The Formal Verification of Generic Interpreters

By

Phillip J. Windley
B.S. (University of Idaho) 1982
M.S. (University of California, Davis) 1988

Dissertation

Submitted in partial satisfaction of the requirements for the degree of

DOCTOR OF PHILOSOPHY

in

Computer Science

in the

Graduate Division

of the

UNIVERSITY OF CALIFORNIA
DAVIS

Approved:

Committee in Charge

1990

Abstract

Verification holds great promise for increasing the trustworthiness of computer systems. Before this promise is realized however, verification must become an engineering activity instead of a research activity. An engineering approach provides a methodology for specification and verification.

This dissertation contains results that provide a methodological approach to microprocessor verification. We present a hierarchical decomposition strategy for specifying microprocessors. The decomposition follows traditional abstraction levels used by microprocessor designers. We show how the explicit representation of these abstraction levels in the specification can lead to an order of magnitude reduction in the number of long, difficult cases in a microprocessor verification.

The dissertation also develops a theory of interpreters that can be used to model microprocessor behavior. The generic interpreter theory abstracts away the details of instruction functionality, leaving a general model of what an interpreter does. The interpreter theory has been formalized using generic theories in a verification system called HOL for use in specifying and verifying microprocessors. The use of generic theories for formalizing the model plays an important role in our methodological approach to microprocessor specification and verification.

The generic interpreter theory formally defines an interpreter and proves a correctness theorem for the generic model stating what it means, in general, for an interpreter to be correctly implemented. The generic interpreter theory provides a methodology for verifying microprocessors:

- The generic interpreter theory clearly states exactly what definitions need to be made to specify a microprocessor.
- The generic interpreter theory also says exactly which lemmas must be proven to verify that the specification is correctly implemented.

The dissertation provides a detailed example, the verification of correctness for a microprocessor called *AVM–1*. The architecture of *AVM–1* provides a RISC–like instruction set operating on a register file with 32 registers. The implementation is microcoded. The example is used to illustrate the utility of hierarchical decomposition and the generic interpreter theory.

Contents

List of Figures

List of Tables

Acknowledgments

There are many people at the University of California, Davis who have helped me with my research:

Karl Levitt, my major professor, has been a constant friend and advisor. I'm sure that people familiar with his past work will see his mark in mine. His musing as to whether a hierarchical decomposition could be formalized in a theorem proving system ultimately led to the concept of generic interpreters described in this dissertation. His guidance, help, and support are the reasons that any of this got done.

The other members of my committee, Bob Redinbo and Bob Keller, helped refine my work by asking questions about things I took for granted. In addition, their comments on the drafts of this dissertation removed many of the rough spots and kept me from making several blunders.

Gene Fisher originally suggested using ML and sequents, instead of redefining the HOL logic, to implement generic theories during one of our many discussions comparing OBJ and HOL.

Tom Schubert read many of the drafts of this dissertation and the technical reports leading up to it. I always found his comments helpful; they frequently cleared up my muddled thoughts.

I also thank Michael Russell and Pat Nelson for the times when they dropped what they were doing to help me get a system working or an account set up so that the theorem proving could go on.

I have also enjoyed the support and friendship of colleagues at locations apart from Davis:

Jerry Cohen, from the Boeing Company, provided many fresh perspectives and much useful discussion. His constant nurturing of the project kept it from derailing several times.

I spent 3 months visiting the System Verification Methods Branch at NASA's Langley Research Center in 1989. Ricky Butler and Jim Caldwell helped me redefine my dissertation topic (with much travail) and provided the initial direction for the work. Good advice is not always graciously received; Ricky and Jim gave it to me anyway and I'm grateful.

In addition to conceiving, building, and maintaining the HOL verification system, Mike Gordon of Cambridge University has always been willing to provide assistance and advice. Much of the initial work in microprocessor verification was done by him or his students and he has had a great influence on the work presented here.

I owe a great debt to Jeff Joyce of the University of British Columbia for his willingness to share both techniques and theories; his help was invaluable. In many ways, my work is an extension of his work in microprocessor verification.

HOL is not an easy system to learn. When I started, Graham Birtwhistle of the University of Calgary generously let me use a draft of his book on hardware verification. In December 1988, I spent a week in Calgary visiting his laboratory, where he took me under his wing and guided me through several difficult proofs. That week made all the difference in my HOL proficiency.

This dissertation was typeset using Donald Knuth's TEX and Leslie Lamport's LATEX. The chapter titles and HOL code were formatted using LATEX macros from the HOL manual.

Finally, I dedicate this dissertation to my wife, Lynne, and my son Bradford, for their constant care, support, encouragement, and love. There are many days I would have given up if they had not been there to help me through. To Lynne and Bradford, my love and gratitude.

The work presented in this dissertation was sponsored under Boeing Contract NAS1-18586, Task Assignment No. 3, with NASA-Langley Research Center.

to Lynne and Bradford

Chapter 1

Introduction

If we do not succeed in solving a mathematical problem,
the reason frequently consists in our failure to recognize
the more general standpoint from which the problem before us
appears only as a single link in a chain of related problems.
After finding this standpoint, not only is this problem frequently
more accessible to our investigation, but at the same time,
we come into possession of a method
which is applicable to related problems.
—*David Hilbert*—

Computers are being used with increasing frequency in areas where the correct implementation of the computer hardware is critical. These include:

- Safety–critical applications where the computer is directly involved in the control of systems that maintain human life. A flight control system on an aircraft or the control system in a nuclear power plant are examples of this type of application.
- Security–critical applications where the computer is used to process information that is economically or politically sensitive. Almost any computer used in government or industry falls into this category to one degree or another.
- Mass produced consumer goods where the computer is an integral part of the product and a mistake in the design or implementation could result in product recalls costing enormous amounts of money.

In these and other applications it is vital that the computer system be correct.

There are two complimentary approaches to computer correctness: fault tolerance and fault exclusion. The former is most useful in handling dynamic faults occurring during system operation due to component failure or other unexpected events. The latter is a static process intended to remove errors in design and implementation before the computer system is in service.

Testing is an example of a fault exclusion technique. Testing can be divided into two distinct kinds. Implementational testing, which is used to verify that a physical device is implemented correctly, and functional testing which is used to verify that a design functions as the designer intended. Because it is impossible to exhaustively testing a computer system, formal verification is an attractive alternative to functional testing.

Formal verification requires at least two descriptions of a system: one of its implementation and one of its specification. Correctness is shown by demonstrating through mathematical proof that the former implies the latter. Since verification entails reasoning about formal logic, producing specifications with the formality needed for verification is difficult. Several steps must be taken to make verification available and acceptable to industry:

- Methodologies that provide a step–by–step approach to system verification must be produced.
- Exemplary verified systems must be provided.

Our goal is to make microprocessor verification tractable. To that end, this dissertation contains a methodology for microprocessor verification that results in a step–by–step approach. We also give an example showing how the methodology can be applied to the verification of a realistic microprocessor.

1.1 Abstraction.

Abstraction is the suppression of irrelevant detail or information. Uses of abstraction abound in everyday life. For example, a map is an abstraction of the area it represents. The irrelevant details of the area being mapped are suppressed so that the map's users are not confused or hampered by unnecessary facts.

Abstraction is a key concept in both mathematics and computer science. Using abstraction, we make complex models more tractable, avoid repeating work, and develop methods for solving general problems. For example, procedural and data abstraction are used frequently in software engineering to ease the burden of programming by suppressing detail, providing reusable structures, and giving general algorithms for computational problems.

Naturally, abstraction has a place in formal verification as well. Melham [Mel88] provides an important discussion of structural, behavioral, data, and temporal abstraction in the verification of computer hardware. Structural abstraction suppresses detail about the internal structure of a device. A behavioral abstraction is a partial specification of a device; the specification may leave out timing details or other functionality not considered important. Data abstraction suppresses implementation details of a data type so that only its functionality is visible. Temporal abstraction relates different views of time in the specification of a device.

Melham discusses the use of abstraction in hardware verification in general; we will concentrate on the application of abstraction to modeling and verifying microprocessors. We ask the questions:

1. Are there particular forms of behavioral and structural abstraction that are more efficacious in the verification of microprocessors than others?

2. Can we formalize a general model that incorporates the behavioral, data, and temporal abstractions used in microprocessor verification so that they can be easily reused?

As we will see in the chapters that follow, we believe that the answer to both of these question is *yes* and we will describe a hierarchical decomposition strategy and a generic interpreter model that make the verification of large microprocessors practical.

1.2 Main Ideas.

This section introduces the main ideas in this dissertation. These concepts will be discussed in detail in later chapters.

1.2.1 Hierarchical Decomposition.

As we mentioned, verification requires at least two formal descriptions of the computer system: one behavioral, **B**, and one structural, **S**. Verification consists of showing through formal proof techniques that

$$\mathbf{S} \Rightarrow \mathbf{B}$$

One need not be limited, of course, to one level of abstraction. Supposing that $\mathbf{B}_1$ through $\mathbf{B}_n$ represent increasingly abstract specifications of the system's behavior, one could verify its correctness by proving

$$\mathbf{S} \Rightarrow \mathbf{B}_1 \Rightarrow \ldots \Rightarrow \mathbf{B}_n$$

Figure 1.1 shows how this principle can be applied to the specification of a microprogrammed microprocessor. At the bottom of the hierarchy is the usual structural specification of the electronic block model. This specification describes the computer's implementation; that is, the connections among its various components. At the top is the behavioral specification corresponding to the programmer's model of the microprocessor. In between these are two additional abstraction levels: one for the microcode interpreter and one specifying the phase (or subcycle) behavior.

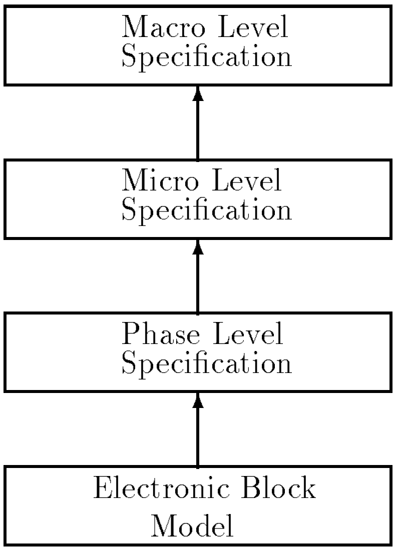

Figure 1.1: A microprocessor specification can be decomposed hierarchically.

Hierarchical decomposition plays an important role in the methodology for verifying microprocessors presented in this dissertation. The use of a hierarchical decomposition can lead to significant reductions in the amount of effort used to structure and complete a correctness proof.

1.2.2 Generic Interpreters.

With one exception, each of the levels in the specification hierarchy shown in Figure 1.1 has the same structure. The bottom level specification is a structural description; but, the other specifications all share a common structure. Each of the abstract behavioral descriptions can be specified using an *interpreter model.*

Perhaps the most distinguishing feature of an interpreter is that it has a *flat* control structure. One of n instructions is chosen based on the current state. The chosen instruction operates on the state and the cycle begins anew. There are a large number interesting computer systems that have a flat control structure: microprocessors, operating systems, language interpreters, and editors are a few.

Since each of the behavioral descriptions in the specification hierarchy are similar, we would prefer to develop a general model of an interpreter and use this model in our specification rather than treating each level in the hierarchy separately. We can ask several interesting questions about the interpreter model:

- How can one interpreter be used to implement another interpreter?
- Can we formalize the data and temporal abstractions between these levels?

- What, if anything, can we say about the correctness of an interpreter's implementation?
- Can we formalize the model in a verification environment so that it can be easily reused in verifying microprocessors?

The chapters that follow provide the answers to these questions.

1.2.3 Composition of Verified Modules.

Verified systems can be constructed using verified components. The composition of verified components that share state is a topic that has not received much attention. Indeed, most of the microprocessor verifications done in the past have assumed that the CPU was the sole user of memory—even when the CPU's designer claimed that input/output was memory mapped.

In this dissertation we take a first step toward the specification of computer components that share state with other devices. We defined the concept of shared state and propose mechanisms for specifying the reading of and writing to shared state. The assumptions on the final proof of correctness clearly state how conflicts regarding shared state are resolved.

1.3 Contributions.

The work described in this dissertation makes the following contributions:

1. The hierarchical decomposition strategy provides a firewall for the structural complexity of the electronic block model specification well below the large case explosion that occurs at the top–level. This firewall results in a substantial savings in effort over past specification methods since the large number of cases in the upper levels can now be handled in a regular, largely automatic manner.
2. Generic proofs formalize a methodology for verifying microprocessors. The generic interpreter proof clearly states what definitions need to be made and what lemmas need to proven about these definitions in order to verify a microprocessor. This is in sharp contrast to previous microprocessor verifications where the specification and verification proceeded on an *ad hoc* basis.
3. Our technique for specifying components with shared state decreases the semantic gap between what the designer intends and what the specification says. Our CPU specification recognizes that other components may change the contents of memory and other shared registers. The proofs that result from these specifications have very satisfying interpretations with respect to the assumptions that indicate how conflicts over shared state are resolved.

In addition to the major benefits listed above, there are a number of other benefits that result from our work:

- The generic theory can be instantiated, resulting in the reuse of large pieces of the generic proof.
- Temporal and data abstraction are handled completely within the generic theory freeing the user from proving theorems about these abstractions.
- The generic proofs show exactly what has been proven. There is no superfluous detail cluttering up the definitions and theorems.
- Our interpreter model recognizes the environment and treats it separately from the state.

1.4 Formal Proofs.

The dissertation deals with the *formal* verification of generic interpreters. What exactly is implied by the word *formal*?

The word *formal* is used to describe many things in mathematics: formal systems, formal logics, formal proofs, and so on. A formal object is one where rigor is maintained through a methodical treatment.

In a formal system, great emphasis is placed on *syntax* (i.e. the form). In a formal logic, for example, the syntax of the logic is set forth unambiguously and inference rules for manipulating the syntax are clearly defined. A formal proof in this logic takes place syntactically through the application of inference rules in a sequential manner. The use of inference rules to transform terms syntactically helps keep the prover's semantic biases from creeping into the proof.

The behavioral and structural models for computer systems can be very large. Proofs of correctness for microprocessors have, in some cases, been done using paper and pencil. Usually, however, the proofs are so large that some form of mechanical proof support is necessary to maintain the required rigor. In one case, the formal verification of a microprocessor found errors in a completed informal proof [Coh88b]. Most proofs involving mechanical theorem provers are done using some sort of formal logic since formalization is a prerequisite to mechanization.

In a strictly proof theoretic sense, the proofs in this dissertation are not formal. A formal proof is like a number—an abstract object that can never be represented in the physical word. Numerals are not numbers, but merely represent them. The proofs in this dissertation are not true formal proofs, but merely represent them. However, the term *formal verification* is used ubiquitously in the verification community to mean a proof in a formal logic, usually with the help of some sort of mechanized theorem prover. We use the term *formal* in this sense.

1.5 Notation and Conventions.

Our notation will be that of standard logic with a few extensions:

- Terms in the logic will be written in `typewriter` font.
- Conjunction, disjunction, negation, implication, universal quantification, existential quantification, and lambda abstraction use the usual symbols: $\wedge$, $\vee$, $\neg$, $\Longrightarrow$, $\forall$, $\exists$, and λ respectively.
- We use a conditional operator that is written `a` $\rightarrow$ `b` $\mid$ `c`, meaning "if `a`, then `b`, else `c`."
- Definitions will be denoted with a prepended $\vdash_{def}$.
- Terms that have been formally proven in the logic will be prepended with $\vdash$.

Other notations and logical expressions will be explained as they are used.

1.6 Chapter Summaries.

The dissertation begins, in Chapter 2, with a discussion of related work. The idea of using abstract representations of theories is not new to our research. There are several specification and theorem proving systems that support generic modules and several examples of their use in the literature. In addition, a number of microprocessors have been specified and verified in formal systems. Most of these verifications used an implicit interpreter model for the behavioral specification.

Our research is, we believe, the first where the interpreter model has been formalized. Chapter 3 discusses our model of interpreters, gives a mathematical definition of an interpreter, and defines what it means to verify one interpreter in terms of another. The chapter also discusses how hierarchical decomposition can be used in the specification of microprocessors and discusses the composition of verified components that share state.

This theme is extended in Chapter 4 where we show how the mathematical definition can be formalized in the HOL verification system. We present two different models: a synchronous interpreter model and an asynchronous model.

Chapter 5 contains an example of the use of the generic interpreter theory in specifying and verifying a microprocessor, *AVM–1*. *AVM–1* is designed to serve as a testbed for the concepts in this dissertation. The architecture and organization of *AVM–1* are described, the formal specification is presented, and the verification is discussed.

Appendix A provides a description of the ML package developed for using generic theories in HOL.

Appendix B presents the technical details of the *AVM–1* proof. The theory hierarchies are discussed and the run times for the proof scripts of the various theories constituting the verification of *AVM–1* are presented.

Chapter 2

Previous Work

This chapter is divided into four sections. The first section discusses research in the verification of sequential machines and its relation to our work. The second section discusses previous microprocessor verifications where a model similar to the one formalized in this dissertation was employed. The third section describes related work in generic theories. The last part of this section describes how hardware behavior and structure are specified in a formal logic.

2.1 Sequential Circuit Verification.

Reasoning about and verifying sequential circuits is a topic that has generated much research interest due to the inherent difficulty involved in reasoning about state. The standard engineering formalism taught in undergraduate switching theory classes is useful for reasoning about small state machines; but when the number of states increases the model suffers from exponential case explosion. In this section we will discuss some of the approaches to the problem.

Sequential machines have been studied for decades. The work in this dissertation is similar in spirit to Gordon's work [Gor80] in the denotational semantics of sequential machines and Plotkin's state transition systems [Plo81]. Our goal is, however, not to simply describe sequential machines, but to verify them.

Several researchers have developed special purpose languages for describing and reasoning about sequential machines. Browne and Clark [BC87] have developed a high–level language, called *SML*, for describing finite state machines. *SML* is based on a temporal logic semantics. The state transition table generated from an *SML* program can be fed to a temporal logic verifier that allows some properties of the state machine to be verified.

Bronstein and Talcott [BT89] have developed a string-functional semantics of synchronous sequential logic and have applied it to the verification of pipe–lines and systolic arrays. The theory is based on finite rather than infinite arrays and thus cannot reason about asynchronous circuits. Because of the finite property of the underlying semantics, the theory can be developed and used in a first–order system such as the Boyer–Moore theorem prover [BM79].

Loewenstein has developed a theory of state machines in HOL [Loe89]. The distinguishing feature of Loewenstein's work is that the theory is developed in the same theorem proving environment that he uses to reason about his state machines. The theory contains theorems that define both deterministic and non-deterministic state machines and derives lemmas that state what it means for one state machine to implement another and what it means for two state machine to be equivalent. Loewenstein's theory is similar to the model that we will define in Chapter 3. The primary difference is that Loewenstein's model does not formalize temporal and data abstraction between a state machine and its implementation.

SDVS (State Delta Verification System) was originally developed and described by Crocker in [Cro77]. A state–delta is a temporal logic formula that describes the changes in the global state of a machine over a time interval. SDVS is currently under development at the Computer Science Laboratory of the Aerospace Corporation [MCL84, Mar87]. The original goal of SDVS was to provide a usable system for proving the correctness of microcode expressed in the ISPS register transfer language. SDVS was also used to reverify Hunt's FM8501 (see Section 2.2.2).

The next section discusses the work in state machine verification that most closely resembles our own work.

2.2 Microprocessor Verification.

There have been numerous efforts to verify microprocessors. Many of these have used the same implicit behavioral model. We will first describe this implicit model and then describe the microprocessor verifications that use it.

In general, the model uses a state transition system to describe the microprocessor. The microprocessor specification has four important parts:

1. A representation of the state, **S**. This representation varies depending on the verification system being used.
2. A set of state transition functions, **J**, denoting the behavior of the individual instructions of the microprocessor. Each of these functions takes the state defined in step (1) as an argument and returns the state updated in some meaningful way.
3. A selection function, **N**, that selects a function from the set **J** according to the current state.
4. A predicate, **I**, relating the state at time $t+1$ to the state at time t by means of **J** and **N**.

In some cases, the individual state transition functions, **J**, and the selection function, **N**, are combined to form one large state transition function. Also, a functional specification would use a function for part (4) instead of a predicate. The specifications, however, are largely the same.

After the microprocessor has been specified, we can verify that a machine description, $\mathbf{M}$, implements it by showing

$$\forall s \in \mathbf{S}.\ \mathbf{M}(s) \Rightarrow \mathbf{I}(s).$$

That is $\mathbf{I}$ has the same effect on the state, s, that $\mathbf{M}$ does. This theorem is typically shown by case analysis on the instructions in $\mathbf{J}$ by establishing the following lemma:

$$\forall j \in \mathbf{J}.\ \mathbf{M}(s) \Rightarrow (\forall t\!:\mathrm{time}.\ \mathcal{C}(j, s, t) \Rightarrow s(t + n_j) = j(s(t)))$$

where $\mathcal{C}$ is a predicate expressing the conditions for instruction j's selection, $s(t)$ is the state at time t, and n_j is the number of cycles that it takes to execute j. This lemma says that if an instruction j is selected, then applying j to the current state yields the state that results by letting the implementing interpreter $\mathbf{M}$ run for n_j cycles. We call this lemma the instruction correctness lemma.

The remaining parts of this section describe microprocessor verifications where some variation of this general model was used.

2.2.1 Tamarack

Tamarack is a small microcoded microprocessor that has been verified by Jeffrey Joyce at the University of Cambridge [Joy89a, Joy88]. Joyce has verified Tamarack to the transistor level using HOL and has fabricated an 8–bit version of the design in CMOS. In addition to verifying the microprocessor, Joyce has also verified a compiler for Tamarack [Joy89b].

Tamarack is a 16–bit computer with a 13–bit address space. The computer has 8 instructions: halt, jump, jump if zero, add, subtract, load, store, and skip (or no operation). The architecture has an accumulator and a program counter visible to the assembly language programmer in addition to the memory. The computer is implemented in microcode and has a single bus connecting each of the blocks in the electronic block model. The microstore is 32 microwords long.

Tamarack is based on a computer designed and verified using LCF-LSM by Mike Gordon [Gor83]. Daniel Weise verified Gordon's design using a Lisp–based system called Silica Pithecus [Wei86] and Harry Barrow verified it using a system called VERIFY [Bar84], making this the most widely verified microcomputer design.

The specification and verification of Tamarack corresponds closely to the general model developed at the beginning of this section. The macro–level specification denotes what each instruction does and ties the descriptions of each instruction together with a predicate stating the relation between the state at time t and time $t + 1$.

The verification of Tamarack is enlightening since it has been done many times with many different verification systems and using many levels of abstraction. Tamarack is, however, small and research is needed to discover methods for scaling the Tamarack experience to larger microprocessors, including those with larger instruction sets and support for operating systems.

2.2.2 FM8501.

FM8501 is a microprocessor designed and verified by Warren Hunt using the Boyer–Moore theorem prover [Hun87]. The architecture has a register file containing eight, 16–bit registers, a 64K–byte memory space, 26 instructions, and four memory addressing modes. FM8501 models memory as an asynchronous process. The implementation is microcoded and has a microstore of 16 microwords.

The specification of FM8501 consists of two recursive functions: one for the behavioral specification and one for the implementation. The functions recurse at each clock cycle, computing a new state. Time and the asynchronous inputs to the CPU are modeled by an oracle. The oracle is represented by a list; it is this list that the specifications recurse on. Time is represented by the current position of the recursive specification in the list. Each member of the list gives whatever asynchronous inputs may exist at that time. The proof shows the equivalence of the two recursive functions using an abstract (uninterpreted) oracle function.

Crocker *et al.* reverified FM8501 using a specification written in ISPS in the SDVS verification system [CCLO88]. The reverification is significant because the work used no part of Hunt's work directly and thus represents an independent verification of the design using a different verification system.

On the surface, the verification of FM8501 appears quite different than the verification of Tamarack, but in fact, they are very similar. The methods of specification for the top–level can be seen as an instance of the general model presented at the beginning of this section. The verification, even though done on a functional specification in a first–order system, uses the a form of the instruction correctness lemma to show that the electronic block model implements the top-level specification.

2.2.3 VIPER.

VIPER was designed by Britain's Royal Signals and Radar Establishment (RSRE) at Malvern to provide a formally verified microprocessor for use in safety critical applications. VIPER's designer's chose not to include a stack and interrupts—anticipating that they might lead to difficulties in the verification. The machine was designed to halt on errors and raise an external exception. The fabrication was carried out by two separate manufacturers and is commercially available.

VIPER is the first microprocessor intended for commercial use where formal verification was used. As we will see, the verification was not completed. While VIPER is significantly simpler than today's general purpose microprocessors, its verification provides a benchmark on the state–of–the–art in microprocessor verification.

VIPER has a 20-bit program counter, a 32-bit general purpose accumulator, and two 32-bit index registers. VIPER has a single instruction format that allows the user to select a source register, one of four memory addressing modes, one of eight destinations, whether or not to compare, and one of sixteen ALU functions. In addition to the fields just mentioned, each instruction contains a 20–bit address. The VIPER design is described in detail in [Cul88]. The implementation is hardwired instead of being microcoded.

The combination of fields in the instruction format (excluding source and destination selections) yields 128 different instruction cases. Recent research on the VIPER design [Aro90] has characterized the VIPER instruction set using only 20 instructions. As we will see, this is an important distinction that bears on the difficulty of verifying VIPER.

The specification of VIPER is hierarchical. The top–level specification of VIPER is similar to that of [Joy89a]. The next level of the specification is called the major–state machine and is a description of VIPER's major states. The next level in the specification is the electronic block model. The top two levels were specified first in LCF-LSM and later in HOL. The electronic block model was specified in HOL. Below the electronic block model the circuit was described using a hardware description language called ELLA and verified by "intelligent exhaustive simulation" [Pyg85].

An paper–and–pencil proof of correctness between the top–level of VIPER and the major–state machine was done by RSRE. Because of the complexity of the lower–level (electronic block model to major state machine) proof, RSRE did not attempt a hand proof of this level. RSRE contracted with Avra Cohn at Cambridge University to formalize the top–level proof and perform the lower–level proof. Cohn describes her formal verification of the major–state machine with respect to the top–level specification in [Coh88b].

Cohn decided to forego the proof of the top–level correspondence in trying to verify the electronic block model since that the major–state level specification and the electronic block model yielded dissimilar structures under cases analysis. Rather, she attempted to show a direct correspondence between the top–level and the electronic block model. The proof is described in detail in [Coh88a]. Cohn's proof of this level remains incomplete because of the large case explosion that occurred and the size of the proofs in each of the cases. This is not to say that the proof could not be completed; but only at large expense.

It seems clear from Cohn's experience with VIPER that abstraction is critical in dealing with the large case explosion that occurs in these kinds of proofs. The major–state machine did provide a level of abstraction in–between the top–level and the electronic block model, but it appears to be the wrong one. In addition, Cohn had almost no access to VIPER's designers and thus had little or no help in deciphering and understanding the informal specification of the electronic block model.

2.2.4 SECD.

Brian Graham *et al.* at the University of Calgary have undertaken the implementation and verification of the SECD machine [GB89]. The SECD machine is an abstract Lisp machine invented by Landin to reduce lambda expressions [Lan64]. The variant of SECD implemented by Graham is described in [Hen80]. Graham's work is part of a larger effort at the University of Calgary to verify a complete system including a LispKit compiler as well as the SECD chip.

The architecture has four registers, called **S**, **E**, **C**, and **D**. The **S** register holds a stack pointer, the **E** register holds a pointer to the environment, the **C** register functions as a program counter, and **D** points to a stack used to dump the state of the machine. There are approximately 20 instructions and the implementation is microcoded.

The remarkable thing about the SECD proof is that even though the architecture is specialized, the specifications and proofs are done in a manner very similar to the proofs of the more conventional architectures described in the last three sections. The behavioral model corresponds to the general model described at the beginning of this section. The top–level specification is based on state–transitions and the description of the electronic block model is a predicate–based circuit description similar to both [Joy89a] and [Coh88a]. The garbage collection mechanism is implemented in hardware and the proof was done without taking it into account. Work is in progress on a second proof that verifies the garbage collection hardware and a second implementation.

2.2.5 Comparison.

Table 2.1 summarizes the designs of the four microprocessors presented in this section. The table, like all such tabulations, cannot hope to capture all of the important characteristics of the microprocessors, but the data presented does provide some basis for judging relative complexities.

2.3 Generic Theories.

Generic theories provide structures to support theorem reuse. Generic theories are similar in spirit to generic modules in programming languages such as Ada [Ada83].

Table 2.1: Comparison of verified microprocessors.

	Tamarack	FM8501	Viper	SECD
User Registers	2	8	4	4
Instructions	8	26	20	21
Microcoded	yes	yes	no	yes
Microstore size	32 words	16 words	N/A	512 words
Interrupts	yes	no	no	no
Memory Model	async	async	sync	sync
Word Width	16-bit	16-bit	32-bit	32-bit
Memory Size	8K	64K	1M	16K

Even accounting for the obvious differences between Ada as a programming language and our use of generics in a verification environment, however, we shall see that the notion of a generic theory is stronger than that of a generic module in Ada.

An generic theory consists of three parts:

1. An *abstract representation* of the uninterpreted constants and types in the theory.
2. A list of *theory obligations* defining relationships between members of the abstract representation.
3. A collection of *abstract theorems* about the representation.

The *abstract representation* contains a set of abstract operations and a set of abstract objects. An abstract object does not necessarily need to be specifically declared, but can be declared through use. The semantics of the abstract representation are unspecified; that is, we don't know (inside the theory) what the objects and operations mean.

The *theory obligations* are a set of predicates. Inside the theory, the obligations represent axiomatic knowledge about the abstract representation. Using the obligations as axioms allows us to prove theorems of interest about the abstract objects and operations. Outside the theory, the obligations represent the criteria that a concrete representation must meet if it is to be used to instantiate the abstract theory.

The theory obligations represent the largest difference between generic theories and the generics of Ada. If we view the generic portion of our theory as the *interface*, the abstract representation can be thought of as the *syntax* of the interface. The abstract representation corresponds to the declaration of the generic parameters in an Ada module. The theory obligations denote the *semantics* of the interface and Ada provides no corresponding structure.

The abstract theorems are a body of facts concerning the abstract objects. Usually, the theorems are based on the theory obligations and can stand alone only after the theory obligations have been met.

Our goal is to instantiate the generic theory with a concrete representation. To effect the instantiation, the concrete representation must meet the syntactic requirements of the abstract representation as well as the semantic requirements of the theory obligations. If the syntactic and semantic requirements are met, then the instantiation provides a collection of concrete theorems about the new representation.

Several specification and verification systems support generic theories. Some, such as OBJ and EHDM, offer explicit support. HOL, the verification environment used in the research reported here, does not explicitly support generic theories; however, HOL's metalanguage, ML, combined with higher–order logic, provides a framework sufficient for implementing generic theories.

2.3.1 OBJ.

OBJ is a specification and programming language developed by Joseph Goguen *et al.* that has most recently been described in [GW88]. OBJ is widely known and the semantics of its theories and views match our use of generic theories much more closely than do Ada generics.

OBJ is based on a many–sorted (or typed) algebraic semantics and supports parameterized specification and programming [Gog84]. OBJ has three kinds of entities:

1. **Objects**, which are concrete modules that encapsulate executable code,
2. **Theories**, which are parameterized modules that correspond to generic theories as used in this dissertation, and
3. **Views**, which bind objects and theories to parameters in another theory.

Objects are said to contain executable code because the expressions in an object module give the initial algebraic semantics of the sorts and operations being defined. The fact that their semantics is initial implies that they describe just one model (up to isomorphism). Theories, on the other hand, are said to have a "loose" semantics since they define a variety of models. A loose semantics describes a class of objects; any member of that class will satisfy the theory.

A view is *not* an instantiation. Instantiation is done using a special command, **make**, after the view has been established. A view can be seen as a mapping of the operators and objects from one module onto a theory, as well as a declaration of intent that the module meets the obligations set forth in the equations of the theory module. OBJ does not require that the user prove that the obligations are met—a simple declaration is sufficient. Of course, if the view is not proper, then the OBJ program will not operate as intended.

2.3.2 EHDM.

EHDM is a specification and verification system that is being developed by SRI International [EHD88]. The language of EHDM is based on first–order predicate logic, but includes some elements of higher–order logic as well. For example, variables can range over functions, functions can return other functions, and functions can appear in quantifications. Parameterized modules are an important part of the EHDM language where they are used to organize specifications. Modules can be parameterized with types, constants, and functions. The module parameters can have constraints placed on them that must be met before the module can be instantiated.

In EHDM, a parameterized module is called a *generic module* and an instantiation is called a *module instance*. EHDM module declarations give the uninterpreted types, constants, and functions over which the module is parameterized. This declaration is analogous to our abstract representation.

The module body contains (among other things) an **ASSUMING** clause that gives the properties of the module parameters. The formulae in the **ASSUMING** clause are analogous to our theory obligations.

The module can also contain declarations of concrete types, constants, and functions that define the theory associated with the module and proofs of theorems about the abstract operations in the theory. These proofs may rely on the formulae in the **ASSUMING** clause.

2.3.3 HOL.

HOL is a general theorem proving system developed at the University of Cambridge [Gor88, CGM87] that is based on Church's theory of simple types, or higher-order logic [Chu40]. Church developed higher-order logic as a foundation for mathematics, but it can be used for describing and reasoning about computational systems of all kinds. Higher–order logic is similar to the more familiar predicate logic, but allows quantification over predicates and functions, not just variables, allowing more general systems to be described.

HOL grew out of Robin Milner's LCF theorem prover [GMW79] and is similar to other LCF progeny such as NUPRL [Con86]. Because HOL is the theorem proving environment used in the body of this work, we will describe it in more detail.

HOL's proof style can be tailored to the individual user, but most users find it convenient to work in a goal–directed fashion. HOL is a tactic based theorem prover. A tactic breaks a goal into one or more subgoals and provides a justification for the goal reduction in the form of an inference rule. Tactics perform tasks such as induction, rewriting, and case analysis. At the same time, HOL allows forward inference and many proofs are a combination of both forward and backward proof styles.

Any theorem proving strategy a user employs in connection with HOL is checked for soundness, eliminating the possibility of incorrect proofs.

HOL provides a metalanguage, ML, for programming and extending the theorem prover. Using the metalanguage, tactics can be put together to form more powerful tactics, new tactics can be written, and theorems can be combined into new theories for later use. The metalanguage makes the HOL verification system extremely flexible.

In HOL, all proofs, even tactic–based proofs, are eventually reduced to the application of inference rules. Most non-trivial proofs require large numbers of inferences. Proofs of large devices such as microprocessors can take many millions of inference steps. In a proof containing millions of steps, what kind of confidence do we have that the proof is correct? One of the most important features of HOL is that it is *secure*, meaning that new theorems can only be created in a controlled manner. HOL is based on 5 primitive axioms and 8 primitive inference rules. All of high–level inference rules and tactics do their work through some combination of the primitive inference rules. Because the entire proof can be reduced to one using only 8 primitive inference rules and 5 primitive axioms, an independent proof checking program could check the proof syntactically.

2.3.3.1 The Language.

The object language of HOL is described in this section. We will discuss HOL's terms and types.

Terms. All HOL expressions are made up of terms. There are four kinds of terms in HOL: variables, constants, function applications, and abstractions (lambda expressions). Variables and constants are denoted by any sequence of letters, digits, underlines and primes starting with a letter. Constants are distinguished in the logic; any identifier that is not a distinguished constant is taken to be a variable. Constants and variables can have any finite arity, not just 0, and thus can represent functions as well.

Function application is denoted by juxtaposition, resulting in a prefix syntax. Thus a term of the form `"t1 t2"` is an application of the operator `t1` to the operand `t2`. Its value is the result of applying `t1` to `t2`.

An abstraction denotes a function and has the form "λ `x. t`". An abstraction "λ `x. t`" has two parts: the bound variable `x` and the body of the abstraction `t`. It represents a function, `f`, such that `"f(x) = t"`. For example, "λ `y. 2*y`" denotes a function on numbers which doubles its argument.

Constants can belong to two special syntactic classes. Constants of arity 2 can be declared to be infix. Infix operators are written `"rand1 op rand2"` instead of in the usual prefix form: `"op rand1 rand2"`. Table 2.2 shows several of HOL's built–in

Operator	*Application*	*Meaning*
=	`t1 = t2`	`t1` equals `t2`
,	`t1,t2`	the pair `t1` and `t2`
$\wedge$	`t1` $\wedge$ `t2`	`t1` and `t2`
$\vee$	`t1` $\vee$ `t2`	`t1` or `t2`
$\Longrightarrow$	`t1` $\Longrightarrow$ `t2`	`t1` implies `t2`

Table 2.2: HOL Infix Operators

Binder	*Application*	*Meaning*
$\forall$	$\forall$ `x. t`	for all `x`, `t`
$\exists$	$\exists$ `x. t`	there exists an `x` such that `t`
ε	ε `x. t`	choose an `x` such that `t` is true

Table 2.3: HOL Binders

infix operators.

Constants can also belong another special class called binders. A familiar example of a binder is $\forall$. If `c` is a binder, then the term `"c x.t"` (where `x` is a variable) is written as shorthand for the term `"c(`λ `x. t)"`. Table 2.3 shows several of HOL's built–in binders.

In addition to the infix constants and binders, HOL has a conditional statement that is written `a` $\rightarrow$ `b` $\mid$ `c`, meaning "if `a`, then `b`, else `c`."

Types. HOL is strongly typed to avoid Russell's paradox and others like it. Russell's paradox occurs in a high–order logic when one can define a predicate that leads to a contradiction. Specifically, suppose that we define `P` as `P(x) =` $\neg$`x(x)` where $\neg$ denotes negation. `P` is true when its argument applied to itself is false. Applying `P` to itself leads to a contradiction since `P(P) =` $\neg$`P(P)` (i.e. $true = false$). This kind of paradox can be prevented by typing since, in a typed system, the type of `P` would never allow it to be applied to itself.

Every term in HOL is typed according to the following recursive rules:

- Each constant or variable has a fixed type.
- If `x` has type α and `t` has type β, the abstraction λ `x. t` has the type $(\alpha \rightarrow \beta)$.
- If `t` has the type $(\alpha \rightarrow \beta)$ and `u` has the type α, the application `t u` has the type β.

Types in HOL are built from type variables and type operators. Type variables are denoted by a sequence of asterisks (`*`) followed by a (possibly empty) sequence of letters and digits. Thus `*`, `***`, and `*ab2` are all valid type variables. All type variables are universally quantified implicitly, yielding type polymorphic expressions.

Operator	*Arity*	*Meaning*
`bool`	0	booleans
`ind`	0	individuals
`num`	0	natural numbers
`(*)list`	1	lists of type `*`
`(*,**)prod`	2	products of `*` and `**`
`(*,**)sum`	2	coproducts of `*` and `**`
`(*,**)fun`	2	functions from `*` to `**`

Table 2.4: HOL Type Operators

Type operators construct new types from existing types. Each type operator has a name (denoted by a sequence of letters and digits beginning with a letter) and an arity. If $\sigma_1, \ldots, \sigma_n$ are types and `op` is a type operator of arity n, the $(\sigma_1, \ldots, \sigma_n)$`op` is a type. Note that type operators are postfix while normal function application is prefix or infix. A type operator of arity 0 is a type constant.

HOL has several built–in types which are listed in Table 2.4. The type operators `bool`, `ind`, and `fun` are primitive. HOL has a special syntax that allows `(*,**)prod` to be written as `(* # **)`, `(*,**)sum` to be written as `(* + **)`, and `(*,**)fun` to be written as `(* -> **)`.

2.3.3.2 The Proof System.

HOL is not an automated theorem prover but is more than simply a proof checker, falling somewhere between these two extremes. HOL has several features that contribute to its use as a verification environment:

1. Several built–in theories, including booleans, individuals, numbers, products, sums, lists, and trees. These theories contain the five axioms that form the basis of higher-order logic as well as a large number of theorems that follow from them.

2. Rules of inference for higher-order logic. These rules contain not only the eight basic rules of inference from higher–order logic, but also a large body of *derived* inference rules that allow proofs to proceed using larger steps. The HOL system has rules that implement the standard introduction and elimination rules for Predicate Calculus as well as specialized rules for rewriting terms.

3. A collection of tactics. Examples of tactics include `REWRITE_TAC` which rewrites a goal according to some previously proven theorem or definition, `GEN_TAC` which removes unnecessary universally quantified variables from the front of terms, and `EQ_TAC` which says that to show two things are equivalent, we should show that they imply each other.

4. A proof management system that keeps track of the state of an interactive proof session.

5. A metalanguage, ML, for programming and extending the theorem prover. Using the metalanguage, tactics can be put together to form more powerful tactics, new tactics can be written, and theorems can be aggregated to form new theories for later use. The metalanguage makes the verification system extremely flexible.

2.3.3.3 Generic Theories in HOL.

HOL provides a non-parameterized module structure called a *theory*. A theory is a set of types, definitions, constants, axioms and parent theories. Higher-order logic is extended by defining new theories. To use a theory, one declares it a parent of the current draft theory; all of the components of the parent and its ancestors are then available for use in the child theory.

HOL does not explicitly support parameterized, or generic, theories and thus might seem a poor vehicle for the research presented in this dissertation. However, HOL's other features, in particular its flexible proof style and programmability, make it a desirable system in which to work. We choose to use HOL and implement generic theories. The fact that generic theories can be defined at the user level in HOL without explicit support for them in the system is a testament to the flexibility of HOL.

Three things are required to implement generic theories in HOL:

1. A way of representing abstract objects and operations.

2. A method for declaring theory obligations and using these obligations in proofs.

3. Functions for instantiating an abstract theory with concrete objects.

Our implementation of abstract theories in HOL is described in Appendix A.

Jeffrey Joyce of Cambridge University presents a method of representing abstract objects and operations using HOL in [Joy89a] where he uses them to provide an abstract view of n–bit words. We use Joyce's methods in our implementation of abstract representations. We have extended Joyce's work by developing full–fledged generic theories including theory obligations and methods of instantiating a generic theory.

HOL has a type polymorphic logic that supports top–level universal quantification over type variables; we use type variables to denote abstract objects. Abstract operations on these objects make use of HOL's ability to quantify over functions. Abstract operations are synthesized by creating variables that hold n–tuples; each entry in the tuple represents one of the abstract operations. The abstract operations *select* the appropriate field from the tuple.

We have implemented an ML function that defines a new abstract representation. The following example shows how our implementation can be used to define an abstract representation for groups. (Recall that type variables are denoted in HOL by prepending an asterisk.)

```
let G = new_abstract_representation
    [
     (`op`,":(*group × *group) → *group")
    ;
     (`e`,":*group")
    ;
     (`inv`,":*group → *group")
    ];;
```

The abstract representation is given as a list of pairs where the first member of the pair is a string giving the name of the abstract operation and the second is an HOL term giving its type. [1]

The theory obligations are declared by giving a list of terms; each term denotes a predicate that will be used as an axiom inside the generic theory. Note that the predicates are not true axioms since we want them to exist only *inside* the generic theory; they will be satisfied and discharged by the instantiation when the theory is used. Continuing the group theory example from above, we present the theory obligations:

```
new_theory_obligations
    ["∀ g:*group.  (op rep (g,e rep) = g)";
     "∀ g:*group.  (op rep (e rep,g) = g)";
     "∀ g:*group.  (op rep (g,inv rep g) = (e rep))";
     "∀ g:*group.  (op rep (inv rep g,g) = (e rep))";
     "∀ g g' g'':*group.
          op rep (g, op rep (g',g'')) = op rep (op rep (g,g'), g'')"
    ];;
```

The abstract operations `op`, `e`, and `inv` are all used as selectors on a variable called `rep`. Recall that function application in HOL is denoted by juxtaposition. The variable `rep` is a 3–tuple in this case; when the theory is instantiated `rep` will be replaced with a tuple containing three concrete functions. The five obligations given in the above example state the usual group theoretic requirements that `e` be an identity element, that `inv` be an inverse, and that `op` be associative.

