathrm{L}(x) \uparrow_p |\mathrm{M}(y)|\right)}_{A}\ \underbrace{(\mathrm{L}(x) \downarrow_p |\mathrm{M}(y)| \sqcup_p \mathrm{L}(y))}_{B}
$$

This yields two parts, identified as A and B above. Part B we recognize as $\mathrm{L}(x /\!\!/ y)$. We then expand LR(x//y) by proposition 43, expand each part using lemma 51, collapse by proposition 50, and finally collapse by proposition 43, whereupon we recognize that part A is indeed $\mathrm{LR}(x /\!\!/ y)$, completing the proof.

$$
\begin{aligned}
&\mathrm{LR}(x /\!\!/ y) = \mathrm{F}(x /\!\!/ y)'\mathrm{M}(x /\!\!/ y) \\
&= (\mathrm{F}(x) \cup_s \mathrm{F}(y) \downarrow_s |\mathrm{M}(x)|)'\left((\mathrm{M}(x) \cup_s \mathrm{F}(y) \uparrow_s |\mathrm{M}(x)|)'\,(\mathrm{M}(y) \cup_p \mathrm{L}(x) \uparrow_p |\mathrm{M}(y)|)\right) \\
&= (\mathrm{F}(x)' \cup_s \mathrm{F}(y) \downarrow_s |\mathrm{M}(x)|)\,(\mathrm{M}(x) \cup_s \mathrm{F}(y) \uparrow_s |\mathrm{M}(x)|)'\,(\mathrm{M}(y) \cup_p \mathrm{L}(x) \uparrow_p |\mathrm{M}(y)|) \\
&= (\mathrm{F}(x)'\mathrm{M}(x) \cup_s \mathrm{F}(y))'\,(\mathrm{M}(y) \cup_p \mathrm{L}(x) \uparrow_p |\mathrm{M}(y)|) \\
&= (\mathrm{LR}(x) \cup_s \mathrm{F}(y))'\,(\mathrm{M}(y) \cup_p \mathrm{L}(x) \uparrow_p |\mathrm{M}(y)|) \\
&= A
\end{aligned}
$$

■

Lemma 58 *If x is 1-marked then* $\mathrm{F}(x) = \mathrm{LR}(x)$ *and* $\mathrm{L}(x) = \mathrm{FR}(x)$.

Proof: Since x is 1-marked, $\mathrm{LR}(x)$ is clean. Since x is marked, $\mathrm{F}(x) = \mathrm{F}(\mathrm{LR}(x))$ by proposition 35. Since $\mathrm{LR}(x)$ is clean, $\mathrm{F}(\mathrm{LR}(x)) = \mathrm{LR}(x)$ by proposition 31.

■

Lemma 3 (p. 57) *If x is 2-marked then* $x = \mathrm{CL}(x) /\!\!/ x$.

Proof: First we observe an identity by expanding the definition of CL, applying corollary 32 and proposition 31 and definition 34. We then observe a similar identity by also applying lemma 58. Then we can expand $\mathrm{CL}(x) /\!\!/ x$ by the definition of $/\!\!/$, and simplify with the identities. Then we can simplify the first part by propositions 43 and 44, and the last part by a lattice property. Applying the definition of $\sqcup_s$ and lattice laws again yields the desired result.

$$\begin{aligned}
\mathrm{L}(\mathrm{CL}(x)) &= \mathrm{L}(\mathrm{LR}(x)\mathrm{L}(x)) = \mathrm{L}(\mathrm{LR}(x))\mathrm{L}(x) = \mathrm{FR}(\mathrm{LR}(x))\mathrm{L}(x) = \mathrm{M}(x)\mathrm{L}(x) \\
\mathrm{LR}(\mathrm{CL}(x)) &= \mathrm{LR}(\mathrm{LR}(x)\mathrm{L}(x)) = \mathrm{LR}(\mathrm{LR}(x)) = \mathrm{F}(x) \\
\mathrm{CL}(x) /\!\!/ x &= (\mathrm{LR}(\mathrm{CL}(x)) \sqcup_s \mathrm{F}(x))' \, (\mathrm{L}(\mathrm{CL}(x)) \sqcup_p \mathrm{FR}(x)) \\
&= (\mathrm{F}(x) \sqcup_s \mathrm{F}(x))' \, (\mathrm{M}(x)\mathrm{L}(x) \sqcup_p \mathrm{FR}(x)) \\
&= \mathrm{F}(x)' \, (\mathrm{M}(x)\mathrm{L}(x) \sqcup_p \mathrm{M}(x)'\mathrm{L}(x)) \\
&= \mathrm{F}(x)'\mathrm{M}(x) \left(\mathrm{L}(x) \sqcup_p' \mathrm{L}(x)\right) \\
&= \mathrm{F}(x)'\mathrm{M}(x)'\mathrm{L}(x) = x
\end{aligned}$$

■

Definition 10 (p. 58) **(Compatibility)** *If strings x and y are 2-marked, we will say that they are compatible, and denote this by $x \approx y$, if and only if neither the expression $\mathrm{F}(\mathrm{CL}(x)) \sqcup_s \mathrm{F}(\mathrm{CL}(y))$ nor $\mathrm{FR}(\mathrm{CL}(x)) \sqcup_p \mathrm{FR}(\mathrm{CL}(y))$ is equal to $\top$.*

Definition 11 (p. 58) *If string x is 2-marked, the* measurements *of x, denoted $||x||$ is the pair $(|\,\mathrm{F}(\mathrm{CL}(x))|, |\,\mathrm{FR}(\mathrm{CL}(x))|)$.*

Lemma 59 *The equality $x \sqcup_p yz = \top$ holds if and only if $x \sqcup_p y'z = \top$ does. The equality $x \sqcup_s yz = \top$ holds if and only if $x \sqcup_s y'z = \top$ does.*

Proof: By structural induction on x and y.

$\hat{x}$	$\hat{y}$	$\hat{x} \sqcup_p \hat{y}z$	$\hat{x} \sqcup_p \hat{y}z$	pf. by
ϵ	y	yz	$y'z$	insp.
x	ϵ	$x \sqcup_p z$	$'(x \sqcup_p z)$	insp.
$\top$	y	$\top$	$\top$	insp.
x	$\top$	$\top$	$\top$	insp.
$'x$	$'y$	$'(x \sqcup_p yz)$	$'(x \sqcup_p y'z)$	ind.
$'x$	y	$'(x \sqcup_p yz)$	$'(x \sqcup_p y'z)$	ind.
x	$'y$	$'(x \sqcup_p yz)$	$'(x \sqcup_p y'z)$	ind.
ax	ay	$a(x \sqcup_p yz)$	$a(x \sqcup_p y'z)$	ind.
ax	by	$\top$	$\top$	insp.

■

Proposition 5 (p. 58) *If x and y are 2-marked then $x \approx y$ iff* $\mathrm{CL}(x) /\!\!/ y \neq \top$.