[1] Lists in HOL take the form `[`x_1`;`$\ldots$`;`x_n`]`. Strings are enclosed in backquotes. HOL types always begin with a colon.

Using the abstract representations and the obligations, we can prove theorems from group theory. For example, we can show that left cancellation holds, that the identity is unique, and that inverse reverses itself:

```
LEFT_CANCELLATION =
⊢ ∀ x y a:*group.
     (op rep (a,x) = (op rep (a,y))) ⟹ (x = y)

IDENTITY_UNIQUE =
⊢ ∀ f:*group.
       (∀ a:*group.  (op rep(a,f) = a) ∧ (op rep(f,a) = a)) ⟹
       (f = (e rep))

INVERSE_INVERSE_LEMMA =
⊢ ∀ a:*group. (inv rep (inv rep a)) = a
```

We can instantiate a generic theory by giving

- the name of the generic theory,
- a list of theorems showing that our instantiation meets the obligations for the generic theory,
- a list of mappings from variables in the generic theory to concrete objects in the instantiation, and
- a string that will be prepended to the names of the generic theorems to make them unique and prevent name clashes.

For example, if we have defined exclusive–disjunction and proven the corresponding group theory obligations, we can instantiate the generic theory for groups as follows:

```
let theorem_list =
    instantiate_abstract_theorems
          'group'
          [LEFT_IDENT; RIGHT_IDENT; LEFT_INVERSE;
           RIGHT_INVERSE; XOR_ASSOC]
          [("rep","(XOR, F, XOR_INV)")]
          'XOR';;
```

This gives a list of all of the theorems in the generic theory specialized for our theory of exclusive–disjunction:

```
XOR_LEFT_CANCELLATION =
⊢ ∀ x y a.  (XOR(a,x) = XOR(a,y)) ⟹ (x = y))

XOR_IDENTITY_UNIQUE =
⊢ ∀ f.  (∀ a.  (XOR(a,f) = a) ∧ (XOR(f,a) = a)) ⟹
            (f = F)

XOR_INVERSE_INVERSE_LEMMA =
⊢ ∀ a. XOR_INV(XOR_INV a) = a
```

Note that there is no mention of any part of the abstract representation in these theorems and the theorems are free of the theory obligations. In fact the theorems appear just as they would had we proven them directly rather than inheriting them from the generic theory.

2.4 Using Logic to Specify Hardware.

A circuit is a collection of devices composed by interconnection. Each of these devices has ports which are used for input, output, or both. The behavior of a device can be expressed in terms of its ports. Each of the devices in a circuit can, in turn, be viewed as a composition of still other devices. This hierarchy of devices eventually leads to the devices that the designer considers primitive. The smallest devices we will deal with in this dissertation are logic gates and indeed, in many cases, we will stop much higher than even gates.

Clocksin describes several ways to specify circuit structure [Clo87]:

- We can use imperative declarations of the circuit structure (this is referred to as the extensional method).
- We can use functions to describe the output in terms of the input.
- We can use predicates in a quantified logic to relate the ports of a device using behavioral or structural constraints.

Each of the methods has advantages and disadvantages. The extensional method has the advantage of being familiar to designers since it resembles imperative languages such as Pascal that most designers have used. The disadvantage of the extensional method is that it is difficult to treat formally, just as traditional imperative languages are hard to treat formally.

The functional model is widely used; Hunt's specification of the FM8501 processor, for example, is functional [Hun87]. To specify the behavior of sequential circuits functionally, the specification language must support recursion. Hunt uses recursion in his specification to describe the sequential operation of his CPU.

In the functional model, circuit interconnection is given by the syntactic structure of function application. This can cause several problems:

- Describing circuits with bi–directional ports is difficult since functional specifications differentiate between input and output syntactically.
- The purpose of a structural specification is to show how components are connected together. Since the only means of expressing connection is function application, even returning a tuple is insufficient for describing circuits with more than one output.
- Sequential circuits feedback on themselves. Recursion is the best alternative; but that can be inadequate for circuits with multiple feedback paths.

The predicate method is the most widely used specification technique in the HOL community [Gor86]. The disadvantage of the predicate method is that designers are likely to find it the most unfamiliar of the three and thus difficult to use. In addition, to use the predicate method, the logic must support existential quantification, either explicitly or implicitly. (Prolog's Horn clause notation is an example of a language with implicit existential quantification.) The predicate method does, however lend itself to a wide variety of circuit types, including those with multiple outputs and bi–directional ports.

The specifications in this report will use a mixture of the functional and predicate methods. Functions will be used inside the specification, but the device structure will be specified using predicates.

2.4.1 Specifying Circuits with Predicates.

As an example of the predicate model, we will specify the behavior and structure of a very simple circuit designated `D`. The predicate that specifies the behavior of the circuit can be given by the following high-order logic definition:

$\vdash_{def}$ `D(a,b,c,d,out) =  out = (a` $\wedge$ `b)` $\vee$ `(c` $\wedge$ `d)`

Notice that the inputs and outputs are all included in the arguments and the behavior is expressed as a constraint among the outputs and the inputs.

One possible implementation for `D` is shown in Figure 2.1. As was mentioned earlier, each device can be thought of as representing a constraint on its inputs and outputs. For example, the top *And* gate constrains `a`, `b`, and `p` in a manner consistent with the behavior of the device.

$\vdash_{def}$ `And(a, b, p) = (p = a` $\wedge$ `b)`

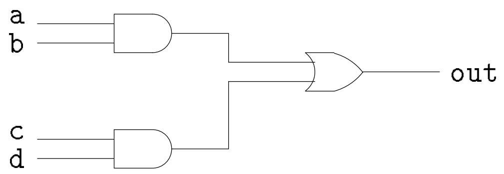

Figure 2.1: Implementation of a simple circuit, D

To get the constraint represented by the entire device, we can compose the individual constraints using conjunction.

```
And(a, b, p) ∧ And(c, d, q) ∧ Or(p, q, out)
```

This expression constrains the values not only on the ports of the device, a, b, c, d, and out, but also on the internal lines p and q. We normally wish to regard such a device as a "blackbox" and consequently are really only interested in the values of the external lines. We can hide the internal lines using existentially quantified variables and define a predicate D_imp that represents the structure of the circuit.

```
⊢def D_imp(a, b, c, d, out) =
     ∃ p q.  And(a, b, p) ∧ And(c, d, q) ∧ Or(p, q, out)
```

For comparison, the following gives a specification of the same circuit using functions:

```
⊢def D(a,b,c,d) =  Or(And(a,b),And(c,d))
```

The outputs are not mentioned explicitly; the result of the function is taken to be the output of the circuit.

Similarly, we can write a extensional specification of the circuit in a language such as VHDL [Arm89]:

```
Entity D_imp is
   port(a, b, c, d :in Bit; outp :out Bit);
end D_imp;

architecture Structure of D_imp is
   component ANDGate port(i1,i2:in Bit; outp :out Bit);
   component ORGate port(i1,i2:in Bit; outp :out Bit);
   signal p, q: Bit

   G1: ANDGate port map (a, b, p);
   G2: ANDGate port map (c, d, q);
   G3: ORGate port map (p, q, outp);

end Structure;
```

The difference between this specification and the predicate model of the circuit structure is largely superficial. The primary difference is the abundance of keywords in the extensional specification. The biggest impediment to using specification languages such as VHDL is that they sometimes lack a clear semantics. This problem can be overcome by defining a semantics of the specification language in the object language of a verification such as HOL. Van Tassel has done just that using VHDL and HOL in [Tas89, TH89].

2.4.2 Specifying Sequential Behavior.

The last section specified a simple combinatorial circuit. We specify the behavior of sequential circuits in higher–order logic using an explicit representation of time.

For example, we can specify the behavior of a simple latch as follows:

$\vdash_{def}$ `latch in out set =` $\forall$ `t. out (t+1) = set t` $\rightarrow$ `in t | out t`

In the specification, `in`, `out`, and `set` are functions of time. The value of a signal at time t is returned when the function representing the signal is applied to t. The specification says that the value of `out` at time $t+1$ gets the value of the input port, `in`, at time t if the `set` line is high and remains unchanged otherwise. Notice the use of the universal quantification over time in defining the predicate.

We can also use existential quantification to describe temporal operators. For example, suppose that we wish to define a predicate that says that a signal will *eventually* go high. The following is a definition of an `EVENTUALLY` operator:

$\vdash_{def}$ `EVENTUALLY d t1 =` $\exists$ `t2.  t2 > t1` $\wedge$ `d t2`

When applied to the signal d, and the current time, t1, the predicate states that there exists a time, t2, in the future when the signal d will be true. The use of existential quantification over time is also used to specify the behavior of asynchronous interconnections between devices.

Many of the specifications in this report will be sequential and will use this explicit representation of time. In addition to these uses of explicit quantification to treat sequential behavior, Joyce [Joy89a] has shown how temporal logic can be embedded in higher–order logic.

2.4.3 Abstraction and Specification.

Specifications can be written for many purposes. For example, in specifying a two input binary decoder, one might write:

```
⊢def decoder_spec s0 s1 o0 o1 o2 o3 =
      (o0 = (s1 → (s0 → F | F) | (s0 → F | T))) ∧
      (o1 = (s1 → (s0 → F | F) | (s0 → T | F))) ∧
      (o2 = (s1 → (s0 → F | T) | (s0 → F | F))) ∧
      (o3 = (s1 → (s0 → T | F) | (s0 → F | F)))
```

While this specification works, its meaning is not very clear.

Here is another specification for the same behavior:

```
⊢def decoder_spec s0 s1 o0 o1 o2 o3 =
      (o0 = ¬s1 ∧ ¬s0) ∧
      (o1 = ¬s1 ∧  s0) ∧
      (o2 =  s1 ∧ ¬s0) ∧
      (o3 =  s1 ∧  s0)
```

This specification closely models one possible implementation for the circuit; consequently, using it as the behavioral specification would make the verification easier, but would not tell us much about the abstract behavior of the decoder.

The next specification is more abstract and says more about the behavior of the decoder:

```
⊢def decoder_spec s0 s1 o0 o1 o2 o3 =
      (o0 ↔ ((s1,s0) = (F,F))) ∧
      (o1 ↔ ((s1,s0) = (F,T))) ∧
      (o2 ↔ ((s1,s0) = (T,F))) ∧
      (o3 ↔ ((s1,s0) = (T,T)))
```

This specification clearly shows the binary numbers being represented by the inputs. Moreover, the specification does not suggest any particular implementation. In general, the more abstract a specification, the easier it is to understand, but more difficult it is to verify.

We could make the above specification even more abstract by defining a function, `pairval`, that converts boolean pairs into numbers and then writing the specification as follows.

```
decoder_spec s0 s1 o0 o1 o2 o3 =
        let n = pairval(s1,s0) in
        (o0 ↔ (n = 0)) ∧
        (o1 ↔ (n = 1)) ∧
        (o2 ↔ (n = 2)) ∧
        (o3 ↔ (n = 3))
```

This specification can be easily generalized to have n inputs and 2^n outputs.

Chapter 3

Interpreters

In Chapter 2, we presented a description of a general model for specifying the behavior of microprocessors. The model had four parts:

1. A representation of the state, **S**.
2. A set of state transition functions, **J**, denoting the behavior of the individual instructions of the microprocessor.
3. A next state function, **N**, that selects a function from the set **J** according to the current state.
4. A predicate, **I**, relating the state at time $t+1$ to the state at time t by means of **J** and **N**.

In this chapter we concentrate on this model, which we call the *interpreter model*.

We begin with an informal discussion of the interpreter model. Much of our discussion follows that of [Anc86]. The chapter continues with a description of hierarchical decomposition. Finally, we present a mathematical definition of the interpreter model. This mathematical definition is formalized in Chapter 4.

The top level view of an interpreter is shown in Figure 3.1. The distinguishing feature of an interpreter is that it has a *flat* control structure. One of n instructions is chosen based on the state. The chosen instruction operates on the state and the cycle begins anew. In a programming language, this model could be described using a case statement in a while loop. There are a large number interesting computer systems that have a flat control structure: microprocessors, low–level system calls in an operating system, language interpreters, and editors are a few. Each of these is an instance of our general interpreter model.

The interpreter model is useful for modeling multiple abstraction levels in a microprocessor specification. So, before discussing the interpreter model, we introduce hierarchical decomposition.

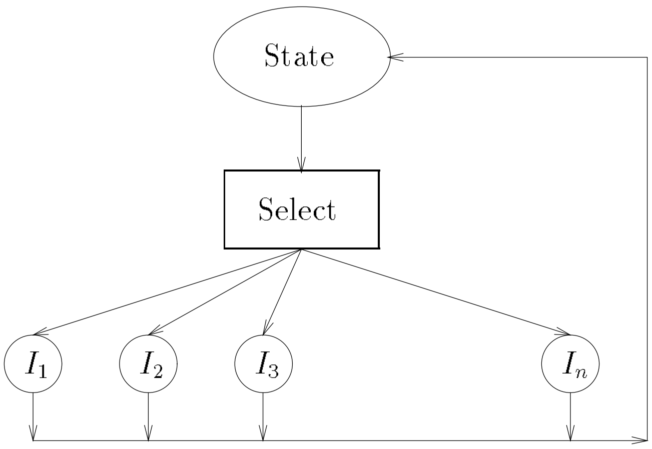

Figure 3.1: An interpreter has a flat control structure.

3.1 Hierarchical Decomposition.

The goal of our work is microprocessor verification. There are two properties of a microprocessor specification that make its verification difficult:

1. The size of $\mathbf{J}_{macro}$, the instruction set at the macro-level, is large. A typical instruction set has on the order of 2^6 to 2^8 instructions. For example, the original VIPER proof had 128 instruction cases.
2. The specification describing the electronic block model, $\mathbf{M}_{EBM}$, is large. The formal specification of the electronic block model for a typical microprocessor can take many pages. The expanded expression describing VIPER's electronic block model is 7 pages long.

According to the instruction correctness lemma (introduced in Section 2.2), we need to show that the electronic block model correctly implements each instruction in the macro-level in order to verify the microprocessor; this results in hundreds of multi-page theorems that must be proven.

3.1.1 The Hierarchy.

In order to reduce the number of long, difficult theorems, we have developed a strategy for describing the specification by means of a series of increasingly abstract interpreters. This strategy, which follows conventional microprocessor design practice, is shown in Figure 3.2.

At the bottom of the hierarchy is a structural description of the electronic block model. By specifying the electronic block model as a circuit rather than an interpreter, we ground the abstract behavioral descriptions in the hierarchy to the circuit

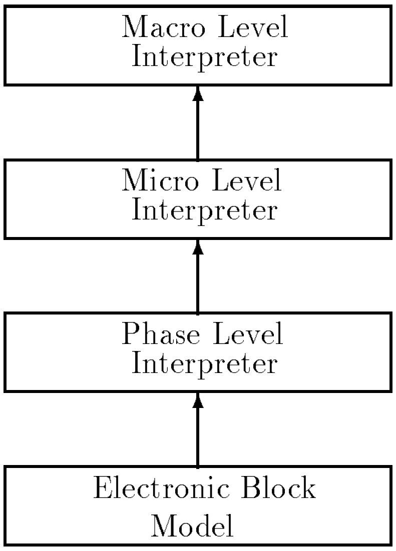

Figure 3.2: A microprocessor specification can be decomposed as a series of interpreters.

model, which is familiar to hardware designers. We specify circuits as conjunctions of predicates as described in Section 2.4.

The electronic block model is the lowest level that we will consider in this dissertation; below the electronic block model, the circuit no longer behaves as a computer, but rather as pieces of a computer. In order to implement the computer, however, the electronic block model would have to be reduced to gates.

The phase–level specification describes the behavior of the electronic block model from the perspective of the register transfer actions. During each phase, or clock sub–cycle, a set of elementary operations is executed in parallel by the machine. The phase–level specification ties each set of operations to a particular phase of the clock and states how the clock is sequenced. The sequencing of phases is usually a trivial serialization, although some conditional operation may be present in order to respond to asynchronous external events and error conditions.

The phase–level either implements the macro–level directly (in a hardwired machine) or implements the micro–level. In the latter case, the actions taken during each phase are conditioned upon the contents of the microstore. Thus, every microinstruction is implemented by a composition of phases operating on the contents of the microstore.

The micro–level description (if present) is a *behavioral* model of the micro–level interpreter. The micro–level is an abstraction of the phase–level:

- Time at the micro–level is more coarsely grained than time at the phase–level. At the micro–level, time is measured by the execution of a single microinstruction. At the phase–level, time is measured by the execution of a single phase.

- The state at the micro–level is a subset of the state at the phase–level. For example, there will be latches at the phase–level that are not important in describing the behavior of the micro–level. These latches would not be included in the description of the micro–level state.
- The behavioral description at the micro–level is concerned with a courser sequence of actions. Rather than concentrating on what happens in parallel in the datapath of the CPU, we concentrate on the state transition effected by an entire microinstruction.

At the top is the macro-level—the level visible to an assembly language programmer. Just as the micro–level is an abstraction of the behavior specified by the phase–level interpreter, the macro–level is an abstraction of the micro–level interpreter.

- Time at the macro–level is measured by the execution of a single macroinstruction. This instruction will be implemented by multiple microinstructions.
- The state at the macro–level is a subset of the state at the micro–level. For example, the instruction register is usually not visible at the macro–level.
- The behavioral specification of the macro–level describes the state transition for an entire macroinstruction.

The hierarchical decomposition, and in particular the explicit representation of the phase–level as a behavioral specification, can significantly reduce the number of long, difficult theorems that must be proven in a microprocessor proof. The next section shows why this is so.

3.1.2 Hierarchical Verification.

We wish to establish that the structure specified in the electronic block model implies the behavior of the macro-level. Past microprocessor verification efforts [Hun87, Joy88, Coh88a] have been done in one step, directly showing that

$$\mathbf{I}_{EBM} \Rightarrow \mathbf{I}_{macro}.$$

As we have seen, this can make the proof intractable for large microprocessors, due to the many long lemmas that need to be proven to establish the instruction correctness lemma. In fact, the VIPER verification [Coh88a] was never completed for this very reason; funding to complete the verification ran out before all of the cases could be considered.

The hierarchical decomposition discussed in the last section provides a way of making the proof tractable: we can establish

$$\mathbf{I}_{EBM} \Rightarrow \mathbf{I}_{macro}$$

in stages by showing

$$\mathbf{I}_{EBM} \Rightarrow \mathbf{I}_{phase} \Rightarrow \mathbf{I}_{micro} \Rightarrow \mathbf{I}_{macro}.$$

It may not be immediately obvious how this decomposition has solved the problem of verifying a large microprocessor. Recall that two things combine to make the verification of a level, ℓ, in terms of another level, ℓ', difficult:

1. the size of the term describing the implementing level, $\mathbf{I}_{\ell'}$, and
2. the number of instructions in the instruction set of the level being verified, $\mathbf{J}_{\ell}$.

The decomposition makes the proof tractable because although $\mathbf{I}_{EBM}$ is still large, $\mathbf{J}_{phase}$ typically contains from 2 to 4 instructions instead of the 2^6 to 2^8 instructions in the macro–level of a typical microprocessor. Thus, the number of long, difficult theorems is reduced by at least an order of magnitude.

The proof that the electronic block model correctly implements the phase-level interpreter is tedious, but can be done since the number of cases is small. The decomposition has, however, *increased* the total number of cases to be considered since we must now prove that the phase-level correctly implements the micro–level and that the micro–level correctly implements the macro-level. Fortunately, the proofs of $\mathbf{I}_{phase} \Rightarrow \mathbf{I}_{micro}$ and $\mathbf{I}_{micro} \Rightarrow \mathbf{I}_{macro}$ are very regular and most of the work in the proofs can be automated. The proofs between interpreter levels are automatable for two reasons:

1. Both specifications have similar structure; they are both interpreters whereas the electronic block model description is a circuit.
2. In a proof that one interpreter implements another, one can generally avoid dealing with the expanded form of the implementation, and so the goals are much smaller.

As we will see in Chapter 5, even though there may be a large number of cases to consider in proving the instruction correctness lemma in these two levels, the cases are all similar and a single tactic suffices at both levels. Thus, the amount of human effort required to complete the proof is not substantially increased by the proofs of these two levels.

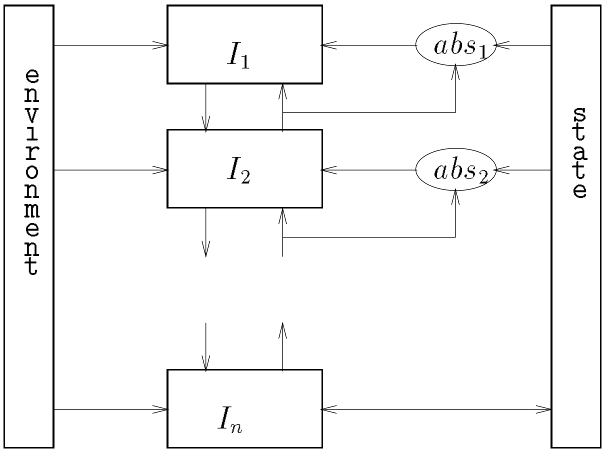

Figure 3.3: A hierarchy of interpreters.

3.2 Interpreter Hierarchies.

The hierarchical decomposition of microprocessor specifications leads us to consider a general model of interpreter hierarchies. The most important issue in the interpreter model with respect to interpreter hierarchies is using one interpreter to implement another interpreter.

Figure 3.3 shows a hypothetical interpreter hierarchy. In the hierarchy, the top level interpreter, $\mathbf{I}_1$, is implemented by the one below it, $\mathbf{I}_2$, which is implemented by the one below it, and so on. Each interpreter in the hierarchy is an abstraction of the one below it. They receive input from the environment, communicate with the interpreters above and below, and use some abstraction of the state.

Because the interpreters on the top of the hierarchy are simply abstractions of the ones below them, they are not *causal agents*. Only the bottom interpreter sees the complete state (not an abstraction). Consequently, the bottom interpreter in the hierarchy is the only one that can modify the state. An interpreter in the hierarchy can only affect the state by issuing instructions to the interpreter below it.

Figure 3.4 shows an individual interpreter from the hierarchy in more detail. The interpreter receives instructions from the interpreter above and issues instructions to the interpreter below. The interpreter does not merely pass the instructions along, but issues a new instruction stream to the interpreter below based on an *interpretation* of the instruction it has been asked to execute. The overall effect of this instruction stream is the state change required by the instruction being interpreted. When the interpreter is ready for the next instruction, it signals the interpreter it is implementing on the `Next` line.

Figure 3.4 shows the state being filtered through an abstraction box before being sent to the interpreter. The filter has a switch line connected to the `Next` line. The

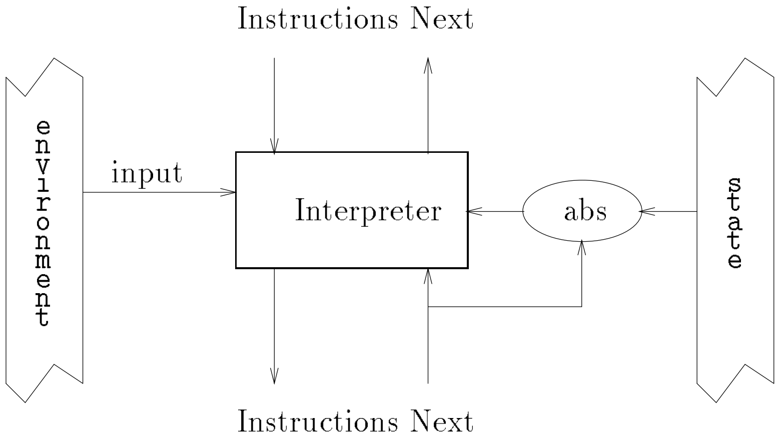

Figure 3.4: Interconnecting interpreters in the hierarchy.

abstraction serves two purposes:

1. When the `Next` line is raised, it takes a snapshot of the current state. The interpreter at this level does not see the finer grained state changes of the interpreters below it; the interpreter only sees the state when it is time to make a decision about which instruction to issue next.

2. The filter also performs a data abstraction on the state. As we will see in more detail later, the state visible at one level is a function of the overall system state.

3.3 A Mathematical Definition of Interpreters.

The rest of this chapter gives a mathematical definition to the interpreter model.

3.3.1 Basic Types.

The basic types for our model are defined in Table 3.1. In addition to these basic types, we also use the following type constructors: **product**, written $(\alpha \times \beta)$; **coproduct**, written $(\alpha + \beta)$; **function**, written $(\alpha \rightarrow \beta)$; and **list**, written $(\alpha)list$. An n-tuple is given by

$$(\alpha_1 \times \alpha_2 \times \ldots \times \alpha_{n-1} \times \alpha_n)$$

which is a shorthand for

$$(\alpha_1 \times (\alpha_2 \times \ldots \times (\alpha_{n-1} \times \alpha_n) \ldots))$$

Table 3.1: Basic types for interpreter definition.

Symbol	Members	Meaning
$\mathbf{T}$	$\{true, false\}$	truth values
$\mathbf{N}$	$\{0, 1, \cdots\}$	natural numbers
$\mathbf{B}$	$\mathbf{N} \rightarrow \mathbf{T}$	bit vectors
$\mathbf{M}$	$\mathbf{N} \rightarrow \mathbf{B}$	stores

3.3.2 State.

Abstractly, we think of state as being something of type $\mathbf{S}$, where $\mathbf{S}$ is an uninterpreted type. This allows us to treat state in an abstract manner, knowing nothing of its structure or content.

More concretely, we can represent state using n-tuples. We let $\mathbf{S}_n$ be the domain of n–tuples representing state. These n–tuples have the type

$$(\alpha_1 \times \alpha_2 \times \ldots \times \alpha_{n-1} \times \alpha_n)$$

where

$$\forall i.\ \alpha_i \in \mathbf{T} + \mathbf{B} + \mathbf{M}$$

We write $\mathbf{S} \leq \mathbf{S}'$ to indicate that $\mathbf{S}$ is an abstraction of $\mathbf{S}'$. The fact that $\mathbf{S}$ is an abstraction of $\mathbf{S}'$ implies that there exists a function, $\mathcal{S} : \mathbf{S}' \rightarrow \mathbf{S}$. The function $\mathcal{S}$ is called the state abstraction function.

3.3.3 Time.

In general, different levels in the interpreter hierarchy will have different views of time. We use temporal abstraction to produce a function that maps time at one level to time at another. Figure 3.5 shows a temporal abstraction function $\mathcal{F}$. The circles represent clock ticks. Notice that the number of clock ticks required at the implementing level to produce one clock tick at the implemented level is irregular. The temporal projection, $\mathcal{F}$, can be defined recursively on time. The resulting function is monotonically increasing and maps time at the implemented level to time at the implementing level.

We will use members of $\mathbf{N}$ to represent time, Thus we define $\mathcal{F} : \mathbf{N} \rightarrow \mathbf{N}$ such that

$$\forall n\ m.\ (n > m) \Rightarrow (\mathcal{F}(n) > \mathcal{F}(m))$$

We will discuss temporal abstraction in detail in Section 4.2.1.

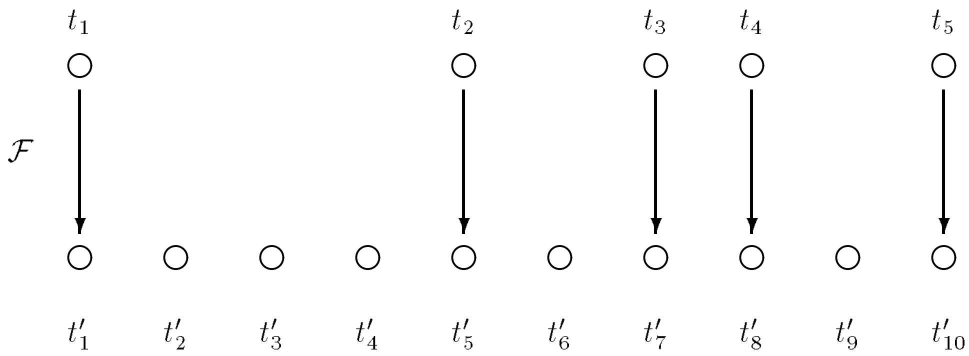

Figure 3.5: A temporal abstraction function maps time at one level to time at another level.

3.3.4 State Streams.

A state stream, $\mathbf{U}$, is a function from time to state, $\mathbf{N} \to \mathbf{S}$. We have chosen n-tuples of booleans, bit-vectors, and stores to represent state. We would like a representation of streams such that the application of a stream to some time, t, yields an n-tuple representing the state at time t. We use a lambda expression for our concrete representation.

$$\lambda t.\ (a_1(t), a_2(t), \ldots, a_{n-1}(t), a_n(t))$$

where

$$\forall i.\ a_i \in \mathbf{N} \ \to \ (\mathbf{T} + \mathbf{B} + \mathbf{M})$$

An important part of our theory will be the abstraction between state streams at different levels. When we say that state stream u is an abstraction of state stream u', we are saying

1. that each member of u is a state abstraction of some member of u' and
2. there is a temporal mapping from time in u to time in u'.

There are two distinct types of abstraction going on: the first is a data abstraction and the second is a temporal abstraction.

Using the state abstraction function, $\mathcal{S}$, and the temporal abstraction function, $\mathcal{F}$, we say that u is an abstraction of u' if and only if

$$\exists(\mathcal{S} : \mathbf{S}' \to \mathbf{S}).\ \exists(\mathcal{F} : \mathbf{N} \to \mathbf{N}).\ u = \mathcal{S} \circ u' \circ \mathcal{F}$$

where $\circ$ denotes function composition. When this is true, we write

$$u \preceq u'$$

3.3.5 Environments.

The environment represents the external world and it plays an important part in our theory. The environment is where interrupt requests originate, reset signals are generated, and so on. In our model, the environment is used only for input; output to the environment is assumed to be simply a function of the state.

At the abstract level, we treat the environment as an abstract object. We know nothing about its structure or content. We denote it as $\mathbf{E}$. Just as we defined $\mathcal{S}$, the state abstraction function, we define an environment abstraction function, $\mathcal{E}$, such that $\mathcal{E} : \mathbf{E}' \to \mathbf{E}$.

Concretely, we represent for the environment using n-tuples of booleans and bit–vectors. We perform the same kinds of abstraction on the environment as on states: we assume that there exists a function, $\mathcal{E}$, that abstracts one environment tuple to another. Temporal abstraction is performed as it was for states. We define abstraction for environment streams in the same manner that we defined it for state streams. Thus, we write $e \preceq e'$ when e is an stream abstraction of e':

$$\exists(\mathcal{E} : \mathbf{E}' \to \mathbf{E}).\ \exists(\mathcal{F} : \mathbf{N} \to \mathbf{N}).\ e = \mathcal{E} \circ e' \circ \mathcal{F}$$

3.3.6 The Interpreter Specification.

The preceding parts of this section have given preliminary definitions for concepts important in the mathematical definition of interpreters. This section presents that definition.

Interpreters are state transition systems. The difference between our model of interpreters and other models of state transition systems such as deterministic finite automata (*dfa*) is that our model accounts for state abstraction and aggregation. By state aggregation, we are referring specifically to *stores*. A store represents a collection of state that we deal with as a monolithic unit. In a *dfa* model, each location in memory would be represented by a different piece of state which would be treated individually. This clearly would not work for sizable memories.

The first step in defining an interpreter is to define a set of instructions. Let $\mathbf{J}^*$ be the set of all functions with domain $(\mathbf{S} \times \mathbf{E})$ and codomain $\mathbf{S}$. Of course, not all functions in $\mathbf{J}^*$ are meaningful; the specifier's job is to choose meaningful functions. We use a subset of $\mathbf{J}^*$ to represent the instruction set; we call this set $\mathbf{J}$. The functions in $\mathbf{J}$ provide a denotational semantics for the instructions that they represent.

In order to uniquely identify each instruction in $\mathbf{J}$, we associate it with a unique key. At the abstract level, we take keys from the domain $\mathbf{K}$. At the concrete level, keys can have various representations, as we will see in the example in Chapter 5.

We must be able to choose instructions from $\mathbf{J}$ according to some predefined selection criteria. Usually, the selection will be based on the current state and environment. We define $\mathcal{K}$ to be a function with domain $(\mathbf{S} \times \mathbf{E})$ and codomain $\mathbf{K}$. Further, we define $\mathcal{C}$ to be a choice function that has domain $(\mathbf{J} \times \mathbf{K})$ and codomain $(\mathbf{S} \times \mathbf{E} \rightarrow \mathbf{S})$. That is, $\mathcal{C}$ picks the state transition function from $\mathbf{J}$ that has a particular key in $\mathbf{K}$.

We define an interpreter, $\mathbf{I}[s, e]$, as a predicate over the state stream, s, and the environment, e. The definition of $\mathbf{I}$ is given as

$$\mathbf{I}[s, e] \;\equiv\; s(t+1) = \mathcal{C}(\mathbf{J}, k\ t)\ (s\ t)\ (e\ t)$$

where

$$k\ t = \mathcal{K}(s\ t, e\ t)$$

The predicate constrains the state of the interpreter at time $t+1$ to be a function of the state and environment at time t. The function is determined by the instruction currently selected by $\mathcal{K}$.

3.3.7 Interpreter Verification.

The goal of this formalization is to prove a correctness relation between the interpreters at different levels of a microprocessor abstraction. In particular, for two state streams, s_ℓ and s_k, and two environments, e_ℓ and e_k, where $s_\ell \preceq s_k$ and $e_\ell \preceq e_k$, we wish to show that

$$\mathbf{I}_k[s_k, e_k] \Rightarrow \mathbf{I}_\ell[s_\ell \circ \mathcal{F}, e_\ell \circ \mathcal{F}]$$

where $\mathcal{F}$ is the temporal abstraction function defined in Section 3.3.3. When this implication is true, $\mathbf{I}_\ell$ is an abstraction of $\mathbf{I}_k$ and $\mathbf{I}_k$ is said to *implement* $\mathbf{I}_\ell$.

We leave the proof of this for the formalization in Chapter 4.

3.4 Composing Specifications.

We have begun to examine how verified components can be composed to implement a more abstract behavioral specification. In the simplest case, where there is no shared state between the components, the problem reduces to the structural specification problem discussed in Section 2.4.1. When the devices share state, however, the problem is more difficult.

Consider a system consisting of a CPU and a memory subsystem with memory mapped I/O. We might be tempted to specify a single-ported memory unit as follows:

```
⊢def MEMORY_UNIT write read memory address port =
     (read t ⟹ (port = fetch(memory,address))) ∧
     (write t → (memory(t+1) = store(memory t, address, port))
              | (memory(t+1) = memory t)
```

This specification says that when the `read` signal is true, the port carries the value of memory at `address`. If the `write` signal is true, the memory is updated by storing the value on the port to the location given by `address`. Otherwise the value of memory *remains unchanged*.

There is a problem with this specification if we expect to use the CPU with memory mapped I/O. The specification assumes that the CPU is the only device that can change memory. Obviously this is not the case if the memory is shared by the CPU and other devices making up the I/O subsystem.

There are two obvious fixes to the problem:

1. Use another kind of I/O that doesn't require sharing the memory.
2. Specify all of the I/O in the system being careful to account for all the changes that can occur to memory.

The first solution is unpalatable since we would like to be able to specify a design with memory mapped I/O. The second is equally distasteful since it requires that we know all of the I/O needs that they system will ever have up front when the system is initially specified, or reverify the system with every change.

Now consider the following specification of the memory unit:

```
⊢def MEMORY_UNIT write read memory address port =
     (read t ⟹ (port = fetch(memory,address))) ∧
     (write t → (memory(t+1) = store(memory t, address, port))
              | (memory(t+1) = trans (memory t))
```

The only difference is the addition of a function, `trans`, that transforms the value of memory at time t to a new value of memory at time $t + 1$ when a write is not occurring.

The transformation function represents all of the changes that are occurring in memory for which the CPU is *not* responsible. Consider the stylized specifications for the simple case of a piece of state, S, shared by two devices. Device A specifies S as

```
S(t+1) = sig t → store_A(S t)
               | trans_A(S t)
```

Device B specifies S as

```
S(t+1) = sig t → store_B(S t)
               | trans_B(S t)
```

At a given time, $\texttt{trans}_B$ is either $\texttt{store}_A$ or I, the identity function. Similarly, $\texttt{trans}_A$ is either $\texttt{store}_B$ or I.

We cannot know at the time of specification what the value of the transformation function will be. The use of an uninterpreted function (from a generic theory) as the transformation function allows the function to appear as a place holder in the proof. Later, when the specification is composed with the specification of another device, the uninterpreted transformation functions in both specifications can be instantiated with the appropriate values.

Having the transformation function appear directly in the specification, as was just shown, has a disadvantage: we must specify every level of the device using transformation functions and, more importantly, must deal with temporal issues between levels as they relate to the transformation functions. For example, the transformation occurring on the state at the micro–level is a composition of the smaller transformations occurring to the state at the phase–level. Thus, we need a different transformation function at each level and we must know how they compose. This places a large burden on the proof, as these assumptions about the abstract transformation functions will all have to be put into the proof and discharged when the devices are composed.

There may be times when we need to know the transformations taking place in a piece of shared state in detail. Many times, however, we can use some abstraction of the transformation and only look at changes to the state using a courser time granularity. We do not want to have the transformation functions appear in the lower levels of the specification. In these cases, we can put the transformation functions in the state abstraction function. By putting the transformation in the state abstraction function for the micro–level state, for instance, the transformation function appears in the specification of the macro–level, but not in the specification of the micro–level (or below). This is the technique used in the specification of *AVM–1*; we present a concrete example of its use in Chapter 5.

Chapter 4

The Formal Models

This section presents a formalization of the theory developed in the last section. There are two points to make before we formalize the mathematical definition:

- We are free to make some of the abstract entities in the mathematical definition more concrete. For example, we will represent the instruction set as a list. What we make concrete and what we leave abstract is a subjective choice. We want to make the definition concrete enough that we can prove interesting theorems about it without restricting the model in unnecessary ways.
- There will be more details to consider concerning types, definitions, and so on since we are dealing with a formal system. HOL's polymorphic type system frees us from some of this, but the details still have to be right.

This chapter formalizes two interpreter models: a synchronous model and an asynchronous model. The terms "synchronous" and "asynchronous" are historical. Our original, synchronous model was too restrictive to support asynchronous memory and thus was described as a model for synchronous memory machines—quickly shortened to the synchronous model. The less restrictive model was naturally called "asynchronous." Perhaps the terms are unfortunate since, as we will see, the temporal abstraction in both models is synchronous. The synchrony is deterministic in the "synchronous" model and non-deterministic in the "asynchronous" model. Of course, calling the models "deterministic" and "non–deterministic" would be confusing as well and we have chosen to keep the historical names. The first part of this chapter will present the theory of synchronous interpreters. The second part of the chapter presents the less restrictive model of asynchronous interpreters.

4.1 Synchronous Interpreters

The theory presented in this section is for synchronous interpreters. In a synchronous interpreter model, the number of instructions in the implementation required to implement each instruction in the interpreter is deterministic. Obviously for microprocessors that have loops in their microcode or use asynchronous memory, the synchronous model will not suffice. We present it, however, because it is more straightforward and somewhat easier to use; there are times when one can live with the restrictions of the synchronous model.

4.1.1 The Abstract Representation.

The abstract representation is the interface to the generic theory. We specify the abstract representation by defining a list of abstract objects and operations. The operations are functions with domains consisting of both abstract and concrete types.

```
let cpu_abs = new_abstract_representation
    [
     (`inst_list`,":(*key#(*state->*env->*state))list")
      ;
     (`select`,":*state->*env->*key")
      ;
     (`key`,":*key->num")
      ;
     (`cycles`,":*key->num")
      ;
     (`substate`,":*state'->*state")
      ;
     (`subenv`,":*env'->*env")
      ;
     (`Impl`,":(time'->*state')->(time'->*env')->bool")
      ;
     (`count`,":*state'->*env'->*key'")
      ;
     (`begin`,":*key'")
      ;
    ];;
```

Not all of the members of the abstract representation are used in the definition of the interpreter. Some of them are only used to specify the theory obligations and formulate the correctness statement.

Before describing the abstract representation, we must emphasize that the representation *is* abstract and therefore, the objects and operations have no definitions. The descriptions that follow are what we *intend* for the representation to mean. The representation is purely syntactic, however; the names are simply convenient mnemonics.

We begin by giving a description of the abstract types used in the representation. We know nothing of the structure or composition of an abstract type.

- `:*state` represents the state and corresponds to **S** in the informal description presented in Chapter 3.
- `:*env` represents the environment and corresponds to **E** in the informal description presented in Chapter 3.
- `:*key` is type containing all of the keys and corresponds to **K** in the informal description presented in Chapter 3.

In addition to these abstract types, the representation makes use of several concrete types: `:time`, `:num`, and `:bool`. The `list` and $\rightarrow$ (function) type constructors are used as well. We add primes to the types to indicate that they represent state, time, etc. at the implementing rather than the implemented level.

As we mentioned earlier, there is a trade–off between the concreteness of the representation and the strength of the final result. We could make the specification of generic interpreters more abstract, but the result would likely be weaker. We could make it more concrete to strengthen the conclusions, but then we risk making it unusable. A good example of this trade–off is the representation of the instruction set.

Our consideration for concreteness led us to discard a completely abstract object such as `:*inst_set`. As a practical issue, the theorems and tools for manipulating lists are codified much better in HOL than they are for sets. Since we will be proving results about the instruction set in order to instantiate the abstract theory, we chose lists over sets as the aggregation mechanism.

The abstract function `inst_list` corresponds to $\mathbf{J}$ in the mathematical definition presented in Chapter 3. The instruction set, `inst_list`, is a collection of state transition functions and is denoted by a list of pairs. The first member of the pair is a key and the second member is a state transition function which operates on a state object and an environment object to produce a new state.

The second member of the representation is the `select` function that picks a key based on the present state and environment. The `select` function corresponds to $\mathcal{K}$ from the definition in Chapter 3.

The key returned by the `select` function is used to choose a member of the instruction list. The third member of the representation is used in indexing the instruction list. Since our representation for the instruction set is a list, `key` maps an object of type `:*key` to a number which is used with the HOL list indexing function `EL` to pick an instruction from the list. Together, `EL` and `key` correspond to the function $\mathcal{C}$ from the definition in Chapter 3.

The model presented here is synchronous and therefore the proof requires that the number of cycles in the implementation be determinate for each instruction in `inst_list`. The function `cycles` returns, for each object of type `:*key`, the number of cycles used to implement the instruction associated with that key.