Proof: First we note that if x is two-marked, the following identities hold.

$$\begin{aligned} \mathrm{LR}(\mathrm{CL}(x)) &= \mathrm{F}(\mathrm{CL}(x)) = \mathrm{F}(x) \\ \mathrm{L}(\mathrm{CL}(x)) &= \mathrm{FR}(\mathrm{CL}(x)) \\ \mathrm{FR}(\mathrm{CL}(x)) &= \mathrm{FR}(\mathrm{LR}(x)\mathrm{L}(x)) = \mathrm{M}(x)\mathrm{L}(x) \end{aligned}$$

Then we expand $x /\!\!/ \mathrm{CL}(y)$ by the definition of $/\!\!/$ and simplify to yield pieces A and B.

$$\begin{aligned} \mathrm{CL}(x) /\!\!/ y &= (\mathrm{LR}(\mathrm{CL}(x)) \sqcup_s \mathrm{F}(y))' \, (\mathrm{L}(\mathrm{CL}(x)) \sqcup_p \mathrm{FR}(y)) \\ &= (\mathrm{F}(x) \sqcup_s \mathrm{F}(y))' \, (\mathrm{M}(x)\mathrm{L}(x) \sqcup_p \mathrm{FR}(y)) \\ &= \underbrace{(\mathrm{F}(\mathrm{CL}(x)) \sqcup_s \mathrm{F}(y))'}_{A} \, \underbrace{(\mathrm{M}(x)\mathrm{L}(x) \sqcup_p \mathrm{M}(y)'\mathrm{L}(y))}_{B} \end{aligned}$$

The entire marked string then differs from $\top$ if both $A \neq \top$ and $B \neq \top$. By lemma 59, $B \neq \top$ iff $(\mathrm{M}(x)\,\mathrm{L}(x) \sqcup_p \mathrm{M}(y)\,\mathrm{L}(y)) \neq \top$, which is half of the definition of $\approx$, definition 10. Observing that the other half of that definition is just $A \neq \top$ completes the proof. ∎

Proposition 6 (p. 59) *If two 2-marked strings have the same measurements, that is, if the equality $||x|| = ||y||$ holds, and $x \approx y$, then* $\mathrm{CL}(x) = \mathrm{CL}(y)$.

Proof: Since $||x|| = ||y||$ we know that $|\,\mathrm{FR}(\mathrm{CL}(x))| = |\,\mathrm{FR}(\mathrm{CL}(y))|$. Similarly, we know that $|\,\mathrm{F}(\mathrm{CL}(x))| = |\,\mathrm{F}(\mathrm{CL}(y))|$. We note that all four marked strings above are clean. We can apply proposition 48 and use symmetry to conclude that $\mathrm{FR}(\mathrm{CL}(x)) = \mathrm{FR}(\mathrm{CL}(y))$ and $\mathrm{F}(\mathrm{CL}(x)) = \mathrm{F}(\mathrm{CL}(y))$. Thus we conclude that $\mathrm{CL}(x) = \mathrm{CL}(y)$. ∎

Definition 12 (p. 59) *If A is a set of 2-marked strings, all of the same measurements, the notation $\tilde{A}$ denotes the set $\{\, x \mid \forall y \in A \,\blacksquare\, ||x|| = ||y|| \wedge x \not\approx y \,\}$.*

Definition 13 (p. 59) *IF x is a marked string, Γ is a superset of the alphabet, and (f, l) is a pair of nonnegative integers, define $\mathrm{ext}^{\Gamma}_{(f,l)}$ as follows. Let the pair $(m, n) = ||x||$ denote the measurements of x, let $\hat{f}$ be $\max(m, f)$ and let $\hat{l}$ be $\max(n, l)$. Then the extension of marked string x to alphabet Γ with lengths (f, l) is defined by the following equation.*

$$\mathrm{ext}^{\Gamma}_{(f,l)} = \{\, y \mid y \approx x \text{ and } ||y|| = (\hat{f}, \hat{l}) \,\}$$

Appendix B

Formal specification of Hector

B.1 Introduction

This appendix contains excerpts of the formal specification of the Hector microprocessor. Incomplete portions of the specification are indicated with suspension points. The specification consists of two main parts.

The first part is the abstract specification. It gives a few definitions, and then a set of assertions. These assertions specify the expected behavior of Hector. The first one specifies Hector's response to its reset signal. The next few specify the response to interrupts. Finally, a large set specify the operation of each of Hector's instructions, except for those instructions that involve looping behavior.

The second part maps the state, inputs, and outputs of the abstract specification onto patterns of operation of the actual circuit. The decomposition of the computer (whose instruction semantics are specified) into a processor (the object actually verified) and a memory system (which is merely assumed to be correct) also occurs in this mapping.

The specification is given in the Scheme-based specification language used by the prototype verifier. This section was typeset directly from the actual specification files. Representative portions of this specification were actually fed to the verifier, which checked the actual switch-level circuit.

B.2 Notation

The specification is given in a language embedded in the Scheme programming language [204], a dialect of Lisp.

B.2.1 Scheme

While not a substitute for an introduction to the language [94], the following terse description will be helpful to those unfamiliar with Scheme. The basic syntactic element is a name or *symbol*. Unlike many programming languages, symbols can include most characters, except parentheses. Other syntactic elements are numbers and strings. The basic syntactic construct is a parenthesized list (which is known as a symbolic expression, or "S-expression"). Most lists represent function calls; the first element (the head, or `car`) of the list denotes the function while the remaining elements (the tail, or `cdr`) denote the arguments. All functions, including those of ordinary arithmetic, follow this syntax. Thus expressions appear in fully parenthesized prefix form. Other lists denote "special forms." The determination of whether a list denotes function application or a special form is made according to the head of the list. Two important special forms are `define` and `let`. The `define` special form establishes a definition which binds to a name (given as the second element of the list) a value (given as the third element of the list). (However, if instead of a name, a parenthesized list is given, then the `define` form defines a function. The head of the parenthesized list gives the name of the function, and the tail gives its arguments.) The `let` form establishes local bindings. For example, `(let ((x 1) (y 2)) (+ x y))` corresponds to the common mathematical usage "let $x = 1$ and $y = 2$ in $x + y$" which denotes $1 + 2$, i.e., 3. Ordinarily, S-expressions are interpreted by these rules, as program text. However, preceding an S-expression with a quotation mark, causes it to be quoted, i.e., interpreted literally and treated as data. The head and the tail may of such expressions may be obtained with the functions `car` and `cdr`; new expressions may be constructed from a head and a tail using `cons`.

B.2.2 Specification language

The specification language used below makes use of numerous functions whose definitions are omitted here. Among the most important of these is `vec-==`, which takes two vectors[1] of Boolean values and returns a Boolean expression indicating whether or not the two vectors are equal. Another important function is `?:`, which denotes a conditional. The function `->` denotes case restriction. The function `is` constructs a formula which is true when a circuit node has the specified symbolic value. The form `decomp` collects conjuncts of the antecedent and the consequent of an assertion, and then decomposes the memory system from the processor.

Most other functions should be evident from their names, particularly to C programmers.

[1] Here "vector" denotes a conceptual vector, rather than a Scheme vector. Vectors are actually implemented in the prototype using Scheme lists.

In the specification below, trailing slash characters are best read as "prime." For example, the variables p/ is best read as p'. (Note that "prime" has nothing to do with the marker in the formulation of marked strings.)

This language is more expressive than the SMAL language defined in Chapter 3. In fact, it is too expressive. For example, since mappings are written in Scheme, they can introduce new case variables. This is not allowed by SMAL, and the semantics is undefined.

B.3 Abstract specification of Hector

```
;;; Hector specification, in a SMAL-like style
;;;                                           ; Derek Beatty 3/93, 5/93.

;;; -------------------------------------------------------------------
;;; Preliminaries
;;; -------------------------------------------------------------------

;;; Types
(define majorOpcode-type (mk-word-type 4))
(define minorOpcode-type (mk-word-type 2))
(define testCode-type (mk-word-type 4))
(define sourceMode-type (mk-word-type 2))
(define destinationMode-type (mk-word-type 2))
(define addr-type (mk-word-type 16))
(define addrSum-type (mk-word-type 17))
(define reg-type (mk-word-type 4))
(define word-type (mk-word-type 16))
(define control-type (mk-enum-type 'reset 'nmi 'irq 'dma 'sst 'wait
                                   'run2 'run3 'run4 'run5 'run6 'run7))

;;; System variables
(define mem (state addr-type word-type))
(define reg (state reg-type word-type))
(define cyCC (state bit-type))
(define ovCC (state bit-type))
(define ngCC (state bit-type))
(define zeCC (state bit-type))
(define intCC (state bit-type))
(define control (state control-type))
(define invariant (state bit-type))
(define pending-interrupt (state bit-type))

;;; -------------------------------------------------------------------
```

```
;;; Global definitions
;;; -------------------------------------------------------------------

;; Special registers
(define PC (encodeU 15 reg-type))
(define SP (encodeU 14 reg-type))
(define INT (encodeU 13 reg-type))
(define NMI (encodeU 12 reg-type))
(define SWI (encodeU 11 reg-type))
(define END (encodeU 10 reg-type))
(define BEGN (encodeU 9 reg-type))
(define SIZE (encodeU 8 reg-type))
(define STEP (encodeU 7 reg-type))
(define R0 (encodeU 0 reg-type))

;; Binary operation op-codes-----
(define add  '(0 0 0 0))        ;
(define addc '(0 0 0 1))        ;
(define sub  '(0 0 1 0))        ;
(define and_op  '(0 0 1 1))     ;
(define subc '(0 1 0 0))        ;
(define or_op   '(0 1 0 1))     ;
(define xor  '(0 1 1 0))        ;
;; Unary operation op-codes----------------------
(define u1Group '(1 0 0 0))     ; Group 1       ;
(define not_op '(0 0))                          ;
(define neg '(0 1))                             ;
(define inc '(1 0))                             ;
(define dec '(1 1))                             ;
(define u2Group '(1 0 0 1))     ; Group 2 ------;
(define shl '(0 0))                             ;
(define rol '(0 1))                             ;
(define shr '(1 0))                             ;
(define ror '(1 1))                             ;
;; Binary test op-codes----------
(define cmp '(1 0 1 0))         ;
(define btst '(1 0 1 1))        ;
;; Conditional branch group op-codes --------------------
(define bjscGroup '(1 1 0 0))                           ;
(define bra '(0 0))                                     ;
;; Tests for conditional branch--                       ;
(define tvc '(0 0 0 0))         ; overflow clear        ;
(define tpl '(0 0 0 1))         ; plus                  ;
(define tge '(0 0 1 0))         ; greater or equal      ;
```

```
(define trn '(0 0 1 1))         ; never                 ;
(define tle '(0 1 0 0))         ; less or equal         ;
(define tne '(0 1 0 1))         ; not equal             ;
(define tls '(0 1 1 0))         ; less                  ;
(define tcc '(0 1 1 1))         ; carry clear           ;
(define tvs '(1 0 0 0))         ; overlow set           ;
(define tmi '(1 0 0 1))         ; minus                 ;
(define tlt '(1 0 1 0))         ; less than             ;
(define tra '(1 0 1 1))         ; always                ;
(define tgt '(1 1 0 0))         ; greater than          ;
(define teq '(1 1 0 1))         ; equal                 ;
(define thi '(1 1 1 0))         ; higher                ;
(define tcs '(1 1 1 1))         ; carry set             ;
(define jsr '(0 1))                                     ;
(define swap '(1 0))                                    ;
(define clr '(1 1))                                     ;
;; Data movement----------
(define move '(1 1 0 1)) ;
;; Flag and push group op-codes -
(define tslpGroup '(1 1 1 0))   ;
(define test_op '(0 0))         ;
(define stf '(0 1))             ;
(define ldf '(1 0))             ;
(define push '(1 1))            ;
;; Special group op-codes--------
(define otherGroup '(1 1 1 1))  ;
(define sec_op '(0 0 0))                ;
(define clc_op '(0 0 1))                ;
(define sei_op '(0 1 0))                ;
(define cli_op '(0 1 1))                ;
(define rti_op '(1 0 0))                ;
(define swi_op '(1 0 1))        ;
(define exch '(1 1 0))          ;
(define srch '(1 1 1))          ;

;;; Addressing modes -----------------------------------------
(define regMode '(0 0))         ; Source, Destination, or Branch
(define indMode '(0 1))         ; Source, Destination, or Branch
(define postIncMode '(1 0))     ; Source or Destination
(define relMode '(1 0))         ; Branch
(define indxMode '(1 1))        ; Source, Destination, or Branch

;;; Functions that instructions compute
(define (addFn u v) (vec-add u v))
```

```
(define (addcFn u v cy) (?: cy  (vec-+1 (vec-add u v))
                                (vec-add u v)))
(define (subFn u v) (vec-subtract u v))
(define (subcFn u v cy) (?: cy  (vec-minus-1 (vec-subtract u v))
                                (vec-subtract u v)))

;;; Functions that binary operations compute

(define (binOp op v u cy ov ng ze)
  (?: (vec-== op add)  (cdr (addFn u v))
   (?: (vec-== op addc) (cdr (addcFn u v cy))
    (?: (vec-== op sub)  (cdr (subFn u v))
     (?: (vec-== op subc) (cdr (subcFn u v cy))
      (?: (vec-== op and_op) (vec-&& u v)
       (?: (vec-== op or_op) (vec-// u v)
        (?: (vec-== op xor) (vec-^ u v)))))))))

;;; Carry condition of binary operations

(define (binCy op v u cy ov ng ze)
  (?: (vec-== op add)  (car (addFn u v))
   (?: (vec-== op addc) (car (addcFn u v cy))
    (?: (// (vec-== op sub) (vec-== op cmp))
        (vec-< (cons 0 u) (cons 0 v))           ; unsigned test
     (?: (vec-== op subc)
         (vec-< (cons 0 (cons 0 u))
                (cons
                 0 (vec-add v `(0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 ,cy))))
      (?: (// (vec-== op and_op) (vec-== op btst)) cy
       (?: (vec-== op or_op) cy
        (?: (vec-== op xor) cy))))))))

;;; Overflow condition of binary operations

(define (binOv op v u cy ov ng ze)
  (?: (vec-== op add)  (&& (!= (cadr (addFn u v)) (car u))
                           (== (car u) (car v)))
   (?: (vec-== op addc) (&& (^ (cadr (addcFn u v cy)) (car u))
                            (! (^ (car u) (car v))))
    (?: (// (vec-== op sub) (vec-== op cmp))
        (// (&&    (car v) (! (car u))   (cadr (subFn u v)))
            (&& (! (car v))   (car u) (! (cadr (subFn u v)))))
     (?: (vec-== op subc)
         (// (&&    (car v) (! (car u))   (cadr (subcFn u v cy)))
```

```
             (&& (! (car v))   (car u) (! (cadr (subcFn u v cy)))))
      (?: (// (vec-== op and_op) (vec-== op btst)) ov
       (?: (vec-== op or_op) ov
        (?: (vec-== op xor) ov))))))))

;;; Negative conditions of binary operations

(define (binNeg op v u cy ov ng ze)
  (?: (vec-== op add)  (cadr (addFn u v))
   (?: (vec-== op addc) (cadr (addcFn u v cy))
    (?: (// (vec-== op sub) (vec-== op cmp))
        (cadr (subFn u v))
     (?: (vec-== op subc)
         (cadr (subcFn u v cy))
      (?: (// (vec-== op and_op) (vec-== op btst)) (car (&& u v))
       (?: (vec-== op or_op) (car (// u v))
        (?: (vec-== op xor) (car (^ u v))))))))))

;;; Zero conditions of binary operations