The function `substate`, which corresponds to $\mathcal{S}$ in Chapter 3, is the state abstraction for the interpreter. Notice that the domain of `substate` is primed indicating that it is from the implementing level. The function `subenv`, which corresponds to $\mathcal{E}$ in Chapter 3, is the environment abstraction function.

The definition in Chapter 3 did not treat the implementation. Because we want to prove correctness results about the interpreter, we must have something to verify it against. The final three functions in the abstract representation provide the necessary abstract definitions for the implementation.

`Impl` is the abstract implementation. We could have chosen to make this function more concrete and define it as we do the interpreter (see Section 4.1.3), but doing so would require that every implementation be an interpreter or at least have some pre-chosen structure. As we will see in the example (Chapter 5), the implementation need not be modeled as interpreter at all. Thus, we say nothing about it besides defining its type. For now, its structure and operation are completely unknown.

The abstract function `count` is analogous to `select` except it operates at the implementing level. Notice that it uses the state and environment at the implementing level to produce a key for the implementing level. As we will see, this function is important in synchronizing the two levels. In the course of the verification we will ensure that the implementation periodically reaches the beginning of its cycle, denoted by the last member of the abstract representation, `begin`.

We must emphasize once again that even though we have spent several paragraphs defining what each of the members of the abstract representation mean, they are truly abstract and have no meaning in the formal theory other than the relationships that will be defined in the theory obligations.

4.1.2 The Theory Obligations.

Theory obligations represent the semantics of the interface to the generic theory. Inside the theory, the only thing we know about the abstract representation presented in the last section is what the theory obligations say about it.

What properties should the theory obligations have?

- We would like the theory obligations to be *sufficient* to prove the correctness result. We make no claim that they are sufficient to prove any other property about our model.

- We also would like, but do not require, that the theory obligations all be *necessary* to prove the correctness result. An unnecessary obligation must be satisfied over and over again for every instantiation, even though it follows from the other obligations and definitions in the abstract theory. We ignore *obviously* unnecessary obligations that are never used in proving the theorems in the abstract theory.

  We cannot *prove* that all of the obligations in the generic interpreter theory are necessary, but there are only three of them and they seem reasonably disjoint.

To prove the correctness result, we must know something about the implementation. Since the implementation is a member of the abstract representation, nothing is known about it except the requirements set forth in the theory obligations. Proving that the implementation implies the interpreter definition is typically done by case analysis on the instructions; we show that when the conditions for an instruction's selection are right, the instruction is implied by the implementation. In Section 2.1 we called this the instruction correctness lemma.

The predicate `INSTRUCTION_CORRECT` expresses the conditions that we require in the instruction correctness lemma.

```
⊢def INSTRUCTION_CORRECT rep s' e' inst =
       (Impl rep s' e') ⟹
        (∀ t:time'.
           let s = (λ t. (substate rep (s' t))) in
           let e = (λ t. (subenv rep (e' t))) in
           let c = (cycles rep (select rep (s t) (e t))) in (
           (select rep (s t) (e t) = (FST inst)) ∧
           (count rep (s' t) (e' t) = (begin rep)) ⟹
             ((SND inst) (s t) (e t) = (s (t + c))) ∧
             (count rep (s' (t + c)) (e' (t + c)) = (begin rep))))
```

`INSTRUCTION_CORRECT` is not really as complicated as it looks. The predicate operates on a single instruction `inst`. The implementation implies that for all time, if `inst` is selected and the implementation's counter is at the beginning, then two things are true:

1. Applying the instruction to the current state yields the same state change that the implementation does in `c` cycles and
2. The counter in the implementation returns to the beginning of its cycle after `c` cycles.

In all cases the number of low–level cycles it takes to implement one upper–level instruction must be determinate. The use of the abstract function `cycles` to determine how long the instruction takes to run, `c`, enforces this condition.

Using `INSTRUCTION_CORRECT` we can define the theory obligations:

```
new_theory_obligations
   [
    "EVERY (INSTRUCTION_CORRECT rep s' e') (inst_list rep)"
    ;
    "∀ k:*key.  (key rep k) < (LENGTH (inst_list rep))"
    ;
    "∀ k:*key.  k = (FST (EL (key rep k) (inst_list rep)))"
    ;
   ];;
```

The first obligation says that every instruction in the instruction list, `inst_list`, satisfies `INSTRUCTION_CORRECT`. The second obligation says that every key maps to some location in the instruction list. The third obligation says that `key` actually maps a key to the instruction with which it is associated (i.e. that the list is ordered correctly).

As mentioned in Section 2.3, the obligations are used in two ways. First they are used axiomatically in proving the correctness result; we will do this in the next section. Second, they are the properties that users of the theory must prove about an instantiation. We will show this in Section 5.3. As we will see, the obligations are really a small burden since the first obligation would have to be proven whether the generic theory was used or not and the other two are simple to prove for most instantiations.

4.1.3 The Correctness Statement.

Before proving the correctness statement, we must define the abstract interpreter. In addition to the state and environment, the interpreter is parameterized by the representation, `rep`. As discussed in Section 2.3.3, the objects in an abstract representation are really selection functions on a higher–order tuple in the logic. Thus the expression `(key rep)` in the interpreter definition selects the key function from the representation. When `rep` is instantiated, the selection function returns the concrete function for `key`.

```
⊢def INTERP rep (s:time → *state) (e:time → *env) =
       ∀ t:time.
        let n = (key rep (select rep (s t) (e t))) in (
        s(t+1) = (SND (EL n (inst_list rep))) (s t) (e t))
```

The interpreter definition corresponds to $\mathbf{I}[s, e]$ in the definition in Chapter 3. The interpreter relates the state at time $t + 1$ to the state and environment at time t through an instruction selected from the instruction list. The instruction is indexed in the list using the number returned from applying `key` to the result of `select` and HOL's list indexing instruction `EL`. The state transition function is the second element in the resulting pair (selected using `SND`).

We wish to show that when the theory obligations are met, the implementation implies the interpreter definition. In order to prove the correctness result, we will need to define the temporal abstraction between the implementation and the interpreter.

When time appears in the theory obligations, it is time at the implementation level. Before we can relate the interpreter and its implementation, we must relate the different time granularities at the two levels. The relationship between the two representations of time can be expressed in a recursive function.

```
(time_shift g s e 0 = 0) ∧
(time_shift g s e (SUC n) = (
    let t = (time_shift g s e n) in
    t + (g (s t) (e t))))
```

When applied to time at the interpreter level, `time_shift` returns time at the implementation level. `Time_shift` is the function $\mathcal{F}$ in Figure 3.5. The function g takes a state value and an environment value and returns the number of cycles for the instruction that is to be executed. We implement g using `select` and `cycles`. The function recurses to determine how many implementation cycles were required to reach the current instruction.

The instruction correctness lemma contains a termination assumption that says that the implementation clock always returns to the beginning of its cycle at every interpreter clock tick. This assumption is too messy to appear in the final result since it seems difficult to discharge. Actually, we can show that a much simpler assumption implies the more complicated one. This is known as the clock lemma.

The clock lemma shows that if `count`, the implementation level clock, is at the beginning of its cycle at time 0, then it will be at the beginning of its cycle for *every* clock tick at the interpreter level.

```
CLOCK_LEMMA =
⊢ (Impl rep) s' e' ∧
     ((count rep) (s' 0) (e' 0) = (begin rep)) ⟹
     let s = (λ t:time. (substate rep (s' t))) and
         e = (λ t:time. (subenv rep (e' t))) in (
     ∀ t.  let t_impl =
             (time_shift
               (λ st env.
                  (cycles rep (select rep st env))) s e t) in
        (count rep) (s' t_impl) (e' t_impl) = (begin rep))
```

We can use a reset button in the implementation to force the clock to the beginning of its cycle at time 0.

Using the clock lemma, the theory obligations and several intermediate lemmas, we can prove the correctness result for our generic, synchronous interpreter.

```
INTERP_CORRECT =
⊢ let s = (λ t:time. (substate rep (s' t))) and
       e = (λ t:time. (subenv rep (e' t))) in (
     (Impl rep) s' e' ∧
     ((count rep) (s' 0) (e' 0) = (begin rep)) ⟹
     let f = time_shift
              (λ st env. (cycles rep (select rep st env))) s e in
     (INTERP rep) (s o f) (e o f))
```

The state and environment variables in the correctness theorem, `s` and `e`, are functions of time at the implementation level. Of course, to use them as arguments to the interpreter definition, `INTERP`, we need to temporally abstract implementation time to interpreter time. Using `time_shift`, we can modify the state and environment streams, producing streams appropriate for the interpreter. The expression `(s o f)` represents the interpreter level state stream whereas `s` is the implementation level state stream.

The correctness theorem states that the implementation implies the definition of the interpreter as long as the implementation clock starts off at the beginning of its cycle. Of course, the result is also predicated on the theory obligations. They are not visible in the theorem, but they must be discharged before it can be used.

4.2 Asynchronous Interpreters

The previous section presented a formal model of generic interpreters where the synchronization function, `time_shift`, operated deterministically. The determinism was provided by the abstract function `cycles` which returns the number of implementation cycles for each member of the instruction set. Often, such deterministic synchrony is not desirable, or even possible.

- The number of implementation cycles may depend not only on the instruction, but on the arguments to the instruction as well. A multiply instruction is one example.
- The number of implementation cycles may depend on some external device or signal. Examples of this include asynchronous memory, an interrupt, or user input.

Since instructions with non-deterministic synchrony in their implementations occur so frequently in computer systems, our model would not be very useful if it excluded them. This section presents a modification of the synchronous model presented in the last section. The new model removes the restriction that the number of implementation cycles for each instruction be deterministic, while maintaining the strong correctness result. The section starts off with a discussion of a more general view of temporal abstraction and then presents the modified theory.

4.2.1 Temporal Abstraction

Section 3.3.4 presented an informal look at stream abstraction. As discussed in that section, a major component of abstraction over streams was temporal abstraction. The function `time_shift`, which appeared in Section 4.1.3, was an attempt to relate

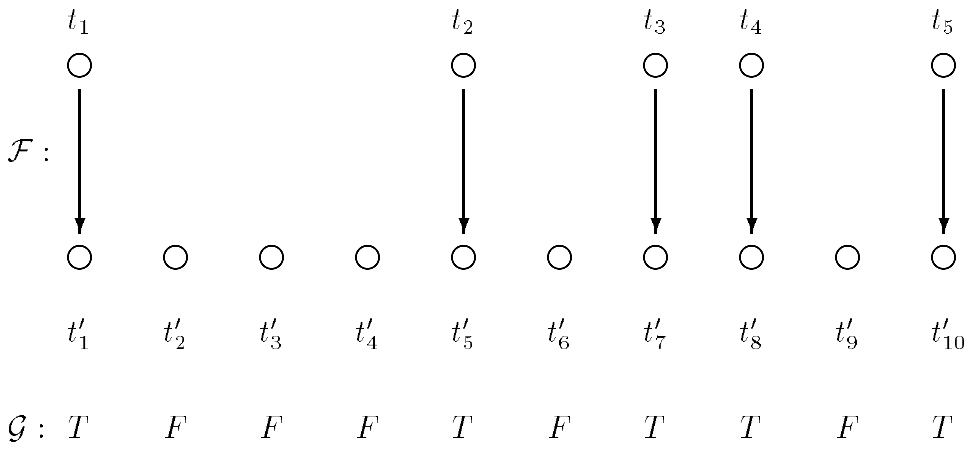

Figure 4.1: The function $\mathcal{F}$, which maps time at one level to another, can be defined in terms of a predicate, $\mathcal{G}$, which is true only when the mapping occurs.

the different views of time at the implementing and implemented levels in the synchronous interpreter. The function was simple in concept and execution, but is too restrictive. This section presents the development of a formal theory for temporal abstraction. The development follows that of [Joy89a, Mel88, Her88]. The ML code creating this theory is contained in [Win90b] .

Figure 4.1 is the same as Figure 3.5 except for the representation of the predicate, $\mathcal{G}$. This predicate is true whenever there is a valid abstraction from the lower level to the upper level. We can define a generic temporal abstraction function in terms of $\mathcal{G}$. It may seem that we have given up having to define `cycles` only to be burdened by having to define $\mathcal{G}$; but as we will see, defining $\mathcal{G}$ is much less restrictive than defining `cycles`. In a microprocessor specification, $\mathcal{G}$ is usually a predicate indicating when the lower level interpreter is at the beginning of its cycle—a condition that is easy to test.

To begin, we can define `First` and `Next`, two predicates that use $\mathcal{G}$ to express two very important concepts. The predicate `First` is true when `t` is the first time that `g` is true.

```
⊢def First g t =
        (∀ p:time. p < t ⟹ ~(g p)) ∧
        (g t)
```

The predicate `Next` is true when `t2` is the next time after `t1` that `g` is true.

```
⊢def Next g (t1,t2) =
       (t1 < t2) ∧
       (∀ t:time. t1 < t ∧ t < t2 ⟹ ~(g t)) ∧
       (g t2)
```

We would like to define $\mathcal{F}$ (see Figure 4.1) using `First` and `Next`. Clearly, at time t_1, `First g` t'_1 is true. In addition, `Next g` (t'_1, t'_5), `Next g` (t'_5, t'_7), `Next g` (t'_7, t'_8), and `Next g` (t'_8, t'_{10}) are true as well. How can we use `First` and `Next`, both predicates, to return the proper values?

The axiomatization of HOL uses Hilbert's choice operator, ε. Given some predicate P, $\varepsilon\ x.\ P(x)$ represents a value satisfying P. For example,

```
ε x:num. x * x = 25
```

denotes 5 (but not -5 as the type `num` only contains the natural numbers). So, using the choice operator, we can define $\mathcal{F}$ as follows ($\mathcal{F}$ has been renamed to `Temp_Abs` which is more mnemonic):

```
⊢def (Temp_Abs g 0 = ε t:time. First g t) ∧
     (Temp_Abs g (SUC n) = ε t:time. Next g (Temp_Abs g n,t))
```

So, `Temp_Abs` at time 0 is the first time `g` is true and `Temp_Abs` at time $n + 1$ is the next time after time n when `g` is true.

The only problem with this definition is that Hilbert's operator is difficult to use in proofs since the methods for handling it in HOL are relatively weak. Fortunately, it is possible to prove theorems about `Temp_Abs` that make it simple to reason about its behavior. Several of these are defined in the temporal abstraction theory found in [Win90b] . One of the most important is the following theorem that says that if `g` is true infinitely often and a relation, `r`, holds between points of time at the upper level, then the same relation holds between the times returned by `Temp_Abs`.

```
INF_Temp_Abs =
⊢ ∀ g r.
     (∃ t:time. g t) ∧
     (∀ t:time. g t ⟹ ∃ n. Next g (t,t+n) ∧ r(t,t+n)) ⟹
     ∀ u. r(Temp_Abs g u, Temp_Abs g (u+1))
```

Another useful theorem describes what happens when `g` is always true.

```
Temp_Abs_DEGENERATE =
⊢ Temp_Abs (λ t:time. T) = I
```

This is a degenerate case and as intuition would suggest, `Temp_Abs` simply reduces to the identity function, `I`. It might not be clear why this last theorem is of use. In defining the generic theory, we will assume that a temporal abstraction always exists between levels in a specification. Such is not the case; sometimes, there is no temporal abstraction. Rather than dealing with this as a special case, it is convenient to use the general theory with a degenerate temporal abstraction function.

4.2.2 The Abstract Representation

The abstract representation for the asynchronous model is identical to the representation for the synchronous model except that the abstract function `cycles` has been eliminated.

```
let cpu_abs = new_abstract_representation
    [
     (`inst_list`,":(*key#(*state->*env->*state))list")
      ;
     (`key`,":*key->num")
      ;
     (`select`,":*state->*env->*key")
      ;
     (`substate`,":*state'->*state")
      ;
     (`subenv`,":*env'->*env")
      ;
     (`Impl`,":(time'->*state')->(time'->*env')->bool")
      ;
     (`count`,":*state'->*env'->*key'")
      ;
     (`begin`,":*key'")
      ;
    ];;
```

The meanings of the abstract functions in the representation are identical to the meanings of the functions in the abstract representation for the synchronous model. Of course, they are only place holders in the definitions that follow.

4.2.3 The Theory Obligations

The major change in the theory obligations for the asynchronous model involves the instruction correctness predicate. The instruction correctness predicate for the synchronous model was able to calculate the number of cycles required to implement an instruction using the abstract operation `cycles`. The length of an instruction cycle in the asynchronous model is indeterminate, but finite. In fact, that is all we need to say about it to prove the correctness statement. We will say that there exists a time in the future when the current cycle will be over. The currently selected instruction applied to the current state should yield the same value as the state at beginning of the next cycle.

```
⊢def INST_CORRECT rep s' e' inst =
     (Impl rep s' e') ⟹
      (∀ t:time'.
         let s = (λ t. (substate rep (s' t))) in
         let e = (λ t. (subenv rep (e' t))) in
         let g = (λ t. (count rep (s' t) (e' t) = (begin rep))) in (
         (select rep (s t) (e t) = (FST inst)) ∧
         (count rep (s' t) (e' t) = (begin rep)) ⟹
           ∃ c.  Next g (t,t+c) ∧
           ((SND inst) (s t) (e t) = (s (t + c)))))
```

As before, `s` and `e` are the abstracted state and environment. We define, `g`, the predicate that is true when the cycle is over, by testing if `count` is equal to `begin`. The predicate `Next` uses `g` to constrain the existentially quantified variable, `c`, to the time when the cycle ends.

Once the instruction correctness predicate has been defined, the theory obligations for the asynchronous model are identical to the theory obligations for the synchronous model.

```
new_theory_obligations
   [
    "EVERY (INST_CORRECT rep s' e') (inst_list rep)"
    ;
    "∀ k:*key.  (key rep k) < (LENGTH (inst_list rep))"
    ;
    "∀ k:*key .  k = (FST (EL (key rep k) (inst_list rep)))"
    ;
   ];;
```

Due to the changes in the instruction correctness predicate, the theory obligations for the asynchronous model are less restrictive than the theory obligations in the synchronous model; nevertheless, they are sufficient for proving the correctness result. As we will see in Chapter 5, however, satisfying the less restrictive obligations can be more difficult than satisfying the obligations for the synchronous model; this can make instantiating the generic theory more difficult.

4.2.4 The Correctness Statement

Just as in the synchronous model, we must define the interpreter before we prove a correctness statement about it. The definition for the interpreter is the same in both models.

```
⊢def INTERP rep s e =
      ∀ t:time.
       let n = (key rep (select rep (s t) (e t))) in (
       s(t+1) = (SND (EL n (inst_list rep))) (s t) (e t))
```

The specification of an interpreter is a predicate relating the contents of the state stream at time $t+1$ to the contents of the state stream at time t. The relationship is defined using the functions from the abstract representation in the same manner as before.

An important step in proving the correctness result is showing that the implementation implies that the next state follows from the currently selected instruction. Of course, the theory obligations play an important part in proving this lemma, which is called the next–state lemma.

```
IMPL_NEXTSTATE_LEMMA =
⊢ let s = (λ t:time. (substate rep (s' t))) and
      e = (λ t:time. (subenv rep (e' t))) and
      f = (λ t. (count rep (s' t) (e' t) = (begin rep))) in (
     (Impl rep s' e') ⟹
       (∀ t:time'.
        (count rep (s' t) (e' t) = (begin rep)) ⟹
         ∃ c.
         Next f (t,t+c) ∧
         ((substate rep (s' (t + c))) =
             (SND (EL (key rep (select rep (s t) (e t)))
                      (inst_list rep))) (s t) (e t))))
```

The implementation–level clock is assumed to start at the beginning of its cycle and the `Next` function is used to constrain the clock so that it terminates, ready to start the cycle again.

We use the next–state lemma to prove the correctness result. There is no need to prove a clock lemma in the asynchronous model. The clock lemma in the synchronous model proved that the temporal abstraction function, `time_shift`, behaved correctly. In the asynchronous model the temporal abstraction function is correct *by definition.* The use of the choice operator says that the value will be correct provided one exists.

```
IMPL_I_CORRECT =
⊢ let s = (λ t:time. (substate rep (s' t))) and
       e = (λ t:time. (subenv rep (e' t))) and
       f = (λ t:time. (count rep (s' t) (e' t) = (begin rep))) in
     let abs = (Temp_Abs f) in (
     (Impl rep s' e') ∧
     (∃ t. f t) ⟹
     (INTERP rep) (s o abs) (e o abs))
```

In the correctness statement, `s'` and `e'` are the state and environment streams in the implementation. The terms `(s o abs)` and `(e o abs)` are the state and environment streams for the interpreter defined in the theory. They are data and temporal abstractions of `s'` and `e'`. The correctness statement says that if the implementation is valid on its state and environment streams and there is a time when the implementing clock is at the beginning of its cycle, then the interpreter is valid on its state and environment streams.

4.3 Conclusions

We have now proven a correctness statement for two different interpreter models. These models each define a class of computational objects. The correctness results provide a verification of *every* microprocessor matching the loose semantics defined in the models.

The most important benefit of the generic models is that they structure the proof. A generic model states explicitly which definitions must be made (one for each of the members of the abstract representation) and which lemmas need to be proven about these definitions (namely, the three theory obligations). This is a large improvement over previous microprocessor verifications where these decisions were made on an *ad hoc* basis.

For each model, the correctness theorem, definitions, and abstractions that make up the theory are important for several reasons.

1. The models show exactly what is required to verify that an interpreter is correct. There is no superfluous detail cluttering up the definitions and theorems.

2. The generic proof is *easier* than the specific proof. This point is subjective, but having completed three different specific interpreter proofs, we found that proving the correctness result for the generic interpreter was easier than similar theorems for specific interpreters. In proving theorems about specific interpreters, there is always some amount of detail that is necessary for the specific interpreter, but not meaningful in the correctness result. Even so, this detail must be manipulated to complete the proof.

3. Temporal abstraction issues are handled completely within the generic theory. This frees the user of the theory from proving theorems about the temporal abstraction; it is only done once, when the theory is built.

4. Similarly, data abstraction between the state and environment streams at the two levels in the theory is clearly defined and consistently performed. The user's contributions are to define the abstractions, the theory uses the abstractions to effect the proof.

5. The generic proof can be instantiated, allowing the theorems to be reused and saving the verifier from having to reverify these theorems.

The use of a generic interpreter theory for specifying and verifying microprocessors provides a methodological approach. Making specification and verification methodological is an important step in turning what has primarily been a research activity into an engineering activity. We believe that the most important contribution of this work may be the *organization* that the generic theory provides.

Chapter 5

A Verified Microprocessor

We have designed a computer designated *AVM–1* (*A Verified Microprocessor*) to demonstrate the use of generic interpreters in verifying hierarchically decomposed microprocessor specifications. There are several reasons why we chose to design our own microprocessor rather than using an existing one:

- In order to verify a commercial microprocessor, we would have to have access to the design, which is likely to be proprietary. Further, the design would have to be correct, which is unlikely.
- A formal specification for a commercial microprocessor is unlikely to be available.
- Any specification written for a commercial microprocessor would be "after the fact" and therefore suspect.
- There are architectural and organizational features that can ease the burden of verification. An existing microprocessor might not have these features. Among these are regular instruction formats and microcoding. We will explain why these features reduce the verification effort.

Our design is an attempt to build a microprocessor that is at once verifiable, implementable, and usable. We have been influenced by our own experience in verifying microprocessors [Win90a], the experience of others [Joy89a, Coh88a], and our desire to provide hardware features in support of operating systems; such features include interrupts, memory management, and supervisory modes. *AVM–1* is part of a verified chip set being designed and verified by the Computer Systems Verification Group at the University of California, Davis. Other pieces of the system include a memory management unit, a floating point unit, an interrupt controller, and a direct memory access chip.

Counter to the current trend in microprocessor design, we have chosen to implement *AVM–1* using microcode. We believe that microcoding a verified design can reduce the amount of effort required to verify the implementation. As we mentioned in Section 3.1, we can hierarchically decompose the specification in order to limit the number of difficult cases in the proof. Recall that the difficulty is caused by the size of the electronic block model description and the fact that it is a structural, rather than a behavioral specification. In verifying a hierarchically decomposed specification, these difficult cases occur when verifying the phase–level with respect to the electronic block model. If the microprocessor is not microcoded, the phase–level description becomes much more complicated and the difficulty of the phase–level proof is exacerbated. This is not to say that hardwired designs cannot be verified, just that they are more difficult.

Another reason for using microcode in the design of a verified microprocessor is the opportunity it affords for easily reverifying the microprocessor when minor changes to the design are made. As we will see, the most difficult part of a microprocessor verification is proving the correspondence between the electronic block model and the phase–level. The phase–level description can be parameterized over the microrom so that the microrom, and consequently the microprocessor's behavior, can be changed without having to redo the difficult phase–level verification. Once a verified phase-level interpreter exists, establishing a proof for a new macro-level can be accomplished with little additional effort.

- Because of the regularity of the proofs for the macro–level and micro–level, general purpose tactics can be devised to verify these levels.

- The proof can be completed by defining the microcode, reverifying the new design using the tactics mentioned above, and instantiating the generic interpreter theory to generate the proof.

Thus a microprocessor design can be customized at very little additional cost after the initial micro-engine has been verified and tactics for verifying the higher levels in the hierarchy have been developed.

This chapter presents a detailed example of how the generic interpreter theory can be used to verify a microprocessor. We begin with a discussion of the architecture and organization of *AVM–1*. The second section of the chapter formally specifies each of the levels in the hierarchical decomposition of *AVM–1*. The last section describes the development of a correctness proof for *AVM–1* using the formal specifications and the HOL verification environment.

5.1 *AVM–1*'s Architecture and Organization.

We distinguish between a computer's architecture and its organization. The former is behavioral in nature and the latter structural. Our goal in this chapter is to show that a particular organization correctly implements our desired architecture. This section will give a brief, natural language description of the architecture and organization of *AVM–1*. Later in this chapter, we will present a formal specification of both using higher–order logic.

5.1.1 An Architectural View.

A computer's architecture is its programming interface; an architecture describes a language and how that language is interpreted. The language definition contains a specification of the computer's state and the instructions available for manipulating that state. The architecture must also define how instructions are selected.

Specifying an architecture amounts to defining a language. This definition can be done in a natural language or in a more formal language; but, still primarily tells the programmer how the machine interprets instructions. This section uses a combination of natural language and a less ambiguous register transfer language (RTL) to describe *AVM–1*. The description is similar to what one would find in a programmer's manual for a commercial microprocessor.

The instruction set was inspired by the RISC I instruction set found in Katevenis [Kat85]. There are a number of differences, but many features in the RISC I instruction set (such as using ALU operations to synthesize a MOVE instruction) were incorporated into the *AVM–1* instruction set. As we will see in the section on organization, however, *AVM–1* cannot be called a RISC architecture since its microcoded implementation is different than today's RISC chips.

One caveat: *AVM–1* was not designed to be a showcase for architecture, but rather to show that microprocessors with modern features such as privileged modes and interrupts could be verified. While one may quibble with the design of *AVM–1*, this in no way affects the usefulness of the example.

5.1.1.1 RTL Notation.

We will use a register transfer language to describe the semantics of the instruction set. There are many register transfer languages in use; the notation and symbols for the RTL used in this dissertation are found in Table 5.1. In general, any capital letter refers to a register. We will define the symbols standing for certain registers later, as the registers are described. Memory is designated by `M`. Most of the other symbols are self–explanatory. The keyword `status` returns the status of the last ALU operation, that is the carry, overflow, negative, and zero flags.

5.1.1.2 The Registers.

AVM–1 has a load–store architecture based on a large register file. The register file (denoted `R` in our RTL) is divided into three portions:

1. Register 0 which is read–only and contains the constant 0.

Table 5.1: Symbols in the Register Transfer Language.

Symbol	Meaning	Example
letters	a register	PSW, PC
subscripts	one or more bits in a register	PSW_4
R[X]	register X of the register file	R[10]
( )	field in a register	PSW(1-4)
M[X]	location X in memory	M[PC]
$\Leftarrow$	transfer of information	PC $\Leftarrow$ R[3]
$p \rightarrow Op_1 \mid Op_1$	if p then Op_1 else Op_2	$PSW_6 \rightarrow (B \Leftarrow C) \mid (B \Leftarrow D)$
,	separates parallel operations	$B \Leftarrow C, D \Leftarrow E$
+	add	$B \Leftarrow C + D$
–	subtract	$B \Leftarrow C - D$
$\vee$	logical–OR	$B \Leftarrow C \vee D$
$\wedge$	logical–AND	$B \Leftarrow C \wedge D$
$\oplus$	logical–exclusive–OR	$B \Leftarrow C \oplus D$
$\neg$	logical–complement	$B \Leftarrow \neg C$
`shl`	logical shift left	B $\Leftarrow$ shl C
`shr`	logical shift right	B $\Leftarrow$ shr C
`asr`	arithmetic shift right	B $\Leftarrow$ asr C
`msb`	most significant bit	PSW_1 $\Leftarrow$ msb C
`lsb`	least significant bit	PSW_1 $\Leftarrow$ lsb C
`status`	status of last ALU operation	PSW(0-3) $\Leftarrow$ status

Table 5.2: The program status word.

Bit	Meaning when set
0	Last ALU result was zero
1	Last ALU operation caused a carry
2	Last ALU result was negative
3	Last ALU operation caused a overflow
4	Interrupts enabled
5	In supervisory mode

2. Seven supervisor–mode registers including a distinguished register for use as the supervisor stack pointer (denoted `SSP`). The supervisor–mode registers are read–only unless the CPU is in supervisor–mode (determined by the 6^{th} bit in the program status word).

3. Twenty–four general purpose registers.

Two additional registers are visible at the architectural level: the program counter and the program status word. The program counter (denoted `PC`) is used to sequence the computer—it indicates which instruction in memory to execute next.

The program status word (denoted `PSW`) is used to keep track of the status of the last ALU operation, whether or not interrupts are enabled, and the privilege level of the CPU. Table 5.2 shows the meaning of the 6 bits in the program status word.

AVM–1 shares a register, `IVEC`, with the interrupt controller. This register contains the interrupt vector and is read–only as far as the CPU is concerned.

5.1.1.3 The Instruction Set.

The instruction set contains 30 instructions. The opcode space has room for 64; the upper half of the opcode space is reserved for future co–processors. As mentioned above, the instruction set is based on a load–store architecture, meaning that most instructions are not allowed to access memory for their operands.

The Instruction Format. The instruction formats are simple and regular. Figure 5.1 shows the four instruction formats. All of the formats use the same opcode field.

In formats 1 and 2, the instruction is divided into four fields. The top 6 bits (31–26) give the opcode of the instructions. The next 5 bits (25–21) denote the destination register in most operations. The third field (bits 20–16) selects the register used as the A operand in most operations. In format 1, the fourth field is comprised of bits 15–11 and is used to select the register used as the B operand. In format 2, the fourth field uses all of the 16 remaining bits to form an immediate number (0 to $(2^{16} - 1)$).

Format 1:

31	25	20	15	10 0
opcode	dest	A	B	unused

Format 2:

31	25	20	15 0
opcode	dest	A	immediate

Format 3:

31	25	20 0
opcode	dest	unused

Format 4:

31	25 0
opcode	unused

Figure 5.1: The instruction formats in *AVM–1*.

Format 3 is identical to formats 1 and 2 except that only the opcode and destination fields are used. Format 4 uses only the opcode field.

There is a trade off between instruction format complexity and verification effort, so in general the instruction format should be kept as simple as possible. A regular instruction format, while not essential to verification, can greatly reduce the amount of detail that has to be dealt with in the proof.

Instruction Set Semantics. The instruction format is essentially an instruction's syntax. Of course, syntax alone is not enough; we must also specify what each instruction means. There are many ways of specifying the semantics of CPU instructions; this dissertation will use two of them. In this section, we give a register transfer language description of the instructions in *AVM–1*. In Section 5.2.6 we give a formal description of the semantics of a sample of the instructions; the complete description can be found in [Win90b] .

The 30 programming level instructions are shown in Table 5.3. There is a group of eight, 3–argument arithmetic instructions and another group of 8 arithmetic instructions that use a 16–bit immediate value. There are 4 instructions for loading and storing registers. In addition, there are instructions for performing user interrupts, jumps, subroutine calls, and shifts.

Table 5.3: The *AVM–1* instruction set.

Mnemonic	*Format*	*Effect*
JMP	2	Jump to new location on condition flags
CALL	2	Call subroutine
INT	2	User interrupt
RTI	4	Return from interrupt
GPSW	3	Get program status word
PPSW	3	Put program status word
LD	1	Load register
ST	1	Store register
LSL	1	Logical shift left
LSR	1	Logical shift right
ASR	1	Arithmetic shift right
RTN	3	Return from subroutine
LDI	2	Load register using immediate value
STI	2	Store register using immediate value
ADD	1	Add
ADDC	1	Add with carry
SUB	1	Subtract
SUBC	1	Subtract with borrow (carry)
BAND	1	Bit–wise conjunction
BOR	1	Bit–wise disjunction
BXOR	1	Bit–wise exclusive disjunction
BNOT	1	Bit–wise negation
ADDI	2	Add using immediate value
ADDCI	2	Add with carry using immediate value
SUBI	2	Subtract using immediate value
SUBCI	2	Subtract with borrow using immediate value
BANDI	2	Bit–wise conjunction using immediate value
BORI	2	Bit–wise disjunction using immediate value
BXORI	2	Bit–wise exclusive disjunction using immediate value
NOOP	4	No operation

Table 5.4: Jump codes for the JMP instruction.

Code	*Meaning*
0	carry
1	no carry
2	overflow
3	no overflow
4	negative
5	positive
6	equal
7	not equal
8	lower or same (unsigned)
9	higher (unsigned)
10	less than (signed)
11	greater or equal (signed)
12	greater than (signed)
13	greater or equal (signed)
14	unconditional
15	unconditional

The remainder of this section provides detailed descriptions of *AVM–1*'s instruction set. The instructions are specified in our register transfer language and described where appropriate. The RTL specification only describes the part of the state that changes; state that is unaffected by the instruction is ignored. In the descriptions, `a` is the value of the A source field in the instruction, `b` is the value of the B source field, `d` is the value of the destination field, and `imm` is the immediate field value.

JMP — jump. The `JMP` instruction jumps on one of 15 different conditions according to the value returned from the function `jc`. The destination field, `d`, is used as an argument to `jc` to select one of the jump conditions listed in Table 5.4. If the result is true, the sum of `R[a]` and `imm` is loaded into the program counter. Otherwise, the program counter is incremented.

```
jc(d) →  (PC ⇐ R[a] + imm) | (PC ⇐ PC + 1)
```

CALL — call a subroutine. The program counter is loaded with the sum of `R[a]` and `imm`. The old value of the program counter is saved on a stack in memory. The destination field points to the stack pointer.

```
PC ⇐ R[a] + imm,
R[d] ⇐ R[d] + 1,
M[R[d]] ⇐ PC + 1
```

Note that the operations in the above RTL description all happen in parallel (denoted by the comma). Thus `M[R[d]]` refers to the memory value at the location pointed to by the *original* value of `R[d]`.

RTN — return from a subroutine. The top of the stack pointed to by register `R[d]` is popped and loaded into the program counter.

$$\texttt{PC} \Leftarrow \texttt{M[R[d] - 1]},$$
$$\texttt{R[d]} \Leftarrow \texttt{R[d] - 1}$$

INT — user interrupt. The `INT` instruction jumps to the location given in the 8 least significant bits of `imm` and stores the old program counter on the supervisor stack pointer. Interrupts are disabled and the CPU goes into supervisory mode.

$$\texttt{PC} \Leftarrow \texttt{imm} \wedge \texttt{255}$$
$$\texttt{R[ssp]} \Leftarrow \texttt{R[ssp] + 1},$$
$$\texttt{M[R[ssp]]} \Leftarrow \texttt{PC + 1},$$
$$\texttt{PSW}_4 \Leftarrow \texttt{false},$$
$$\texttt{PSW}_5 \Leftarrow \texttt{true}$$

RTI — return from interrupt. The program counter gets the value on top of the supervisor stack, the value is popped from the top of the stack, interrupts are enabled, and the CPU leaves supervisory mode.

$$\texttt{PC} \Leftarrow \texttt{M[R[ssp] - 1]},$$
$$\texttt{R[ssp]} \Leftarrow \texttt{R[ssp] - 1},$$
$$\texttt{PSW}_4 \Leftarrow \texttt{true},$$
$$\texttt{PSW}_5 \Leftarrow \texttt{false}$$

GPSW — get program status word. The program status word is stored in the register selected by the destination field, `R[d]`.

$$\texttt{R[d]} \Leftarrow \texttt{psw},$$
$$\texttt{PC} \Leftarrow \texttt{PC + 1}$$

PPSW — put program status word. The register selected by the destination field, `R[d]`, is moved to the program status word if the CPU is in supervisory mode.

$$\texttt{PSW}_5 \rightarrow (\texttt{psw} \Leftarrow \texttt{R[d]}),$$
$$\texttt{PC} \Leftarrow \texttt{PC + 1}$$

LD — load from memory. Register `R[d]` is loaded with the contents of memory at the location given by the sum of registers `R[a]` and `R[b]`.

$$\texttt{R[d]} \Leftarrow \texttt{M[R[a] + R[b]]},$$
$$\texttt{PC} \Leftarrow \texttt{PC + 1}$$

LDI — load from memory using immediate value. `LDI` operates exactly like `LD` except that the address is given by the sum of `R[a]` and `imm`.

$$\texttt{R[d]} \Leftarrow \texttt{M[R[a] + imm]},$$
$$\texttt{PC} \Leftarrow \texttt{PC + 1}$$

ST — store to memory. The contents of the destination register, `R[d]`, are stored in memory at the address given by the sum of registers `R[a]` and `R[b]`.

$$\texttt{M[R[a] + R[b]]} \Leftarrow \texttt{R[d]},$$
$$\texttt{PC} \Leftarrow \texttt{PC + 1}$$

STI — store to memory using immediate value. `STI` operates exactly like `ST` except that the address is given by the sum of `R[a]` and `imm`.

$$\texttt{M[R[a] + imm]} \Leftarrow \texttt{R[d]},$$
$$\texttt{PC} \Leftarrow \texttt{PC + 1}$$

LSL — logical shift left. The destination register, `R[d]`, gets the contents of `R[a]` shifted left one position. The carry field of the program status word gets the value of the bit that was shifted out.

$$\texttt{R[d]} \Leftarrow \texttt{shl R[a]},$$
$$\texttt{PSW}_1 \Leftarrow \texttt{msb R[a]},$$
$$\texttt{PC} \Leftarrow \texttt{PC + 1}$$

LSR — logical shift right. The destination register, `R[d]`, gets the contents of `R[a]` shifted right one position. The carry field of the program status word gets the value of the bit that was shifted out.

$$\texttt{R[d]} \Leftarrow \texttt{shr R[a]},$$
$$\texttt{PSW}_1 \Leftarrow \texttt{lsb R[a]},$$
$$\texttt{PC} \Leftarrow \texttt{PC + 1}$$

ASR — arithmetic shift right. The destination register, `R[d]`, gets the contents of `R[a]` shifted right arithmetically one position. That is, the most significant bit is retained in its position during the shift. The carry field of the program status word gets the value of the bit that was shifted out.

$$\texttt{R[d]} \Leftarrow \texttt{asr R[a]},$$
$$\texttt{PSW}_1 \Leftarrow \texttt{lsb R[a]},$$
$$\texttt{PC} \Leftarrow \texttt{PC + 1}$$

NOOP — no operation. No state changes take place except that the program counter is incremented.

$$\texttt{PC} \Leftarrow \texttt{PC + 1}$$

ADD — add. The destination register, `R[d]`, gets the sum of the `R[a]` and `R[b]` registers. The program status word is updated with the status from the ALU.

$$\texttt{R[d]} \Leftarrow \texttt{R[a] + R[b]},$$
$$\texttt{PSW(0-3)} \Leftarrow \texttt{status},$$
$$\texttt{PC} \Leftarrow \texttt{PC + 1}$$

ADDI — add immediate. The result is identical to that of the `ADD` instruction except that the value of the immediate field is used instead of `R[b]`.

$$\texttt{R[d]} \Leftarrow \texttt{R[a] + imm},$$
$$\texttt{PSW(0-3)} \Leftarrow \texttt{status},$$
$$\texttt{PC} \Leftarrow \texttt{PC + 1}$$

ADDC — add with carry. The destination register, `R[d]`, gets the sum of the `R[a]` and `R[b]` registers plus the value of the carry bit in the program status word. The program status word is updated with the status from the ALU.

$$\texttt{R[d]} \Leftarrow \texttt{R[a] + R[b] + PSW}_1,$$
$$\texttt{PSW(0-3)} \Leftarrow \texttt{status},$$
$$\texttt{PC} \Leftarrow \texttt{PC + 1}$$

ADDCI — add immediate with carry. The result is identical to that of the `ADDC` instruction except that the immediate field is used instead of the `R[b]` register.

$$\texttt{R[d]} \Leftarrow \texttt{R[a] + imm + PSW}_1,$$
$$\texttt{PSW(0-3)} \Leftarrow \texttt{status},$$
$$\texttt{PC} \Leftarrow \texttt{PC + 1}$$

SUB — subtract. The destination register, `R[d]`, gets the value produced by subtracting `R[b]` from `R[a]`. The program status word is updated with the status from the ALU.

$$\texttt{R[d]} \Leftarrow \texttt{R[a] - R[b]},$$
$$\texttt{PSW(0-3)} \Leftarrow \texttt{status},$$
$$\texttt{PC} \Leftarrow \texttt{PC + 1}$$

SUBI — subtract immediate. The result is identical to that of the `SUB` instruction except that the immediate field is used instead of the `R[b]` register.

$$\texttt{R[d]} \Leftarrow \texttt{R[a] - imm},$$
$$\texttt{PSW(0-3)} \Leftarrow \texttt{status},$$
$$\texttt{PC} \Leftarrow \texttt{PC + 1}$$

SUBC — subtract with borrow (carry). The destination register, `R[d]`, gets the value produced by subtracting the contents of the `R[b]` register and the value of the carry bit from the contents of the `R[a]` register. The program status word is updated with the status from the ALU.

$$\texttt{R[d]} \Leftarrow \texttt{R[a] - R[b] - PSW}_1,$$
$$\texttt{PSW(0-3)} \Leftarrow \texttt{status},$$
$$\texttt{PC} \Leftarrow \texttt{PC + 1}$$

SUBCI — subtract immediate with borrow. The result is identical to that of the `SUBC` instruction except that the immediate field is used instead of the contents of the `R[b]` register.

$$\texttt{R[d]} \Leftarrow \texttt{R[a] - imm - PSW}_1,$$
$$\texttt{PSW(0-3)} \Leftarrow \texttt{status},$$
$$\texttt{PC} \Leftarrow \texttt{PC + 1}$$

BAND — bit–wise conjunction. The destination register, `R[d]`, gets the value produced by taking the bit–wise conjunction of the contents of the `R[a]` register with the contents of the `R[b]` register. The negative and zero flags in the program status word are updated with the status from the ALU.

$$\texttt{R[d]} \Leftarrow \texttt{R[a]} \wedge \texttt{R[b]},$$
$$\texttt{PSW(0,2)} \Leftarrow \texttt{status},$$
$$\texttt{PC} \Leftarrow \texttt{PC + 1}$$

BANDI — bit–wise conjunction with immediate. The result is identical to that of the BAND instruction except that the immediate field is used instead of the contents of the R[b] register.

$$\texttt{R[d]} \Leftarrow \texttt{R[a]} \wedge \texttt{imm},$$
$$\texttt{PSW(0,2)} \Leftarrow \texttt{status},$$
$$\texttt{PC} \Leftarrow \texttt{PC + 1}$$

BOR — bit–wise disjunction. The destination register, R[d], gets the value produced by taking the bit–wise disjunction of the contents of the R[a] register with the contents of the R[b] register. The negative and zero flags in the program status word are updated with the status from the ALU.

$$\texttt{R[d]} \Leftarrow \texttt{R[a]} \vee \texttt{R[b]},$$
$$\texttt{PSW(0,2)} \Leftarrow \texttt{status},$$
$$\texttt{PC} \Leftarrow \texttt{PC + 1}$$

BORI — bit–wise disjunction with immediate. The result is identical to that of the BOR instruction except that the immediate field is used instead of the contents of the R[b] register.

$$\texttt{R[d]} \Leftarrow \texttt{R[a]} \vee \texttt{imm},$$
$$\texttt{PSW(0,2)} \Leftarrow \texttt{status},$$
$$\texttt{PC} \Leftarrow \texttt{PC + 1}$$

BXOR — bit–wise exclusive disjunction. The destination register, R[d], gets the value produced by taking the bit–wise exclusive disjunction of the contents of the R[a] register with the contents of the R[b] register. The negative and zero flags in the program status word are updated with the status from the ALU.

$$\texttt{R[d]} \Leftarrow \texttt{R[a]} \oplus \texttt{R[b]},$$
$$\texttt{PSW(0,2)} \Leftarrow \texttt{status},$$
$$\texttt{PC} \Leftarrow \texttt{PC + 1}$$

BXORI — bit–wise exclusive disjunction with immediate. The result is identical to that of the BXOR instruction except that the immediate field is used instead of the contents of the R[b] register.

$$\texttt{R[d]} \Leftarrow \texttt{R[a]} \oplus \texttt{imm},$$
$$\texttt{PSW(0,2)} \Leftarrow \texttt{status},$$
$$\texttt{PC} \Leftarrow \texttt{PC + 1}$$

Table 5.5: Synthesizing addressing modes using *AVM–1*'s load and store instructions.

Mode	*HLL Usage*	*Synthesizing in AVM–1*
Direct	Global Scalar	M[R[a] + imm]
Indirect	Pointer Dereferencing	M[R[A] + R[0]]
Indexed	Record Field	M[R[a] + imm]
Indexed	Array Element	M[R[a] + R[b]]

BNOT — bit–wise conjunction. The destination register, `R[d]`, gets the value produced by taking the bit–wise negation of the contents of the `R[a]` register. The negative and zero flags in the program status word are updated with the status from the ALU.

```
R[d] ⇐ ¬ R[a],
PSW(0,2) ⇐ status,
PC ⇐ PC + 1
```

Synthesizing Addressing Modes. Besides the `CALL` and `INT` instructions which must access a stack, only the load and store instructions can access memory. All of the other instructions only operate on the internal registers. This makes the implementation of the instruction set easier and results in faster operation of most of the instructions.

The addresses for the load and store instructions are calculated using the sum of two numbers: a register and either a register or an immediate value. This is a flexible scheme which allows most popular addressing modes to be synthesized.

Table 5.5 (adapted from [Kat85]) shows how the memory addressing scheme in *AVM–1* can be used to support common constructs in modern high–level languages.

- In direct mode, the A register holds the base of the data segment and the immediate value allows addressing within $\pm 2^{15}$ of the base.
- In indirect mode, the A register holds the value of the pointer. `R[0]` holds the constant 0.
- To perform memory operations on records, the A register holds the base address of the record and the immediate field is used to hold the field offsets into the record.
- Array operations are performed by using the A register to hold the base address of the array and the B register hold the index.

Table 5.6: Synthesizing instructions using *AVM–1*'s instruction set.

Instructions	*Synthesizing in AVM–1*
Move s to d	ADD R[d] R[s] R[0]
Clear d	ADD R[d] R[0] R[0]
Set bit x in s	BORI R[s] R[s] $2^{(x+1)}$
Clear bit x in s	BANDI R[s] R[s] $2^{16} - 2^{(x+1)}$
Test s	ADD R[0] R[s] R[0]
Increment s	ADDI R[s] R[s] 1
Decrement s	SUBI R[s] R[s] 1
Complement s	SUB R[s] R[0] R[s]

Synthesizing Other Instructions. Even though the instruction set of *AVM–1* is quite simple, many common instructions can be synthesized using only one instruction. For example, a move instruction can be synthesized by adding the register to be moved to `R[0]` which always contains 0. Table 5.6 shows the implementation of this and other instructions.

The idea behind the simple instruction set of *AVM–1* is to implement the operations that are used frequently in hardware and synthesize operations that are used less frequently by composing simple operations. For example, the memory addressing scheme allows the implementation of the common stack operations in just a few primitive instructions. Another example is clearing or setting a bit in the program status word. This takes at least three operations and a temporary register. This is acceptable, however, since toggling program status word bits is an operation that occurs much less frequently than the operations that are built into the instruction set.

5.1.1.4 Selecting Instructions.

We select instructions in the instruction set using the opcode portion of the word in memory pointed to by the current value of the program counter. We will only use the 5 least significant bits of the opcode field, allowing 32 instructions.

Table 5.7 gives a breakdown of the opcodes for *AVM–1*. The instruction set is divided into four groups depending on the value of the first 2 bits in the opcode. The first two groups contain miscellaneous instructions, the third group contains ALU operations and the fourth group contains the immediate version of the instructions in group 3.

Table 5.7: Opcode breakdowns for *AVM–1*'s instruction set.

	00XXX	01XXX	10XXX	11XXX
000	JMP	LSL	ADD	ADDI
001	CALL	LSR	ADDC	ADDCI
010	INT	ASR	SUB	SUBI
011	RTI	RTN	SUBC	SUBCI
100	GPSW	NOOP	BAND	BANDI
101	PPSW	NOOP	BOR	BORI
110	LD	LDI	BXOR	BXORI
111	ST	STI	BNOT	NOOP

5.1.2 An Organizational View.

A computer's organization is its structure—what components are used and to what effect. An organization must define the behavior of the components and how they are connected together. Abstractly, the goal of the organization is to implement a particular architecture; but, there may be system requirements not expressed at the architectural level (such as the memory interface) that are specified and met at the organizational level.

There are many ways of describing a computer organization. Circuit diagrams, computer programs, natural language, CAD tools, and mixtures of all of these have been used. This section will describe the implementation of *AVM–1* using circuit diagrams, pictures, and natural language descriptions.

The implementation of *AVM–1* can be divided into two major parts: the datapath and the control unit. We will discuss each of these.

5.1.2.1 The *AVM–1* Datapath.

The *AVM–1* datapath is loosely based on the AMD 2903 bit-sliced datapath [Adv83] and is shown in Figure 5.2. The signals shown at the right-hand side of the figure connect to the control unit. The signals on the left go to or come from the environment. Note that none of the clocking signals are shown.

The datapath has three buses, a register file containing 32 registers, and numerous support registers and latches. Two buses, `A` and `B`, are connected to the output ports on the register file and system registers. The `C` bus is connected to the input port on the register file and the system registers. In addition, the interrupt vector register is attached to the `B` bus through a special port to the interrupt controller.

The `A` and `B` buses feed the inputs to the ALU through two latches. The memory buffer register can also serve as the `A` input to the ALU through a multiplexor on the ALU input. The ALU performs simple arithmetic and boolean operations on the

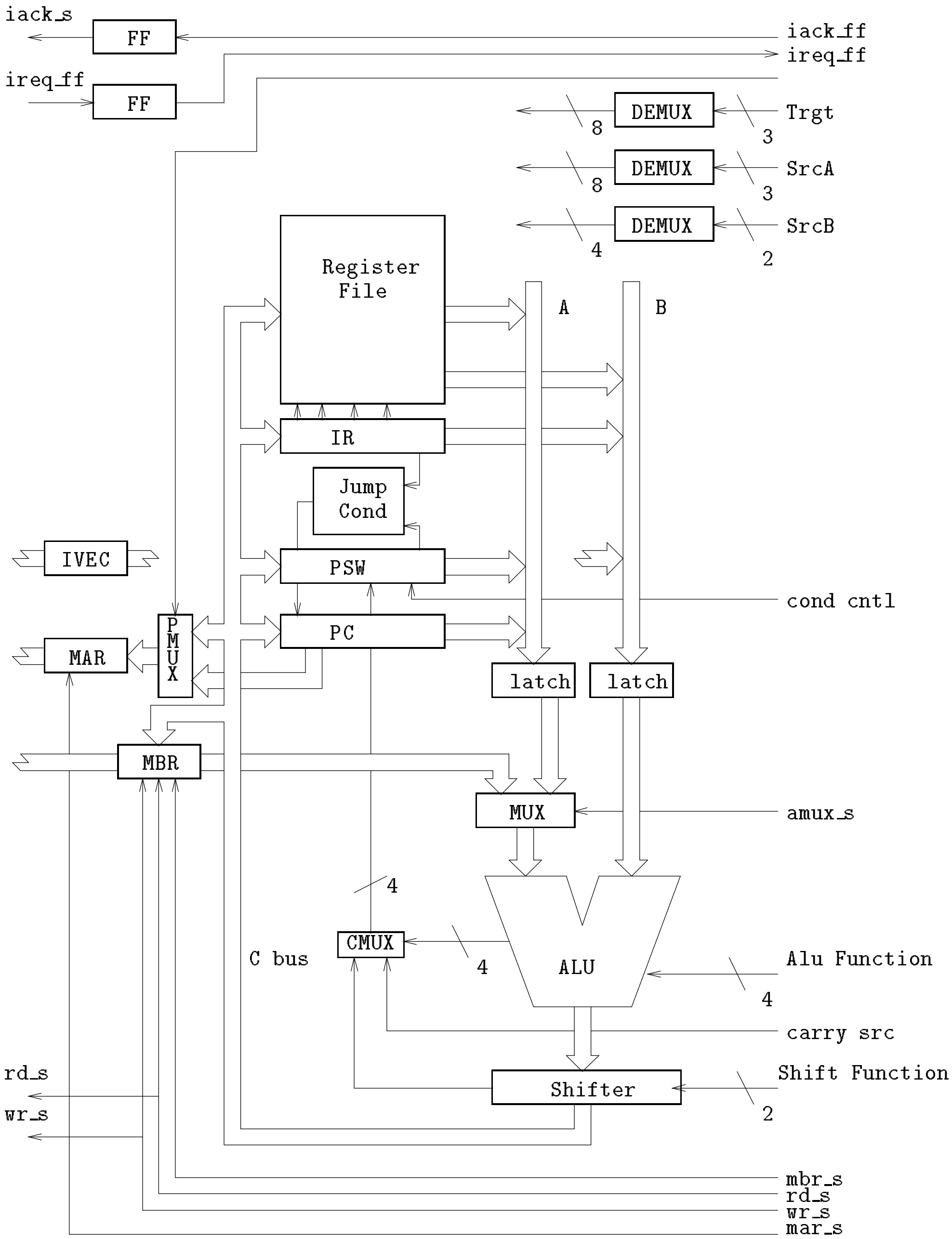

Figure 5.2: The *AVM–1* Datapath

Table 5.8: Implementation of the jump codes for the `JMP` instruction. `cf` is the carry flag in the PSW, `zf` is the zero flag, etc.

Code	*Implementation*
0	cf
1	¬cf
2	vf
3	¬vf
4	nf
5	¬nf
6	zf
7	¬zf
8	(¬cf ∨zf)
9	¬(¬cf ∨zf)
10	(nf xor vf)
11	¬(nf ⊕vf)
12	¬((nf ⊕vf) ∨zf)
13	((nf ⊕vf) ∨zf)
14	true
15	true

values on its `A` and `B` inputs. The results of the ALU operation are fed to the shifter which can perform logical and arithmetic shifts. The result from the shifter is put onto the `C` bus for distribution.

In addition to a result, the ALU produces a set of status bits (negative, zero, carry, and overflow) which can be saved in the program status word directly. A one–bit multiplexor also allows the bit shifted out of the shifter to be saved in the carry field of the PSW. The control lines to the PSW allow the supervisor and interrupt enable bits to be set and cleared and each of the status bits to be loaded individually.

The status from the PSW and the destination field of the instruction register are fed into the jump code circuitry. This combinatorial circuit calculates the jump conditions shown in Table 5.8 and supplies a boolean result which is used to determine if the program counter should be loaded from the `C` bus. The program counter can also be loaded unconditionally.

The instruction register can be loaded from the `C` bus, but only the immediate portion of the instruction register can be placed on the `B` bus.

The memory address register can be loaded directly from the program counter or from the `C` bus. This allows the MAR to be loaded quickly for instruction fetches while still allowing calculated addresses for loads and stores.

The datapath has two flipflops for holding the status of interrupt actions and three demultiplexors for decoding register selection signals from the control unit.

5.1.2.2 The Control Unit.

The control unit for *AVM–1* is shown in Figure 5.3. The control unit has four major blocks: the microprogram counter, the microinstruction register, the clock, and the microrom.

The microprogram counter is the most complex of the four. The purpose of the microprogram counter is to compute the next address for the microprogram based on the current system state. The microprogram counter is fed the condition and address (`addr`) fields from the microinstruction register, the opcode from the instruction register, and the supervisory and interrupt enable bits from the program status word. There are 5 jump conditions:

1. No jump; the microprogram counter is incremented. This is the default operation.
2. Jump to `addr` unconditionally
3. Jump to the location given by the `opcode` signal and an offset (4 in this case). This allows us to use a table lookup approach to instruction decoding in the microcode. We only use the 5 least significant bits of the 6–bit opcode; the top half of the instruction set is reserved for a coprocessor.
4. Jump to `addr` if the interrupt signal is true and interrupts are enabled.
5. Jump to `addr` if the supervisory mode signal is true.

The microinstruction register is a 40–bit register that holds the current microinstruction. The only special feature of the register is that each of the fields from the microinstruction are available through separate ports for use elsewhere in the control unit and datapath.

The microinstruction format is shown in Table 5.9. A microinstruction consists of 40 bits in 24 fields. The fields in a microinstruction can be broken into 4 groups: those affecting the operation of the microprocessor, those affecting the program status word, those dealing with external signals, and those that are used for microinstruction sequencing.

The operational group consists of the following fields:

- **AMUX** – If set, the A–latch (feeding the ALU) is loaded from the memory buffer register, otherwise the A–latch is loaded from the A–bus.
- **SHFT** – This field is passed unchanged to the shifter where it is used to select the shifter operation.

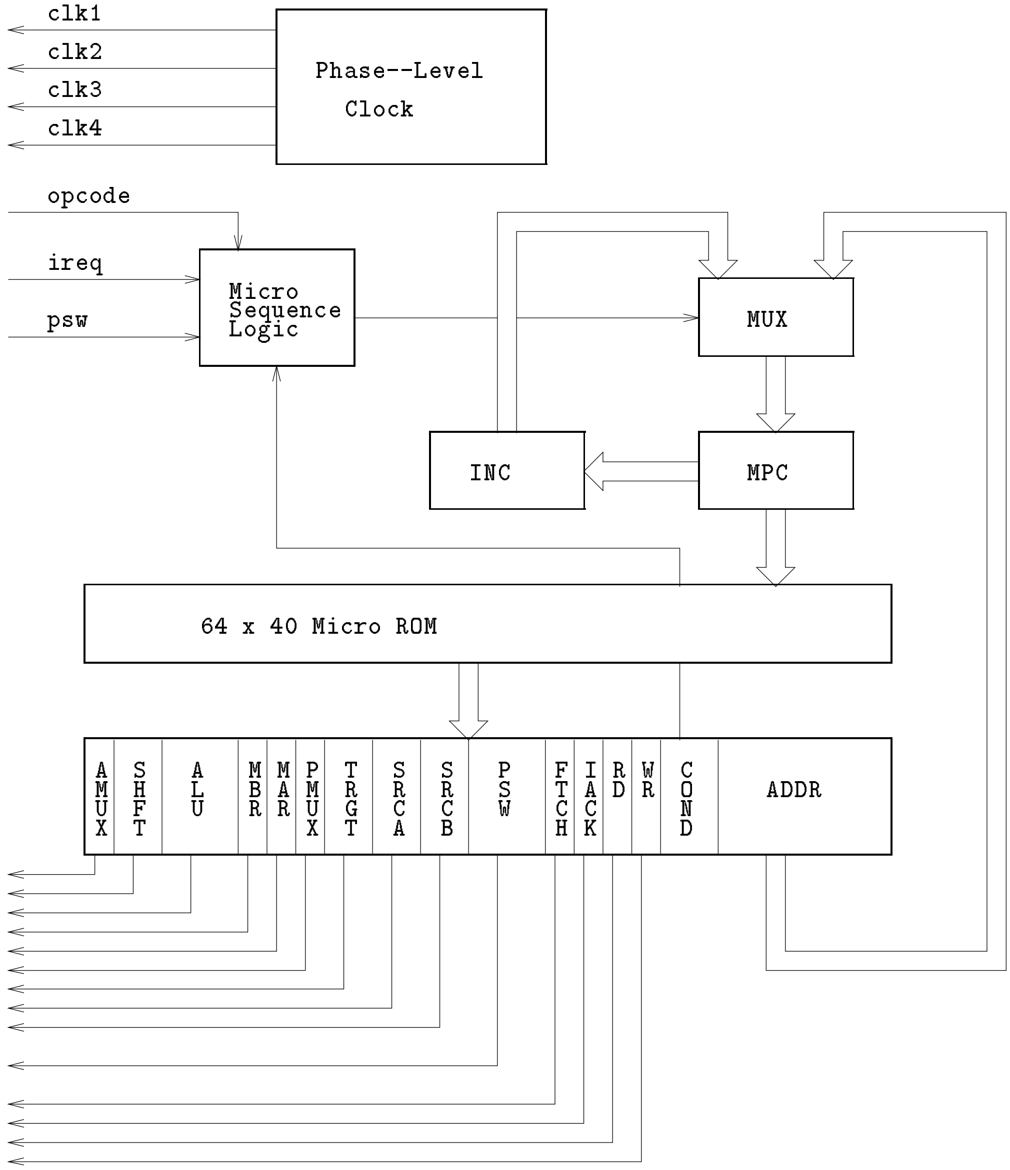

Figure 5.3: The *AVM–1* Control Unit

Table 5.9: The microinstruction format for *AVM–1*.

Operation Group

Bits	**Mnemonic**	**Description**
1	`AMUX`	Toggle MUX on A-bus
2	`SHFT`	Shifter function
4	`ALU`	ALU function
1	`MAR`	Load MAR from P-Mux
1	`MBR`	Load MBR from C-bus
1	`PMUX`	Toggle MUX loading MAR
3	`SRCA`	A-bus source
2	`SRCB`	B-bus source
3	`TRGT`	C-bus target

Program Status Word Group

Bits	**Mnemonic**	**Description**
1	`S_SM`	Set supervisory mode bit in PSW
1	`C_SM`	Clear supervisory mode bit in PSW
1	`S_IE`	Set interrupt enable bit in PSW
1	`C_IE`	Clear interrupt enable bit in PSW
1	`LD_C`	Load carry bit in PSW
1	`LD_V`	Load overflow bit in PSW
1	`LD_N`	Load negative bit in PSW
1	`LD_Z`	Load zero bit in PSW
1	`CSRC`	Source of carry (shifter or alu)

External Signals Group

Bits	**Mnemonic**	**Description**
1	`IACK`	Interrupt acknowledge signal
1	`FTCH`	Fetch signal
1	`RD`	Read signal
1	`WR`	Write signal

Microprogram Counter Group

Bits	**Mnemonic**	**Description**
3	`COND`	Microcode jump condition
6	`ADDR`	Next address

- **ALU** – This field is passed unchanged to the ALU where it is used to select the ALU operation.
- **MAR** – If high, the MAR is loaded with the value on the output port of the PMUX.
- **MBR** – If high, the MBR is loaded from the C-Bus
- **PMUX** – Determines the value of the PMUX output. If high, the output is equal to the value in the program counter, otherwise the output is equal to the value on the C-Bus.
- **SRCA** – Determines the source of the value on the A-Bus.
- **SRCB** – Determines the source of the value on the B-Bus.
- **TRGT** – Selects a register in which to store the value on the C-Bus.

The program status word group consists of the following fields:

- **S_SM** – When high, the supervisory mode bit in the PSW is set.
- **C_SM** – When high, the supervisory mode bit in the PSW is cleared.
- **S_IE** – When high, the interrupt enable bit in the PSW is set.
- **C_IE** – When high, the interrupt enable bit in the PSW is cleared.
- **LD_C** – When high, the carry bit in the PSW is loaded from the carry–bit input port.
- **LD_V** – When high, the overflow bit in the PSW is loaded from the overflow–bit input port.
- **LD_N** – When high, the negative bit in the PSW is loaded from the negative–bit input port.
- **LD_Z** – When high, the zero bit in the PSW is loaded from the zero–bit input port.
- **CSRC** – The ALU and shifter both produce a carry out. This bit controls a multiplexor that selects which of these carry signals is fed to the carry-bit input port on the PSW.

The external signals group consists of the following fields:

- **IACK** – This value is passed to the interrupt acknowledge flipflop to control the external interrupt acknowledge signal.

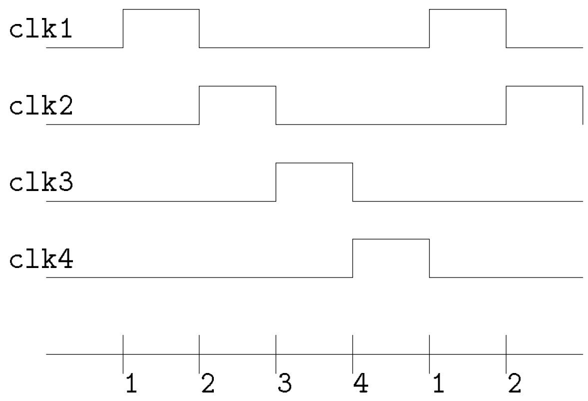

Figure 5.4: The clock signals in *AVM–1*.

- **FTCH** – Passed to the environment to inform external devices that the CPU is in fetch mode.
- **RD** – Used to control loading of the MAR and MBR. It is also passed to the environment to control reading from memory and other devices.
- **WR** – Used to control loading of the MAR and MBR. It is also passed to the environment to control writing to memory and other devices.

The microprogram counter group consists of the following fields:

- **COND** – Selects one of 8 possible jump conditions for the microprogram counter. Every microinstruction is a potential control point in the microprogram. Sequencing is done explicitly.
- **ADDR** – The next address for the microprogram counter. This may or may not be used depending on the value of the `cond` field.

The clock is a simple four–phase counter with a strobe line for each phase. Figure 5.4 shows the output timing for the clock. The `clk1` line, for example, is only true during phase 1, the `clk2` line is true during phase 2, and so on.

The microrom holds the microcode and is made from a read–only memory that is 40–bits wide and 64 words long.

5.1.2.3 Timing.

The timing of *AVM–1* is based on a four phase clock (see Figure 5.5). During the four phases, the machine performs the following state transitions:

1. In phase 1, the microinstruction register is loaded from the microrom.

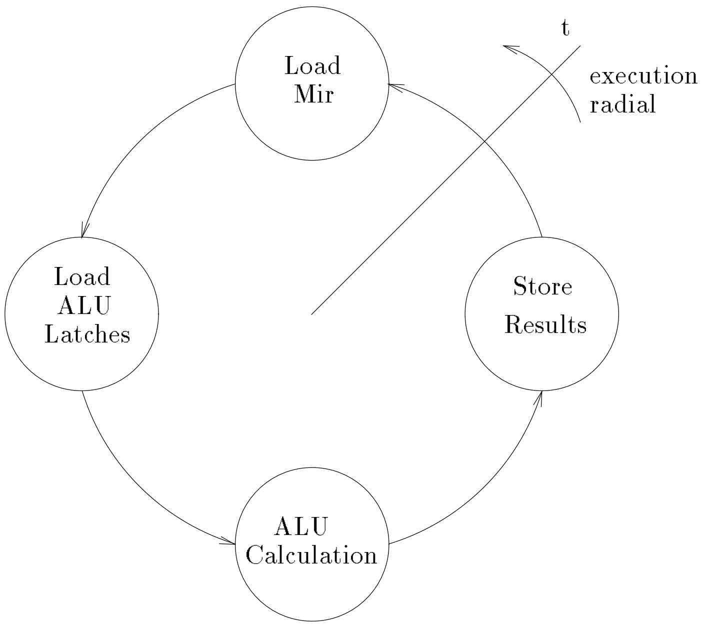

Figure 5.5: A PERT phase diagram for *AVM–1*.

2. In phase 2, the latches feeding ALU are loaded from the register file and system registers.
3. In phase 3, the results from the ALU and shifter are calculated. In addition, the MAR can be loaded from the PC in this phase.
4. In phase 4, the result calculated in phase 3 is stored back into the register file and system registers.

Every microinstruction is executed by this phase sequence.

Since microinstructions are used to implement the macroinstructions, the timing for a macroinstruction is dependent on the number of microinstruction in its implementation. In most cases this number is 4.

5.1.3 Comparisons.

Table 5.10 compares the design of *AVM–1* to the designs of the four microprocessors discussed in Chapter 2. The table, like all such tabulations, cannot hope to capture all of the important characteristics of the microprocessors, but the data presented does provide some basis for judging relative complexities.

Table 5.10: Comparison of verified microprocessors and *AVM–1*.

	AVM–1	Tamarack–3	FM8501	Viper	SECD
User Registers	31	2	8	4	4
Instructions	30	8	26	20	21
Microcoded	yes	yes	yes	no	yes
Microstore size	64 words	32 words	16 words	N/A	512 words
Interrupts	yes	yes	no	no	no
Supervisory Mode	yes	no	no	no	no
Memory Model	sync	async	async	sync	sync
Word Width	32–bit	16-bit	16-bit	32-bit	32-bit
Memory Size	4G	8K	64K	1M	16K

5.1.4 Observations.

Having completed the description of *AVM–1*'s architecture and organization, we have several observations.

The design of *AVM–1* is not intended to push the architectural envelope, but rather to serve as a test bed for experimenting with using generic interpreter proofs in microprocessor verification. To this end, we have tried to include interesting features (such as a privileged mode and interrupts), but have not been overly anxious about small inefficiencies.

For example, the implementation of *AVM–1* is not optimal. A good example of where the implementation could be improved is the `FETCH--ISSUE--DECODE` cycle. The only purpose of the `ISSUE` microinstruction is to transfer the contents of the memory buffer register to the instruction register. Being able to read the memory bus directly into the instruction register during the `FETCH` cycle would eliminate this step and result in nearly a 25% speed–up in the execution time since almost every instruction is implemented in 4 microinstructions. Making this modification to the design of *AVM–1* would have very little impact on the verification.

The implementation makes the assumption that memory can be read in a single machine cycle. This is not unreasonable given the speed of today's high–speed memory devices, but limits the usefulness of the chip. A more versatile approach would be to interface memory to the CPU asynchronously. Eventually, *AVM–1* will have an asynchronous memory interface so that it can be coupled with the memory management unit being designed as part of the UC Davis Verified Chip Set. In anticipation of this change to the design, the specification in the next section uses the asynchronous generic interpreter theory.

5.2 *AVM–1*'s Formal Specification.

Section 5.1 presented an informal description of *AVM–1*. This section presents the formal specification of the microprocessor at each level in the decomposition hierarchy. We begin by describing the electronic block model and then present the phase–level specification, the micro–level specification, and finally the macro–level specification.

Turning an informal description of a microprocessor into a formal specification is a difficult task. Avra Cohn, in [Coh89], describes her specification of VIPER's electronic block model from informal descriptions supplied by VIPER's designers as follows:

> ...VIPER's top–level specification and its major–state level were both supplied in a logical language; but its block–level model was given partly formally and partly pictorially (as was natural). Combining these two parts required both ingenuity and some guesswork. The guesses were based on the coincidence of line names, on the names of bound variables in the functional definitions, and on the annotations in the text of the definitions. None of these notational devices can be regarded as formal specification.

This quote not only tells of the difficulties of developing formal specifications from the kinds of informal descriptions commonly in use, but also alludes to the inadequacies of those descriptions. The formal specification of *AVM–1* was probably easier than VIPER since the designer and the specifier were the same person.

The rest of this section is organized as follows: We begin by describing the theory of abstract words that is used in the specification of *AVM–1*. Following that, we present the specifications of the electronic block model, phase–level, micro–level, and macro–level in turn. We also describe the definition of the microcode. There is a fair amount of detail and it is easy to get lost. Each of the sections describing a particular level have been further divided into important subparts.

The electronic block model specification is unique. The electronic block model is a composition of two large blocks: the datapath and the control unit. Within these two blocks are many major blocks. Each of the major blocks are described in a separate subpart of the section. Many of these can be skipped by readers not interested in the details without losing continuity.

The descriptions of the abstract interpreter levels all follow the pattern imposed by the generic theory. The generic theory requires that we make definitions for each of the abstract objects in the representation; the following abstract objects will be defined in each section: `inst_list`, `select`, `key`, `substate`, and `subenv`. We will break each chapter into parts defining each of these abstract objects. (Note that we do not have to define `Impl`, `count`, and `begin`; they are defined by the lower level in the hierarchy.)

One note about the following descriptions: this section attempts to describe the meaning of the formal specifications in English text. In all cases, the true meaning should be taken from the logic, not the English description accompanying it.

5.2.1 A Theory of Abstract Words.

The specification of any microprocessor is based upon the fundamental data type that the microprocessor is to manipulate and a set of primitive operations on that data type. Usually, the data type is a bit–vector and the primitive operations define addition, subtraction, and so on for bit–vectors. Sometimes a single specification may use more than one representation. For example, the verification of *MAC–2* [Win90a] used natural numbers as the base type in the abstract representations and a bit–vector representation in the electronic block model.

The verification of the microprocessor is orthogonal to the concrete representation of the fundamental data type. Using concrete data representations for defining the fundamental data type clutters the proof with the implementation details of the data type; these are frequently a bother to manipulate and usually irrelevant in the correctness proof.

We can solve this problem by choosing an abstract representation for the fundamental data type. Our abstract data type is called `:*wordn` and we have defined a number of abstract operations on it.

The fact that there are two abstract representations used in this dissertation might be a point for some confusion. The generic interpreter theory uses an abstract representation to specify the operations of the generic interpreter. This representation is instantiated with the definitions for the various levels in the decomposition in the course of completing the verification.

The definitions for the various levels in the design are also parameterized over the abstract representation for the fundamental data type for *AVM–1*. Thus, the correctness result for the microprocessor forms yet another generic theory. The generic theory for the microprocessor must be instantiated with a concrete representation for bit–vectors in order to arrive at the gate–level implementation of the electronic block model and complete the implementation.

The abstract theory of n-bit words defines the following abstract objects through use:

- `*wordn` – the type for n–bit words.
- `*memory` – the type for memories.
- `*address` – the type for memory addresses.
- `*reg_len` – the type for bit-vectors used to select registers.

The operations in the abstract theory form the set of primitive operations for defining the blocks in the electronic block model and specifying the actions taken by instructions at all levels. Some may object to using abstract operations for defining the behavior of the macro–level as well as electronic block model. This is really no different, however, than using the + symbol at both levels. The fact that one operation has a concrete definition and the other does not, makes no difference. In fact, the concrete definition attached to the + symbol may fool the reader of the specification into believing that the microprocessor has been proven to correctly add, when in fact, it has not. The use of abstract representations for this purpose makes it clear which operations are taken as primitive and consequently not verified.

The abstract representation for n-bit words is large and contains several sections. We will deal with each of them individually.

ALU Functions. The n–bit word theory defines the following abstract functions for defining ALU operations:

- `(‘add‘,":(*wordn × *wordn → *wordn)")` – add two n-bit words.
- `(‘addc‘,":(*wordn × *wordn × bool → *wordn")` – add two n-bit words with carry.
- `(‘addp‘,":(*wordn × *wordn × *wordn) → bool")` – predicate that uses the arguments to and result from the `add` operation to determine if carry–out has occurred.
- `(‘addcp‘,":(*wordn ×*wordn ×*wordn) → bool")` – determine if carry–out has occurred using the arguments to and result from the `addc` operation.
- `(‘aovfl‘,":(*wordn × *wordn × *wordn) → bool")` – determine if overflow has occurred using the arguments to and result from the `add` and `addc` operations.
- `(‘inc‘, ":(*wordn → *wordn)")` – increment an n–bit word.
- `(‘sub‘, ":(*wordn × *wordn → *wordn)")` – subtract two n-bit words.
- `(‘subc‘, ":(*wordn × *wordn × bool) → *wordn")` – subtract two n-bit words with carry (borrow).
- `(‘subp‘,":(*wordn × *wordn × *wordn) → bool")` – predicate that uses the arguments to and result from the `sub` and `subc` operations to determine if carry–out has occurred.
- `(‘sovfl‘,":(*wordn × *wordn × *wordn) → bool")` – determine if overflow has occurred using the arguments to and result from the `sub` and `subc` operations.

- (‘dec‘, ":(*wordn $\rightarrow$ *wordn)") – decrement an n–bit word.
- (‘band‘, ":(*wordn $\times$ *wordn $\rightarrow$ *wordn)") – perform bitwise conjunction of two n–bit words.
- (‘bxor‘, ":(*wordn $\times$ *wordn $\rightarrow$ *wordn)") – perform bitwise exclusive–disjunction of two n–bit words.
- (‘bor‘, ":(*wordn $\times$ *wordn $\rightarrow$ *wordn)") – perform bitwise disjunction of two n–bit words.
- (‘bnot‘, ":(*wordn $\rightarrow$ *wordn)") – perform bitwise negation of an n–bit word.

Test functions. In addition to the operations used to define the ALU operations, two predicates for testing whether a number is negative and whether it is zero are used in the specification.

- (‘negp‘, ":(*wordn $\rightarrow$ bool)") – is the argument negative?
- (‘zerop‘, ":(*wordn $\rightarrow$ bool)") – is the argument zero?

SHIFTER functions. The shifter has a set of primitive operations as well:

- (‘shl‘, ":(*wordn $\rightarrow$ *wordn)") – shift the argument left one bit.
- (‘shr‘, ":(*wordn $\rightarrow$ *wordn)") – shift the argument right one bit.
- (‘asr‘, ":(*wordn $\rightarrow$ *wordn)") – arithmetically shift the argument right one bit (i.e. preserve the sign bit).

Bit functions. We do not need a full range of bit manipulation functions in the specification, but we do need to select the most significant and least significant bits.

- (‘msb‘, ":(*wordn $\rightarrow$ bool)") – select the most significant bit in the argument.
- (‘lsb‘, ":(*wordn $\rightarrow$ bool)") – select the least significant bit in the argument.

Coercion functions. Coercion functions convert objects from one type to another.

- (`val`, ":(*wordn → num)") – returns the numeric value of an n-bit word.
- (`wordn`, ":(num → *wordn)") – return the n-bit word representation of number.
- (`reg_len`, ":(*reg_len → num)") – coerces a value of type :*reg_len to a number.
- (`address`, ":(*wordn → *address)") – return the address representation of an n-bit word.

The use of type *address gives the user of the abstract word representation the freedom to use only portions of a word for an address or to manipulate them in some way prior to use.

Subranging functions. Subranging functions return a portion of an n-bit word corresponding to some meaningful component. The following functions are used to implement the instruction formats in *AVM–1*.

- (`opcode`, ":(*wordn → bt6)") – return the opcode portion of an n-bit word which is represented as a boolean 6–tuple.
- (`dest`, ":(*wordn → *reg_len)") – return the portion of an n-bit word designating the destination register of an operation.
- (`srca`, ":(*wordn → *reg_len)") – return the portion of an n-bit word designating the source A register of the operation.
- (`srcb`, ":(*wordn → *reg_len)") – return the portion of an n-bit word designating the source B register of the operation.
- (`imm`, ":(*wordn → *wordn)") – return the portion of an n-bit word designating the immediate value used in the operation.

The use of type :*reg_len to describe the size of the sub–word designating registers makes the proof independent of the size of the register file. The opcode, however, is returned as a boolean 6–tuple. Making it concrete has advantages in the verification.

Constructor and selectors for the Program Status Word. The program status word is a register that keeps track of the status of the most recent ALU operation as well as recording whether or not the CPU is in supervisory mode and whether or not interrupts are enabled. The following operations represent a constructor and 6 selectors on the program status word.

- (`mk_psw`, ":(bt6 $\rightarrow$ *wordn)") – construct a new program status word.
- (`get_ie`, ":(*wordn $\rightarrow$ bool)") – select the interrupt enable bit in the program status word.
- (`get_sm`, ":(*wordn $\rightarrow$ bool)") – select the supervisory mode bit in the program status word.
- (`get_cf`, ":(*wordn $\rightarrow$ bool)") – select the carry bit in the program status word.
- (`get_vf`, ":(*wordn $\rightarrow$ bool)") – select the overflow bit in the program status word.
- (`get_zf`, ":(*wordn $\rightarrow$ bool)") – select the zero bit in the program status word.
- (`get_nf`, ":(*wordn $\rightarrow$ bool)") – select the negative bit in the program status word.

Memory functions. We need special functions for interacting with memory because it represents shared state. The CPU cannot assume that it is the only device that changes memory. The `fetch` and `store` operation are fairly self–explanatory. The use of the abstract transformation functions is described in Section 3.4.

- (`fetch`,":(*memory $\times$ *address) $\rightarrow$ *wordn") – retrieve a word from memory at a particular address.
- (`store`,":(*memory $\times$ *address $\times$ *wordn)$\rightarrow$*memory") – store a word to memory at a particular address.
- (`trans`,":*memory $\rightarrow$ *memory") – transform memory.

Interrupt instructions. The interrupt vector is another example of shared state. We will use the following functions to interact with the interrupt vector.

- (`int_fetch`, ":*wordn $\rightarrow$ *wordn") – fetch the interrupt vector
- (`int_trans`, ":*wordn $\rightarrow$ *wordn") – transform the interrupt vector

5.2.2 Defining the Electronic Block Model.

As we mentioned before, the electronic block model is a structural description and is modeled using existentionally linked conjunctions of predicates as described in

Section 2.4. We choose blocks, define predicates to specify their behavior, connect them together using hidden internal lines, and connect the remaining lines to the external buses.

We have some leeway in choosing the blocks. Each block will be specified using a behavioral description. We will not continue the proof below the electronic block model level in this dissertation; to completely verify the circuit making up the CPU to the gate–level, we would have to specify implementations for each of the blocks and prove that the implementation implies the behavioral specification. These proofs could be used along with the proof we give here to prove a correctness statement showing that the gate–level circuit implies the macro–level interpreter specification.

The level to which the proof should be performed is a subjective consideration. We could carry the proof to the transistor level, but there is a point where the benefits of the proof are outweighed by its difficulty. For example, we could expend effort showing that all the gates we use are correctly implemented in some transistor model, but such effort would probably be wasted since a verification is only as good as the model used in the specification. Given the current state-of-the-art in the mathematical modeling of transistors, it is probably more reasonable to assume that an `AND` gate is correctly implemented than it is to assume we have a good transistor model.

The Datapath Blocks. Some of the blocks in the datapath are fairly small (a flip–flop for instance) and others are fairly large (the ALU is specified as a single block). Still, we believe that our block model is a good compromise between circuit detail and proof effort. Much more detail in the electronic block model would have made the verification of the phase–level even more difficult. Much less detail would have made the verification of the phase–level trivial. As a general rule of thumb, we have tried to keep our blocks simple enough that there would be little doubt that a device could be made which satisfies the specification.

Simple Blocks. We begin by defining some simple blocks.

```
⊢def GND out = (out = F)

⊢def MUX_SPEC ctl a b c = (c = (ctl → a | b))

⊢def MUX_1_SPEC ctl a b c = (c = (ctl → a | b))

⊢def LATCH_SPEC i ld out =
       ∀  t:time .  out(t+1) = ld t  → i t | out t

⊢def FF_SPEC i ld q =
       ∀  t:num .  q(t+1) = ((ld t)  → i | (q t))

⊢def REG_SPEC i ld prt out contents =
       ∀  t:time .
       (contents (t+1) = ld t  → i t | contents t) ∧
       (prt t ⟹ (out = contents))

⊢def C255_SPEC rep prt out =
        prt ⟹ (out = (wordn rep 255))
```

`GND` is the ground line. Its output is always false. `MUX_SPEC` is a simple n–bit, two–to–one multiplexor. `MUX_1_SPEC` is a 1–bit multiplexor; it is identical to `MUX_SPEC` except for its type (which is not shown).

`FF_SPEC` and `LATCH_SPEC` specify a flip–flop and a latch respectively. The only difference between the two specifications is that `FF_SPEC` operates on a single bit, while `REG_SPEC` operates on n–bit words.

`REG_SPEC` specifies a register. For our purposes, the difference between a register and a latch is that a register has a tri–stated output port (controlled by the signal `prt`).

`C255_SPEC` specifies a hard–wired constant that is tri–stated to the port `out`. In this case, the constant is 255. The function `wordn` is from the abstract word package that was just discussed. The abstract functions must be applied to an abstract representation, so `wordn rep` returns a function that coerces an integer into an n–bit word.

The Register Block. The register block is a triple–ported register file with 32 registers. (The formal specification does not actually say how many registers there are until the abstract word package has been instantiated with a concrete representation.) The basic operation in the register block is described by the function `UPDATE_REG`

```
⊢def UPDATE_REG rep psw n reg_list value =
       let sm   = (get_sm rep psw) in
       (n = zero_reg)         → reg_list |
       (IS_SUP_REG n ∧ ¬sm)  → reg_list |
                                (SET_EL n reg_list value)
```

This function is used to update the list representing the register file when certain conditions have been met. Register 0 is has a constant value of 0 which cannot be changed. Registers 1 through 7 are reserved for privileged mode; they cannot be changed unless the supervisory mode bit of the program status word is set. The function SET_EL changes the value of the n^{th} element of a list.

In general, the register file reads values on the in port and write values to the outA and outB ports. In addition to the three ports, there are two lines that control loading the register from the input port, four lines controlling the two output ports, and three switch lines of type :*reg_len that select registers in the block for various reasons. There is one distinguished register in the register file used as the stack pointer when the CPU is in supervisory mode called ssp_reg.

```
⊢def REGISTER_BLOCK rep c a b ld ld_ssp prt_A prt_D ssp prt_B
                        in outA outB psw reg_list =
  ∀ t:time .
   (reg_list (t+1) =
      (ld t) →
          (UPDATE_REG rep (psw t) (reg_len rep (c t))
                          (reg_list t) (in t)) |
      (ld_ssp t) →
          (UPDATE_REG rep (psw t) ssp_reg
                          (reg_list t) (in t)) |
          (reg_list t)) ∧
   (prt_A t ⟹ (outA t = (EL (reg_len rep (a t)) (reg_list t)))) ∧
   (prt_D t ⟹ (outA t = (EL (reg_len rep (c t)) (reg_list t)))) ∧
   (ssp t   ⟹ (outA t = (SSP_REG (reg_list t)))) ∧
   (prt_B t ⟹ (outB t = (EL (reg_len rep (b t)) (reg_list t))))
```

The register file is designated reg_list in the specification and is represented as a list. The list indexing function EL is used to select specific registers. The register block operates as follows:

- When ld is high, the register selected by the c line is updated with the value on the input port in.
- When ld_ssp is high, ssp_reg is updated with the value on the input port in.
- When prt_A is high, outA has the value of the register selected by the value on the a line.