(define (binZer op v u cy ov ng ze)
  (?: (vec-== op add)
      (vec-== (cdr (addFn u v)) (encodeU 0 word-type))
   (?: (vec-== op addc)
       (vec-== (cdr (addcFn u v cy)) (encodeU 0 word-type))
    (?: (// (vec-== op sub) (vec-== op cmp))
        (vec-== (cdr (subFn u v)) (encodeU 0 word-type))
     (?: (vec-== op subc)
         (vec-== (cdr (subcFn u v cy)) (encodeU 0 word-type))
      (?: (// (vec-== op and_op) (vec-== op btst))
          (vec-== (&& u v) (encodeU 0 word-type))
       (?: (vec-== op or_op)
           (vec-== (// u v) (encodeU 0 word-type))
        (?: (vec-== op xor)
            (vec-== (^ u v) (encodeU 0 word-type))))))))))

;;; Functions and condition code update of unary operators

(define (negate v)
   (vec-difference (encodeU 0 word-type) v))
(define (increment v) (vec-+1 v))
(define (decrement v) (vec-minus-1 v))

(define (unOp major minor v cy ov ng ze)
```

```
  (?: (&& (vec-== major u1Group) (vec-== minor not_op)) (! v)
   (?: (&& (vec-== major u1Group) (vec-== minor neg)) (negate v)
    (?: (&& (vec-== major u1Group) (vec-== minor inc)) (increment v)
     (?: (&& (vec-== major u1Group) (vec-== minor dec)) (decrement v)
      (?: (&& (vec-== major u2Group) (vec-== minor shl))
          (cdr (append v '(0)))
       (?: (&& (vec-== major u2Group) (vec-== minor rol))
           (cdr (append v (list cy)))
        (?: (&& (vec-== major u2Group) (vec-== minor shr))
            (rdc (append (list (car v)) v))
         (?: (&& (vec-== major u2Group) (vec-== minor ror))
             (rdc (append (list cy) v)))))))))))

(define (unCy major minor v cy ov ng ze)
  (?: (&& (vec-== major u1Group) (vec-== minor not_op)) cy
   (?: (&& (vec-== major u1Group) (vec-== minor neg))
       (vec-== (negate v) (encodeU 0 word-type))
    (?: (&& (vec-== major u1Group) (vec-== minor inc))
        (vec-== v (encodeS -1 word-type))
     (?: (&& (vec-== major u1Group) (vec-== minor dec))
         (vec-==  v (encodeU 0 word-type))
      (?: (&& (vec-== major u2Group) (vec-== minor shl)) (car v)
       (?: (&& (vec-== major u2Group) (vec-== minor rol)) (car v)
        (?: (&& (vec-== major u2Group) (vec-== minor shr)) (rac v)
         (?: (&& (vec-== major u2Group) (vec-== minor ror))
             (rac v))))))))))

(define (unOv major minor v cy ov ng ze)
  (?: (&& (vec-== major u1Group) (vec-== minor not_op)) ov
   (?: (&& (vec-== major u1Group) (vec-== minor neg))
       (vec-== (negate v) (encodeU #x8000 word-type))
    (?: (&& (vec-== major u1Group) (vec-== minor inc))
        (vec-== v (encodeU #x7fff word-type))
     (?: (&& (vec-== major u1Group) (vec-== minor dec))
         (vec-== v (encodeU #x8000 word-type))
      (?: (&& (vec-== major u2Group) (vec-== minor shl)) ov
       (?: (&& (vec-== major u2Group) (vec-== minor rol)) ov
        (?: (&& (vec-== major u2Group) (vec-== minor shr)) ov
         (?: (&& (vec-== major u2Group) (vec-== minor ror)) ov)))))))))

(define (unNeg major minor v cy ov ng ze)
  (?: (&& (vec-== major u1Group) (vec-== minor not_op)) (! (car v))
   (?: (&& (vec-== major u1Group) (vec-== minor neg))
       (car (negate v))
```

```
    (?: (&& (vec-== major u1Group) (vec-== minor inc))
        (car (increment v))
     (?: (&& (vec-== major u1Group) (vec-== minor dec))
         (car (decrement v))
      (?: (&& (vec-== major u2Group) (vec-== minor shl)) (cadr v)
       (?: (&& (vec-== major u2Group) (vec-== minor rol)) (cadr v)
        (?: (&& (vec-== major u2Group) (vec-== minor shr)) (car v)
         (?: (&& (vec-== major u2Group) (vec-== minor ror)) cy)))))))))

(define (unZer major minor v cy ov ng ze)
  (?: (&& (vec-== major u1Group) (vec-== minor not_op))
      (vec-== (! v) (encodeU 0 word-type))
   (?: (&& (vec-== major u1Group) (vec-== minor neg))
       (vec-== (negate v) (encodeU 0 word-type))
    (?: (&& (vec-== major u1Group) (vec-== minor inc))
        (vec-== (increment v) (encodeU 0 word-type))
     (?: (&& (vec-== major u1Group) (vec-== minor dec))
         (vec-== (decrement v) (encodeU 0 word-type))
      (?: (&& (vec-== major u2Group) (vec-== minor shl))
          (vec-== (cdr (append v '(0))) (encodeU 0 word-type))
       (?: (&& (vec-== major u2Group) (vec-== minor rol))
           (vec-== (cdr (append v (list cy)))
                   (encodeU 0 word-type))
        (?: (&& (vec-== major u2Group) (vec-== minor shr))
            (vec-== (rdc (append (list (car v)) v))
                    (encodeU 0 word-type))
         (?: (&& (vec-== major u2Group) (vec-== minor ror))
             (vec-== (rdc (append (list cy) v))
                     (encodeU 0 word-type))))))))))

(define (testCond t cy ov ng ze)
   (?: (vec-== t tvc) (! ov)
    (?: (vec-== t tpl) (! ng)
     (?: (vec-== t tge) (! (^ ng ov))
      (?: (vec-== t trn) 0
       (?: (vec-== t tle) (// (ze (^ ng ov)))
        (?: (vec-== t tne) (! ze)
         (?: (vec-== t tls) (// cy ze)
          (?: (vec-== t tcc) (! cy)
           (?: (vec-== t tvs) ov
            (?: (vec-== t tmi) ng
             (?: (vec-== t tlt) (^ ng ov)
              (?: (vec-== t tra) 1
               (?: (vec-== t tgt) (! (// ze (^ ng ov)))
```

```
                (?: (vec-== t teq) ze
                 (?: (vec-== t thi) (! (// cy ze))
                  (?: (vec-== t tcs) cy))))))))))))))))

;;; ----------------------------------------------------------------------
;;;    A S S E R T I O N S
;;; ----------------------------------------------------------------------

;;; Initialization

(define (init)
  (decomp ((control 'reset))
     ; ==>
          ((invariant 0)              ; NB: 0 creates proof obligation.
           (reg PC  (encodeU 0 addr-type))
           (reg SP  (encodeU 0 addr-type))
           (reg INT (encodeU 4 addr-type))
           (reg NMI (encodeU 2 addr-type)))))

;;; Non maskable interrupt: push flags and PC, set interrupt flag,
;;; and begin executing at NMI location.

(define (nmi-int)
  (decl p addr-type)
  (decl s addr-type)
  (decl n addr-type)
  (decl l addr-type) (decl d word-type)
  (decl cy bit-type) (decl ov bit-type) (decl ng bit-type)
  (decl ze bit-type) (decl int bit-type)
  (decl r reg-type) (decl w word-type)
  (let* ((s- (decrement s))
         (s-- (decrement s-))
         (flags (append (list cy ov ng ze int)
                        '(- - - - - - - - - - -))))
    (decomp ((control 'nmi)
             (invariant 1)
             (mem l d '#f)
             (-> (&& (vec-!= r NMI) (vec-!= r SP) (vec-!= r PC))
                 (reg r w))
             (reg NMI n)
             (reg SP s)
             (reg PC p)
             (cyCC cy) (ovCC ov) (ngCC ng) (zeCC ze) (intCC int)
             (-> (vec-!= r R0)
```

```
                 (exists (w) (reg R0 w))))
      ; ==>
            ((invariant 0)
             (-> (&& (vec-!= l s-) (vec-!= l s--))
                 (mem l d '#f))
             (-> (vec-!= r SP) (reg r w))
             (mem s- flags -4)
             (mem s-- p -3)
             (cyCC cy) (ovCC ov) (ngCC ng) (zeCC ze) (intCC 1)
             (reg NMI n)
             (reg SP s--)
             (reg PC n)))))

(define (irq-int) ...)
(define (dma-int) ...)
(define (sst-int) ...)
(define (wait-int) ...)

;;; Two-operand register-to-register;  2 execution cycles.

(define (twrgrg)
 (decl p addr-type)
 (decl u word-type) (decl v word-type)
 (decl l addr-type) (decl d word-type)
 (decl dst reg-type) (decl src reg-type)
 (decl cy bit-type) (decl ov bit-type) (decl ng bit-type)
 (decl ze bit-type) (decl int bit-type)
 (decl r reg-type) (decl w word-type)
 (-> (// (vec-== op add) (vec-== op addc)
         (vec-== op sub) (vec-== op subc)
         (vec-== op and_op) (vec-== op or_op) (vec-== op xor))
    (let* ((p/ (increment p))
           (u/ (?: (vec-== src PC) p/ u))
           (v/ (?: (vec-== dst PC) p/
                   (?: (vec-== src dst) u v))))
      (decomp ((control 'run2)
               (invariant 1)
               (mem l d '#f)
               (reg PC p)
               (mem p (append op regMode regMode src dst) 0)
               (cyCC cy) (ovCC ov) (ngCC ng) (zeCC ze) (intCC int)
               (-> (&& (vec-!= r PC)
                       (vec-!= r src) (vec-!= r dst))
                   (reg r w))
```

```
              (-> (vec-!= src PC) (reg src u))
              (-> (&& (vec-!= dst PC) (vec-!= src dst)) (reg dst v))
              (-> (&& (vec-!= r R0) (vec-!= src R0) (vec-!= dst R0))
                  (exists (w) (reg R0 w))))
        ; ==>
             ((invariant 0)
              (-> (&& (vec-!= r PC) (vec-!= r src) (vec-!= r dst))
                  (reg r w))
              (-> (&& (vec-!= src dst) (vec-!= src PC)) (reg src u/))
              (-> (vec-!= dst PC) (reg PC p/))
              (reg dst (binOp op u/ v/ cy ov ng ze))
              (cyCC (binCy op u/ v/ cy ov ng ze))
              (ovCC (binOv op u/ v/ cy ov ng ze))
              (ngCC (binNeg op u/ v/ cy ov ng ze))
              (zeCC (binZer op u/ v/ cy ov ng ze))
              (intCC int)
              (mem l d '#f))))))


(define (twrgid) ...)
(define (twrgic) ...)
(define (twrgnx) ...)
(define (twidrg) ...)
(define (twidid) ...)
(define (twidic) ...)
(define (twidnx) ...)
(define (twicrg) ...)
(define (twicid) ...)
(define (twicic) ...)
(define (twicnx) ...)
(define (twnxrg) ...)
(define (twnxid) ...)
(define (twnxic) ...)
(define (twnxnx) ...)
(define (mvrgrg) ...)
(define (mvrgid) ...)
(define (mvrgic) ...)
(define (mvrgnx) ...)
(define (mvidrg) ...)
(define (mvidid) ...)
(define (mvidic) ...)
(define (mvidnx) ...)
(define (stfrg) ...)
(define (stfid) ...)
```

```
(define (stfic) ...)
(define (stfnx) ...)
(define (onerg) ...)
(define (oneid) ...)
(define (oneic) ...)
(define (onenx) ...)
(define (nsrgrg) ...)
(define (nsrgid) ...)
(define (nsrgic) ...)
(define (nsrgnx) ...)
(define (nsidrg) ...)
(define (nsidid) ...)
(define (nsidic) ...)
(define (nsidnx) ...)
(define (nsicrg) ...)
(define (nsicid) ...)
(define (nsicic) ...)
(define (nsicnx) ...)
(define (nsnxrg) ...)
(define (nsnxid) ...)
(define (nsnxic) ...)
(define (nsnxnx) ...)
(define (mvicrg) ...)
(define (mvicid) ...)
(define (mvicic) ...)
(define (mvicnx) ...)
(define (mvnxrg) ...)
(define (mvnxid) ...)
(define (mvnxic) ...)
(define (mvnxnx) ...)
(define (ldfrg) ...)
(define (ldfid) ...)
(define (ldfic) ...)
(define (ldfnx) ...)
(define (pshrg) ...)
(define (pshid) ...)
(define (pshic) ...)
(define (pshnx) ...)
(define (sec) ...)
(define (rti) ...)
(define (clc) ...)
(define (swi-assn) ...)
(define (sei) ...)
(define (cli) ...)
```

```
(define (cbric-n) ...)
(define (cbrnx-n) ...)
(define (cbrrg-n) ...)
(define (cbrid-n) ...)
(define (cbric-t) ...)
(define (cbrnx-t) ...)
(define (cbrrg-t) ...)
(define (cbrid-t) ...)
(define (ccarg-n) ...)
(define (ccaid-n) ...)
(define (ccaic-n) ...)
(define (ccanx-n) ...)
(define (ccarg-t) ...)
(define (ccaid-t) ...)
(define (ccaic-t) ...)
(define (ccanx-t) ...)
(define (swaprg) ...)
(define (swapid) ...)
(define (swapic) ...)
(define (swapnx) ...)

;;; Clear a register; 2 execution cycles.

(define (clrrg)
 (decl l addr-type)
 (decl d word-type)
 (decl p addr-type)
 (decl src reg-type)
 (decl cy bit-type) (decl ov bit-type) (decl ng bit-type)
 (decl ze bit-type) (decl int bit-type)
 (decl r reg-type) (decl w word-type)
 (decomp
    ((control 'run2)
     (pending-interrupt 0)
     (invariant 1)
     (mem l d '#f)
     (reg pc p)
     (mem p (append bjscGroup regMode clr src src) 0)
     (cyCC cy) (ovCC ov) (ngCC ng) (zeCC ze) (intCC int)
     (-> (&& (vec-!= r PC) (vec-!= r src))
         (reg r w))
     (-> (vec-!= src PC)
         (exists (w) (reg src w)))
     (-> (&& (vec-!= r R0) (vec-!= src R0))
```

```
         (exists (w) (reg R0 w))))
; ==>
    ((invariant 0)
     (mem l d '#f)
     (-> (&& (vec-!= r pc) (vec-!= r src))
         (reg r w))
     (-> (vec-!= src pc)
         (reg pc (increment p)))
     (cyCC cy) (ovCC ov) (ngCC ng) (zeCC ze) (intCC int)
     (reg src (encodeU 0 word-type)))))

;;; Clear memory location addressed by a register; 3 execution cycles.

(define (clrid)
 (decl l addr-type)
 (decl d word-type)
 (decl p addr-type)
 (decl b addr-type)
 (decl src reg-type)
 (decl cy bit-type) (decl ov bit-type) (decl ng bit-type)
 (decl ze bit-type) (decl int bit-type)
 (decl r reg-type) (decl w word-type)
 (let ((b/ (?: (vec-== src PC) (vec-+1 p) b)))
   (decomp
    ((control 'run3)
     (invariant 1)
     (mem l d '#f)
     (reg pc p)
     (mem p (append bjscGroup indMode clr src src) 0)
     (cyCC cy) (ovCC ov) (ngCC ng) (zeCC ze) (intCC int)
     (-> (&& (vec-!= r PC) (vec-!= r src))
         (reg r w))
     (-> (vec-!= src PC)
         (reg src b))
     (-> (&& (vec-!= r R0) (vec-!= src R0))
         (exists (w) (reg R0 w))))
  ; ==>
    ((invariant 0)
     (mem l d '#f)
     (-> (&& (vec-!= r PC) (vec-!= r src))
         (reg r w))
     (-> (vec-!= src PC)
         (reg src b))
     (reg PC (increment p))
```

```
     (cyCC cy) (ovCC ov) (ngCC ng) (zeCC ze) (intCC int)
     (mem b/ (encodeU 0 word-type) -1)))))

;;; Clear mem addressed by a register & incr. reg.; 3 execution cycles.

(define (clric)
 (decl l addr-type)
 (decl d word-type)
 (decl p addr-type)
 (decl b addr-type)
 (decl src reg-type)
 (decl cy bit-type) (decl ov bit-type) (decl ng bit-type)
 (decl ze bit-type) (decl int bit-type)
 (decl r reg-type) (decl w word-type)
 (let ((b/ (?: (vec-== src PC) (vec-+1 p) b)))
   (decomp
    ((control 'run3)
     (invariant 1)
     (mem l d '#f)
     (reg pc p)
     (mem p (append bjscGroup postIncMode clr src src) 0)
     (cyCC cy) (ovCC ov) (ngCC ng) (zeCC ze) (intCC int)
     (-> (&& (vec-!= r PC) (vec-!= r src))
         (reg r w))
     (-> (vec-!= src PC)
         (reg src b))
     (-> (&& (vec-!= r R0) (vec-!= src R0))
         (exists (w) (reg R0 w))))
  ; ==>
    ((invariant 0)
     (mem l d '#f)
     (-> (&& (vec-!= r PC) (vec-!= r src))
         (reg r w))
     (reg src (increment b/))
     (-> (vec-!= src PC)
         (reg PC (increment p)))
     (cyCC cy) (ovCC ov) (ngCC ng) (zeCC ze) (intCC int)
     (mem b/ (encodeU 0 word-type) -1)))))

;;; Clear mem addr. given by base + register; 5 execution cycles.

(define (clrnx)
 (decl l addr-type)
 (decl d word-type)
```

```
 (decl p addr-type)
 (decl b addr-type)
 (decl j addr-type)
 (decl w word-type)
 (decl src reg-type) (decl r reg-type)
 (decl cy bit-type) (decl ov bit-type) (decl ng bit-type)
 (decl ze bit-type) (decl int bit-type)
 (let ((j/ (?: (vec-== src PC) (vec-+1 p) j)))
   (decomp
    ((control 'run5)
     (invariant 1)
     (mem l d '#f)
     (reg PC p)
     (mem p (append bjscGroup indxMode clr src src) 0)
     (mem (increment p) b 2)
     (cyCC cy) (ovCC ov) (ngCC ng) (zeCC ze) (intCC int)
     (-> (&& (vec-!= r PC) (vec-!= r src))
         (reg r w))
     (-> (vec-!= src PC)
         (reg src j))
     (-> (&& (vec-!= r R0) (vec-!= src R0))
         (exists (w) (reg R0 w))))
  ; ==>
    ((invariant 0)
     (mem l d '#f)
     (-> (&& (vec-!= r PC) (vec-!= r src))
         (reg r w))
     (-> (vec-!= src PC)
         (reg src j))
     (reg PC (increment (increment p)))
     (cyCC cy) (ovCC ov) (ngCC ng) (zeCC ze) (intCC int)
     (mem (vec-+ b j/) (encodeU 0 word-type) -1)))))

(define (tstrg) ...)
(define (tstid) ...)
(define (tstic) ...)
(define (tstnx) ...)
```