- When `prt_D` is high, `outA` has the value of the register selected by the value on the `c` line.
- When `ssp` is high, `outA` has the value of `ssp_reg`.
- When `prt_B` is high, `outB` has the value of the register selected by the value on the `b` line.

The Instruction Register. The instruction register is similar to the register defined in `REG_SPEC`; but it has four additional ports that supply the opcode, destination, A source, and B source fields from the register.

```
⊢def IR_SPEC rep set prt in out contents
                opc_port dest_port srca_port srcb_port =
      ∀ t:time.
        (contents (t+1) = (set t) → in t | contents t) ∧
        (opc_port t  = opcode rep (contents t)) ∧
        (dest_port t = dest rep (contents t)) ∧
        (srca_port t = srca rep (contents t)) ∧
        (srcb_port t = srcb rep (contents t)) ∧
        (prt t ⟹ (out t = (imm rep (contents t))))
```

The value on the output port (when the `port` line is high) is the immediate field, not the entire instruction. There is no way to read the complete contents of the instruction register onto the bus.

The PSW Register. The register that holds the program status word (PSW) is the most complicated register specification. Each of the 6 bits used for the CPU status are individually addressable for the input and output, much as if they were 6 independent flipflops. The unit functions as a register as well, with input and output ports for reading and writing the entire for the program status word at once.

```
⊢def PSW_SPEC rep set clk prt in out ie sm contents
                 vf nf cf zf
                 s_sm c_sm s_ie c_ie ld_v ld_n ld_c ld_z =
  ∀ t:time.
      (contents (t+1) =
          ((set t) ∧ (get_sm rep (contents t))) →
             (in t) |
          (clk t) →
             (mk_psw rep (
              (s_sm t → T  |
                  c_sm t → F | (get_sm rep (contents t))),
              (s_ie t → T  |
                  c_ie t → F | (get_ie rep (contents t))),
              (ld_v t → vf | (get_vf rep (contents t))),
              (ld_n t → nf | (get_nf rep (contents t))),
              (ld_c t → cf | (get_cf rep (contents t))),
              (ld_z t → zf | (get_zf rep (contents t))))) |
             (contents t)) ∧
         (sm t = get_sm rep (contents t)) ∧
         (ie t = get_ie rep (contents t)) ∧
         (prt t ⟹ (out = contents))
```

The PSW register operates as follows:

- When the `set` line is high and the supervisory mode bit is set, then the current contents are replaced by the value on the `in` port.
- When the `clk` line is high, the new value of the PSW is constructed from the input ports for the individual fields, provided that their associated load lines are high.
- The `sm` port gets the current value of the supervisory mode bit.
- The `ie` port gets the current value of the interrupt enable mode bit.
- When the `prt` line is high, the output port holds the current contents of the register.

The Jump Circuitry. As mentioned in Section 5.1, the jump instruction in *AVM–1* uses the 4 least significant bits of the destination field to select a jump condition. Calculating jump codes could be done in the microcode, but would be extremely slow. The electronic block model contains a special block for calculating jump codes based on the current PSW and the destination field of the instruction.

```
⊢def JUMP_SPEC rep d psw out =
      ∀ t:time .
      (out t) = JUMP_COND rep (reg_len rep (d t)) (psw t)
```

The definition relies on an auxiliary definition, JUMP_COND

```
⊢def JUMP_COND rep d psw =
       let cf = (get_cf rep psw) and
           vf = (get_vf rep psw) and
           nf = (get_nf rep psw) and
           zf = (get_zf rep psw) in (
       (d =  0) → cf                   |
       (d =  1) → ~ cf                 |
       (d =  2) → vf                   |
       (d =  3) → ~ vf                 |
       (d =  4) → nf                   |
       (d =  5) → ~ nf                 |
       (d =  6) → zf                   |
       (d =  7) → ~ zf                 |
       (d =  8) → (~cf ∨ zf)           |
       (d =  9) → ~(~cf ∨ zf)          |
       (d = 10) → (nf xor vf)          |
       (d = 11) → ~(nf xor vf)         |
       (d = 12) → ~((nf xor vf) ∨ zf)  |
       (d = 13) → ((nf xor vf) ∨ zf)   |
                  T                    )
```

The meanings of the jumps codes can be found in Table 5.4.

The Memory Buffer Register. The memory buffer register has a complicated porting arrangement. The register has one bi–directional port, mem_port, a second input port, in, and a second output port, bus.

```
⊢def MBR_SPEC set clk rd_s wr_s in value bus mem_port =
      (∀ t:time.
          ((value (t+1) = (((clk t) ∧ (rd_s t)) → mem_port t |
                          ((clk t) ∧ (set t)) → in t | value t)) ∧
           (wr_s t ⟹ (mem_port = value)))) ∧
      (bus = value)
```

The specification describes three different parts of the register:

1. The new value of the register is the value on the memory port if the clock, clk, and the read line, rd_s are high. Otherwise, if the clock and the set line are high, the new value is the value of the input port. If neither of these conditions is true, then the value of the register is unchanged.

2. Then memory port carries the value of the register only if the write line, wr_s in high.

3. The value on the output bus is always the value of the register.

The Interrupt Vector Register. IVEC_SPEC describes the interrupt vector register. This register does not actually reside on the CPU, but is shared with the interrupt controller. Thus, the following specification can be thought of as a partial specification for the interrupt controller; the only part specified is the part that the CPU actually uses to read the interrupt vector.

```
⊢def IVEC_SPEC rep prt out contents =
       ∀  t:time .
        (contents (t+1) = (contents t)) ∧
        (prt t ⟹ (out t = (int_fetch rep (contents t))))
```

The Demultiplexors. The specification of the electronic block model makes use of several demultiplexors. The following specification describes a 2–to–4 demultiplexor. The specification of a 3–to–8 is similar.

```
⊢def DEMUX_2_SPEC s o0 o1 o2 o3 =
         (∀ t .  o0 t = ((s t) = (F,F))) ∧
         (∀ t .  o1 t = ((s t) = (F,T))) ∧
         (∀ t .  o2 t = ((s t) = (T,F))) ∧
         (∀ t .  o3 t = ((s t) = (T,T)))
```

The Memory Block. The operation of the memory block is based on the operation of two abstract functions: `store` and `fetch`. Memory has a single bi–directional data port, `data`, an address port, a read signal, `rd_s` and a write signal, `wr_s`.

```
⊢def MEM rep wr_s rd_s addr data mem =
     ∀ t:time .
       (mem (t+1) =
         (wr_s t → store rep (mem t, address rep (addr t), (data t))
                | mem t)) ∧
        (rd_s t ⟹
           (data t = (fetch rep (mem t, address rep (addr t)))))
```

When the write signal is high, the new value of memory is the result returned by applying `store` to the old memory, the address, and the data. Otherwise, the value of memory is unchanged. If the read signal is true, then the value of memory returned by the `fetch` function is placed on the data port.

We specify the memory as one of the blocks in the electronic block model. There are other ways of specifying memory. For example, Joyce [Joy89a] separates the memory block from the electronic block model and then uses the combination to verify the macro–level. There is, however, little difference in meaning.

```
⊢ AVM_ALU_SPEC rep switch(in_A,in_B,cin)out(neg,zero,ovfl,carry) =
    ((switch = F,F,F,F) →
     ADD_WITHOUT_CARRY rep(in_A,in_B,cin)out(neg,zero,ovfl,carry) |
    ((switch = F,F,F,T) →
     ADD_WITH_CARRY rep(in_A,in_B,cin)out(neg,zero,ovfl,carry) |
    ((switch = F,F,T,F) →
     INCREMENT rep(in_A,in_B,cin)out(neg,zero,ovfl,carry) |
    ((switch = F,F,T,T) →
     SUB_WITHOUT_CARRY rep(in_A,in_B,cin)out(neg,zero,ovfl,carry) |
    ((switch = F,T,F,F) →
     SUB_WITH_CARRY rep(in_A,in_B,cin)out(neg,zero,ovfl,carry) |
    ((switch = F,T,F,T) →
     DECREMENT rep(in_A,in_B,cin)out(neg,zero,ovfl,carry) |
    ((switch = F,T,T,F) →
     BITWISE_AND rep(in_A,in_B,cin)out(neg,zero,ovfl,carry) |
    ((switch = F,T,T,T) →
     BITWISE_XOR rep(in_A,in_B,cin)out(neg,zero,ovfl,carry) |
    ((switch = T,F,F,F) →
     BITWISE_OR rep(in_A,in_B,cin)out(neg,zero,ovfl,carry) |
    ((switch = T,F,F,T) →
     BITWISE_NOT rep(in_A,in_B,cin)out(neg,zero,ovfl,carry) |
    ((switch = T,F,T,F) →
     ALU_NOOP rep(in_A,in_B,cin)out(neg,zero,ovfl,carry) |
    ((switch = T,F,T,T) →
     ALU_NOOP rep(in_A,in_B,cin)out(neg,zero,ovfl,carry) |
    ((switch = T,T,F,F) →
     ALU_NOOP rep(in_A,in_B,cin)out(neg,zero,ovfl,carry) |
    ((switch = T,T,F,T) →
     ALU_NOOP rep(in_A,in_B,cin)out(neg,zero,ovfl,carry) |
    ((switch = T,T,T,F) →
     ALU_NOOP rep(in_A,in_B,cin)out(neg,zero,ovfl,carry) |
     ALU_NOOP rep(in_A,in_B,cin)out(neg,zero,ovfl,carry)
    )))))))))))))))
```

Figure 5.6: The ALU Specification for *AVM-1*.

The ALU Block. The ALU definition used in the specification of the electronic block model is shown in Figure 5.6. The ALU selects one of 16 functions based on the value of a 4–bit input, `switch`. Only 10 of the 16 available functions are used in our implementation. The complete specification gives the formal definition of each of the functions, including how flags are set for each operation. The ALU performs addition, with and without carry; incrementing; subtraction, with and without carry; decrementing; and bitwise disjunction, conjunction, exclusive disjunction, and negation. The 16 functions are filled out with a NOOP operation that passes the `A` input through unchanged, but sets the appropriate flags. The functions operate on the `A` and `B` inputs (`in_A` and `in_B` respectively) and produce the output, `out`. In addition, there is a carry in, `cin`, and four result flags indicating a negative result, a zero result, overflow and carry.

The auxiliary functions used to define the ALU are defined in terms of the abstract word package. For example, here is the auxiliary function used to define addition without carry:

```
⊢def ADD_WITHOUT_CARRY rep (in_A,in_B,cin) out
                        (neg,zero,ovfl,carry) =
        let result = (add rep) (in_A,in_B) in
        let c = (addp rep)  (in_A,in_B,result) and
            n = (negp rep) result and
            z = (zerop rep) result and
            v = (aovfl rep) (in_A,in_B,result) in
        ((out = result) ∧
         (neg = n) ∧
         (zero = z) ∧
         (ovfl = v) ∧
         (carry = c))
```

This predicate specifies addition without carry simply because it uses the auxiliary functions that we have *decided* describe that operation. In fact, this specification makes no statement about what addition without carry means. Furthermore, we will not prove that the ALU adds correctly or performs any other mathematical operation. What we will prove is that the primitive operations are called in such a way that the top level specification is met.

The Shifter Block. The shifter has four functions: 1-bit shift left, 1-bit shift right, 1-bit arithmetic shift right, and a NOOP. The functions are selected by a 2-bit switch.

```
⊢def SHIFTER_SPEC rep switch in_A out c_flag =
      ((switch = (F,F)) → ((out = (shl rep) in_A) ∧
                           (c_flag = (msb rep) in_A))    |
       (switch = (F,T)) → ((out = (shr rep) in_A) ∧
                           (c_flag = (lsb rep) in_A))    |
       (switch = (T,F)) → ((out = (asr rep) in_A) ∧
                           (c_flag = (lsb rep) in_A))    |
                          ((out = in_A)  ∧
                           (c_flag = F))                 )
```

The specification of the shifter is also given in terms of the abstract representation for the microprocessor. In addition to calculating the output of the shifter, the specification produces a carry corresponding to the bit shifted out of the word.

Miscellaneous Logic. In addition to several and–gates and or–gates, the specification makes use of several larger chunks of logic to describe the selection signals for the memory address register and the program counter register.

```
⊢def MAR_LOGIC_SPEC pmux clk_3 clk_4 mar out =
     ∀ t:time.  (out t) =
      ((((pmux t) ∧ (clk_3 t)) ∨
        (~(pmux t) ∧ (clk_4 t))) ∧ (mar t))

⊢def PC_LOGIC_SPEC clk pc_enable pc_jmp_enable jump_flag out =
     ∀ t:time.  (out t) = (clk t) ∧
                         ((pc_enable t) ∨
                          ((pc_jmp_enable t) ∧ (jump_flag t)))
```

The Datapath Specification. The datapath definition (shown in Figure 5.7) is made from the blocks that we have specified. We specify the internal lines using existential quantification. The specification of the datapath is difficult to read; it is also difficult to write. There is, however, a close correspondence between the major blocks, the internal lines, and the external lines in the specification given in Figure 5.7 and the circuit diagram shown in Figure 5.2. In a production setting, the structural specification could be derived from a CAD description of the circuit, given the appropriate definitions for the blocks. This would make the specification of the electronic block model much easier.

Some of the blocks in the datapath expect arguments that are functions of time and others do not. The use of the blocks in the specification of the datapath reflects this. For example, `MUX_SPEC` uses `(MuxIn t)` as an argument, while `MBR_SPEC` simply uses `MuxIn`.

```
⊢def DATAPATH rep mem reg mar mbr alatch blatch ir pc psw ivec
              iack_ff ireq_ff ireq_e amux_s alu_s shft_s mbr_s
              mar_s pmux_s cselect aselect bselect
              s_sm c_sm s_ie c_ie ld_c ld_v ld_n ld_z csrc_s
              iack_s rd_s wr_s opc ie sm clk_1 clk_2 clk_3 clk_4 =
     ∀ t:time.
     ∃  Abus Bbus Cbus MuxOut MuxIn MemData AluOut Gnd MarIn
        regd_enable ssp_enable psw_enable ir_enable pc_enable
        pc_jmp_enable reg_a_enable reg_sa_enable ssp_a_enable
        psw_a_enable C255_enable pc_a_enable reg_b_enable
        ivec_enable ir_b_enable ld_reg_block ld_ssp ld_ir
        ld_psw ld_mar ld_pc do_write dest_s srca_s srcb_s
        alu_c shift_c cf nf vf zf jump_flag pc_a_1 pc_a_2
        pc_a_3 ir_b_1 ir_b_2 float0 float1 .
      (GND (Gnd t)) ∧
      (DEMUX_3_SPEC cselect regd_enable ssp_enable psw_enable
                            ir_enable pc_enable pc_jmp_enable
                            float0 float1) ∧
      (DEMUX_3_SPEC aselect reg_a_enable reg_sa_enable ssp_a_enable
                            psw_a_enable C255_enable pc_a_1
                            pc_a_2 pc_a_3) ∧
      (OR_3_SPEC pc_a_1 pc_a_2 pc_a_3 pc_a_enable) ∧
      (DEMUX_2_SPEC bselect reg_b_enable
                    ivec_enable ir_b_1 ir_b_2) ∧
      (OR_SPEC ir_b_1 ir_b_2 ir_b_enable) ∧
      (AND_SPEC clk_4 regd_enable ld_reg_block) ∧
      (AND_SPEC clk_4 ssp_enable ld_ssp) ∧
      (REGISTER_BLOCK rep dest_s srca_s srcb_s
                          ld_reg_block ld_ssp reg_a_enable
                          reg_sa_enable ssp_a_enable reg_b_enable
                          Cbus Abus Bbus psw reg) ∧
      (AND_SPEC clk_4 ir_enable ld_ir) ∧
      (IR_SPEC rep ld_ir ir_b_enable Cbus Bbus ir
                   opc dest_s srca_s srcb_s) ∧
      ...
```

Figure 5.7: The specification for the datapath (continued on next page).

```
... ∧
(LATCH_SPEC Abus clk_2 alatch) ∧
(LATCH_SPEC Bbus clk_2 blatch) ∧
(IVEC_SPEC rep ivec_enable Bbus ivec) ∧
(FF_SPEC iack_s clk_2 iack_ff) ∧
(FF_SPEC ireq_e clk_1 ireq_ff) ∧
(MUX_SPEC (amux_s t) (MuxIn t) (alatch t) (MuxOut t)) ∧
(MAC2_ALU_SPEC rep
         (alu_s t)
         (MuxOut t,blatch t,get_cf rep (psw t))
         (AluOut t) (nf t, zf t,vf t,alu_c t)) ∧
(SHIFTER_SPEC rep (shft_s t) (AluOut t) (Cbus t)
                  (shift_c t)) ∧
(MUX_1_SPEC (csrc_s t) (alu_c t) (shift_c t) (cf t)) ∧
(MBR_SPEC mbr_s clk_4 rd_s wr_s Cbus mbr MuxIn MemData) ∧
(AND_SPEC clk_4 psw_enable ld_psw) ∧
(PSW_SPEC rep ld_psw clk_4 psw_a_enable Cbus Abus ie sm psw
              (vf t) (nf t) (cf t) (zf t)
              s_sm c_sm s_ie c_ie ld_v ld_n ld_c ld_z) ∧
(JUMP_SPEC rep dest_s psw jump_flag) ∧
(PC_LOGIC_SPEC clk_4 pc_enable pc_jmp_enable
               jump_flag ld_pc) ∧
(REG_SPEC Cbus ld_pc pc_a_enable Abus pc) ∧
(C255_SPEC rep (C255_enable t) (Abus t)) ∧
(MUX_SPEC (pmux_s t) (pc t) (Cbus t) (MarIn t)) ∧
(MAR_LOGIC_SPEC pmux_s clk_3 clk_4 mar_s ld_mar) ∧
(LATCH_SPEC MarIn ld_mar mar) ∧
(AND_SPEC clk_4 wr_s do_write) ∧
(MEM rep do_write rd_s mar MemData mem)
```

Figure 5.8: The specification for the datapath (continued).

The Control Unit Blocks. Now that we have specified the datapath, we will turn our attention to the control unit. The control unit has three main parts: (1) the microprogram counter and its associated logic, (2) the microinstruction register, and (3) the clock. The microrom, is specified as a variable. The microrom specification for the microcode in *AVM–1* is described in Section 5.2.4.

The Microprogram Unit Block. The microprogram unit calculates the next value of the microprogram counter from the current value, the contents of the microinstruction register, and some signals from the datapath. The microprogram counter is 6 bits wide; the function `add_bt6` adds a boolean 6-tuple and a number.

```
⊢def MPC_UNIT mpc opc addr cond ireq_f ie sm =
        let bt6_inc n     = (add_bt6 n 1) in
        ((cond = (F,F,F)) → (bt6_inc mpc) |
         (cond = (F,F,T)) → addr |
         (cond = (F,T,F)) → (add_bt6 (F,(SND opc)) 4) |
         (cond = (F,T,T)) →
               ((ireq_f ∧ ie) → addr |
                                  (bt6_inc mpc)) |
         (cond = (T,F,F)) → (sm → addr | (bt6_inc mpc)) |
                            (bt6_inc mpc))
```

There are 5 jump conditions:

1. No jump; the microprogram counter is incremented. This is the default operation.
2. Jump to `addr` unconditionally
3. Jump to the location given by the `opc` signal plus an offset (4 in this case). This allows us to use a table lookup approach to instruction decoding in the microcode. Note that no matter what the opcode is, we only use the 5 least significant bits for a value. The top half of the instruction set is reserved for a coprocessor.
4. Jump to `addr` if the interrupt signal is true and interrupts are enabled.
5. Jump to `addr` if the supervisory mode signal is true.

We use the above definition in specifying the microprogram counter:

```
⊢def MPC_SPEC rep mpc clk opc irq ie sm addr_s cond_s =
      ∀ t:time.
       mpc (t+1) =
        ((clk t) →
           (MPC_UNIT (mpc t) (opc t) (addr_s t)
                     (cond_s t) (irq t) (ie t) (sm t)) |
           mpc t)
```

When the `clk` signal is high, the new value of the microprogram counter is calculated using `MPC_UNIT`. Otherwise, the value remains unchanged.

The Microinstruction Register Block. The microinstruction register is simple in concept, but rather unwieldy to specify. The specification describes a register with 25 ports—one corresponding to each of the fields in the microinstruction. The following specification uses selection functions on microinstructions to produce the various fields. For example, `Alu` is a selector on a microinstruction that returns a 4–bit field giving the ALU operation.

```
⊢def MIR_SPEC mir clk in
          amux_s sh_s alu_s mbr_s mar_s pmux_s cselect
          aselect bselect s_sm_s c_sm_s s_ie_s c_ie_s
          ld_c_s ld_v_s ld_n_s ld_z_s csrc_s ftch_s
          iack_s rd_s wr_s addr_s cond_s =
      ∀ t:time .
          (mir (t+1) = (clk t → (in t) | (mir t))) ∧
          (amux_s t =  (Amux (mir t))) ∧
          (sh_s t =  (Shift (mir t))) ∧
          (alu_s t = (Alu (mir t))) ∧
          (mbr_s t = (Mbr (mir t))) ∧
          (mar_s t = (Mar (mir t))) ∧
          (pmux_s t = (Pmux (mir t))) ∧
          (cselect t = (Trgt (mir t))) ∧
          (aselect t = (SrcA (mir t))) ∧
          (bselect t = (SrcB (mir t))) ∧
          (s_sm_s t = (S_sm (mir t))) ∧
          (c_sm_s t = (C_sm (mir t))) ∧
          (s_ie_s t = (S_ie (mir t))) ∧
          (c_ie_s t = (C_ie (mir t))) ∧
          (ld_c_s t = (Ld_c (mir t))) ∧
          (ld_v_s t = (Ld_v (mir t))) ∧
          (ld_n_s t = (Ld_n (mir t))) ∧
          (ld_z_s t = (Ld_z (mir t))) ∧
          (csrc_s t = (Csrc (mir t))) ∧
          (ftch_s t = (Ftch (mir t))) ∧
          (iack_s t = (Iack (mir t))) ∧
          (rd_s t = (Rd (mir t))) ∧
          (wr_s t = (Wr (mir t))) ∧
          (addr_s t = (Address (mir t))) ∧
          (cond_s t = (Cond (mir t)))
```

The Clock Block. The clock is a four–valued counter with a strobe line for each of the phases. The counter sequences from 0 to 3. The strobe `clk_1` is only high in the first clock phase, the strobe `clk_2` is only high in the second clock phase, and so on.

```
⊢def CLOCK_SPEC clk clk_1 clk_2 clk_3 clk_4 =
         ∀ t:time.
         (clk (t+1) = (((clk t) = (F,F)) → (F,T) |
                       ((clk t) = (F,T)) → (T,F) |
                       ((clk t) = (T,F)) → (T,T) | (F,F))) ∧
         (clk_1 t = (clk t = (F,F))) ∧
         (clk_2 t = (clk t = (F,T))) ∧
         (clk_3 t = (clk t = (T,F))) ∧
         (clk_4 t = (clk t = (T,T)))
```

The Control Unit. The control unit is specified by connecting the behavioral specifications for the microprogram counter, microinstruction register, and clock. The only internal lines carry the address and jump condition portions of the microinstruction from the microinstruction register to the microprogram counter unit.

```
⊢def CONTROL_UNIT rep
         mpc mir clk urom
         clk_1 clk_2 clk_3 clk_4
         amux_s sh_s alu_s mbr_s mar_s pmux_s cselect aselect
         bselect s_sm c_sm s_ie c_ie ld_c ld_v ld_n ld_z csrc_s
         ftch_s iack_s rd_s wr_s opc sm ie ireq_f =
      ∃  addr_s cond_s .
      (MPC_SPEC rep mpc clk_4 opc ireq_f ie sm addr_s cond_s) ∧
      (MIR_SPEC mir clk_1 (λ t. (urom t (bt6_val (mpc t))))
                amux_s sh_s alu_s mbr_s mar_s pmux_s cselect
                aselect bselect s_sm c_sm s_ie c_ie ld_c ld_v
                ld_n ld_z csrc_s ftch_s
                iack_s rd_s wr_s addr_s cond_s) ∧
      (CLOCK_SPEC clk clk_1 clk_2 clk_3 clk_4)
```

EBM State. Before we put the datapath and the control unit together to specify the structure of the electronic block model, we describe the state that is visible at this level. The following state–tuple is used to describe the state at the electronic block model level.

```
(reg, psw, pc, mem, ivec, ir, mar, mbr, mpc,
 alatch, blatch, ireq_ff, iack_ff, mir, urom, clk)
```

The state–tuples for more abstract levels will contain a subset of the members of the state–tuple at this level. We have kept the names consistent between levels for clarity.

- **reg** – A variable of type `:(*wordn)list` used to represent the register file.
- **psw** – A variable of type `:*wordn` used to represent the program status word.
- **pc** – A variable of type `:*wordn` used to represent the program counter.
- **mem** – A variable of type `:*memory` used to represent external memory.
- **ivec** –A variable of type `:*wordn` used to represent the interrupt vector.
- **ir** – A variable of type `:*wordn` used to represent the instruction register.
- **mar** – A variable of type `:*wordn` used to represent the memory address register.

- **mbr** – A variable of type `:*wordn` used to represent the memory buffer register.
- **mpc** – A variable of type `:bt6` (boolean 6–tuple) used to represent the microprogram counter.
- **alatch** – A variable of type `:*wordn` used to represent the latch feeding the `A` side of the ALU.
- **blatch** – A variable of type `:*wordn` used to represent the latch feeding the `B` side of the ALU.
- **ireq_ff** – A variable of type `:bool` used to represent the interrupt request flipflop.
- **iack_ff** – A variable of type `:bool` used to represent the interrupt acknowledge flipflop.
- **mir** – A variable of type `:ucode` (a complex bit–string) used to represent the microinstruction register.
- **urom** – A variable of type `:num → ucode` used to represent the microrom.
- **clk** – A variable of type `:bt2` (boolean 2–tuple) used to represent the phase–level clock.

The EBM Specification. The electronic block model is specified by connecting the datapath and the control unit using existential quantification to represent internal lines. We want a definition of the electronic block model that can be used with the generic interpreter specification. The electronic block model is used to instantiate the abstract implementation, `Impl`, which has the abstract type

```
:(time' → *state') → (time' → *env') → bool
```

The definition must take two functions of time, one representing the state stream and the other the environment stream and return a boolean. We use tuples, abstracted over time to represent these state and environment streams.

```
⊢ EBM
    rep
    (λ t.
      (reg t,psw t,pc t,mem t,ivec t,ir t,mar t,mbr t,mpc t,
       alatch t,blatch t,ireq_ff t,iack_ff t,mir t,urom,clk t))
    (λ t. (ireq_e t)) =
    (∃ amux_s alu_s shft_s mbr_s mar_s pmux_s cselect aselect
       bselect s_sm c_sm s_ie c_ie ld_c ld_v ld_n ld_z csrc_s
       iack_s rd_s wr_s ftch_s opc ie sm clk_1 clk_2 clk_3 clk_4.
      DATAPATH rep
          mem reg mar mbr alatch blatch ir pc psw
          ivec iack_ff ireq_ff ireq_e amux_s alu_s
          shft_s mbr_s mar_s pmux_s cselect aselect bselect
          s_sm c_sm s_ie c_ie ld_c ld_v ld_n ld_z csrc_s
          iack_s rd_s wr_s opc ie sm
          clk_1 clk_2 clk_3 clk_4 ∧
      CONTROL_UNIT rep
          mpc mir clk (λ t. urom) clk_1 clk_2 clk_3 clk_4
          amux_s shft_s alu_s mbr_s mar_s pmux_s
          cselect aselect bselect s_sm c_sm s_ie c_ie
          ld_c ld_v ld_n ld_z csrc_s ftch_s iack_s
          rd_s wr_s opc sm ie ireq_ff)
```

The above specification is not a definition, but rather a theorem; a definition cannot have lambda abstractions on the left–hand side. To create this theorem, we define the electronic block model using single variables for the state and the environment and selectors on those variables. Using that definition and the definition of the state selectors, we can derive the theorem given above.

The EBM Clock. There are two other parts of the abstract representation that need to be instantiated with definitions related to the specification of the electronic block model. We must define a function representing `count`, which takes the electronic block model state and environment streams and returns the clock. We must also define a constant `begin` that designates the beginning state for the electronic block model clock. There's one small problem: the electronic block model clock and the phase–level clock are the same; in other words, there is no temporal abstract between those two levels. We can still use the generic interpreter theory, however, since we can model this using a 1-phase clock at the electronic block model level.

The following definitions for `GetEBMClock` and `EBM_Begin`, which represent `count` and `begin` respectively, implement a 1-phase clock. There are many ways of doing this; we chose to use an arbitrary boolean value to represent the single phase.

```
⊢def GetEBMClock rep (reg, psw, pc, mem, ivec, ir, mar,
                     mbr, mpc, alatch, blatch, ireq_ff,
                     iack_ff, mir, urom, clk)
                    (int_e) = ε x:bool. F

⊢def EBM_Begin = ε x:bool. F
```

The expression ε `x:bool. F` chooses a boolean value for `x` such that the expression `F` is true. Since `F` can never be true, we get an arbitrary value of the same type as `x`, boolean.

5.2.3 Defining the Phase Level.

The phase-level represents the lowest level interpreter in our hierarchy. It is really a reflection of the electronic block model rather than an abstraction. All of the state present in the electronic block model is present in the phase-level and they share the same clock. Although we only show that the electronic block model implies the phase–level, we could show that they are equivalent. We first present an informal description of the phase–level interpreter and then present the formal definitions.

Defining the Phase–Level State. The state–tuple that describes the phase–level interpreter state is identical to the tuple describing the state of the electronic block model.

```
(reg, psw, pc, mem, ivec, ir, mar, mbr, mpc,
 alatch, blatch, ireq_ff, iack_ff, mir, urom, clk)
```

The variables have the same meaning as they did in the electronic block model.

Defining the Instruction List. The operation of the phase–level interpreter is fairly simple. We associate each phase in the system clock with an instruction in the phase–level interpreter. The instructions define the state transitions that occur during each phase of the clock. This same information is available in the electronic block model, but is not as apparent. During the four phases, the machine performs the following state transitions:

1. In phase 1, the microinstruction register is loaded from the microrom.

2. In phase 2, the latches feeding ALU are loaded from the register file and system registers.

3. In phase 3, the ALU and shifter calculate a result based on their inputs.

4. In phase 4, the result calculated in phase 3 is stored back into the register file and system registers.

The formal definitions for these phases describe in detail what happens at each phase.

Phase–One. During the first phase, the microinstruction register is loaded with the contents of the microrom at the location given by the current microprogram counter, the flip-flop holding the interrupt request is latched from the interrupt request line in the environment, and the clock is updated so that the second phase is selected next.

```
⊢def phase_one rep (reg, psw, pc, mem, ivec, ir, mar, mbr, mpc,
                    alatch, blatch, ireq_ff, iack_ff, mir, urom,
                    clk)
                   (int_e) =
       let new_mir = urom (bt6_val mpc) and
           new_ireq_ff = int_e and
           new_clk = (F,T) in
       (reg, psw, pc, mem, ivec, ir, mar, mbr, mpc,
        alatch, blatch, new_ireq_ff, iack_ff, new_mir,
        urom, new_clk)
```

Phase–Two. During the second phase, the latches that feed the ALU are loaded from the register file and system registers according to the `SrcA` and `SrcB` fields in the microinstruction register. In addition, the interrupt acknowledge flip-flop is set if the interrupt acknowledge field is set in the microinstruction register. The clock is updated to select the third phase.

```
⊢def phase_two rep (reg, psw, pc, mem, ivec, ir, mar, mbr, mpc,
                    alatch, blatch, ireq_ff, iack_ff, mir, urom,
                    clk)
                   (int_e) =
   let new_alatch = (
       ((SrcA mir) = (F,F,F)) →
                       (EL (reg_len rep (srca rep ir)) reg) |
       ((SrcA mir) = (F,F,T)) →
                       (EL (reg_len rep (dest rep ir)) reg) |
       ((SrcA mir) = (F,T,F)) → (SSP_REG reg) |
       ((SrcA mir) = (F,T,T)) → psw |
       ((SrcA mir) = (T,F,F)) → (wordn rep 255) |
                                 pc) in
   let new_blatch = (
       ((SrcB mir) = (F,F)) →
                       (EL (reg_len rep (srcb rep ir)) reg) |
       ((SrcB mir) = (F,T)) → (int_fetch rep ivec) |
                               (imm rep ir)) in
   let new_iack_ff = Iack mir and
       new_clk = (T,F) in
   (reg, psw, pc, mem, ivec, ir, mar, mbr, mpc,
    new_alatch, new_blatch, ireq_ff, new_iack_ff,
    mir, urom, new_clk)
```

Note that setting the interrupt acknowledge flip-flop in this phase is not conditioned upon the value of the interrupt request flip-flop set in phase one, but the current contents of the microinstruction register. Any connection between the values on these lines is established in the microcode, not in the hardware.

Phase–Three. The primary function of the third phase is to allow the result from the ALU and shifter to stabilize. In addition, the memory address register is loaded from the program counter if the `Mar` and `Pmux` fields are high in the current microinstruction. The clock is updated to select phase four.

```
⊢def phase_three rep (reg, psw, pc, mem, ivec, ir, mar, mbr, mpc,
                      alatch, blatch, ireq_ff, iack_ff, mir, urom,
                      clk)
                     (int_e) =
       let new_mar = (((Pmux mir) ∧ (Mar mir)) → pc | mar) and
           new_clk = (T,T) in
       (reg, psw, pc, mem, ivec, ir, new_mar, mbr, mpc,
        alatch, blatch, ireq_ff, iack_ff, mir, urom, new_clk)
```

Phase–Four. Phase four (shown in Figure 5.9) is the busiest of the four phases. The program status word is updated, the results from the ALU and shifter are stored

```
⊢def phase_four rep (reg, psw, pc, mem, ivec, ir, mar, mbr, mpc,
                     alatch, blatch, ireq_ff, iack_ff, mir, urom,
                     clk)
                    (int_e) =
      let a_input = ((Amux mir) → mbr | alatch) in
      let carry_in = (get_cf rep psw) in
      let alu_result =
          ALU_FUNC rep (Alu mir) a_input blatch carry_in in
      let cf =
          ALU_CARRY_FUNC rep (Alu mir) a_input blatch carry_in in
      let vf =
          ALU_OVFL_FUNC rep (Alu mir) a_input blatch carry_in in
      let nf =
          ALU_NEG_FUNC rep (Alu mir) a_input blatch carry_in in
      let zf =
          ALU_ZERO_FUNC rep (Alu mir) a_input blatch carry_in in
      let result = SHIFTER_FUNC rep (Shift mir) alu_result in
      let shft_c = SHIFTER_CARRY_FUNC rep (Shift mir) alu_result in
      let opc  = (opcode rep ir) in
      let ie   = (get_ie rep psw) and
          sm   = (get_sm rep psw) in
      let new_psw = (
          (((Trgt mir) = (F,T,F)) ∧ sm) → result |
             (mk_psw rep (
               ((S_sm mir) → T | (C_sm mir) → F | sm),
               ((S_ie mir) → T | (C_ie mir) → F | ie),
               ((Ld_v mir) → vf | (get_vf rep psw)),
               ((Ld_n mir) → nf | (get_nf rep psw)),
               ((Ld_c mir) →
                   ((Csrc mir) → cf | shft_c) | (get_cf rep psw)),
               ((Ld_z mir) → zf | (get_zf rep psw))))) in
      ...
```

Figure 5.9: Phase four of the phase–level interpreter (continued on next page).

```
...
let new_reg = (
    ((Trgt mir) = (F,F,F)) →
          (UPDATE_REG rep psw
                  (reg_len rep (dest rep ir)) reg result) |
    ((Trgt mir) = (F,F,T)) →
          (UPDATE_REG rep psw ssp_reg reg result) |
          reg) in
let new_mpc = (
     MPC_UNIT mpc opc (Address mir)
              (Cond mir) ireq_ff ie sm) in
let new_ir = (((Trgt mir) = (F,T,T)) → result | ir) in
let jmp = (JUMP_COND rep (reg_len rep (dest rep ir)) psw) in
let new_pc = (
    ((Trgt mir) = (T,F,F)) → result |
    (((Trgt mir) = (T,F,T)) ∧ jmp) → result | pc) in
let new_mbr = (
    (Rd mir)  → (fetch rep (mem, address rep mar)) |
    (Mbr mir) → result |
                mbr) in
let new_mar =
    ((~(Pmux mir) ∧ (Mar mir)) → result | mar) in
let new_mem =
    ((Wr mir) → store rep (mem,address rep mar,mbr)
              | mem) in
let new_clk = (F,F) in
  (new_reg, new_psw, new_pc, new_mem, ivec, new_ir, new_mar,
   new_mbr, new_mpc, alatch, blatch, ireq_ff, iack_ff, mir,
   urom, new_clk)
```

Figure 5.10: Phase four of the phase–level interpreter (continued).

back in the register file and other system registers, the microprogram counter is updated, the memory buffer register is updated, and a new value of memory is calculated.

The specification of the fourth phase is dependent on several auxiliary functions. For example, the result from the ALU is calculated by `ALU_FUNC`.

```
⊢def ALU_FUNC rep s a_input blatch carry_in  =
       ((s = (F,F,F,F)) → (add  rep (a_input,blatch)) |
        (s = (F,F,F,T)) → (addc rep (a_input,blatch,carry_in)) |
        (s = (F,F,T,F)) → (inc  rep a_input) |
        (s = (F,F,T,T)) → (sub  rep (a_input,blatch)) |
        (s = (F,T,F,F)) → (subc rep (a_input,blatch,carry_in)) |
        (s = (F,T,F,T)) → (dec  rep a_input) |
        (s = (F,T,T,F)) → (band rep (a_input,blatch)) |
        (s = (F,T,T,T)) → (bxor rep (a_input,blatch)) |
        (s = (T,F,F,F)) → (bor  rep (a_input,blatch)) |
        (s = (T,F,F,T)) → (bnot rep a_input) |
                          a_input)
```

The auxiliary functions keep the specification of the fourth phase from being more unwieldy than it already is and significantly reduce the amount of time to verify the phase–level since the time to rewrite a term grows exponentially with its size.

An interesting point in the specification of the fourth phase is that the we calculate the value of the microprogram counter using the same function, `MPC_UNIT`, that we do in the electronic block model. The specification for `MPC_UNIT` represents a functionality assumed at every level in the specification. Thus, we have not proven very much about the microprogram unit, only that it is hooked up correctly. As we mentioned earlier, we have not implemented and verified the blocks in the electronic block model in this proof. The goal of this work is a demonstration that the generic interpreter theory and hierarchical decomposition work. The proof of low-level objects is orthogonal to this goal. We believe, however, that the abstract specifications used at different levels are reasonable and that circuits meeting our specification could be built.

Defining `select`. The abstract function `select` returns a key based on the value of the state and the environment. In the case of the phase–level, the key is simply the clock.

```
⊢def GetPhaseClock rep
          (reg, psw, pc, mem, ivec, ir, mar, mbr, mpc,
           alatch, blatch, ireq_ff, iack_ff, mir, urom, clk)
          (int_e) = clk
```

Defining key. Key transforms a key into a number. Our clock is represented by a boolean 2-tuple, so the tuple function bt2_val serves as the representation for key.

Defining substate. The state is identical at the phase–level and the electronic block model; therefore, the substate function is represented using the built-in identity function, I.

Defining subenv. The environment is identical at the phase–level and the electronic block model; therefore, the subenv function is represented using the built-in identity function, I.

Defining the Phase–Level Interpreter. Unlike the electronic block model specification, we do not combine the phase–level definitions together into a specification for the phase–level. We will use a properly instantiated form of the definition of the generic interpreter from Section 2.3 as our phase–level specification.

In Section 2.3, we defined a generic interpreter, INTERP. The first argument to INTERP is the representation. The representation tuple contains the concrete instantiations for the abstract objects from the abstract representation, in the order that they appear in the abstract representation. Table 5.11 shows the functions used to instantiate the abstract representation. The result is a specification of the phase–level interpreter:

```
⊢def Phase_Int rep s e =
       INTERP
          ([(F,F),phase_one rep;
            (F,T),phase_two rep;
            (T,F),phase_three rep;
            (T,T),phase_four rep],
           bt2_val,
           GetPhaseClock rep,
           I,
           I,
           EBM rep,
           GetEBMClock rep,
           EBM_Start) s e
```

Note that the first argument to the phase–level description, Phase_Int, is rep. This is a different abstract representation than the one used to describe the generic interpreter theory. As we mentioned earlier, the definition of the microprocessor is given in terms of an abstract representation for n–bit words, :*wordn. The variable rep in the above definition is a representation variable for the abstract word data type.

The definition of our microprocessor has two layers of abstraction. We instantiate the generic interpreter theory to get an abstract representation of the microprocessor,

Table 5.11: The functions used to instantiate the abstract representation of the generic interpreter theory for the phase–level.

Operation	*Instantiation*
`inst_list`	list of phase instructions
`key`	`bt2_val`
`select`	`GetPhaseClock`
`substate`	The identity function, `I`
`subenv`	The identity function, `I`
`Impl`	`EBM`
`count`	`GetEBMClock`
`begin`	`EBM_Begin`

which is then instantiated with a word package (for example, `word32`, for a 32–bit microprocessor) to yield a completely specified microprocessor. Thus, `rep` in the above definitions, and all of the definitions and theorems in this section, denotes the abstract representation for the microprocessor's basic data type.

The definition of `Phase_Int` is not very satisfying since it does not look like the predicate that we expect to see in an microprocessor specification. We can instantiate the definition using a function from the abstract theory package as follows:

```
let Phase_Int = save_thm
   (`Phase_Int`,
    BETA_RULE (
    EXPAND_LET_RULE
       (instantiate_abstract_definition
           `gen_I`
           `INTERP`
           Phase_Int_def))
   );;
```

The string `gen_I` in the above expression is the name of the generic interpreter theory and `INTERP` gives the name of the definition to instantiate. We expand the `let` terms in the result to create the more familiar top–level predicate specification:

```
⊢ Phase_Int rep s e =
    (∀ t.
      s(t + 1) =
      SND
      (EL
       (bt2_val(GetPhaseClock rep (s t)(e t)))
         [(F,F),phase_one rep;
          (F,T),phase_two rep;
          (T,F),phase_three rep;
          (T,T),phase_four rep]) (s t) (e t))
```

This theorem defines the phase–level interpreter by relating the state at time $t+1$ to the state and environment at time t. The relationship is based on the n^{th} member of the instruction list where n is calculated from the phase–level clock.

5.2.4 Defining the Microcode.

The phase–level interpreter definition is independent of the contents of the microrom; the microrom appears as a variable. Thus, the microcode is not a level in the abstraction, but rather the data that the phase–level interpreter will act upon to implement the micro–level interpreter.

Recall from the discussion of the microinstruction register in Section 5.1.2 that a microinstruction consists of 40 bits in 24 fields which can be broken into 4 groups: those affecting the operation of the microprocessor, those affecting the program status word, those dealing with external signals, and those that are used for microinstruction sequencing. Table 5.12 briefly reviews the meaning of these fields. Refer to Section 5.1.2 for a more detailed description.