B.4 Mapping onto the Hector chip

```
; State mappings for Hector                    ; Derek Beatty 3/93, 5/93.
```

```
;;; This file defines the following state mapping functions:
;;;
;;;  MEM  REG
;;;  cyCC  ovCC  ngCC  zeCC  intCC
;;;  CONTROL  INVARIANT   PENDING-INTERRUPT
;;;

;;; --------------------------------------------------------------------
;;; Variable ordering
;;; --------------------------------------------------------------------

(define-var-classes 16 ctl-class reg-class data-class bit-class)
(define-type-class reg-type reg-class)
(define-type-class word-type data-class)
(define-type-class addr-type data-class)
(define-type-class bit-type data-class)

;;; --------------------------------------------------------------------
;;; Constants
;;; --------------------------------------------------------------------

(define idle-microstate '(1 1 0 1 1 0 1 0))
;;;     Special registers.
(define R0 '(0 0 0 0))
(define T1 '(1 0 0 0 0))                ; Not visible to programmer.
(define T2 '(1 0 0 0 1))                ; Not visible to programmer.


;;; --------------------------------------------------------------------
;;; Circuit Nodes
;;; --------------------------------------------------------------------

;;;     Control lines:
(define-nodes (phi1 "p1") (phi2 "p2"))  ; Clocks.
(define-nodes _sst _int _nmi _dma _wait); Active-low control inputs.
(define-nodes (reset-nd "reset") test)  ; Active-high control inputs.
(define-nodes rd_wr data_prg _dmack _vma); Bus control outputs.
;;;     Bus control line drivers:
(define-nodes
  (rd_wr-drive "200_0/REFL_0/PROC_0/rd_wr/fake_assert")
  (rd_wr-in "200_0/REFL_0/PROC_0/rd_wr/fake_in")
  (data_prg-drive "200_0/REFL_0/PROC_0/data_prg/fake_assert")
  (data_prg-in "200_0/REFL_0/PROC_0/data_prg/fake_in")
  (_dmack-drive "200_0/REFL_0/PROC_0/_dmack/fake_assert")
```

```
  (_dmack-in "200_0/REFL_0/PROC_0/_dmack/fake_in")
  (_vma-drive "200_0/REFL_0/PROC_0/_vma/fake_assert")
  (_vma-in "200_0/REFL_0/PROC_0/_vma/fake_in"))
;;;     Busses:
(define data-bus (make-nvec "d~a" (name-range 0 16)))
(define addr-bus (make-nvec "a~a" (name-range 0 16)))
;;;     Bus drivers:
(define data-bus-drive
  (make-nvec "200_0/REFL_0/PROC_0/d~a/fake_assert" (name-range 0 16)))
(define data-bus-in
  (make-nvec "200_0/REFL_0/PROC_0/d~a/fake_in" (name-range 0 16)))
(define addr-bus-drive
  (make-nvec "200_0/REFL_0/PROC_0/a~a/fake_assert" (name-range 0 16)))
(define addr-bus-in
  (make-nvec "200_0/REFL_0/PROC_0/a~a/fake_in" (name-range 0 16)))
;;;     Internal nodes:
(define-nodes fc  fv  fn  fz  fi        ; Condition codes and flags.
              fci fvi fni fzi fii)      ; Their input bits.
(define ccodes (list fc fv fn fz fi))
(define regfile                         ; Reg. array, 18 w. of 16 bits.
  (make-ary "r~a.~~a" (name-range 0 18) (name-range 0 16)))
(define cc-bits-of-regfile              ; Upper 5 bits of reg. array.
  (make-ary "r~a.~~a" (name-range 0 18) (name-range 11 16)))
(define non-cc-bits-of-regfile          ; Lower 11 bits.
  (make-ary "r~a.~~a" (name-range 0 18) (name-range 0 11)))

(define src (make-nvec "sr~adrv" (name-range 0 16))); Int. source bus.
(define dst (make-nvec "ds~adrv" (name-range 0 16))); " destination bus.
(define u-adr (make-nvec "ra~a" (name-range 0 8))) ; Microcode address.
(define src-adr (make-nvec "src_adr~a" (name-range 0 5))) ; Source adr.
(define dst-adr (make-nvec "dst_adr~a" (name-range 0 5))) ; Dest. "
(define-nodes
  (nmi-lat "200_0/REFL_0/PROC_0/DPTH_0/CCB_0/NMILAT_0/bit")) ; NMI latch

;;; -----------------------------------------------------------------
;;; ALU state to be weakend to keep BDD sizes down:

(define nano-addr
  (make-nvec "200_0/REFL_0/PROC_0/DPTH_0/ALUF_0/ALUMP_0/in~a"
             (name-range 0 5)))
(define nano-word
  (make-nvec "200_0/REFL_0/PROC_0/DPTH_0/ALUF_0/ALUMP_0/out~a"
             '(0 1 2 3 4 5 6 7
               9 10 11 12 13 14
```

```
               16 17 18 19 20
               22 23 24 25 26 27 28 29 30 31)))
(define src-inv
 (append
 (make-nvec
  "200_0/REFL_0/PROC_0/DPTH_0/ALUF_0/ALU_0/LALUB_~a/ALUB_0/6_2906_2384#"
  (name-range 0 15))
 (make-nvec
  "200_0/REFL_0/PROC_0/DPTH_0/ALUF_0/ALU_0/HALUB_~a/ALUB_0/6_2906_2384#"
  '(0))))
(define dst-inv
 (append
 (make-nvec
  "200_0/REFL_0/PROC_0/DPTH_0/ALUF_0/ALU_0/LALUB_~a/ALUB_0/6_2874_2384#"
  (name-range 0 15))
 (make-nvec
  "200_0/REFL_0/PROC_0/DPTH_0/ALUF_0/ALU_0/HALUB_~a/ALUB_0/6_2874_2384#"
  '(0))))
(define cchain
 (append
 (make-nvec
  "200_0/REFL_0/PROC_0/DPTH_0/ALUF_0/ALU_0/LALUB_~a/ALUB_0/SRFF_0/d"
  (name-range 0 15))
 (make-nvec
  "200_0/REFL_0/PROC_0/DPTH_0/ALUF_0/ALU_0/HALUB_~a/ALUB_0/SRFF_0/d"
  '(0))))
(define cchain-inv
 (append
 (make-nvec
  "200_0/REFL_0/PROC_0/DPTH_0/ALUF_0/ALU_0/LALUB_~a/ALUB_0/SRFF_0/_d"
  (name-range 0 15))
 (make-nvec
  "200_0/REFL_0/PROC_0/DPTH_0/ALUF_0/ALU_0/HALUB_~a/ALUB_0/SRFF_0/_d"
  '(0))))
(define wr (make-nvec "wr~a" (name-range 0 16))); " destination bus.

(define weaken-alu-state
  (append
   wr src dst src-inv dst-inv cchain cchain-inv nano-addr nano-word))

;;; ------------------------------------------------------------------
;;;  Timing
;;; ------------------------------------------------------------------
;;; Timing:
```

```
;;;     Time points:
(define (begin-cycle N) (* 7 N))
(define (end-cycle N) (begin-cycle (+ N 1)))
;;;     Intervals:
(define (during-cycle N f) (over (begin-cycle N) (end-cycle N) f))
(define (before-phi1 N f)
  (over (+ (begin-cycle N) 0) (+ (begin-cycle N) 1) f))
(define (when-phi1-rises N f) (at (+ (begin-cycle N) 1) f))
(define (during-phi1 N f)
  (over (+ (begin-cycle N) 1) (+ (begin-cycle N) 3) f))
(define (amid-phi1 N f) (at (+ (begin-cycle N) 2) f))
(define (after-phi1 N f)
  (over (+ (begin-cycle N) 3) (+ (begin-cycle N) 4) f))
(define (during-phi2 N f)
  (over (+ (begin-cycle N) 4) (+ (begin-cycle N) 6) f))
(define (as-phi2-falls N f)
  (over (+ (begin-cycle N) 5) (+ (begin-cycle N) 6) f))
(define (after-phi2 N f)
  (over (+ (begin-cycle N) 6) (+ (begin-cycle N) 7) f))
;;;     Clock cycle:
;         (when-phi1-rises
;          N (itf-add-extra true-tempula
;                           (map xtra-weaken-module alu-subntwks)))
;    (conj (before-phi1
;          N (itf-add-extra true-tempula
;                           (map xtra-weaken-node (append src dst))))
(define (cycle N)
  (let* ((hi1 (is phi1 1))
         (lo1 (is phi1 0))
         (hi2 (is phi2 1))
         (lo2 (is phi2 0))
         (low (conj lo1 lo2)))
    (conj (before-phi1 N (conj lo1 lo2))
          (when-phi1-rises
           N (itf-add-extra
              true-tempula
              (map xtra-freeze-node-at-X weaken-alu-state)))
          (during-phi1 N (conj hi1 lo2))
          (amid-phi1
           N (itf-add-extra
              true-tempula
              (map xtra-thaw-node weaken-alu-state)))
          (after-phi1 N (conj lo1 lo2))
          (during-phi2 N (conj lo1 hi2))
```

```
          (after-phi2 N (conj lo1 lo2))
          ;; slight overlap w/ next cycle:
          (before-phi1 (+ N 1) (conj lo1 lo2)))))

;;;     Multiple cycles:
(define (n-cycles N) (apply conj (map cycle (range 0 N))))
(define (during-cycles beg end f)
  (over (begin-cycle beg) (end-cycle end) f))

(define (after n p) (at (end-cycle n) p))

;;;     Bidirectional bus:
(define zero-word '(0 0 0 0  0 0 0 0  0 0 0 0  0 0 0 0))
(define ones-word '(1 1 1 1  1 1 1 1  1 1 1 1  1 1 1 1))
(define (drive-data-bus) (vec-is data-bus-drive ones-word))
(define (undrive-data-bus) (vec-is data-bus-drive zero-word))
(define (drive-addr-bus) (vec-is addr-bus-drive ones-word))
(define (undrive-addr-bus) (vec-is addr-bus-drive zero-word))
(define (drive-ctl-bus)
  (conj (is rd_wr-drive 1) (is data_prg-drive 1)
        (is _dmack-drive 1) (is _vma-drive 1)))
(define (undrive-ctl-bus)
  (conj (is rd_wr-drive 0) (is data_prg-drive 0)
        (is _dmack-drive 0) (is _vma-drive 0)))
(define (undrive-busses)
  (conj (undrive-data-bus) (undrive-addr-bus) (undrive-ctl-bus)))



;;; -----------------------------------------------------------------------
;;;  State Mapping Functions
;;; -----------------------------------------------------------------------

;;; The MEM predicate.
;;;
;;; This predicate indicates data storage in the memory system.
;;; Since the memory system is actually separate from the processor
;;; that we are verifying, the predicate is modified to decompose
;;; the memory state into the actions that the processor takes to
;;; act on that state.  It takes an extra argument, a "hint"
;;; establishing the time (in cycles) when a memory operation occurs.

(define (mem a w hint)
  (check-type a addr-type)
```

```
  (check-type w word-type)
  (if (not hint)
      ;; Hint is '#F: no memory operation actually occurs.
      true-tempula
      ;; Hint has a value, telling us when the memory operation occurs.
      ;; If in the antecedent, a READ operation must occur soon, and
      ;; if in the consequent, a WRITE operation must have just occured.
      (let (
            ;; Read: consequent checks address and control bus.
            (ca (conj (as-phi2-falls hint
                                     (conj (is _vma 0)
                                           (vec-is addr-bus a)))
                      (during-phi2 hint
                                   (conj (is rd_wr 1)
                                         (is _dmack 1)))))
            ;; Read: antecedent asserts data onto data bus.
            (aa (conj (as-phi2-falls hint
                                     (conj (drive-data-bus)
                                           (vec-is data-bus-in w)))
                      (after-phi2 hint (undrive-data-bus))))
            ;; Write: no antecedent.
            (ac true-tempula)
            ;; Write: consequent checks address, data, and control bus.
            (cc (conj (during-phi2 hint
                                   (conj (is rd_wr 0)
                                         (is _dmack 1)))
                      (as-phi2-falls hint
                                     (conj (is _vma 0)
                                           (vec-is addr-bus a)
                                           (vec-is data-bus w))))))
        (make-asp '() '() (make-adf aa ca ac cc)))))

;;; -------------------------------------------------------------------

;;; The REG predicate.
;;;
;;; This predicate establishes data storage in the internal
;;; register file.

(define (extend lst len)
    (append! (make-list (- len (length lst)) 0) lst))

(define (reg r w)
  (check-type r reg-type)
```

```
  (check-type w word-type)
  (vec-is (word regfile (extend r 6)) w))

;;; ------------------------------------------------------------------

;;; The condition code predicates.

;;; These establish the values of the processor's condition-code flags.

(define (cyCC b)
  (check-type b bit-type)
  (is fc b))

(define (ovCC b)
  (check-type b bit-type)
  (is fv b))

(define (ngCC b)
  (check-type b bit-type)
  (is fn b))

(define (zeCC b)
  (check-type b bit-type)
  (is fz b))

(define (intCC b)
  (check-type b bit-type)
  (is fi b))

;;; ------------------------------------------------------------------

;;; The CONTROL predicate.
;;;
;;; This predicate establishes the control signals that are applied
;;; to the processor by its environment.  This is the only external
;;; input that the (abstract) system (processor plus memory) receives.
;;; Thus, this defines the clocking, the duration of instructions,
;;; and "abnormal" inputs such as interrupts and the reset sequence.

(define (control ctl)
  (check-type ctl control-type)
  ;; Abbreviation for mapping RUN controls of various lengths.
  (let* ((normal-controls (conj (is _sst 1)
                                (is _int 1)
```

```
                              (is _dma 1)
                              (is _wait 1)
                              (is reset-nd 0)
                              (is test 0)))
        (make-run
         (lambda (run-cycles)
           (make-asp ;; Duration:
                     (end-cycle (- run-cycles 1))
                     ;; Mapped temporal formula:
                     (conj (at (begin-cycle -2)
                               (n-cycles (+ run-cycles 2)))
                           (during-cycles -2 (- run-cycles 3)
                               (conj normal-controls
                                     (is _nmi 1)))
                           (during-cycles (- run-cycles 2)
                                          (- run-cycles 1)
                               (conj normal-controls
                                     (let ()
                                       (decl nmi-bit bit-type)
                                       (exists (nmi-bit)