5.2.4.1 The Microcode Assembler.

We use ML to assemble the microcode into the bit–strings that will be used by the phase–level interpreter to implement the micro–level interpreter. The goal in writing this assembler was not to produce a production quality assembler, but rather to allow mnemonic names to be used to define the microcode so that errors can be reduced. This section will describe how the microassembler is used. For implementation details, see [Win90b] .

The microcode assembler is implemented using four functions, one for each of the four groups of fields in the microinstruction.

The Operations Group. The operations group is specified using a function `Oper` which takes the following 6 arguments:

1. Specifies the target register for the operation using the mnemonic values shown in Table 5.13.
2. Specifies the shifter operation using the mnemonic values shown in Table 5.14.
3. Specifies the A source register using the mnemonic values shown in Table 5.13.
4. Specifies the ALU operation using the mnemonic values shown in Table 5.15.
5. Specifies the B source register using the mnemonic values shown in Table 5.13.

Table 5.12: The microinstruction format for *AVM–1*.

Operation Group

Bits	Mnemonic	Description
1	AMUX	Toggle MUX on A-bus
2	SHFT	Shifter function
4	ALU	ALU function
1	MAR	Load MAR from P-Mux
1	MBR	Load MBR from C-bus
1	PMUX	Toggle MUX loading MAR
3	SRCA	A-bus source
2	SRCB	B-bus source
3	TRGT	C-bus target

Program Status Word Group

Bits	Mnemonic	Description
1	S_SM	Set supervisory mode bit in PSW
1	C_SM	Clear supervisory mode bit in PSW
1	S_IE	Set interrupt enable bit in PSW
1	C_IE	Clear interrupt enable bit in PSW
1	LD_C	Load carry bit in PSW
1	LD_V	Load overflow bit in PSW
1	LD_N	Load negative bit in PSW
1	LD_Z	Load zero bit in PSW
1	CSRC	Source of carry (shifter or alu)

External Signals Group

Bits	Mnemonic	Description
1	IACK	Interrupt acknowledge signal
1	FTCH	Fetch signal
1	RD	Read signal
1	WR	Write signal

Microprogram Counter Group

Bits	Mnemonic	Description
3	COND	Microcode jump condition
6	ADDR	Next address

Table 5.13: Register mnemonics for the microassembler.

Mnemonic	Meaning
`reg_file`	Register file
`ssp`	Supervisor stack pointer
`ir`	Instruction register
`psw`	Program status word
`pc`	Program counter (unconditional)
`pcj`	Program counter (using jump conditions)
`mar`	Memory address register
`mbr`	Memory buffer register
`noreg`	No register
`mar_gets_pc`	Load MAR from PC
`reg_dest`	Register file (using dest field from IR)
`C255`	Constant value (255)
`ivec`	Interrupt vector

Table 5.14: Shifter mnemonics for the microassembler.

Mnemonic	Meaning
`shl`	Shift left
`shr`	Shift right
`asr`	Arithmetic shift right
`nsh`	No shift

6. Specifies special operations related to the memory address register and the memory buffer register using the mnemonic values shown in Table 5.13.

Note that not all of the mnemonic values for the registers are allowable in every position. For example, `mbr` can appear in the target field or the source A field, but not the B source field. This is not checked by the assembler, so improper use can give unexpected results.

Here are a few examples of the use of `Oper`:

```
Oper(reg_file,nsh,reg_file,add,pc,noreg);;

Oper(reg_file,shl,mbr,band,reg_file,mar);;
```

The first example adds the contents of the register selected by the A source field in the instruction register to the program counter and stores the result in the register selected by the destination field of the instruction register. The second example takes bit–wise conjunction of the MAR with the register selected by the B source field in the instruction register, shifts the result left and stores it in the register selected by the destination field of the instruction register; the MAR is loaded with the result as well.

Table 5.15: ALU mnemonics for the microassembler.

Mnemonic	Meaning
`add`	Add without carry
`addc`	Add with carry
`inc`	Increment
`sub`	Subtract without borrow
`subc`	Subtract with borrow
`dec`	Decrement
`band`	Bit–wise conjunction
`bxor`	Bit–wise exclusive disjunction
`bor`	Bit–wise disjunction
`bnot`	Bit–wise negation
`nop`	No ALU operation

Table 5.16: Program status word mnemonics for the microassembler.

Mnemonic	Meaning
`set_sm`	Set the supervisory mode bit
`clr_sm`	Clear the supervisory mode bit
`set_ie`	Set the interrupt enable bit
`clr_ie`	Clear the interrupt enable bit
`pass`	Take no action
`ld_from_alu`	Load carry from the ALU
`ld_from_shifter`	Load carry from the Shifter
`ld_vf`	Load the overflow bit
`ld_nf`	Load the negative bit
`ld_zf`	Load the zero bit

The PSW Group. Table 5.16 gives the names and meanings for the mnemonics affecting the program status word (PSW). The value of the PSW group of bits is declared using function `Set_PSW` which has 6 arguments. The meaning of the 6 arguments is given below:

1. Set, clear, or pass (leave unchanged) the supervisory mode bit.
2. Set, clear, or pass the interrupt enable bit.
3. Load the carry bit from either the ALU or the Shifter or take no action.
4. Load the overflow bit or takes no action.
5. Load the negative bit or takes no action.
6. Load the zero bit or takes no action.

Table 5.17: External signal mnemonics for the microassembler.

Mnemonic	Meaning
`rd`	A read is in progress
`wr`	A write is in progress
`no_mem_op`	No memory operation is in progress
`i_ack`	Set the interrupt acknowledge flag
`off`	Turn the signal off
`in_fetch`	CPU is in a fetch cycle

The following examples show how the `Set_PSW` function is used:

```
Set_PSW (set_sm, clr_ie, pass, pass, pass, pass);;

Set_PSW (pass, pass, ld_from_alu, ld_vf, ld_nf, ld_zf);;

Set_PSW (pass, pass, ld_from_shifter, ld_vf, ld_nf, ld_zf);;
```

The first example, sets the supervisory mode bit, clears the interrupt enable bit, and leaves the others unchanged. The second example leaves the supervisory mode bit and the interrupt enable bit unchanged and loads the carry bit from the ALU as well as setting the other status bits from the last ALU operation. The third example differs from the second only in that the carry bit is loaded from the Shifter instead of the ALU. Like the `Oper` function, the `Set_PSW` function does not check for most errors.

The External Signals Group. Table 5.17 give the names and meanings for the mnemonics affecting the external signals. The value of the group of bits for external signals is declared using function `ExtSig` which has 3 arguments. The meaning of the 3 arguments is given below:

1. Specifies whether or not an interrupts being acknowledged.
2. Specifies whether or not the CPU is in fetch mode.
3. Specifies the current memory operation.

The following examples show how the `ExtSig` function is used:

```
ExtSig(off,off,rd);;

ExtSig(i_ack,in_fetch,no_mem_op);;
```

Table 5.18: Microprogram counter mnemonics for the microassembler.

Mnemonic	Meaning
`step`	Increment the program counter and go there
`jmp`	Jump unconditionally
`jop`	Jump relative to mpc based on current opcode
`jint`	Jump on interrupt
`jsm`	Jump when in supervisory mode

In the first example, the microcode turns off interrupt acknowledge, is not in fetch mode, and is performing a read. The second example is acknowledging an interrupt, is in the fetch portion of its cycle, and has no memory operation occurring.

The MPC Group. Table 5.18 give the names and meanings for the mnemonics affecting the microprogram counter. This group of bits is declared using the function `Mpc` which has 2 arguments:

1. The jump condition.
2. The address of the next microinstruction for all jump conditions except the sequencing operator `step`

When a conditional jump fails, the microprogram counter is incremented. The following examples show how the `Mpc` function is used:

```
let TEST_ADDR = "(F,F,F,F,F,F)";;

Mpc(step,TEST_ADDR);;

Mpc(jint,TEST_ADDR);;
```

The first example goes to the next instruction in the microrom regardless of the address given. The second example jumps to location `TEST_ADDR` when the interrupt flipflop is set.

Assembling Microcode. Each of the four functions returns an HOL term consisting of the appropriately sized n–tuple of boolean values. The four functions can be used to specify a microinstruction using HOL's antiquotation operator:

```
    "(^(Oper(noreg,nsh,noreg,nop,noreg,mar_gets_pc)),
      ^(Set_PSW (pass, pass, pass, pass, pass, pass)),
      ^(ExtSig(off,off,rd)),
      ^(Mpc(jint,EINT_u1_ADDR)))"
```

The antiquotation operator (caret) allows an expression that results in a term to be incorporated into an explicit term declaration. This example returns a bit–string broken into four groups—one for each of the four operations just described.

5.2.4.2 The Microinstructions.

Using the microassembler, we can define the microprogram. The microprogram is 53 microwords long and begins at location 0 of the microrom. Most of the macroinstructions are implemented in 4 microinstructions. This section will briefly describe the important features of the microprogram and give several examples of microinstructions that implement macroinstructions. The complete program is contained in [Win90b] .

The `FETCH` Instruction. Every macroinstruction begins with the same, three microinstruction sequence. The only exception is when an external interrupt is being processed. The first microinstruction fetches the instruction from memory to be executed next (pointed to by the program counter).

```
⊢def FETCH_mc =
     (^(Oper(noreg,nsh,noreg,nop,noreg,mar_gets_pc)),
      ^(Set_PSW (pass, pass, pass, pass, pass, pass)),
      ^(ExtSig(off,off,rd)),
      ^(Mpc(jint,EINT_u1_ADDR))
```

If the interrupt flipflop is set, the program branches to the routine that handles external interrupts (located at `EINT_u1_ADDR`).

The definition of `FETCH` given above actually gets assembled before it is stored in the theory. Here is what the assembled version looks like.

```
⊢def FETCH_mc =
   (F,(T,T),(T,F,F,T),F,T,T,(T,T,F),(T,F,T),T,F),
   (F,F,F,F,F,F,F,F,F),
   (F,F,T,F),
   (F,T,T),T,T,F,F,F,T
```

Throughout this section, we will show the unassembled versions, but they are actually stored as bit–strings.

The `ISSUE` Instruction. If the interrupt flipflop is not set, the `FETCH` instruction is followed by the `ISSUE` instruction. The `ISSUE` instruction moves the word that was just fetched from memory to the instruction register.

$\vdash_{def}$
```
ISSUE_mc =
  (^(Oper(ir,nsh,mbr,nop,noreg,noreg)),
   ^(Set_PSW (pass, pass, pass, pass, pass, pass)),
   ^(ExtSig(off,off,no_mem_op)),
   ^(Mpc(step,DUMMY)))
```

The `DECODE` Instruction. The next instruction is the `DECODE` instruction which increments the program counter (in preparation for the next cycle) and branches to the locations in the microcode given by the opcode of the word in the instruction register plus an offset of 4. Thus, locations 4 through 35 of the microrom are a look–up table of microinstructions. The correct microinstruction is selected by the opcode of the current macroinstruction.

$\vdash_{def}$
```
DECODE_mc =
  (^(Oper(pc,nsh,pc,inc,noreg,noreg)),
   ^(Set_PSW (pass, pass, pass, pass, pass, pass)),
   ^(ExtSig(off,off,no_mem_op)),
   ^(Mpc(jop,DUMMY)))
```

The `jop` jump condition does not use the address portion of the microinstruction, so a dummy address is used as the `addr` field.

The `JMP_u1` Instruction. After the instruction has been decoded, the work specific to the macroinstruction being implemented is performed. For example, if the opcode of the current instruction has a value of 0 (the `JMP` instruction), then `DECODE` would jump to location 4 and execute the following microinstruction:

$\vdash_{def}$
```
JMP_u1_mc =
  (^(Oper(pcj,nsh,reg_file,add,ir,noreg)),
   ^(Set_PSW (pass, pass, pass, pass, pass, pass)),
   ^(ExtSig(off,off,no_mem_op)),
   ^(Mpc(jmp,FETCH_ADDR)))
```

This microinstruction conditionally loads the program counter with the value of immediate portion of the instruction register plus the contents of the register in the register file selected by the current instruction. The conditional load is based on the value of the destination field of the current instruction and the values of the status bits in the program status word. After loading the program counter, the microinstruction returns to the beginning of the microprogram.

The `ADD_u1` Instruction. The `ADD` macroinstruction is implemented by the following microinstruction.

```
⊢def ADD_u1_mc =
     (^(Oper(reg_file,nsh,reg_file,add,reg_file,noreg)),
      ^(Set_PSW (pass, pass, ld_vf, ld_nf, ld_from_alu, ld_zf)),
      ^(ExtSig(off,off,no_mem_op)),
      ^(Mpc(jmp,FETCH_ADDR)))
```

This instruction takes both operands from the register file and stores the result of adding them to the register file. The A source, B source, and destination registers in the register file are all selected by the respective fields in the instruction register. The `ADD` instruction sets the appropriate bits in the program status word based on the results of the addition.

The MICROROM Definition. The microrom is a function with domain `:num` and range `:ucode`. we define it by using `EL` to select the n^{th} element from a list of the microinstructions.

```
⊢def micro_rom n =
      EL n
         [FETCH_mc;
          ISSUE_mc;
          DECODE_mc;
          NOOP_u1_mc;
          JMP_u1_mc;
            .
            .
            .
          ADD_u1_mc;
            .
            .
            .
          NOOP_u1_mc]
```

5.2.5 Defining the Micro–Level.

The micro–level interpreter is an abstraction of the behavior described by the phase–level interpreter. At the micro-level, we are concerned with the behavior of the microinstructions, not their implementation.

Defining the Micro–Level State. The state–tuple that describes the micro–level interpreter state is an abstraction of the state–tuple describing the state of the phase–level.

```
(reg, psw, pc, mem, ivec, ir, mar, mbr, mpc)
```

These variables have the same meaning as they did at the electronic block model level. Note that state–variables such as the latches feeding the ALU are no longer available. The only state visible at the micro–level is that seen by someone writing microcode.

Defining the Instruction List. The instruction list at the micro–level is the same length as the microrom and the keys associated with each instruction are identical to the instruction's location in the microrom (rather than being an opcode). We will give abstract behavioral descriptions of each of the microinstructions that were described in section 5.2.4.

The `FETCH` Instruction. The memory buffer register is loaded with the instruction currently pointed to by the program counter. If the interrupt flag is high and interrupts are enabled, the CPU enters the interrupt routine in the microcode, otherwise the next instruction in the microrom is executed.

```
⊢def FETCH rep (reg, psw, pc, mem, ivec, ir, mar, mbr, mpc)
               (int_e, reset_e) =
     let new_mar = pc and
         new_mbr = fetch rep (mem, address rep pc) and
         new_mpc = (int_e ∧ (get_ie rep psw)) → ^EINT_u1_ADDR
                                               | add_bt6 mpc 1 in
     (reg, psw, pc, mem, ivec, ir, new_mar, new_mbr, new_mpc)
```

The `ISSUE` Instruction. The contents of the memory buffer register are moved into the instruction register. The program continues with the next instruction in the microrom.

```
⊢def ISSUE rep (reg, psw, pc, mem, ivec, ir, mar, mbr, mpc)
               (int_e, reset_e) =
     let new_ir = mbr and
         new_mpc = add_bt6 mpc 1 in
     (reg, psw, pc, mem, ivec, new_ir, mar, mbr, new_mpc)
```

The `DECODE` Instruction. During this instruction, the program counter is incremented. The most important action, however, is the jump at the end of the instruction to a location based on the current opcode portion of the instruction register.

```
⊢def DECODE rep (reg, psw, pc, mem, ivec, ir, mar, mbr, mpc)
                (int_e, reset_e) =
     let new_pc = inc rep pc and
         new_mpc =  (add_bt6 (F,(SND(opcode rep ir))) 4) in
     (reg, psw, new_pc, mem, ivec, ir, mar, mbr, new_mpc)
```

Note that the value used for the look–up is not the entire 6–bit opcode, but only the 5 least significant bits, padded with a false value in the most significant bit. The effect of this is to decrease the opcode space to 32 instructions. This was adequate for *AVM–1*; the top 32 instruction are reserved for a future co–processor.

The `JMP_u1` Instruction. This microinstruction changes the program counter to the new value (computed by adding `R[a]` to `imm`) if the jump conditions are met. The microprogram counter is set so that control returns to the beginning of the microprogram (`FETCH_ADDR` is the address of the `FETCH` instruction).

```
⊢def JMP_u1 rep (reg, psw, pc, mem, ivec, ir, mar, mbr, mpc)
              (int_e, reset_e) =
       let a = EL (reg_len rep (srca rep ir)) reg and
           i = imm rep ir and
           d = reg_len rep (dest rep ir) in
       let result = add rep (a, i) in
       let jump_cond = JUMP_COND rep d psw in
       let new_pc = (jump_cond → result | pc) and
           new_mpc = ^FETCH_ADDR in
       (reg, psw, new_pc, mem, ivec, ir, mar, mbr, new_mpc)
```

The boolean valued `jump_cond` is calculated using the function `JUMP_COND` defined in section 5.2.2.

The `ADD_u1` Instruction. This microinstruction adds the values in `R[a]` and `R[b]` and stores the result back into `R[d]`. In addition, the program status word is updated to reflect the status of the ALU after the operation, and the microprogram counter is loaded with the address of the `FETCH` microinstruction.

```
⊢def ADD_u1 rep (reg, psw, pc, mem, ivec, ir, mar, mbr, mpc)
              (int_e, reset_e) =
    let a = EL (reg_len rep (srca rep ir)) reg and
        b = El (reg_len rep (srcb rep ir)) reg and
        d = reg_len rep (dest rep ir) in
    let result = (add rep (a, b)) in
    let cflag  = addp rep (a, b, result) and
        vflag  = aovfl rep (a, b, result) and
        nflag  = negp rep result and
        zflag  = zerop rep result and
        sm     = get_sm rep psw and
        ie     = get_ie rep psw in
    let new_reg = UPDATE_REG rep psw d reg result and
        new_psw =
           mk_psw rep (sm, ie, vflag, nflag, cflag, zflag) and
        new_mpc = ^FETCH_ADDR in
       (new_reg, new_psw, pc, mem, ivec, ir, mar, mbr, new_mpc)
```

The Instruction List. Once we have defined all of the state transition functions denoting the microinstructions, we can put them together in a list suitable for use with the generic interpreter theory. The micro–level instruction set is represented by a list of pairs, where the first member of the pair is the key for selecting it (the location of the microinstruction in the microrom in this case) and the second member of the pair is the state transition function.

```
⊢def micro_inst_list rep =
      [((F,F,F,F,F,F),(FETCH rep));
       ((F,F,F,F,F,T),(ISSUE rep));
       ((F,F,F,F,T,F),(DECODE rep));
       ((F,F,F,F,T,T),(NOOP_u1 rep));
       ((F,F,F,T,F,F),(JMP_u1 rep));
                     .
                     .
                     .
       ((F,T,F,T,F,F),(ADD_u1 rep));
                     .
                     .
                     .
       ((T,T,T,T,T,T),(NOOP_u1 rep))]
```

Defining `select`. At the micro–level, we will view each location in the microrom as constituting a new instruction. Actually this is not true since there are a several instructions in the microrom that appear more than once. In fact, the no–operation instruction appears 13 times. Due to the largely horizontal nature of the microcode, however, most of the instructions are unique. Because we are treating each location in the microrom as a separate instruction, we select the next instruction to execute using the value of the microprogram counter.

```
⊢def  GetMPC (reg, psw, pc, mem, ivec, ir, mar, mbr, mpc)
              (int_e, reset_e) = mpc
```

Defining `key`. `Key` transforms a key into a number. The microprogram counter is represented by a boolean 6-tuple, so the tuple function `bt6_val` serves as the representation for `key`.

Defining `substate`. At the micro–level, `substate` is a function for performing the data abstraction on the phase–level state to produce the micro–state tuple shown earlier:

Table 5.19: The functions used to instantiate the abstract representation of the generic interpreter theory for the micro–level.

Operation	*Instantiation*
inst_list	`micro_inst_list`
key	`bt6_val`
select	`GetMPC`
substate	`Phase_Substate`
subenv	`I`
Impl	`Phase_Int`
clock	`GetPhaseClock`
begin	`PhaseClockBegin`

```
⊢def Phase_Substate rep
       (reg, psw, pc, mem, ivec, ir, mar, mbr, mpc,
        alatch, blatch, ireq_ff, iack_ff, mir, urom, clk) =
       (reg, psw, pc, mem, ivec, ir, mar, mbr, mpc)
```

Defining `subenv`. The environment is identical at the micro–level and the phase–level; therefore, the `subenv` function is represented using the built-in identity function, `I`.

Defining the Micro–Level Interpreter. The definitions given in this section (along with selected definitions from the previous section) are sufficient to instantiate the interpreter definition in the generic interpreter theory. Table 5.19 shows the functions used to instantiate the abstract representation for the micro–level. Just as we did at the phase–level, we can use these definitions to produce a top–level specification of the interpreter at the micro–level:

```
Micro_Int =
⊢ Micro_Int rep s e =
    (∀ t.
      s(t + 1) =
         SND (EL (bt6_val(GetMPC(s t)(e t))) (micro_inst_list rep))
          (s t)
          (e t))
```

5.2.6 Defining the Macro–Level.

The macro–level is the topmost specification in our hierarchy—making it the most abstract. The macro–level specification is a formal specification of what one would generally find in a programmer's manual for a microprocessor (see Section 5.1.1). The specification describes the effect of each of the macro–level instructions on the processor's state and defines how the instructions are selected. The major difference between the formal specification of the microprocessor and the programmer's manual is that the formal specification is less ambiguous and usually more concise.

Defining the Macro–Level State. The state–tuple that describes the macro–level interpreter state is an abstraction of the state–tuple describing the state of the micro–level.

```
(reg, psw, pc, mem)
```

These variables have the same meaning as they did at the micro–level. Note that registers such as the instruction register and the memory address register are no longer available. The only state visible at the macro–level is that seen by someone writing assembly code.

Auxiliary Definitions. Before we specify the instructions, there are a few auxiliary functions that are used to define the behavior of almost every instruction.

```
⊢def GetSrcA rep pc mem =
       reg_len rep (srca rep (fetch rep (mem, address rep pc)))

⊢def GetSrcB rep pc mem =
       reg_len rep (srcb rep (fetch rep (mem, address rep pc)))

⊢def GetImm rep pc mem =
       (imm rep (fetch rep (mem, address rep pc)))

⊢def GetDest rep pc mem =
       reg_len rep (dest rep (fetch rep (mem, address rep pc)))
```

These functions return the values of the instruction fields from the word in memory pointed to by the program counter. Note that they reference memory and not the instruction register; at the macro–level, the instruction register is not visible. Also, not every instruction will use all of these auxiliary functions since the B and immediate fields overlap and some of the formats do not use all of the fields.

Defining the Instruction List. We will not specify every instruction at the macro–level in this section, but will highlight a few example instructions. The complete specification for the macro–level is contained in [Win90b] .

The `JMP` Instruction. The `JMP` instruction has a simple description. The value of the program status word and the contents of the destination field of the current instruction are used to determine if a jump should occur. If so, the program counter is loaded with the sum of the A register and the value of the immediate field from the current instruction. Otherwise, the program counter is incremented.

```
⊢def JMP rep (reg, psw, pc, mem, ivec) =
      let a = EL (GetSrcA rep pc mem) reg and
          i = GetImm rep pc mem and
          d = GetDest rep pc mem in
      let jump_cond = JUMP_COND rep d psw in
      let new_pc = (jump_cond → (add rep (a, i)) | inc rep pc) in
      (reg, psw, new_pc, mem, ivec)
```

The ADD Instruction. The ADD instruction adds the contents of the registers selected by the A and B fields in the current instruction and stores the result in the register selected by the destination field of the current instruction. In addition, the program status word is updated to reflect the results of the calculation and the program counter is incremented.

```
⊢def ADD rep (reg, psw, pc, mem, ivec) =
      let a = EL (GetSrcA rep pc mem) reg and
          b = EL (GetSrcB rep pc mem) reg and
          d = GetDest rep pc mem in
      let result = add rep (a, b) in
      let cflag  = addp rep (a, b, result) and
          vflag  = aovfl rep (a, b, result) and
          nflag  = negp rep result and
          zflag  = zerop rep result and
          sm     = get_sm rep psw and
          ie     = get_ie rep psw in
      let new_reg = UPDATE_REG rep psw d reg result and
          new_psw = mk_psw rep (sm, ie, vflag, nflag, cflag, zflag) and
          new_pc = inc rep pc in
     (new_reg, new_psw, new_pc, mem, ivec)
```

The EINT Instruction. The EINT instruction describes the behavior of the microprocessor upon an external interrupt. The selection criteria for the external interrupt instruction distinguishes it from the other instructions specified at this level. Every other instruction is selected based on the value of the opcode portion of the word in memory pointed to by the program counter; the EINT instruction is selected whenever the external interrupt line in the environment is set. Because its selection criteria differs substantially from that of the other instructions (and because an assembly language programmer would not really think of it as an instruction) we term EINT a "pseudoinstruction." Even though we have not described the implementation of this instruction in earlier sections, we include it here because it has interest both in its own right and in showing how pseudoinstructions can be specified.

Every state variable in the macro–level state except the interrupt vector is changed in the execution of the EINT instruction. The program status word is updated to enter supervisory mode and disable further interrupts. The contents of the program counter are pushed onto the supervisory stack, the supervisory stack pointer (SSP) is incremented, and the program counter is loaded with the 8 least significant bits of the interrupt vector.

```
⊢def EINT rep (reg, psw, pc, mem, ivec) =
     let cd = SSP_REG reg and
          d = ssp_reg in
     let cflag  = get_cf rep psw and
         vflag  = get_vf rep psw and
         nflag  = get_nf rep psw and
         zflag  = get_zf rep psw and
         sm     = T and
         ie     = F in
     let new_psw =
              mk_psw rep (sm, ie, vflag, nflag, cflag, zflag) in
     let new_reg = UPDATE_REG rep new_psw d reg (inc rep cd) and
         new_pc = band rep (wordn rep 255, int_fetch rep ivec) and
         new_mem = store rep (mem, address rep cd, pc) in
     (new_reg, new_psw, new_pc, new_mem, ivec)
```

Note that the value of the interrupt vector is retrieved using the `int_fetch` operation from the abstract theory. This is required because the interrupt vector is shared state.

The Instruction List. Before defining the instruction list and the selection function for the macro–level, we must decide upon a representation for the keys. The instruction's opcode seems particularly well suited to be used as the key since it uniquely identifies the instruction and is a natural part of the description of an assembly language. However, there is one instruction, `EINT`, that has no opcode. We could assign an unused opcode to `EINT`, but this raises the issue of what to do if that opcode appears in a program.

We chose to represent the keys at the macro–level using a coproduct of boolean five–tuples (`:bt5`) and the type containing exactly one object (`:one`). Left injections on the type represent real instructions and right injections represent pseudoinstructions. We chose boolean five–tuples because there were approximately 32 instructions. There is only one pseudoinstruction, so `:one`, the type with only one member, was the logical choice for its representation. There was nothing special about associating `:one` with the pseudoinstructions; if there had been more than one pseudoinstruction, another representation (such as boolean n–tuples) would have worked.

Another small hurdle in defining the instruction list is that since none of the instructions used the environment vector, the state transition functions defined above take only one argument—the state. The second member of an instruction is defined in the generic theory to take two arguments: the state and the environment. We define `ABS_ENV`, which takes a function of type `:(macro_state → macro_state)` and creates a function of type :(`macro_state` $\rightarrow$ `macro_env` $\rightarrow$ `macro_state`).

```
⊢def ABS_ENV f x y = f x
```

We can now define the macro–level instruction list. Every instruction uses the environment abstraction function to give it the proper type. The keys readily distinguish between the real instructions and the pseudoinstructions—clearly specifying the opcodes associated with each real instruction.

```
⊢def macro_inst_list rep =
     [(INL(F,F,F,F,F),ABS_ENV (JMP rep));
                     .
                     .
                     .
      (INL(T,F,F,F,F),ABS_ENV (ADD rep));
                     .
                     .
                     .
      (INR(one),      ABS_ENV (EINT rep));
     ]
```

Defining `select`. The instruction selection function `Opcode` uses the environment and the state to determine which instruction to execute.

```
⊢def Opcode rep (reg, psw, pc, mem, ivec) (int_e, reset_e) =
        (int_e ∧ (get_ie rep psw)) →
            INR(one) |
            INL(SND (opcode rep (fetch rep (mem, address rep pc))))
```

If the interrupt line in the environment is high and interrupts are enabled, then the key associated with the external interrupt instruction, `INR(one)`, is returned. Otherwise, a left injection of the 5 least significant bits of the opcode portion of the word in memory pointed to by the program counter is returned.

Defining `key`. To instantiate the generic interpreter theory, we must be able to turn a key into a number that indexes the instruction associated with that key in the instruction list. The function `Opc_Val` performs that task:

```
⊢def Opc_Val (x:(bt5 + one)) =
        (ISL x) → (bt5_val (OUTL x))
                | 32
```

Table 5.20: The functions used to instantiate the abstract representation of the generic interpreter theory for the macro–level.

Operation	*Instantiation*
inst_list	`macro_inst_list`
key	`Opc_Val`
select	`Opcode`
substate	`Micro_Substate`
subenv	`I`
Impl	`Micro_Int`
clock	`GetMPC`
begin	`FETCH_ADDR`

The function determines whether its argument is a left or right injection and then uses the appropriate function to return the value. Because there is only one possible right injection, we can return 32 without any further work.

Defining `substate`. `Micro_Substate` is the function used to transform a micro–level state–tuple into the macro–level state tuple shown above. `Micro_Substate` is not as straightforward as the substating functions from the previous levels. In particular, the variables representing memory and the interrupt vector register both represent shared state. The interrupt vector register is shared with the interrupt controller and the memory is shared with a variety of devices.

```
⊢def Micro_Substate rep
       (reg, psw, pc, mem, ivec, ir, mar, mbr, mpc) =
          (reg, psw, pc, trans rep mem, int_trans rep ivec)
```

In Section 3.4, we discussed the specification of shared state. The definition of `Micro_Substate` presents a concrete example of the theory in application. The memory at the macro–level is a function of the memory at the micro–level. This function takes into account the changes that are occurring in memory due to the actions of other devices. In this way, the lower levels of the implementation can assume that they own memory without the top–level specification making the same assumption. The interrupt vector is handled similarly. As we will see in the verification of the macro–level, the use of the transformation functions on shared state leads to requirements in the proof that have very satisfying interpretations.

Defining `subenv`. The environment is identical at the macro–level and the micro–level; therefore, the `subenv` function is represented using the built-in identity function, `I`.

Defining the Macro–Level Interpreter. The definitions given in this section (along with selected definitions from the previous section) are sufficient to instantiate the interpreter definition in the generic interpreter theory for the macro–level. Table 5.20 shows the functions used to instantiate the abstract representation. Just as we did at the micro–level, we can use these definitions to produce a top–level specification of the interpreter at the macro–level:

```
Macro_Int =
⊢ Macro_Int rep s e =
    (∀ t.
      s(t + 1) =
      SND
      (EL(Opc_Val(Opcode rep(s t)(e t)))(macro_inst_list rep))
      (s t)
      (e t))
```

5.2.7 Observations.

Having completed the formal specification of *AVM–1*, we have several observations.

This section has shown how a variety of architectural and organizational features can be modeled using the generic interpreter theory. One should not assume that we are claiming that every architectural feature will map onto the models presented in Chapter 4. Indeed, many may not. For example, we have not explored the use of our generic interpreter theory in pipelined architectures.

What does this say then for the utility of generic theories? Certainly, many interesting features, such as interrupts, can be mapped onto the models given in this dissertation. Furthermore, formalizing new models is not a difficult process. We expect that our models will change and new models will be developed to suit new features. The major utility of generic theories, structuring the proof, is not diminished.

Each of the interpreter levels uses a different concept of "key." The phase–level, for example, uses the value of a polyphase clock as the instruction key. The micro–level, on the other hand, uses location in memory as the key to select an instruction. The macro–level uses an opcode as the key. Thus a program that is thousands of instructions long at the micro–level implies that there are thousands of instructions. A program that is thousands of lines long at the macro–level would still only uses the 30 instructions given here.

Another interesting point concerning keys is their use at the macro–level to distinguish between user instructions and pseudoinstructions. When specifying an interpreter, it is important to be flexible about the concept of an instruction. We would not have been able to model the external interrupts using the interpreter theory if we had not been willing to think of it as just another instruction that is selected using an environment signal instead of the program counter.

The use of coproducts to specify the user instructions and pseudoinstruction keys also points out the utility of having a specification language that is powerful and expressive. Because HOL had coproducts, we were easily able to specify the distinction between these two types of instructions while continuing to use the opcode to select user instructions.

The phase–level instructions perform the same action on every cycle. The only difference between one cycle and the next is the data in the microinstruction register. The phase–level could have been structured differently. We could have used the values in the microinstruction register to select among several instructions. For example, instead of selecting the second phase instruction when the clock was `(F,T)`, we could take action conditioned upon the contents of the microinstruction register. This would have made the specification of the phase–level much more complex and subsequently increased the amount of effort required to establish the electronic block model to phase–level correctness result given in the next section.

The specifications of the electronic block model and phase–level provide an interesting point of discussion. The results from the ALU and shifter are calculated in the fourth phase in the phase–level interpreter even though in the electronic block model the results are calculated in the third phase. The difference is that the calculations in the phase–level interpreter happen instantaneously (from the state's perspective) and are therefore calculated and stored in the last phase. The phase when the values are *stored* is what is important in verifying that the phase–level implies the electronic block model, not the phase when they are calculated. This is a good example of the kind of design mistake that our model *will not* catch. Because we are not concerned with gate delays, there is no way to model that the result from the ALU will not be available for some time after the latches are loaded. We probably could have left the third phase out of the design and still verified that the design was correct; obviously, the chip would not have worked even though the design was verified.

In order to deal with timing issues, gate–delays would have to be built into the models. There is nothing to keep us from building specifications that model gate–delay; however, the models would be more complex and the verification that much more difficult. Given current state–of–the–art, it is probably better to leave timing analyses to CAD systems for VLSI layout. In time, the timing analysis may also be done in the formal system; but for now, it seems prudent to let the CAD tools do what they do well and let formal verification so what it does well, namely verify abstract functionality of structural descriptions.

One of the merits of an abstract specification can be clearly seen in the phase–level specification. The interrupt request environment signal, `ireq_e`, is latched into the interrupt flip-flop in the datapath during the first phase. The value of the flip–flop is not used until the fourth phase when its contents are used by the `MPC_UNIT` to calculate the new contents for the microprogram counter. One could legitimately ask why the line is latched so early. The point of this discussion is not to debate that issue, but to point out that the phase–level specification is a useful tool for exploring these kinds of design issues. The circuit diagram and specification of the electronic block model contain this information, but it is more difficult to extract.

Each level in the decomposition hierarchy corresponds to a real level in the microprocessor. We could introduce levels that do not correspond to these real levels. For example, we might add an additional level of abstraction between the micro–level and phase–level to reduce the size of the instruction set that we have to use at the micro–level. This is an area that needs further exploration.

The specification of interrupts is incomplete until the specification presented in this chapter is composed with a priority interrupt controller that receives signals from devices and sets the interrupt vector accordingly.

The specification treats `ivec` as a piece of state. Actually, the specification never changes the value of `ivec` and it seems that it could probably be treated as a member of the environment rather than the state. There is no set rule about what should be in the environment and what should be in the state, but in general, the environment is a good place to put signals that are read–only. A respecification of *AVM–1* should place `ivec` in the environment instead of the state. This would simplify the specification since we would not have to treat `ivec` as shared state. More importantly, the composition of *AVM–1* with a priority interrupt controller would be easier.

5.3 *AVM–1*'s Formal Verification.

Microprocessor verification involves generating a correctness result of the form

$\vdash$ `Structure` $\Longrightarrow$ `Behavior`

from the microprocessor's structural and behavioral specifications. The specifications presented in the last section were written in the object language of HOL and can be manipulated in the HOL system.

The opening part of this section describes how the generic interpreter theory can be used to prove the correctness of the macro–level with respect to the electronic block model using the hierarchical decomposition presented in the last section.

Next, each level in the proof is examined, showing in detail how the proof of correctness for that level was obtained. The three levels are interesting in that different methods of proof were necessary in each.

The last part of this section describes how the proofs of correctness for the three levels can be combined into a overall proof of correctness for *AVM–1*.

5.3.1 Instantiating the Generic Interpreter Theory.

Before describing the actual instantiations of the generic interpreter theory, we discuss exactly what we hope to gain by this instantiation. Figure 5.11 shows how a combination of the generic interpreter and the definitions leads to specifications for the three interpreter levels. We want more than a description however, we want a verified correctness statement.

The diagonal lines in from the interpreter specification at one level to the definitions at the level above represent the proofs that must be done to satisfy the theory obligations. Because of

1. the definitional relationship between the generic interpreter and the specification on one level and
2. the theory obligations relating the implementation and the definitions between levels,

we can conclude that the electronic block model implies the phase–level, the phase–level implies the micro–level, and that the micro–level implies the macro–level. Using these theorems we can prove a result about the overall correctness of our microprocessor.

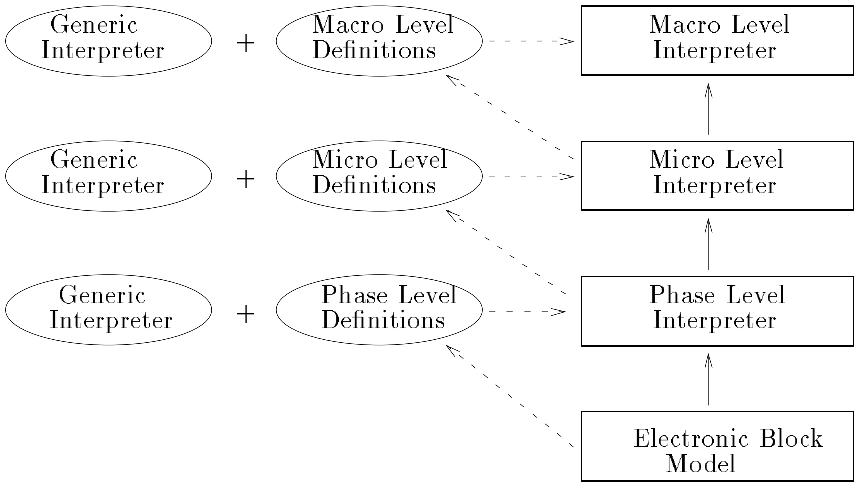

Figure 5.11: The generic interpreter theory can be instantiated with definitions of the various levels from the hierarchical decomposition to yield a proof of the microprocessor.

In the sections that follow, we will be instantiating the generic interpreter proof to provide the desired correctness lemmas at each level. In each case, we will follow the following plan:

1. Instantiate the generic interpreter definition, providing a specification of the interpreter at that level.
2. Instantiate the generic correctness predicate so that it can be used in the proofs of the theory obligations.
3. Prove the three theory obligations for the instantiation. This step constitutes the bulk of each section that follows.
4. Using the proofs of the theory obligations, instantiate the correctness result from the generic theory.

The sections that follow will be divided into subparts roughly corresponding to this plan.

The instantiations, for the most part, are done by calling functions defined in the library package `abstract` which is discussed in Appendix A. We will describe the functions from that package as they are used. All of the instantiation functions are secure; that is, they do their work entirely through primitive inference in the object world of HOL.

Table 5.21: The functions used to instantiate the abstract representation of the generic interpreter theory for the phase–level.

Operation	*Instantiation*
`inst_list`	list of phase instructions
`key`	`bt2_val`
`select`	`GetPhaseClock`
`substate`	The identity function, `I`
`subenv`	The identity function, `I`
`Impl`	`EBM`
`count`	`GetEBMClock`
`begin`	`EBM_Begin`

5.3.2 Verifying the Phase Level.

We would like to show that the phase–level is implemented by the electronic block model. Logically, this amounts to showing that the electronic block model implies the phase–level by proof.

Table 5.21 gives the concrete functions used to instantiate the generic interpreter theory at this level. These functions were all defined in Section 5.2 with the exception of `bt2_val` which gives a numerical value to a boolean 2–tuple.

The Definition. The definition of the phase–level specification was given in Section 5.2.3. Using the function for instantiating definitions from the abstract package and expanding the let terms in the result we get the following theorem:

```
⊢ Phase_I rep s e =
    (∀ t.
      s(t + 1) =
      SND
      (EL
       (bt2_val(GetPhaseClock(s t)(e t)))
       [(F,F),phase_one rep;
        (F,T),phase_two rep;
        (T,F),phase_three rep;
        (T,T),phase_four rep]) (s t) (e t))
```

This theorem defines the phase–level interpreter by relating the state at time $t + 1$ to the state and environment at time t. The relationship is based on the n^{th} member of the instruction list where n is calculated from the phase–level clock.

The Correctness Predicate. After instantiating the top–level specification for the phase–level, we instantiate the instruction correctness predicate for the phase–

level. Each of the phase–level instructions must satisfy this predicate if we are to meet the theory obligations. We first apply the generic instruction correctness definition `INSTRUCTION_CORRECT` to the concrete representation given in Table 5.21.

```
⊢def Phase_Int_Inst_Correct rep s' e' =
    INST_CORRECT
      ([(F,F),phase_one rep;
        (F,T),phase_two rep;
        (T,F),phase_three rep;
        (T,T),phase_four rep],
       bt2_val,
       GetPhaseClock rep,
       I,I,
       EBM rep,
       GetEBMClock rep,EBM_Start)  s' e'
```

After calling the function for instantiating definitions from the abstract package and some minor manipulation we get a predicate that can be used in subsequent proofs.

```
Phase_Int_Inst_Correct =
⊢ Phase_Int_Inst_Correct rep s' e' p =
    EBM rep s' e' ⟹
    (∀ t.
      (GetPhaseClock rep(s' t)(e' t) = FST p) ∧
      (GetEBMClock rep(s' t)(e' t) = EBM_Start) ⟹
      (∃ c.
       Next(λ t'. GetEBMClock rep(s' t')(e' t') = EBM_Start)(t,t + c) ∧
       (SND p (s' t) (e' t) = s'(t + c)))
```

It is interesting to compare this version of the instruction correctness predicate with the generic one. The structure is the same, but the names have changed.

The Theory Obligations. There are three theory obligations that we are required to meet before we can instantiate the generic theory.

1. We must show that each instruction in the phase–level specification is correct with respect to the electronic block model. Specifically, we must prove that the instruction correctness predicate, `Phase_Int_Inst_Correct` is true for every instruction in the phase–level specification.

2. We must show that every key selects an instruction.

3. We must show that every key selects the right instruction.

The Instruction Correctness Lemma. To establish the first theory obligation for the generic interpreter theory, we will prove that the phase–level instruction correctness predicate applies to each of the phases and then use these results to establish that the predicate applies to *every* instruction.

In order to prove the correctness lemma, we will need a lemma about `Next`. `NEXT_LEMMA` states the following:

```
NEXT_LEMMA =
⊢ ∀  t.  t < (t + 1) ∧ (∀ t'.  ~(t < t' ∧ t' < (t + 1)))
```

This is a special form of the `Next` predicate when the existential variable is 1. It says that `t` is less the `t + 1` and that no natural number exists between `t` and `t + 1`.