                                               (is _nmi nmi-bit))))))
                     ;; Decomposition formulas:
                     '()))))

    ;; Mappings for individual abstract control signals.

    (case ctl

      ;; mapping for RESET operation.
      ((reset)
       (let ((reset-cycles 7))
         (make-asp ;; Duration:
                   (end-cycle reset-cycles)
                   ;; Mapped temporal formula:
                   (conj
                    (n-cycles (+ reset-cycles 1))
                    (undrive-busses)
                    (during-cycles 0 1
                       (conj (is reset-nd 1)
                             (is _nmi 1)
                             (is _sst 1)
                             (is _int 1)
                             (is _dma 1)
                             (is _wait 1)
```

```
                        (is test 0)))
               (during-cycles 2 (- reset-cycles 2)
                  (conj (is reset-nd 0)
                        (is _nmi 1)
                        (is _sst 1)
                        (is _int 1)
                        (is _dma 1)
                        (is _wait 1)
                        (is test 0)))
               (during-cycles (- reset-cycles 1) reset-cycles
                  (conj (is reset-nd 0)
                        (let ()
                          (decl nmi-bit bit-type)
                          (exists (nmi-bit)
                             (is _nmi nmi-bit)))
                        (is _sst 1)
                        (is _int 1)
                        (is _dma 1)
                        (is _wait 1)
                        (is test 0)))

               ;; These are, strictly speaking, wrong.
               ;; They do not map onto circuit inputs.
               ;; They are needed because of the
               ;; conservativism of the switch-level model.
               (let ()
                 (decl w word-type)
                 (conj
                  (before-phi1 2 (exists (w) (reg NMI w)))
                  (before-phi1 2 (exists (w) (reg R0 w)))
                  (before-phi1 5 (exists (w) (reg INT w)))
                  (before-phi1 5 (exists (w) (reg SP w)))
                  (before-phi1 5 (exists (w) (reg PC w))))))
              ;; Decomposition formulas:
              '())))

;; Mapping for NMI operation --- non maskable interrupt.
((nmi)
 (let ((nmi-cycles 7))
   (make-asp ;; Duration:
              (end-cycle (- nmi-cycles 1))
              ;; Mapped temporal formula:
              (conj
               (at (begin-cycle -2)
```

```
                    (n-cycles (+ nmi-cycles 2)))
                (during-cycles -2 -2
                               (conj normal-controls
                                     (is _nmi 0)))
                (during-cycles -1 (- nmi-cycles 3)
                               (conj normal-controls
                                     (is _nmi 1)))
                (during-cycles (- nmi-cycles 2) (- nmi-cycles 1)
                               (conj normal-controls
                                     (let ()
                                       (decl nmi-bit bit-type)
                                       (exists (nmi-bit)
                                               (is _nmi nmi-bit)))))

                ;; Needed because of switch-level conservatism
                ;; and model weakening: this says that the temp
                ;; register bits that don't hold flags do hold
                ;; some value.
                (let ()
                  (decl w word-type)
                  (before-phi1 3
                     (exists (w)
                       (vec-is (word non-cc-bits-of-regfile
                                     (extend t2 6))
                               (list-tail w 5))))))
               ;; Decomposition formulas:
               '())))

      ((irq) ...)
      ((dma) ...)
      ((sst) ...)
      ((wait) ...)

      ;; Mapping for RUN controls:
      ((run2) (make-run 2))
      ((run3) (make-run 3))
      ((run4) (make-run 4))
      ((run5) (make-run 5))
      ((run6) (make-run 6))
      ((run7) (make-run 7))
      (else
       (error 'control "can't happen")))))

;;; ---------------------------------------------------------------------
```

```
;;; The system invariant.
;;;
;;; The system invariant is established by the RESET operation and
;;; maintained by all other operations... .  Actually, a weaker form
;;; of the invariant is checked, but a stronger one assumed.
;;; The difference is made up by electrical effects (bistablity
;;; of postive feedback with gain) that are not captured by the
;;; switch-level model.

(define ctl-type (mk-word-type 6))
(define-type-class ctl-type ctl-class)

(define i0 '(1 1 0 1  0 1 1 0))
(define i1 '(0 0 0 1  1 0 0 0))
(define i2 '(0 0 0 0  0 0 0 1))
(define i3 '(1 1 1 0  0 1 0 0))
(define i4 '(1 0 1 1  0 1 0 0))
(define i5 '(0 1 1 1  0 1 0 0))
(define i6 '(0 0 0 0  1 1 0 1))

(define (invariant b)
  (check-type b bit-type)
  (decl msel ctl-type)
  (decl ipend bit-type)
  (conj
   (exists (ipend) (after-phi1 -1 (is nmi-lat ipend)))
   (after-phi2 -2
    (exists (msel)
      (vec-is u-adr
        (?: (vec-== msel (encodeU 0 ctl-type)) i0
         (?: (vec-== msel (encodeU 1 ctl-type)) i1
          (?: (vec-== msel (encodeU 2 ctl-type)) i2
           (?: (vec-== msel (encodeU 3 ctl-type)) i3
            (?: (vec-== msel (encodeU 4 ctl-type)) i4
             (?: (vec-== msel (encodeU 5 ctl-type)) i5
                 i6)))))))))

   (-> b
       ;; Binary values in temporary registers.
       ;; These must be guaranteed by studying electrical effects.
       (let ()
         (decl w word-type)
         (after-phi2 -1
```

```
          (conj
           (exists (w) (vec-is (word regfile (extend t1 6)) w))
           (exists (w) (vec-is (word regfile (extend t2 6)) w))
           ))))))

;;; --------------------------------------------------------------------

;;; The NMI latch predicate.
;;;
;;; This predicate establishes whether or not an interrupt is pending.

(define (pending-interrupt b)
  (check-type b bit-type)
  (after-phi1 -2 (is nmi-lat b)))


;;; --------------------------------------------------------------------
;;; End of file containing state mappings for hector.
;;; --------------------------------------------------------------------
```