The following theorem says that the instruction correctness predicate applied to the first instruction, `phase_one`, is a tautology.

```
PHASE_ONE_EBM_LEMMA =
⊢ Phase_Int_Inst_Correct rep
    (λ t.
      (reg t,psw t,pc t,mem t,ivec t,ir t,
       mar t,mbr t,mpc t,alatch t,blatch t,
       ireq_ff t,iack_ff t,mir t,urom,clk t))
    (λ t. (ireq_e t))
    ((F,F),phase_one rep)
```

We proved the instruction correctness lemma for the first phase using the following tactic:

```
    PURE_ONCE_REWRITE_TAC [Phase_Int_Inst_Correct]
    THEN REPEAT GEN_TAC
    THEN BETA_TAC
    THEN REWRITE_TAC [GetPhaseClock;Next;
                      GetEBMClock;EBM_Start;phase_one_def;]
    THEN SUBST_TAC [EBM_expanded]
    THEN REPEAT STRIP_TAC
    THEN POP_ASSUM_LIST
         (λ asl. (MAP_EVERY (STRIP_ASSUME_TAC o SPEC_ALL) asl))
    THEN EXISTS_TAC "1"
    THEN ASM_REWRITE_TAC [PAIR_EQ;NEXT_LEMMA]
```

This tactic performs the following actions:

1. Rewrite with the definition of the instruction correctness predicate.

2. Strip the universal quantification using `GEN_TAC`.

3. Beta reduce the goal to remove the lambda expressions using `BETA_TAC`.

4. Rewrite with the definitions of functions from the instantiation and the definition of the first phase.

5. Substitute the expanded form of the electronic block model definition. The expanded form has all of the definitions completely expanded and is about 4 pages long. Substitution does not perform unification the way that rewriting does and is thus faster than rewriting. Substitution is sufficient for our purposes.

6. Strip the antecedent of the implication (the expanded form of the electronic block model) and place it in the assumption list.

7. Break the expanded form of the electronic block model into the definitions of the individual blocks using `STRIP_ASSUME_TAC`. This tactic picks arbitrary constants for the existential variables and then splits any conjunctions into two assumptions.

8. Pick a witness for the existential variable in the instruction correctness predicate. For this level, finding an existential witness is easy; because there is no temporal abstraction taking place, the existential variable is always 1.

9. Rewrite using the assumptions, `NEXT_LEMMA`, and a theorem about the equality of pairs.

The above tactic only proves the first instruction correctness lemma. The tactics to prove the other instructions at the phase–level are more involved than this one. The tactic that proves the fourth phase is quite long.

The instruction correctness lemma is difficult to prove at the phase–level since every instruction requires a different proof. As we will see, at the micro and macro–levels, one tactic suffices to prove every instruction correctness lemma. We will not show all of the proofs for the phase–level here, but they are contained in [Win90b] .

After we have shown that the instruction correctness predicate is true for each of the instructions, we can show that it is true for every instruction. This satisfies the first theory obligation.

```
Phase_Int_Correct_LEMMA =
⊢ EVERY
       (Phase_Int_Inst_Correct rep
          (λ t.
             (reg t,psw t,pc t,mem t,ivec t,ir t,mar t,mbr t,
              mpc t,alatch t,blatch t,ireq_ff t,iack_ff t,
              mir t,urom,clk t))
          (λ t. (ireq_e t)))
       [(F,F),phase_one rep;
        (F,T),phase_two rep;
        (T,F),phase_three rep;
        (T,T),phase_four rep]
```

The Length Lemma. The second theory obligation is easy to show. The theorem says that the numeric value of a boolean 2-tuple is always less than the length of a four element list.

```
Phase_Int_LENGTH_LEMMA =
⊢ bt2_val clk < (LENGTH [(F,F),phase_one rep;
                         (F,T),phase_two rep;
                         (T,F),phase_three rep;
                         (T,T),phase_four rep])
```

The Order Lemma. The third theory obligation says that the numeric value of the first part of the pair denoting an instruction is the index of that instruction in the instruction list (i.e. that the list is correctly ordered).

```
Phase_Int_ORDER_LEMMA =
⊢ clk = FST (EL (bt2_val clk)
                   [(F,F),phase_one rep;
                    (F,T),phase_two rep;
                    (T,F),phase_three rep;
                    (T,T),phase_four rep])
```

This lemma is also quite easy to show by case analysis.

Instantiating the Correctness Theorem. Having proven the theory obligations, we can instantiate the generic interpreter theory. The function from the `abstract` package which does this takes several arguments.

1. The name of the theory to instantiate.

2. A list of the lemmas proving the theory obligation.

3. A list of substitutions for the parameters in the generic theorems. These substitutions take the form of a pair, where the first member of the pair gives the variable to specialize and the second gives the term with which to specialize it.

4. A string to prepend to the names of the theorems resulting from the instantiation. This is used to prevent name clashes with the names in the generic theory.

The instantiation of the generic interpreter theory for the *AVM–1* microengine is shown in Figure 5.12.

The Final Result. The result of the instantiation can be simplified through minor rewriting and beta reduction. In particular, we note that the temporal abstraction function, `Temp_Abs` is equivalent to the identity function at this level since the clock for the electronic block model and the phase–level are the same.

```
Temp_Abs_DEGENERATE = ⊢ Temp_Abs(λ t. T) = I
```

Using the last theorem and a few minor manipulations, the result of the instantiation is a correctness result for the electronic block model and phase–level becomes:

```
PHASE_LEVEL_CORRECT_LEMMA =
⊢ EBM rep
       (λ t.
         (reg t,psw t,pc t,mem t,ivec t,ir t,
          mar t,mbr t,mpc t,alatch t,blatch t,
          ireq_ff t,iack_ff t,mir t,urom,clk t))
       (λ t. (ireq_e t)) ⟹
    Phase_Int rep
       (λ t.
         (reg t,psw t,pc t,mem t,ivec t,ir t,
          mar t,mbr t,mpc t,alatch t,blatch t,
          ireq_ff t,iack_ff t,mir t,urom,clk t))
       (λ t. (ireq_e t))
```

This result is the same theorem that we would have proven about the phase–level and the electronic block model if we had not used the generic interpreter theory. The result says that the electronic block model implies the phase–level for the concrete state and environment in our model. The result is a little cleaner than the proofs of correctness for other levels since it does not include a temporal projection function and there are no assumptions.

```
let theorem_list =
    instantiate_abstract_theorems
        `gen_I`
        [Phase_Int_Correct_LEMMA;
         Phase_Int_LENGTH_LEMMA;
         Phase_Int_ORDER_LEMMA]
        [
         ("rep:^I_rep_ty",
          "([(F,F),phase_one rep;
             (F,T),phase_two rep;
             (T,F),phase_three rep;
             (T,T),phase_four rep],
            bt2_val,
            GetPhaseClock rep,
            I,
            I,
            EBM rep,
            GetEBMClock rep");
         ("e':time'->*env'",
          "(λ t. (ireq_e t))");
         ("s':time->*state'",
          "(λ t. (reg t, psw t, pc t, mem t, ivec t,
                 ir t, mar t, mbr t, mpc t,
                 alatch t, blatch t, ireq_ff t,
                 iack_ff t, mir t, urom, clk t))");
      ]
      `PHASE`;;
```

Figure 5.12: Instantiating the abstract theory for the phase–level.

5.3.3 Verifying the Micro–Level.

The verification of the micro–level is at once the most straightforward and the largest of the proofs presented here. In the proof of correctness for this level, we are showing that the phase–level specification implements the micro–level specification. Again, we do this by instantiating the generic interpreter proof.

The instantiation is possible even though the implementation for this level, the phase–level specification, is vastly different in structure from the implementation in the proof we just completed. The electronic block model is a structural specification and the phase–level is a behavioral, interpreter–based specification. The reason that these two different types of specifications can be used in the instantiation for the implementation is that the generic interpreter theory places very few restrictions on the abstract operator representing the implementation.

Table 5.22 gives the concrete functions used to instantiate the generic interpreter theory at this level. These functions were all defined in Section 5.2 with the exception of `bt6_val` which gives a numerical value to a boolean 6–tuple.

The Definition. The definition of the micro–level specification was given in Section 5.2.5. Using the function for instantiating definitions from the abstract package and expanding the let terms in the result we get the following theorem:

```
Micro_Int =
⊢ Micro_Int rep s e =
    (∀ t.
      s(t + 1) =
       SND
        (EL (bt6_val (GetMPC (s t) (e t)))
            (micro_inst_list rep))
        (s t)
        (e t))
```

This theorem defines the micro–level state at time $t + 1$ in terms of the state at time t using the instruction in the instruction list selected by the current value of the microprogram counter.

The Correctness Predicate. After instantiating the top–level specification for the micro–level, we instantiate an instruction correctness predicate for the micro–level. Each of the micro–level instructions must satisfy this predicate if we are to meet the theory obligations. We first apply the generic instruction correctness definition `INSTRUCTION_CORRECT` to the concrete representation given in Table 5.22.

Table 5.22: The functions used to instantiate the abstract representation of the generic interpreter theory for the micro–level.

Operation	*Instantiation*
inst_list	`micro_inst_list`
key	`bt6_val`
select	`GetMPC`
substate	`Phase_Substate`
subenv	`I`
Impl	`Phase_Int`
clock	`GetPhaseClock`
begin	`PhaseClockBegin`

```
⊢def Micro_Int_Inst_Correct rep s e =
       INST_CORRECT
           (micro_inst_list rep,
            bt6_val, GetMPC,
            Phase_Substate rep, I, Phase_Int rep,
            GetPhaseClock rep, PhaseClockBegin) s e
```

After applying the function for instantiating definitions from the abstract package and some minor manipulation we get a predicate that can be used in subsequent proofs.

```
Micro_Int_Inst_Correct =
⊢ Micro_Int_Inst_Correct rep s e p =
    Phase_Int rep s e ⟹
    (∀ t.
      (GetMPC(Phase_Substate rep(s t))(e t) = FST p) ∧
      (GetPhaseClock rep(s t)(e t) = PhaseClockBegin) ⟹
      (∃ c.
        Next
        (λ t'.  GetPhaseClock rep(s t')(e t') = PhaseClockBegin)
        (t,t + c) ∧
        (SND p(Phase_Substate rep(s t))(e t) =
         Phase_Substate rep(s(t + c)))))
```

The instruction correctness predicate for the micro–level looks very similar to the instruction correctness predicate for the phase–level; only the names are different. This should not come as a surprise since they were generated by instantiating the same generic definition.

The Theory Obligations. Just as we did at the phase–level, we must meet the three theory obligations of the generic theory before we can instantiate it.

The Instruction Correctness Lemma. The first theory obligation for this instance of the generic interpreter theory is that `Micro_Int_Inst_Correct` applies to every instruction in `micro_inst_list`. We do this by case analysis, first showing that it applies to each instruction in the instruction set, and then using those lemmas to show that it applies to every instruction.

There are 64 instructions at the micro–level. In order to prove this large number of lemmas, we use the meta–language of HOL, ML, to automate most of the proof. We write an ML function that when applied to a number, returns the instruction correctness lemma for the instruction in `micro_inst_list` corresponding to that number. This function is mapped onto a list of numbers from 0 to 63 to create a list of lemmas—one for each instruction in the list. The regularity of the proof for the micro–level makes this possible.

The first step is to write a function to produce the desired goal. The following function, when applied to a number, returns the goal for the instruction corresponding to that number.

```
let MK_INST_CORRECT_GOAL n =
    let inst = term_list_el n
                 (snd(dest_eq(
                  snd(dest_forall(concl micro_inst_list))))) in
    "∀ (rep:^rep_ty) reg mem
       psw pc ivec ir mar mbr alatch blatch
       mpc clk urom mir ireq_ff iack_ff int_e.
     (∀ p. mk_psw rep
           (get_sm rep p,get_ie rep p,get_vf rep p,
            get_nf rep p,get_cf rep p,get_zf rep p) = p) ⟹
     Micro_Int_Inst_Correct rep
          (λ t. (reg t,psw t,pc t,mem t,
                 ivec t,ir t,mar t,mbr t,mpc t,
                 alatch t, blatch t, ireq_ff t, iack_ff t,
                 mir t, micro_rom, clk t))
          (λ t. (int_e t)) ^inst";;
```

For example, when applied to 4, `MK_INST_CORRECT_GOAL` returns the goal for the `JMP_u1` microinstruction:

```
"∀ (rep:^rep_ty) reg mem
    psw pc ivec ir mar mbr alatch blatch
    mpc clk urom mir
    ireq_ff iack_ff int_e.
 (∀ p. mk_psw rep
          (get_sm rep p,get_ie rep p,get_vf rep p,
           get_nf rep p,get_cf rep p,get_zf rep p) = p) ⟹
 Micro_Int_Inst_Correct rep
       (λ t. (reg t,psw t,pc t,mem t,
              ivec t,ir t,mar t,mbr t,mpc t,
              alatch t, blatch t, ireq_ff t, iack_ff t,
              mir t, micro_rom, clk t))
       (λ t. (int_e t)) ((F,F,F,F,F,F),JMP_u1 rep)";;
```

In order to establish the correctness of the micro–level, the goal contains an assumption about the abstract word representation:

```
∀ p. mk_psw rep
       (get_sm rep p,get_ie rep p,get_vf rep p,
        get_nf rep p,get_cf rep p,get_zf rep p) = p
```

This assumption requires that the constructors and selectors for the program status word be mutually consistent.

We can solve goals of this form though symbolic execution. As we said, the regularity of the goals allows us to write a single tactic that solves all 64 microinstruction correctness goals. The complete tactic is to large to include here; it can be found in [Win90b] . Rather than include it, we will describe the theory behind how the tactic works.

We will establish an intermediate lemma for each instruction at the phase–level to aid in the symbolic execution. This lemma gives relationships between the various state variables at time t and $t+1$ provided that the phase–level interpreter is valid and the clock selects that instruction at time t. For example, for phase–one, we can easily prove the following lemma using the definition of the phase–level interpreter and the definition of the first instruction at the phase–level.

```
PHASE_ONE_LEMMA =
⊢ Phase_Int rep
        (λ t.  (reg t,psw t,pc t,mem t,ivec t,ir t,mar t,
                 mbr t,mpc t,alatch t,blatch t,ireq_ff t,
                 iack_ff t,mir t,urom,clk t))
        (λ t. (int_e t)) ⟹
    (∀ t. (clk t = F,F) ⟹
          (reg(t + 1) = (reg t)) ∧
          (psw(t + 1) = (psw t)) ∧
          (pc(t + 1) = (pc t)) ∧
          (mem(t + 1) = (mem t)) ∧
          (ivec(t + 1) = (ivec t)) ∧
          (ir(t + 1) = (ir t)) ∧
          (mar(t + 1) = (mar t)) ∧
          (mbr(t + 1) = (mbr t)) ∧
          (mpc(t + 1) = (mpc t)) ∧
          (alatch(t + 1) = (alatch t)) ∧
          (blatch(t + 1) = (blatch t)) ∧
          (ireq_ff(t + 1) = (ireq_ff t)) ∧
          (iack_ff(t + 1) = (iack_ff t)) ∧
          (mir(t + 1) = (urom(bt6_val(mpc t)))) ∧
          (clk(t + 1) = (F,T)))
```

Note that the selection is based on the clock being `(F,F)` for phase–one. Just as we expect from the definition of phase–one, the `mir` and `clk` are the only variables that change. We would also establish `PHASE_TWO_LEMMA`, `PHASE_THREE_LEMMA`, and `PHASE_FOUR_LEMMA`. These can all be proven using a single inference rule.

Now we turn our attention to the symbolic execution that establishes the instruction correctness lemma. We begin by stripping the universally quantified variables and the antecedents of the implication from the goal and rewriting it with the definition of the instruction correctness predicate for the micro–level, the definition of the microinstruction, and the definition of `Next`. This is the result:

```
∃ c.
  (t < (t + c) ∧ (∀ t'. t < t' ∧ t' < (t + c) ⟹
                              ¬ (λ t''. clk t'' = F,F)t') ∧
  (λ t'. clk t' = F,F)(t + c)) ∧
  (reg t,psw t,
    (JUMP_COND rep(reg_len rep(dest rep(ir t)))(psw t) =>
       add rep(EL(reg_len rep(srca rep(ir t)))(reg t),imm rep(ir t)) |
       pc t),
  mem t,ivec t,ir t,mar t,mbr t,F,F,F,F,F,F,F =
  reg(t + c),psw(t + c),pc(t + c),mem(t + c),ivec(t + c),
  ir(t + c),mar(t + c),mbr(t + c),mpc(t + c))
   [ "∀ p. mk_psw rep
       (get_sm rep p,get_ie rep p,get_vf rep p,
        get_nf rep p,get_cf rep p,get_zf rep p) = p" ]
   [ "Phase_Int rep
      (λ t. (reg t,psw t,pc t,mem t,ivec t,ir t,mar t,mbr t,mpc t,
             alatch t,blatch t,ireq_ff t,iack_ff t,mir t,micro_rom,clk t))
      (λ t. (int_e t))" ]
   [ "mpc t = F,F,F,T,F,F" ]
   [ "clk t = F,F" ]
```

The assumption list holds the antecedents of the implication in PHASE_ONE_LEMMA. We can resolve PHASE_ONE_LEMMA with the assumptions using *Modus Ponens* to perform one step in the execution. The results are put back on the assumption list.

```
    ...
    [ "reg(t + 1) = reg t" ]
    [ "psw(t + 1) = psw t" ]
    [ "pc(t + 1) = pc t" ]
    [ "mem(t + 1) = mem t" ]
    [ "ivec(t + 1) = ivec t" ]
    [ "ir(t + 1) = ir t" ]
    [ "mar(t + 1) = mar t" ]
    [ "mbr(t + 1) = mbr t" ]
    [ "mpc(t + 1) = F,F,F,T,F,F" ]
    [ "alatch(t + 1) = alatch t" ]
    [ "blatch(t + 1) = blatch t" ]
    [ "ireq_ff(t + 1) = int_e t" ]
    [ "iack_ff(t + 1) = iack_ff t" ]
    [ "mir(t + 1) =(F,(T,T),(F,F,F,F),F,F,F,(T,F,T),(F,F,F),T,F),
           (F,F,F,F,F,F,F,F,F),(F,F,F,F),(F,F,T),F,F,F,F,F,F" ]
    [ "clk(t + 1) = F,T" ]
```

Note that the value of `urom` has been expanded so that the microinstruction register holds the actual bit string for the microinstruction currently selected by the microprogram counter. Also note that the clock, `clk`, has advanced to `(F,T)`.

We can now use PHASE_TWO_LEMMA to symbolically execute the second phase. We resolve it with the assumption that the phase–level is valid and the clock at time $t+1$ to obtain the following step–wise changes to the phase–level state.

```
...
[ "reg((t + 1) + 1) = reg(t + 1)" ]
[ "psw((t + 1) + 1) = psw(t + 1)" ]
[ "pc((t + 1) + 1) = pc(t + 1)" ]
[ "mem((t + 1) + 1) = mem(t + 1)" ]
[ "ivec((t + 1) + 1) = ivec(t + 1)" ]
[ "ir((t + 1) + 1) = ir(t + 1)" ]
[ "mar((t + 1) + 1) = mar(t + 1)" ]
[ "mbr((t + 1) + 1) = mbr(t + 1)" ]
[ "mpc((t + 1) + 1) = mpc(t + 1)" ]
[ "alatch((t + 1) + 1) =
   EL(reg_len rep(srca rep(ir(t + 1))))(reg(t + 1))" ]
[ "blatch((t + 1) + 1) = imm rep(ir(t + 1))" ]
[ "ireq_ff((t + 1) + 1) = ireq_ff(t + 1)" ]
[ "~iack_ff((t + 1) + 1)" ]
[ "mir((t + 1) + 1) = (F,(T,T),(F,F,F,F),F,F,F,(T,F,T),(F,F,F),T,F),
          (F,F,F,F,F,F,F,F,F),(F,F,F,F),(F,F,T),F,F,F,F,F,F" ]
[ "clk((t + 1) + 1) = T,F" ]
```

The `alatch` and `blatch` have been loaded at time $(t+1)+1$ just as we expect and the clock has advanced to `(T,F)`.

To execute the third phase, we resolve `PHASE_THREE_LEMMA` with the assumption list and add the changes that occur during phase–three to the assumption list.

```
[ "reg(((t + 1) + 1) + 1) = reg((t + 1) + 1)" ]
[ "psw(((t + 1) + 1) + 1) = psw((t + 1) + 1)" ]
[ "pc(((t + 1) + 1) + 1) = pc((t + 1) + 1)" ]
[ "mem(((t + 1) + 1) + 1) = mem((t + 1) + 1)" ]
[ "ivec(((t + 1) + 1) + 1) = ivec((t + 1) + 1)" ]
[ "ir(((t + 1) + 1) + 1) = ir((t + 1) + 1)" ]
[ "mar(((t + 1) + 1) + 1) = mar((t + 1) + 1)" ]
[ "mbr(((t + 1) + 1) + 1) = mbr((t + 1) + 1)" ]
[ "mpc(((t + 1) + 1) + 1) = mpc((t + 1) + 1)" ]
[ "alatch(((t + 1) + 1) + 1) = alatch((t + 1) + 1)" ]
[ "blatch(((t + 1) + 1) + 1) = blatch((t + 1) + 1)" ]
[ "ireq_ff(((t + 1) + 1) + 1) = ireq_ff((t + 1) + 1)" ]
[ "iack_ff(((t + 1) + 1) + 1) = iack_ff((t + 1) + 1)" ]
[ "mir(((t + 1) + 1) + 1) =
   (F,(T,T),(F,F,F,F),F,F,F,(T,F,T),(F,F,F),T,F),
   (F,F,F,F,F,F,F,F,F),(F,F,F,F),(F,F,T),F,F,F,F,F,F" ]
[ "clk(((t + 1) + 1) + 1) = T,T" ]
```

The only change in this phase is the new clock value.

The fourth phase is executed in the same manner, using PHASE_FOUR_LEMMA to obtain the state changes during the fourth phase.

```
...
[ "reg((((t + 1) + 1) + 1) + 1) = reg(((t + 1) + 1) + 1)"]
[ "psw((((t + 1) + 1) + 1) + 1) =
   mk_psw rep
     (get_sm rep(psw(((t + 1) + 1) + 1)),
      get_ie rep(psw(((t + 1) + 1) + 1)),
      get_vf rep(psw(((t + 1) + 1) + 1)),
      get_nf rep(psw(((t + 1) + 1) + 1)),
      get_cf rep(psw(((t + 1) + 1) + 1)),
      get_zf rep(psw(((t + 1) + 1) + 1)))" ]
[ "pc((((t + 1) + 1) + 1) + 1) =
    (JUMP_COND rep
         (reg_len rep(dest rep(ir(((t + 1) + 1) + 1))))
         (psw(((t + 1) + 1) + 1)) =>
     add rep(alatch(((t + 1) + 1) + 1),
             blatch(((t + 1) + 1) + 1)) | pc(((t + 1) + 1) + 1))" ]
[ "mem((((t + 1) + 1) + 1) + 1) = mem(((t + 1) + 1) + 1)" ]
[ "ivec((((t + 1) + 1) + 1) + 1) = ivec(((t + 1) + 1) + 1)" ]
[ "ir((((t + 1) + 1) + 1) + 1) = ir(((t + 1) + 1) + 1)" ]
[ "mar((((t + 1) + 1) + 1) + 1) = mar(((t + 1) + 1) + 1)" ]
[ "mbr((((t + 1) + 1) + 1) + 1) = mbr(((t + 1) + 1) + 1)" ]
[ "mpc((((t + 1) + 1) + 1) + 1) =
    MPC_UNIT (mpc(((t + 1) + 1) + 1))(opcode rep(ir(((t + 1) + 1) + 1)))
             (F,F,F,F,F,F) (F,F,T)(ireq_ff(((t + 1) + 1) + 1))
             (get_ie rep(psw(((t + 1) + 1) + 1)))
             (get_sm rep(psw(((t + 1) + 1) + 1)))" ]
[ "alatch((((t + 1) + 1) + 1) + 1) = alatch(((t + 1) + 1) + 1)" ]
[ "blatch((((t + 1) + 1) + 1) + 1) = blatch(((t + 1) + 1) + 1)" ]
[ "ireq_ff((((t + 1) + 1) + 1) + 1) = ireq_ff(((t + 1) + 1) + 1)" ]
[ "iack_ff((((t + 1) + 1) + 1) + 1) = iack_ff(((t + 1) + 1) + 1)" ]
[ "mir((((t + 1) + 1) + 1) + 1) =
     (F,(T,T),(F,F,F,F),F,F,F,(T,F,T),(F,F,F),T,F),
     (F,F,F,F,F,F,F,F,F,F),(F,F,F,F),(F,F,T),F,F,F,F,F,F" ]
[ "clk((((t + 1) + 1) + 1) + 1) = F,F" ]
```

In the fourth phase, the program counter is finally updated with the new value (provided the jump condition is true). The clock returns to (F,F), signalling that we are through.

The assumption list now contains the step–wise changes for each phase in the phase–level instruction sequence for the JMP_u1 microinstruction. We can solve the goal by rewriting with the assumptions and a few auxiliary lemmas.

The symbolic execution technique can be used to prove each of the instruction correctness lemmas for the 64 microinstructions. Using the instruction correctness lemmas, we can prove that every instruction in the microinstruction list is correct.

```
Micro_Int_CORRECT_LEMMA =
⊢ (∀ p.  mk_psw rep
            (get_sm rep p,get_ie rep p,get_vf rep p,
             get_nf rep p,get_cf rep p,get_zf rep p) = p) ⟹
     EVERY (Micro_Int_Inst_Correct rep
             (λ t. (reg t,psw t,pc t,mem t,
                    ivec t,ir t,mar t,mbr t,mpc t,
                    alatch t, blatch t, ireq_ff t, iack_ff t,
                    mir t, micro_rom, clk t))
             (λ t. (int_e t))) (micro_inst_list rep)
```

The Length Lemma. The length lemma in the micro–level is similar to the length lemma at the phase–level. The only difference is that the keys are represented by boolean 6–tuples, so there are many more cases to consider.

```
Micro_Int_LENGTH_LEMMA =
⊢ bt6_val mpc < (LENGTH (micro_inst_list rep))
```

The Order Lemma. The order lemma at the micro–level is also similar to the order lemma at the micro–level. Again the number of cases is greater, but the proof is straightforward.

```
Micro_Int_ORDER_LEMMA =
⊢ mpc = (FST (EL (bt6_val mpc) (micro_inst_list rep)))
```

Instantiating the Correctness Theorem. After we have established the instruction correctness lemma, the length lemma, and the order lemma for the micro–level, we are ready to instantiate the generic interpreter theory for the micro–level. Figure 5.13 shows the instantiation. The variable `rep` (in the generic theory) gets the concrete representation shown in Table 5.22, `s'` gets the phase–level state stream, and `e'` gets the phase–level environment stream.

The Final Result. After the instantiation is complete and some minor rewriting and beta reduction, the correctness lemma for the micro–level becomes

```
let theorem_list =
    instantiate_abstract_theorems
        `gen_I`
        [Micro_Int_CORRECT_LEMMA;
         Micro_Int_LENGTH_LEMMA;
         Micro_Int_ORDER_LEMMA]
        [
         ("rep:^I_rep_ty",
          "(micro_inst_list rep,
            bt6_val,
            GetMPC,
            Phase_Substate rep,
            I,
            Phase_Int rep,
            GetPhaseClock rep,
            PhaseClockBegin)");
         ("e':time'->*env'",
          "(λ t. int_e t)");
         ("s':time->*state'",
          "(λ t. (reg t,psw t,pc t,mem t,
                  ivec t,ir t,mar t,mbr t,mpc t,
                  alatch t, blatch t, ireq_ff t, iack_ff t,
                  mir t, micro_rom, clk t))")
       ]
      `MICRO`;;
```

Figure 5.13: Instantiating the abstract theory for the micro–level.

```
MICRO_LEVEL_CORRECT_LEMMA =
⊢ (∀ p.  mk_psw rep
        (get_sm rep p,get_ie rep p,get_vf rep p,
         get_nf rep p,get_cf rep p,get_zf rep p) = p) ⟹
   Phase_Int rep
     (λ t. (reg t,psw t,pc t,mem t,ivec t,ir t,
           mar t,mbr t,mpc t,alatch t,blatch t,
           ireq_ff t,iack_ff t,mir t,micro_rom,clk t))
     (λ t. (int_e t)) ∧
   (∃ t.  clk t = F,F) ⟹
   Micro_Int rep
     ((λ t. (reg t,psw t,pc t,mem t,ivec t,
            ir t,mar t,mbr t,mpc t)) o
      (Temp_Abs(λ t. clk t = F,F)))
     ((λ t. (int_e t)) o
      (Temp_Abs(λ t. clk t = F,F)))
```

The lambda expression

```
(λ t. (reg t,psw t,pc t,mem t,ivec t,ir t,mar t,mbr t,mpc t))
```

in the above theorem models a state vector that is a function of time, that is a state stream. It is important to note, however, that this expression represents a data abstraction of the phase–level state stream and thus is not a micro–level state stream until it is composed with the temporal abstraction function

```
(Temp_Abs(λ t. clk t = F,F))
```

which maps micro–level time onto phase–level time.

The correctness result also contains the assumption

```
(∃ t. clk t = F,F)
```

This assumption must be met for the correctness result to be valid. That is, unless we can guarantee that at some time the clock will be at the beginning of its cycle, we cannot say that the computer will function correctly. Of course, we can guarantee this using a reset button.

It is useful to compare the proof at this level with what would have happened had the phase–level specification not been used. We could have still proven the instruction correctness predicate for the microinstructions, but the form of the proof would have been quite different. Instead of being able to write a single tactic that uses symbolic execution to verify the instruction correctness lemma, the irregular types of proof done for the phase–level would have had to have be done *for each of the* 64

Table 5.23: The functions used to instantiate the abstract representation of the generic interpreter theory for the macro–level.

Operation	*Instantiation*
inst_list	`macro_inst_list`
key	`Opc_Val`
select	`Opcode`
substate	`Micro_Substate`
subenv	`I`
Impl	`Micro_Int`
clock	`GetMPC`
begin	`FETCH_ADDR`

instructions at the micro–level. The proof of the phase level is the most difficult one because of the length of the terms and its irregularity; having to repeat it 64 times would have made the proof intractable. The explicit specification of the phase–level is vital to the successful completion of a large microprocessor proof because it places a firewall between the structural specification of the electronic block model and the large instruction case explosion of the upper levels.

We should also point out that even though the size of the microrom was fairly small (64 microwords), proofs containing larger microstores could be completed with very little extra human effort. Some effort could certainly be invested in making the symbolic execution faster; but, once the tactic to prove the instruction correctness lemma is written, the difference between proving 64 microinstructions or 512 is simply a matter of computer time.

For even larger microstores, the proof would have to be restructured. In the proof presented here, we assumed that every word in the microrom was unique and used the position of the instruction in the microrom as the key. By fixing a set of microinstructions that are repeated often and using keys to identify identical instruction, much larger microroms could be verified. This amounts to a nanoprogramming level that may or may not reflect the actual structure of the machine. Only the instruction set would be verified at the micro–level and the microrom would not be used until the microprogram was needed to verify the macro–level.

5.3.4 Verifying the Macro–Level.

The goal of the macro–level verification is to show that the micro–level implements the macro–level. At this level, the micro–level specification becomes the implementation and the macro–level interpreter is used as the abstract behavioral model. We want to show that under some small set of assumptions, the micro–level specification implies the macro–level specification.

Table 5.23 gives the concrete functions used to instantiate the generic interpreter theory at this level. These functions were all defined in Section 5.2.

The Definition. We define the macro–level in the usual manner, using the function for instantiating abstract definitions from the abstract theory package. Using the concrete representation in Table 5.23 we produce the following specification of the macro–level interpreter.

```
Macro_Int =
⊢ Macro_Int rep s e =
    (∀ t.
      s(t + 1) =
      SND
      (EL(Opc_Val(Opcode rep(s t)(e t)))(macro_inst_list rep))
      (s t)
      (e t))
```

The Correctness Predicate. Just as we did at the phase–level and the micro–level, we instantiate the instruction correctness predicate for the macro–level. The instruction correctness predicate, once instantiated, says exactly what must be proven about the instructions at the macro–level to meet the theory obligations and instantiate the generic theory.

```
Macro_Inst_Correct =
⊢ Macro_Inst_Correct rep s' e' p =
    Micro_Int rep s' e' ⟹
    (∀ t.
      (Opcode rep(Micro_Substate rep(s' t))(e' t) = FST p) ∧
      (GetMPC(s' t)(e' t) = F,F,F,F,F,F) ⟹
      (∃ c.
        Next(λ t'.  GetMPC(s' t')(e' t') = F,F,F,F,F,F)(t,t + c) ∧
        (SND p(Micro_Substate rep(s' t))(e' t) =
         Micro_Substate rep(s'(t + c)))))
```

The Theory Obligations. We must satisfy the same three theory obligations at the macro–level as we did at the phase–level and micro–level. The instruction correctness lemma and the order lemma are a little more interesting at this level than they were at the micro–level because of our use of coproducts to represent the keys.

The Instruction Correctness Lemma. We wish to show that every instruction in the macroinstruction set meets the instruction correctness lemma. The instruction list for the macro–level can be broken into two parts based on whether the key is a right or left injection to the coproduct used as the instruction key. If the key is a right injection, then the instruction is a pseudoinstruction. If it is a left injection, then the instruction is a user instruction.

We develop a tactic that will prove the instruction correctness lemma for every instruction in the set. At the macro–level, however, there is only one pseudoinstruction and so handling the pseudoinstruction as a special case makes more sense than developing a tactic general enough to solve both types of instructions. We will not deal with the proof of EINT here, but rather refer the interested reader to [Win90b] . We do note, however, that the techniques used for solving the user instructions are similar to the method used to verify the pseudoinstruction.

Every user instruction at the macro–level has the same three microinstructions in common for the first part of its execution cycle. The FETCH, ISSUE, and DECODE microinstructions are always executed in that order before microinstructions specific to a macroinstruction are executed. Because of this, we prove the following lemma which gives the state at time $t + 3$ as a function of the state at time t.

```
FID_LEMMA =
⊢ Micro_Int rep (λ t. (reg t,psw t,pc t,mem t,ivec t,
                        ir t,mar t,mbr t,mpc t))
                 (λ t. (int_e t)) ⟹
     ∀ t.  (int_e t ∧ get_ie rep (psw t) = F) ∧
         (mpc t = (F,F,F,F,F,F)) ⟹
         ((reg(t + 3),psw(t + 3),pc(t + 3),mem(t + 3),ivec(t + 3),
           ir(t + 3),mar(t + 3),mbr(t + 3),mpc(t + 3)) =
         (reg t,psw t,inc rep(pc t),mem t,ivec t,
          fetch rep(mem t,address rep(pc t)),pc t,
          fetch rep(mem t,address rep(pc t)),
          add_bt6 (F,SND(opcode rep
                         (fetch rep
                          (mem t,address rep(pc t))))) ^OFFSET)) ∧
     ~(mpc(t + 1) = F,F,F,F,F,F) ∧
     ~(mpc((t + 1) + 1) = F,F,F,F,F,F) ∧
     ~(mpc(((t + 1) + 1) + 1) = F,F,F,F,F,F)"),
```

Using this lemma in the proof allows the FETCH--ISSUE--DECODE sequence to be symbolically executed in one step instead of three. Since we will do this for each of the 32 user instructions, this results in substantial savings in time.

Using the same strategy that we used at the micro–level to prove the instruction correctness lemma through symbolic execution, we can prove the instruction correctness lemma for each instruction at the macro–level. For example, here is the instruction correctness lemma for the first instruction in the list, JMP.

```
MAC_INST_0 =
⊢  (∀ m a. fetch rep (trans rep m,a) = fetch rep (m,a)) ∧
    (∀ m a x. store rep (trans rep m,a,x) =
              trans rep (store rep (m,a,x))) ∧
    (∀ m. int_fetch rep (int_trans rep m) = (int_fetch rep m)) ⟹
      Macro_Inst_Correct rep
         (λ t. reg t, psw t, pc t, mem t, ivec t,
              ir t, mar t, mbr t, mpc t)
         (λ t. int_e t)
         (INL(F,F,F,F,F),ABS_ENV (JMP rep))"
```

Note that the instruction correctness lemma is predicated on three assumptions:

```
(∀ m a. fetch rep (trans rep m,a) = fetch rep (m,a))
(∀ m a x. store rep (trans rep m,a,x) = trans rep (store rep (m,a,x)))
(∀ m. int_fetch rep (int_trans rep m) = (int_fetch rep m))
```

Recall that memory is shared state. The function `trans` is a memory transformation function that represents what other devices that share memory with the CPU are doing to memory. Using `trans` we can write a specification that allows changes to memory besides those resulting from CPU action.

The first assumption says that fetching something from memory when another device is changing it is the same as fetching the same thing from memory when no changes are occurring. The second assumption says that the order of memory write operations is not important. In effect, these statements are assumptions of non–interference between the CPU and other devices that use memory. This is exactly what we want to have happen, of course, if we are to say anything reasonable about the reliable operation of a system built using *AVM–1*. The third assumption is a similar statement about the interrupt vector which is shared by the CPU and the interrupt controller.

These three assumptions will appear in the final correctness result. When the correctness result for *AVM–1* is used to verify some more abstract specification relating to the connection of the CPU chip with some other device that uses memory, these assumptions will have to be met to complete the verification. This kind of non–interference might be guaranteed with a hand–shaking protocol. The handshaking protocol would result in lemmas that would be used to discharge these assumptions.

Using the individual results about each instruction in the list, we can prove the instruction correctness lemma for the macro–level.

```
Macro_Int_CORRECT_LEMMA =
⊢  (∀ m. int_fetch rep (int_trans rep m) = (int_fetch rep m)) ∧
    (∀ m a. fetch rep (trans rep m,a) = fetch rep (m,a)) ∧
    (∀ m a x. store rep (trans rep m,a,x) =
                    trans rep (store rep (m,a,x))) ⟹
      EVERY (Macro_Inst_Correct rep
              (λ t. reg t, psw t, pc t, mem t, ivec t,
                      ir t, mar t, mbr t, mpc t)
              (λ t. int_e t))
            (macro_inst_list rep)
```

The Length Lemma. In the length lemma, the opcode variable `opc` has the type `:bt5+one`. The representation of the keys as coproducts makes the proof of the length lemma slightly more interesting than the proof of the length lemma for the other levels; but, not substantially more difficult.

```
Macro_Int_LENGTH_LEMMA =
⊢ Opc_Val opc < (LENGTH (macro_inst_list rep))
```

The Order Lemma. The proof of the order lemma for the macro–level is also different from the proof of the order lemma for the other levels due to the coproduct representation of the keys.

```
Macro_Int_ORDER_LEMMA =
⊢ opc = (FST (EL (Opc_Val opc) (macro_inst_list rep)))
```

Again, the result is not difficult to prove.

Instantiating the Correctness Theorem. After the theory obligations for the macro–level have been established, we can instantiate the generic theory to provide a correctness result for this level. The concrete representation matches that of Table 5.23. The generic environment stream stream, `e'`, is instantiated with the micro–level environment stream and the generic state stream, `s'`, is instantiated with the micro–level state stream.

The Final Result. After the instantiation is complete, some minor rewriting and beta reduction lead to the final result for this level.

```
let theorem_list =
    instantiate_abstract_theorems
        `gen_I`
        [Macro_Int_CORRECT_LEMMA;
         Macro_Int_LENGTH_LEMMA;
         Macro_Int_ORDER_LEMMA]
        [
         ("rep:^I_rep_ty",
          "(macro_inst_list rep,
            Opc_Val,
            Opcode rep,
            Micro_Substate rep,
            I,
            Micro_Int rep,
            GetMPC, ^FETCH_ADDR)");
         ("e':time'->*env'",
          "(λ t:time.  int_e t)");
         ("s':time->*state'",
          "(λ t:time.  reg t, psw t, pc t, mem t, ivec t,
                     ir t, mar t, mbr t, mpc t)")
       ]
      `MACRO`;;
```

Figure 5.14: Instantiating the abstract theory for the macro–level.

```
MACRO_LEVEL_CORRECT_LEMMA =
⊢ (∀ m.  int_fetch rep(int_trans rep m) = int_fetch rep m) ∧
   (∀ m a.  fetch rep(trans rep m,a) = fetch rep(m,a)) ∧
   (∀ m a x.  store rep(trans rep m,a,x) =
                    trans rep(store rep(m,a,x))) ⟹
   Micro_Int rep
       (λ t. (reg t,psw t,pc t,mem t,
                  ivec t,ir t,mar t,mbr t,mpc t))
       (λ t. (int_e t)) ∧
   (∃ t.  mpc t = F,F,F,F,F,F) ⟹
   Macro_Int rep
       ((λ t. (reg t,psw t,pc t,
              trans rep(mem t),int_trans rep(ivec t))) o
        (Temp_Abs(λ t. mpc t = F,F,F,F,F,F)))
       ((λ t. (int_e t)) o
        (Temp_Abs(λ t. mpc t = F,F,F,F,F,F)))
```

The expression

```
(Temp_Abs(λ t. mpc t = F,F,F,F,F,F))
```

is the temporal abstraction function for the macro–level state stream.

5.3.5 *AVM–1* Is Correct.

We have successfully instantiated the generic interpreter theory for each of the levels in our hierarchical decomposition.

As discussed in Section 3.1, we can establish

$$\mathbf{I}_{EBM} \Rightarrow \mathbf{I}_{macro}$$

in stages by showing

$$\mathbf{I}_{EBM} \Rightarrow \mathbf{I}_{phase} \Rightarrow \mathbf{I}_{micro} \Rightarrow \mathbf{I}_{macro}.$$

We will use the correctness results from each of the levels and *Modus Ponens* to prove the correctness result for the entire CPU.

```
AVM_CORRECT =
|- let micro_abs = Temp_Abs(λ t. clk t = F,F) in
   let abs = micro_abs o
             (Temp_Abs(λ t. (mpc o micro_abs)t = F,F,F,F,F,F)) in
   ((∀ m.  int_fetch rep(int_trans rep m) = int_fetch rep m) ∧
    (∀ m a.  fetch rep(trans rep m,a) = fetch rep(m,a)) ∧
    (∀ m a x.
        store rep(trans rep m,a,x) =
              trans rep(store rep(m,a,x))) ⟹
   (∀ p.  mk_psw rep
        (get_sm rep p,get_ie rep p,
         get_vf rep p,get_nf rep p,
         get_cf rep p,get_zf rep p) = p) ⟹
   EBM rep
     (λ t. (reg t,psw t,pc t,mem t,ivec t,ir t,
              mar t,mbr t,mpc t,alatch t,blatch t,
              ireq_ff t,iack_ff t,mir t,micro_rom,clk t))
     (λ t. (ireq_e t)) ∧
   (∃ t.  clk t = F,F) ∧
   (∃ t.  (mpc o micro_abs)t = F,F,F,F,F,F) ⟹
   Macro_Int rep
     ((λ t. (reg t,psw t,pc t,
           trans rep(mem t),int_trans rep(ivec t))) o abs)
     ((λ t. (ireq_e t)) o abs))
```

We can make several points about the final correctness result for *AVM–1*:

- The function abs, which is defined as

  ```
  micro_abs o
    (Temp_Abs(λ t. (mpc o micro_abs) t = F,F,F,F,F,F))
  ```

  where

  ```
  micro_abs = Temp_Abs(λ t. clk t = F,F)
  ```

  is a temporal abstraction function that maps time at the macro-level to time at the electronic block model.

- The assumption that the shared state operations are non–interfering and the assumption that the selectors and constructors on the program status word are consistent both appear in the final result. The first will be satisfied when the CPU is used correctly in conjunction with other devices. The second represents a constraint on the abstract word package (the only one).

- We must also require that there is a time when the clk and the mpc are at the beginning of their cycles. The composition of mpc with micro_abs further requires that this time be congruent for both variables. As we mentioned earlier, both these assumptions can be met using a reset button.

- The definition of `EBM` uses a variable to represent the microrom, `urom`. In the correctness theorem, `EBM` been specialized to use the program specified by `micro_rom`. The electronic block model only implements `Macro_Int` when coupled with a correctly written microprogram.

5.4 Observations.

Having completed the verification of *AVM–1*, we have several observations:

The verification presented in this section has said nothing about whether the macro–level specification is any good. The specification could be wrong—that is, not correctly specify the behavior that the designer had in mind. All we have done is show that we have a machine that implements this behavior, not that it is the behavior we want. We could prove properties about the instructions. For example, we could show that calling a subroutine and then returning from it leaves the program counter with the correct value. We could also come up with a method of executing the specification so that it could be tested. While we have not done either of these in this dissertation, they would be important if we were going to implement *AVM–1*.

The verification of *AVM–1* is dependent upon the high–level specifications of the blocks in the electronic block model. In order to build *AVM–1*, of course, we would need to decide upon implementations for these blocks and show that these implementations satisfy the behavioral requirements imposed upon them by the high–level specifications in the electronic block model. Thus, the verification of *AVM–1* is independent, in a sense, of the particular implementations used for the individual blocks. This gives the designer the flexibility to change the implementation of the blocks without affecting the verification. For example, a designer might include an adder with no look ahead or an adder with 4-bit look ahead depending on the power and space budgets for the chip.

The proof of the instruction correctness lemma was done using a single tactic at the micro–level and another tactic at the phase–level. These tactics both operate through symbolic execution. Because of the great regularity imposed on the proofs of correctness by the generic interpreter theory, it should be possible to write a tactic which solves the instruction correctness lemma for any instantiation (provided that the implementation was an interpreter). This would be an important step since the instruction correctness lemma is the largest part of the effort involved in instantiating the theory.

The verification highlights the fact that the generic interpreter theory uses the same temporal abstraction for the environment and the state streams. This does not have to be so, but seems reasonable for our purposes.

Chapter 6

Summary

6.1 Summary of Major Results.

This dissertation has described a theory of generic interpreters and shown how that theory can aid in the verification of a microprocessor. We believe that several important benefits accrue from our work.

We have provided a methodology for verifying microprocessors that changes what has been primarily a research activity into an engineering activity. The generic interpreter theory structures the specification by stating what definitions must be made. The generic interpreter theory also structures the proof by stating what lemmas must be proven about those definitions.

We believe that the structure provided by the generic interpreter theory, coupled with the savings afforded by the hierarchical decomposition strategy, make the verification of usable microprocessors a viable engineering activity. We are currently in the process of conducting an experiment that will test this hypothesis. We have begun a project to reverify VIPER using graduate students not familiar with HOL or microprocessor verification. We plan to complete the verification using less than 6 man–months of effort. The project is about two–thirds complete. A preliminary report describing the specification of the hierarchy and the verification of the hierarchy's two lowest levels can be found in [Aro90].

We have demonstrated that a hierarchical decomposition of the specification can lead to an order of magnitude reduction in the number of difficult cases that must be considered to complete a microprocessor proof. If we had verified *AVM–1* using the standard approach of directly establishing the macro–level from the electronic block model, we would have to prove 32 long, difficult instruction correctness lemmas. In the verification of *AVM–1* from Chapter 5, the number of these lemmas was reduced to 4 due to the hierarchical decomposition. Machines with larger instruction sets or fewer cycles, provide the opportunity for even larger savings.

In addition to reducing the number of difficult cases in a verification, the hierarchical decomposition leads to proofs of the other cases that are readily automatable. We have shown that at each level in the hierarchy above the electronic block model a single tactic suffices for verifying the resulting lemmas. This regularity can be easily exploited in HOL using ML.

We have demonstrated how generic theories can be used to make the verification of hardware easier. Certainly, the generic interpreter theory is not the only useful generic theory. We found that the generic proof for the interpreter was considerably *easier* than the specific proofs reported in [Win90a]. Because of this, new models for various architectural features can be easily developed and catalogued.

The generic proofs show exactly what a correctness statement for a microprocessor means. Because there is no superfluous detail cluttering up the definitions and theorems, we are less likely to mistakenly think that we have proven that the microprocessor adds, for example, when we look at the generic proof. The final result is a simple statement of the correctness of an interpreter with respect to its implementation.

```
IMPL_I_CORRECT =
⊢ let s = (λ t:time. (substate rep (s' t))) and
      e = (λ t:time. (subenv rep (e' t))) and
      f = (λ t:time. (count rep (s' t) (e' t) =
                          (begin rep))) in
     let abs = (Temp_Abs f) in (
     (Impl rep s' e') ∧ (∃ t. f t) ⟹
     (INTERP rep) (s o abs) (e o abs))
```

The correctness theorem simply states that any true statement about the implementation is similarly true about the abstract interpreter describing its behavior.

Generic theories are a powerful mechanism for reusing theorems. We have demonstrated how a generic interpreter theory can be instantiated—saving the user from having to reverify a number of difficult theorems. The generic theory can be thought of as a structured library that not only provides useful theorems, but also provides a framework for using those theorems. For example, temporal and data abstraction between the interpreter and its implementation are handled entirely within the generic interpreter theory; the user can define the temporal and data abstractions without having to explicitly prove theorems about them.

We have provided the first, to our knowledge, microprocessor specification with provisions for shared state. The work reported in this dissertation does not compose the microprocessor specification with other specifications. However, the inclusion of the transformation functions in the specification leads to conditions on the final result that have very satisfying interpretations.

```
(∀ m a. fetch rep (trans rep m,a) = fetch rep (m,a))
(∀ m a x. store rep (trans rep m,a,x) = trans rep (store rep (m,a,x)))
```

These assumptions amount to non–interference requirements between the memory actions of the CPU and the other devices in the system. The first assumes that a fetch will not be interfered with and the second assumes that a store will not be interfered with. These assumptions must be met when the CPU is composed with other devices if we are to be able to say anything reasonable about the reliable operation of the system. The non–interference proofs are similar to critical regions in concurrent programming. Most likely, we would compose the devices using a handshaking protocol to prove that neither accesses the same location in memory at the same time.

6.2 Future Work.

The work presented here has shown how the generic interpreter theory and hierarchical decomposition strategy can be used in microprocessor verification. The success of this effort has led to us to begin exploring several related areas.

The Computer Systems Verification Group at the University of California, Davis is designing and verifying a complete chip set including a memory management unit, an interrupt controller, a direct memory access controller, and a floating point co-processor. Composing these and other devices will provide an important test of our methodology for specifying shared state.

The specifications for the various levels in our hierarchy are all very regular due to the use of the generic interpreter theory. Imposing regularity in this way leads to possibility of writing tools that make use of this structure.

- For example, we believe that a general tactic for verifying the instruction correctness lemmas could be written that would reduce the amount of effort on the part of the human verifier to simply writing the specification.
- Another example of a general purpose tool that would benefit from the structure imposed by the generic interpreter theory is a tool for executing the specifications. Being able to execute the specifications would eliminate the need for separate simulators (which may or may not match the specifications) for code development.
- We believe that the regular structure imposed by the theory will also prove useful in connecting a verification system with a CAD system or a silicon compiler. Being able to link high–level functional verifications to low–level tools for implementation and design would greatly increase our confidence in a device.

The generic models presented here were used exclusively in microprocessor verification. Of course, microprocessors are not the only hardware devices that act like interpreters. For example, an interpreter theory can be used to describe co–processors, such as the floating point unit and the memory management unit. We are exploring general models of these devices and how these models relate to the generic interpreter theory presented here.

6.3 Conclusion.

The goal of our work has been to make the verification of usable microprocessors tractable. In this dissertation we have described a strategy for hierarchically decomposing specifications that reduces the number of difficult cases by an order of magnitude. We have also described a generic theory useful for specifying and verifying microprocessors. The generic theory structures both the specification and the verification. This structure not only says what has to be done, but provides a framework for building tools to further support the verification. The combination of hierarchical decomposition and the generic interpreter theory represents a substantial improvement over past methods for verifying microprocessor designs.

References

[Ada83] *Reference Manual for the Ada Programming Language.* U.S. Department of Defense, ANSI/MIL-STD-1815A, 1983.

[Adv83] Advanced Micro Devices. *Bipolar Microprocessor Logic and Interface Data Book.* AMD Inc., 1983.

[Anc86] Francois Anceau. *The Architecture of Microprocessors.* Addison-Wesley Publishing Company, 1986.

[Arm89] James R. Armstrong. *Chip–Level Modeling with VHDL.* Prentice Hall, 1989.

[Aro90] Tejkumar Arora. *The Formal Verification of the VIPER Microprocessor: EBM to Microcode Level.* Master's thesis, University of California, Davis, 1990.

[Bar84] Harry G. Barrow. VERIFY: a program for proving correctness of digital hardware designs. *Artificial Intelligence*, 24:437–491, 1984.

[BC87] M. C. Browne and E. M. Clarke. *SML*–a high level language for the design and verification of state machines. In D. Borrione, editor, *IFIP Working Conference: From HDL Descriptions to Guaranteed Circuit Designs*, Elsevier Science Publishers (North-Holland), 1987.

[BM79] R. S. Boyer and J S. Moore. *A Computational Logic.* Academic Press, 1979.

[BT89] Alexandre Bronstein and Carolyn L. Talcott. *Formal Verification of Pipelines based on String-Functional Semantics.* Technical Report, Department of Computer Science, Stanford University, 1989.

[CCLO88] S. D. Crocker, E. Cohen, S. Landauer, and H. Orman. Reverification of a microprocessor. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 166–176, April 1988.

[CGM87] Albert Camilleri, Mike Gordon, and Tom Melham. Hardware verification using higher order logic. In D. Borrione, editor, *From HDL Descriptions to Guaranteed Correct Circuit Designs*, Elsevier Scientific Publishers, 1987.

[Chu40] Alonzo Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5, 1940.

[Clo87] W. F. Clocksin. Logic programming and digital circuit analysis. *The Journal of Logic Programming*, 4:59–82, 1987.

[Coh88a] Avra Cohn. *Correctness Properties of the Viper Block Model: The Second Level.* Technical Report 134, University of Cambridge Computer Laboratory, May 1988.

[Coh88b] Avra Cohn. A proof of correctness of the viper microprocessor: the first level. In G. Birtwhistle and P. Subrahmanyam, editors, *VLSI Specification, Verification, and Synthesis*, pages 27–72, Kluwer Academic Publishers, 1988.

[Coh89] Avra Cohn. The notion of proof in hardware verification. *Journal of Automated Reasoning*, 5:127–139, 1989.

[Con86] Robert L. Constable. *Implementing Mathematics with the NUPRL Proof Development System.* Prentice Hall, 1986.

[Cro77] Stephen D. Crocker. *A Formalism for Representing Segments of Computation.* PhD thesis, University of California, Los Angeles, 1977.

[Cul88] W. J. Cullyer. Implementing safety critical systems: the viper microprocessor. In G. Birtwhistle and P.A Subrahmanyam, editors, *VLSI Specification,Verification, and Synthesis*, pages 1–25, Kluwer Academic Press, 1988.

[EHD88] *EHDM Specification and Verification System: User's Guide,* Version 4.1. SRI International Computer Science Laboratory, 1988.

[GB89] Brian Graham and Graham Birtwhistle. Formalizing the design of an SECD chip. In *Proceedings of the Mathematical Sciences Institute Workshop on Hardware Specification, Verification, and Synthesis:Mathematical Aspects*, Cornell University, July 1989.

[GMW79] M. Gordon, R. Milner, and C. Wadsworth. *Edinburgh LCF. Lecture Notes in Computer Science No. 78*, Springer Verlag, 1979.

[Gog84] J. Goguen. Parameterized programming. *IEEE Transactions on Software Engineering*, SE-10(5):528–543, September 1984.

[Gor80] Michael J.C. Gordon. The denotational semantics of sequential machines. *Information Processing Letters*, 10(1), February 1980.

[Gor83] Michael J.C. Gordon. *Proving a Computer Correct.* Technical Report 41, Computer Lab, University of Cambridge, 1983.

[Gor86] Michael J.C. Gordon. Why higher-order logic is a good formalism for specifying and verifying hardware. In G. J. Milne and P. A. Subrahmanyam, editors, *Formal Aspects of VLSI Design*, pages 153–177, Elsevier Science, 1986.

[Gor88] Michael J.C. Gordon. Hol: a proof generating system for higher-order logic. In G. Birtwhistle and P.A Subrahmanyam, editors, *VLSI Specification,Verification, and Synthesis*, Kluwer Academic Press, 1988.

[GW88] J. Goguen and T. Winkler. *Introducing OBJ3*. Technical Report SRI-CSL-88-9, SRI International, August 1988.

[Hen80] P. Henderson. *Functional Programming; Applications and Implementation*. Prentice–Hall, London, 1980.

[Her88] John Herbert. Temporal abstraction of digital designs. In G.J. Milne, editor, *The Fusion of Hardware Design and Verification, Proceedings of the IFIP WG 10.2 International Working Conference, Glasgow, Scotland*, North–Holland, 1988.

[Hun87] W. A. Hunt. The mechanical verification of a microprocessor design. In D. Borrione, editor, *From HDL Descriptions to Guaranteed Correct Circuit Designs*, North-Holland, 1987.

[Joy88] Jeffrey J. Joyce. Formal verification and implementation of a microprocessor. In G. Birtwhistle and P.A Subrahmanyam, editors, *VLSI Specification, Verification, and Synthesis*, Kluwer Academic Press, 1988.

[Joy89a] Jeffrey J. Joyce. *Multi–Level Verification of Microprocessor–Based Systems*. PhD thesis, Cambridge University, December 1989.

[Joy89b] Jeffrey J. Joyce. *Totally Verified Systems: Linking Verified Software to Verified Hardware*. Technical Report, University of Cambridge, 1989.

[Kat85] Manolis G. H. Katevenis. *Reduced Instruction Set Computer Architectures for VLSI*. MIT Press, 1985.

[Lan64] P. J. Landin. The mechanical evaluation of expressions. *Computer Journal*, 6(4):308–320, 1964.

[Loe89] Paul Loewenstein. Reasoning about state machines in higher–order logic. In M. Leeser and G. Brown, editors, *Workshop on Hardware Specification, Verification, and Synthesis: Mathematical Aspects*, Springer-Verlag, 1989.

[Mar87] L. Marcus. *SDVS 6 Users' Manual*. Aerospace Report No. ATR-86A(2778)-4, 1987.

[MCL84] L. Marcus, S. D. Crocker, and J. R. Landauer. Sdvs: a system for verifying microcode correctness. In 17^{th} *Microprogramming Workshop*, pages 246–255, oct 1984.

[Mel88] Thomas Melham. Abstraction mechanisms for hardware verification. In G. Birtwhistle and P. A. Subrahmanyam, editors, *VLSI Specification, Verification and Synthesis*, Kluwer Academic Publishers, 1988.

[Plo81] Gordon Plotkin. *A Structural Approach to Operational Semantics.* Technical Report DAIMI FN-19, University of Aarhus, Denmark, September 1981.

[Pyg85] C. Pygott. *Formal Proof of Correspondence Between a Hardware Module and its Gate–level Implementation.* Memo No. 85012, Royal Signals and Radar Establishment, June 1985.

[Tas89] John P. Van Tassel. *The Semantics of VHDL with VAL and HOL: Towards Practical Verification Tools.* Master's thesis, Department of Computer Science and Engineering, Wright State University, 1989.

[TH89] John P. Van Tassel and David Hemmendinger. Toward formal verification of vhdl specifications. In Luc Claesen, editor, *Applied Formal Methods for Correct VLSI Design*, Elsevier Science Publishers, Leuven, Belgium, November 1989.

[Wei86] Daniel Weise. *Formal Multilevel Hierarchical Verification of Synchronous MOS VLSI Circuits.* PhD thesis, Massachusetts Institute of Technology, August 1986.

[Win90a] Phillip J. Windley. A hierarchical methodology for the verification of microprogrammed microprocessors. In *Proceedings of the IEEE Symposium on Security and Privacy*, May 1990.

[Win90b] Phillip J. Windley. *The Verification of AVM–1.* Technical Report CSE-90-21, University of California, Davis, Division of Computer Science, 1990.

Appendix A

Abstract Theories in HOL

A theory is a set of types, definitions, constants, axioms and parent theories. Logics are extended by defining new theories. A generic or abstract theory (we will use the two terms interchangeably) is parameterized so that some of the types and constants defined in the theory are undefined inside the theory except for their syntax and an algebraic specification of their semantics. Group theory provides an example of a generic theory from mathematics. The multiplication operator is undefined except for its syntax (a binary operator on type `:group`) and a semantics given in terms of the axioms of group theory.

Generic theories are useful because they provide proofs about generic structures which can then be used to reason about specific instances of the structure. In groups, for example, after showing that addition over the integers satisfies the axioms of group theory, we can use the theorems from group theory to reason about addition on the integers.

This appendix describes the use and documents the implementation of generic theories in the HOL theorem prover. The abstract theory package was not designed to be a final implementation of generic theories in HOL, but rather is seen as an interim solution until the system can be modified to provide them as full–fledged objects. The appendix describes how to use abstract theories in HOL and briefly describes the implementation of abstract theories in HOL. The implementation is interesting because it displays the flexibility of the HOL theorem prover.

A.1 Abstract Theories.

The key components of an generic theory are a set of abstract objects and a set of abstract operations. This *abstract representation* is unspecified, that is, we don't know (inside the theory) what the objects and operations mean. Their meaning is specified through a set of predicates that define relationships among members of the abstract representation. The theory describes a model. Any structure with objects and operations that satisfy the predicates is a homomorphism of that model.

The theory obligations axiomatize the theory. Using the obligations as axioms allows us to prove theorems of interest about the abstract objects and operations. Our goal is to be able to instantiate the generic theory with a concrete representation meeting the obligations. The instantiation specializes the generic theorems, resulting in a set of theorems about the concrete representation. The concrete representation is an instance of the generic theory and represents a *member* of the class of abstract objects that it describes.

An generic theory consists of three parts:

1. An *abstract representation* where the abstract operations and their types are declared.
2. A list of *theory obligations* defining the relationships between members of the abstract representation.
3. A collection of *abstract theorems* which are proven with respect to the obligations.

A.1.1 Using the Abstract Theory Package.

The remainder of this section describes the functions in the abstract theory package. Before beginning a abstract theory, the ML file `abstract` should be loaded. This sets up the commands in the abstract package and modifies some of the standard HOL commands to support its operation.

One declares a new abstract theory in the same way that one declares a standard theory, using `new_theory`. One is free to use any of the standard HOL commands for manipulating a draft theory in their usual manner. For example, definitions can be done in the usual way using `new_definition`.

A.1.2 Abstract Representations.

The abstract representation describes the abstract objects and operators in the generic theory. The abstract theory package defines `new_abstract_representation` for declaring the abstract representation. The function is applied to a list of pairs. The first member of the pair is a string giving the name of the abstract object and the second member of the pair is the type of the operator. There is no limit on the length of the list.

The system does not require that abstract objects be specifically declared. We represent abstract objects as type variables in HOL (denoted by a prepended asterisk). Since HOL does not require that type variables be declared, we are free to use them wherever we wish. The declaration of abstract objects is implicit, being the set of type variables occurring in the abstract representation.

The result of declaring a new abstract representation is a list of definitions. The definitions can usually be ignored. There is one exception: After declaring the abstract representation, you should apply the function `make_inst_thms` to the resulting list of definitions if you intend to use the instantiation functions described later. This will automatically prove a lemma about each definition for use during any subsequent instantiations. This is not done in the declaration of the representation to save time when abstract theories are being drafted.

In order to use the abstract representation, we will need to know its type. The abstract package provides a function for determining the type of an abstract representation. The ML function `abstract_type` is applied to two strings. The first is the name of the abstract theory defining the representation and the second is the name of *any* of the objects in the representation.

When one defines a constant in the abstract theory, by convention, the first argument to the constant will be a variable with the same type as the abstract representation. This variable must, in turn be the first argument to any of the abstract constants from the abstract representation used in the definition. Later, during instantiation, the definition will be applied to a concrete representation and the instantiation functions will replace the abstract constants with the appropriate concrete constants in the instantiation.

A.1.3 Theory Obligations.

The theory obligations are declared using the ML function `theory_obligations`. The function is applied to a list of HOL terms. Each term should represent an axiom concerning the abstract objects. These obligations will be available for use in the draft theory. The system will automatically add them to the assumption list when the standard HOL commands for declaring goals and proving theorems, such as `set_goal`, are used. The HOL command `close_theory` closes the current draft and after it has been issued, the system no longer automatically appends the theory obligations to the assumption list.

One note on writing theory obligations: the representation variable and any variables with abstract types that are to be instantiated must not be included in the universally quantified variables of any of the theory obligations.

A.1.4 Instantiating Theories.

One makes use of a generic theory by instantiating it. The first step is to make the generic theory a parent of the draft theory using the ML function `new_parent`.

HOL theories differentiate between definitions and theorems. In order to instantiate a theory we need to be able to instantiate both. Instantiating definitions is the easier of the two. By convention, the first variable in an abstract definition has the same type as the representation. To use this definition, the following steps are performed:

1. Make an auxiliary definition that uses the abstract definition and applies it to a concrete representation (an ordered n–tuple containing, in order, a concrete constant to instantiate each abstract constant in the abstract representation).

2. Use the ML function `instantiate_abstract_definition` to produce an instance of the abstract definition. This function is applied to three arguments. The first is the name of the abstract theory where the abstract definition was defined. The second is the name of the abstract definition. The third is the name of the definition from step (1).

The result of this instantiation is a theorem that defines a concrete instance of the abstract definition and makes no reference to the abstract definition.

As part of drafting an abstract theory, one normally proves theorems about the abstract representation using the theory obligations as axioms. In addition, the theorems may make use of some of the abstract definitions in the abstract theory. When the abstract theory is used, we instantiate the theorems in it so that the theory obligations are discharged and the new concrete theorems stand on their own.

The ML function `instantiate_abstract_theorems` instantiates all of the abstract theorems in the theory. The function takes four arguments:

1. `th` – the name of the abstract theory where theorems reside.

2. `axiom_list` – a list of theorems that satisfy the theory obligations and thereby discharge the antecedents of the abstract theorems.

3. `tm_list` – a list of term pairs that instantiate variables with concrete representations. The first term in the pair is the variable to instantiate and the second is the concrete representation.

4. `base` – a name to prepend to newly created theorems. This is done to avoid name clashes with existing theorems.

A.2 Implementational Considerations.

This section briefly describes the principles behind the implementation of abstract theories used in this report. The section is not intended to provide a full discussion of the implementation, but rather to describe how the facilities of HOL were used to reason about generic theories. The ML code that implements the abstract theory package is contained in Section A.4.

There are two features of HOL that allowed generic theories to be implemented without changing the HOL system. The first is higher–order logic which is necessary for implementing abstract representations. The second is the meta–language ML which allowed the theory obligations to be declared and used in the proofs.

The idea of using n–tuples of functions to implement abstract representations in HOL is due to Jeff Joyce [Joy89a]. The idea is that abstract types can be represented by type variables and that abstract functions can be represented as selectors on a n–tuple. Each member of the tuple has the type of the corresponding member in the abstract representation. Since the abstract functions are selectors on the representation variable, we can use them in an abstract representation by applying them to the representation (thus producing the right type) and we can instantiate them by applying them to a n–tuple containing concrete functions.

For example, suppose we declare an abstract representation containing three functions as follows:

```
new_abstract_representation
    [
     (`f`,":*t1->*t2")
      ;
     (`g`,":*t2->*t3")
      ;
     (`h`,":*t3->*t1")
      ;
    ];;
```

The abstract package described in this report creates a representation with the type

```
    :(*t1->*t2 # *t2->*t3 # *t3->*t1)
```

The package also makes the definitions:

$\vdash_{def}$ $\forall$ `rep.  f rep = FST rep`

$\vdash_{def}$ $\forall$ `rep.  g rep = FST (SND rep)`

$\vdash_{def}$ $\forall$ `rep.  h rep = SND (SND rep)`

and proves the theorems

$\vdash_{def}$ $\forall$ `x y z.  f (x,y,z) = x`

$\vdash_{def}$ $\forall$ `x y z.  g (x,y,z) = y`

$\vdash_{def}$ $\forall$ `x y z.  h (x,y,z) = z`

The implementation of theory obligations depends on the use of sequents as the underlying structure for goals and theorems and the meta–language used to program the system. A sequent is a pair where the first member is a list of terms representing the assumption list and the second member is a term representing the conclusion. Goals are represented as sequents and transformed into theorems (which have the same structure) when they have a proof.

The HOL system has three ML functions that are used for proof and goal management.

```
set_goal: goal -> void

TAC_PROOF: (goal # tactic) -> void

prove_thm: (string # goal # tactic) -> void
```

The first sets a goal in the proof management system for subsequent interactive proof. The second proves the goal using the tactic (if it can). The third solves the goal using the tactic and saves the resulting theorem in the draft theory using the name given in the string.

The function `theory_obligations` takes a single argument, a list of terms. This list of terms is saved in a variable. When one of the above ML functions for setting up a goal is called, the list of terms in the theory obligations is appended to the (usually null) list of terms in the assumption list of the goal. These terms appear on the assumption list and can be used to prove the goal. The theory obligations remain on the assumption list of any resulting theorem, serving as a reminder that the theorem cannot stand on its own.

A.3 Limitations.

There are several limitations to the `abstract` package that should be fixed if this package is not superseded by a full-fledged abstract theory implementation in the HOL system.

- The entire abstract theory must be declared in one file. The primary problem is that there is no way for a theory to know whether or not it is an abstract theory. The theory obligations from an abstract parent are not available in an abstract child. This could be fixed by storing them in the theory and creating a special `new_parent` command for recalling them.
- The package only supports goal–directed proofs. Forward proof styles could be supported with a little more work. Some of the things necessary for supporting

multiple file abstract theories mentioned in the previous item would be used here as well.

A.4 Implementing Abstract Theories in HOL.

This appendix provides the complete source code for the implementation of abstract theories in HOL.

```
%-----------------------------------------------------------------

   File:         abstract.ml

   Description:

   Defines ML functions for defining generic structures.

   Author:       (c) P. J. Windley 1989
   Date:         29 DEC 89

-----------------------------------------------------------------%

let new_abstract_representation lst = (
    letrec make_type lst =
       null lst => ":one"
                 | let rest = make_type (tl lst) in
                   ": ^(snd (hd lst)) # ^rest" in
    let rep_type = make_type lst in
    letrec make_definitions lst n  =
        null lst => nil |
        let f = (make_definitions (tl lst) (n+1)) and
            name = (fst (hd lst)) and
            nterm = (int_to_term n) in
        letrec make_tuple_term n =
            (n=0) => "rep:^rep_type" |
            let f = make_tuple_term (n-1) in
            "SND ^f" in
        let tuple_term = "FST ^(make_tuple_term (n-1))" in
        let op_type = ":^rep_type -> ^(snd(hd lst))" in
         (name,
          "! rep:^rep_type.^(mk_var(name, op_type)) rep =
           ^tuple_term") . f in
    map new_definition (make_definitions lst 1))
    ? failwith `new_abstract_representation` ;;

let abstract_type th const = (
   hd(snd(dest_type
      (snd(dest_const(hd(filter (\x. (fst o dest_const) x = const)
                             (constants th))))))))
   ? failwith `abstract_type`;;

let make_inst_thms th_list = (
    let is_FST_term t =
        fst(dest_const(fst(strip_comb t))) = `FST` in
    let is_SND_term t =
        fst(dest_const(fst(strip_comb t))) = `SND` in
    let FST_CONV t =
```

```
        if is_FST_term t then
           let op,pr = dest_comb t in
           let op,[t1;t2] = strip_comb pr in
           SPECL [t1;t2] (
              INST_TYPE [((type_of t1),":*");
                         ((type_of t2),":**")] FST)
        else fail in
    let SND_CONV t =
        if is_SND_term t then
           let op,[t1;t2] = strip_comb (snd (dest_comb t)) in
           SPECL [t1;t2] (
              INST_TYPE [((type_of t1),":*");
                         ((type_of t2),":**")] SND)
        else fail in
    let make_inst_thm th = (
        letrec MY_DEPTH_CONV conv t =
           (SUB_CONV (MY_DEPTH_CONV conv) THENC (TRY_CONV conv)) t in
        let rep_type =
             (snd(dest_var (rand (rand (rator (concl (SPEC_ALL th))))))) in
        letrec make_spec_term tp n = (
           if tp = ":one" then "y:one" else
           let new_types = (snd(dest_type tp)) in
           let new_term = make_spec_term (hd(tl new_types)) (n+1) in
           let term_str = concat `elm` (string_of_int n) in
           "^(mk_var (term_str,hd new_types)), ^new_term")
           ? failwith `make_spec_term` in
        let spec_th = SPEC (make_spec_term rep_type 0) th in
        CONV_RULE ((RAND_CONV (MY_DEPTH_CONV SND_CONV)) THENC
                   (RAND_CONV FST_CONV)) spec_th) in
    let save_thm_list basename th_lst = (
        letrec process_lst th_lst n =
           if null th_lst then [] else
           let name = concat basename (string_of_int n) in
           (save_thm (name,hd th_lst)). process_lst (tl th_lst) (n+1) in
        (process_lst th_lst 0)) in
    save_thm_list (current_theory()) (map make_inst_thm th_list))
    ? failwith `make_inst_thms`;;

let get_abstract_thms th_name =
    letrec retrieve_thms n = (
       let name = (concat th_name (string_of_int n)) in
       (theorem th_name name) . (retrieve_thms (n+1))) ? [] in
    retrieve_thms 0;;

let instantiate_abstract_definition th_name defn1 defn2 =
    let th_list = get_abstract_thms th_name in
    (ONCE_REWRITE_RULE th_list
       (ONCE_REWRITE_RULE [definition th_name defn1] defn2));;

%------------------------------------------------------------------

 instantiate_abstract_theorems

 th         -- abstract theory where theorems reside
```

```
 axiom_list -- list of theorems that discharge antecedents in
               abstract theorems

 tm_list    -- list of term pairs that instantiate free variables.
               The first term in the pair is the variable to
               instatiate and the second is the instantiation.

 base       -- name to prepend to newly created theorems

----------------------------------------------------------------%

let instantiate_abstract_theorems th axiom_list tm_list base =
   let abs_thms = get_abstract_thms th in
   letrec add_one_at_end p = (
      let f,s = dest_pair p in
      mk_pair(f,add_one_at_end s) ? mk_pair(f,mk_pair (s,"@x:one.F"))) in
   letrec build_type_list tm_pair_list =
      if null tm_pair_list then [] else
      let (gen_tm,spec_tm) = hd(tm_pair_list) in
      let alt_spec_tm = (add_one_at_end spec_tm) ? spec_tm  in
      let type_list = snd((match gen_tm spec_tm)?
                          (match gen_tm alt_spec_tm)) in
      type_list @ (build_type_list (tl tm_pair_list)) in
   let type_list = build_type_list tm_list in
   letrec GEN_FROM_LIST tm_pair_list thm = (
      if null tm_pair_list then thm else
      let (gen_tm,spec_tm) = hd(tm_pair_list) in
      let gen_thm = (GEN gen_tm thm) in
      GEN_FROM_LIST (tl tm_pair_list) gen_thm)
      ? thm in
   letrec SPEC_FROM_LIST tm_pair_list thm = (
      if null tm_pair_list then thm else
      let (gen_tm,spec_tm) = hd(tm_pair_list) in
      let alt_spec_tm = (add_one_at_end spec_tm) ? spec_tm  in
      let spec_thm = ((SPEC spec_tm thm) ?
                      (SPEC alt_spec_tm thm)) in
      SPEC_FROM_LIST (tl tm_pair_list) spec_thm)
      ? thm in
   let multi_mp thm alist =
       letrec multi_mp_aux thm alist =
          if null alist then thm else
          let new_thm = PROVE_HYP (hd alist) thm in
          multi_mp_aux new_thm (tl alist) in
       DISCH_ALL (multi_mp_aux (UNDISCH_ALL thm) alist) in
   let instantiate_one_thm thm = (
       let undisch_thm = (DISCH_ALL thm) in
       let gen_thm = GEN_FROM_LIST tm_list undisch_thm in
       let inst_thm = INST_TYPE (build_type_list tm_list) gen_thm in
       let spec_thm = SPEC_FROM_LIST (rev tm_list) inst_thm in
       let new_thm =
          (PURE_REWRITE_RULE abs_thms spec_thm) in
       multi_mp new_thm axiom_list) ? thm in
   letrec generate_names th_name n =
```

```
      let name = concat th_name (string_of_int n) in
      (theorem th_name name);(name.(generate_names th_name (n+1))) ? [name] in
   let th_thms =  (theorems th) in
   let real_thms = subtract th_thms
      (filter (\x:(string#thm). (mem (fst x) (generate_names th 0)))
              th_thms) in
   let new_base = concat base `_` in
   letrec make_save_list nt_list =
      if null nt_list then [] else
      let name,thm = hd(nt_list) in
      (concat new_base name,instantiate_one_thm thm) .
         (make_save_list (tl nt_list)) in
   make_save_list real_thms;;



% set up obligation lists %

letref theory_obligation_list = []:(term)list;;

let new_theory_obligations tm_list =
   theory_obligation_list := tm_list;;


%----------------------------------------------------------------
 Modify the standard commands so that they know about obligation
 lists.
---------------------------------------------------------------%
%Prove and store a theorem%
let prove_thm(tok, w, tac:tactic) =
    let gl,prf = tac (([] @ theory_obligation_list),w) in
    if null gl then save_thm (tok, prf[])
    else
       (message (`Unsolved goals:`);
        map print_goal gl;
        print_newline();
        failwith (`prove_thm -- could not prove ` ^ tok));;


% TAC_PROOF (g,tac) uses tac to prove the goal g                      %
let TAC_PROOF : (goal # tactic) -> thm =
    set_fail_prefix `TAC_PROOF`
       (\(g,tac).
           let new_g = ((fst g) @ theory_obligation_list,snd g) in
           let gl,p = tac new_g in
           if null gl then p[]
           else (
              message (`Unsolved goals:`);
              map print_goal gl;
              print_newline();
              failwith `unsolved goals`));;

%Set the top-level goal, initialize %
let set_goal g =
```

```
          let new_g = ((fst g) @ theory_obligation_list,snd g) in
          change_state (abs_goals (new_stack new_g));;

let g = \t. set_goal([],t);;

let close_theory_orig = close_theory;;

let close_theory x =
   theory_obligation_list := [];
   close_theory_orig x;;

let new_theory_orig = new_theory;;

let new_theory x =
   theory_obligation_list := [];
   new_theory_orig x;;
```

Appendix B

The Organization of the Proof

This appendix presents the organization of the proof of *AVM–1* in HOL. The appendix discusses the overall proof organization, gives a description of the theories making up the proof and gives some measurements of the complexity of the proof.

B.1 Proof organization

The proof for *AVM–1* contains more than 25 theories. This section presents the general proof organization (the hierarchy of theories) and briefly describes the contents of each theory.

Figure B.1 shows how the main theories of the proof of *AVM–1* are related. This hierarchy shows `avm.th` as the child theory of a long ancestry that follows the hierarchical decomposition discussed in the body of this dissertation. The picture is not complete; there are many theories not shown. For example, `aux_def.th` is the ancestor of almost every theory in the proof.

The rest of this section gives a taxonomy of the major theories in the proof of *AVM–1*.

Generic Interpreters. The generic interpreter theories include the synchronous model, the temporal abstraction theory, and the asynchronous model.

- **gen_I_sync.th** — Defines and verifies a synchronous version of the generic interpreter theory.
- **time_abs.th** — Defines a temporal abstraction function and proves several useful lemmas concerning it.
- **gen_I.th** — Contains the generic definition of an interpreter used in the definition and proof of the various levels in *AVM–1*.

Auxiliary Theories. There are a number of auxiliary theories that are used throughout the proof of *AVM–1*.

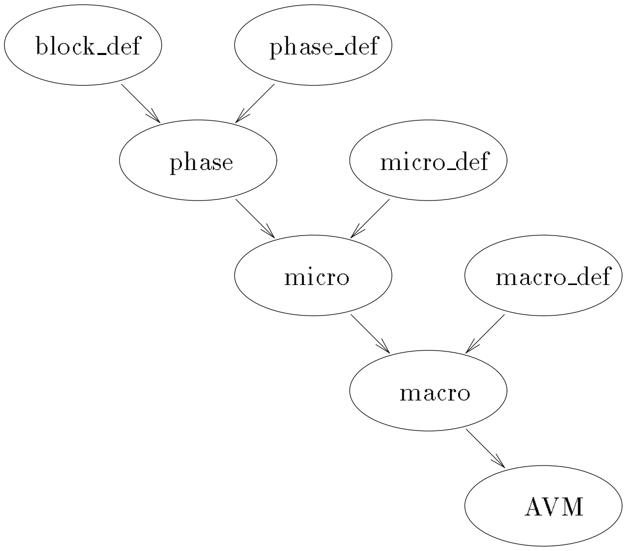

Figure B.1: The theory hierarchy for the proof of *AVM–1*.

- **aux_defs.th** — Contains the abstract definition for n–bit words. The definition is accomplished using the functions in `abstract.ml`, the ML code for producing abstract theories.
- **aux_thms.th** — Contains auxiliary definitions and theorems. The theory is an ancestor of many of the main theories in the proof.
- **jump_def.th** — Contains the definition of the jump condition logic that is used at every level.
- **regs_def.th** — Contains the definition of the register file. Several distinguished registers are defined and the function for updating the register file is given.

The Electronic Block Model. The electronic block model description depends on a number of theories. The definition makes use of a generic ALU that is subsequently instantiated to define the ALU used in *AVM–1*. The shifter and microprogram counter are also defined separately.

- **mux16_def.th** — Contains the definition of a 16 input multiplexor that is used in the definition of the generic ALU theory.
- **gen_alu.th** — Contains the abstract definition and verification of a 16 function ALU.
- **alu_def.th** — Contains the instantiation of the generic ALU theory presented in the last section for a specific set of functions. The correctness result is meaningless since the modules used to implement the functions are null modules.

This does not affect the validity of the proof presented here since only the definition is used in subsequent theories. A number of theorems about the ALU's output are proven here and are used in subsequent proofs.

- **shifter_def.th** — Contains the definition of a 4 function shifter that is used in defining the electronic block model. A number of theorems about the shifter's output are proven here and are used in subsequent proofs.
- **mpc_def.th** — Contains the definition of the microprogram counter unit that is used in the definition of the electronic block model and the phase–level.
- **mpc_def.th** — Contains the definition of the state selectors for the electronic block model.
- **block_def.th** — This theory contains the definition of the electronic block model. The theory contains the definition of most of the blocks used to construct the electronic block model.

The Phase–Level. This section presents the theories that define the phase–level interpreter. Also presented is the theory that verifies the phase–level interpreter with respect to the electronic block model.

- **ucode_aux.ml** — Contains the ML code that defines the microcode assembler. No theory is created; the assembler is an ML program that creates the appropriate terms for a given program statement.
- **ucode_def.th** — Defines the type for the microcode as well as a number of selector functions that return the various fields that make up a microinstruction.
- **phase_def.th** — Defines the abstract behavior of the 4 phase–level instructions and gives several auxiliary definitions used in instantiating the abstract interpreter theory.
- **phase.th** — Contains the correctness result for the phase–level. The result is obtained by instantiating the generic interpreter theory contained in `gen_I.th`.

The Micro–Level. This section presents the theories that define the micro–level interpreter. Also presented is the theory that verifies the micro–level interpreter with respect to the phase–level interpreter.

- **micro_def.th** — Defines the abstract behavior of the 64 micro–level instructions and gives several auxiliary definitions used in instantiating the abstract interpreter theory.
- **uinst_def.th** — Defines the microinstructions and combines them together into the microrom.

- **micro.th** — Contains the correctness result for the micro–level. The result is obtained by instantiating the generic theory `gen_I.th`.

The Macro–Level. This section presents the theories that define the macro–level interpreter. Also presented is the theory that verifies the macro–level interpreter with respect to the micro–level interpreter.

- **macro_def.th** — Defines the abstract behavior of the 32 macro–level instructions and gives several auxiliary definitions used in instantiating the abstract interpreter theory.
- **macro.th** — Contains the correctness result for the macro–level. The result is obtained by instantiating the generic theory `gen_I.th`.

The Final Result. This section presents the theory that proves that *AVM–1* is correct. The theory is the descendant of all of the theories presented earlier.

- **avm.th** — Contains the correctness result for the microprocessor. The final result is obtained by combining the correctness results from `phase.th`, `micro.th`, and `macro.th`.

B.2 Proof Metrics.

Table B.1 presents the run–times for the various theories in the proof on a SPARC-Station with 16 Mbytes of memory. The times are CPU seconds. The table also gives the number of primitive inferences required to run the corresponding ML script in HOL. We were using version 1.11 of HOL built using the Austin Kyoto Common Lisp compiler.

The total time to run the proof was 208029.1 CPU seconds, or nearly 58 CPU hours. The proof took almost a week of elapsed time because the core images were quite large (as high as 29 Mbytes) and caused the operating system to thrash when garbage collecting.

There are several files in the table that were not discussed in the last section. Due to size limitations, the files `mk_mic_x1.ml` and `mk_mic_x2.ml` were broken out of `mk_micro.ml` and `mk_mac_I.ml`, `mk_mac_1.ml`, and `mk_mac_2.ml` were broken out of `mk_macro.ml`.

Table B.1: Script run–times on a SPARCStation with 16M of memory.

File Name	*Time (CPU sec.)*	*Inferences*
def_aux.ml	3070.7	88
mk_aux.ml	1117.5	33852
def_regs.ml	41.0	14
def_jump.ml	50.7	4
def_macro.ml	2373.5	84
mk_time.ml	126.8	7256
mk_I.ml	229.9	11727
def_micro.ml	7063.6	48460
def_mpc.ml	6.4	4
def_ucode	115.6	50
def_phase.ml	915.2	32
def_mux16.ml	344.2	29211
mk_gen_alu.ml	8038.4	101155
def_alu.ml	2325.3	70815
def_shift.ml	129.0	2891
def_select.ml	1969.0	43903
def_block.ml	1316.0	14738
mk_phase.ml	12818.4	355161
def_uinst	568.5	107
mk_mic_x1.ml	54846.2	1589683
mk_mic_x2.ml	51300.6	1500604
mk_micro.ml	13505.3	295744
mk_mac_I.ml	688.3	3985
mk_mac_1.ml	16774.1	389738
mk_mac_2.ml	20256.1	457606
mk_macro.ml	7247.9	200120
mk_avm.ml	790.9	10031
	208029.1	5167063