Formal Modeling and Verification of Microprocessors

Phillip J. Windley

Abstract— **Formal verification has long been promised as a means of reducing the amount of testing required to ensure correct VLSI devices. Verification requires at least two mathematical models: one that describes the structure of a computer system and another that models its intended behavior. These models are called *specifications*. Verification is a mathematical analysis showing that the behavior follows from the structure. Formal verification of microprocessor designs has been quite successful. Indeed, several verified microprocessors have been presented in the literature, and one microprocessor where formal modeling has been applied is commercially available. These efforts were virtuoso performances—largely academic exercises carried out by experts in logic and specification.**

This paper presents a methodology for microprocessor verification that significantly reduces the learning curve for performing verification. The methodology is formalized in the HOL theorem-proving system. The paper includes a description of a large case study performed to evaluate the methodology.

The novel aspects of this research include the use of abstract theories to formalize hardware models. Because our model is described using abstract theories, it provides a framework for both the specification and the verification. This framework reduces the number of ad hoc modeling decisions that must be made to complete the verification. Another unique aspect of our research is the use of hierarchical abstractions to reduce the number of difficult lemmas in completing the verification. Our formalism frees the user from directly reasoning about the difficult aspects of modeling the hierarchy, namely the temporal and data abstractions.

We believe that our formalism, coupled with case studies and tools, allows microprocessor verification to be done by engineers with relatively little experience in microprocessor specification or logic. We are currently testing that hypothesis by using the methodology to teach graduate students formal microprocessor modeling.

I. Introduction

COMPUTERS are being used with increasing frequency in areas in which the correct implementation of the computer hardware is critical. Testing has traditionally been used to exclude faults in computers; however, the effectiveness of testing is limited by the combinatorial explosion inherent in any testing technique. The limitations of testing, coupled with the ever-increasing size of VLSI devices, have led to a search for alternatives to testing, such as mathematical modeling and analysis.

Manuscript received July 29, 1991; revised December 2, 1992; July 28, 1993. This work was supported by NASA under Space Engineering Research Center Grant NAGW-1406 and under Boeing Contract NAS1-18586, Task Assignment No. 3, with NASA-Langley Research Center.

The author is with the Laboratory for Applied Logic, Department of Computer Science, Brigham Young University, Provo, UT 84602-6576 USA; E-mail: windley@cs.byu.edu.

IEEE Log Number 9407556.

Formal models of VLSI designs are usually called *specifications*; specifications provide a concise description of the behavior of the device that can be used by design engineers, layout technicians, production engineers, test engineers, technical writers, and users. The application of symbolic mathematical analysis to these models is usually called *verification*.

Verification is largely an exercise in demonstrating that a design has certain properties. The primary property that concerns us is functional correctness; that is, showing that a design has an intended behavior. This paper is largely concerned with verifying functional correctness, but other work by the author has been aimed at using specifications to demonstrate, for example, the integrity of supervisory mode in a RISC-like microprocessor [20] and the correctness of rules used in instruction stream reordering [22].

Correctness verification uses at least two descriptions of a system: one that describes *how* the circuit is constructed, called the *structural specification*, and one that describes *what* the circuit is supposed to do, called the *behavioral specification*. Correctness is shown by demonstrating through mathematical proof that the former implies the latter. Design faults are discovered as part of demonstrating correctness and are corrected as the verification proceeds. Thus, verification can be viewed as part of the design process itself, not as an *ex post facto* process that gives a seal of approval. Typically, some sort of mechanical proof tool is used in conjunction with the verification to reduce the tedium associated with manipulating large specifications.

Treating microprocessor design formally can be a difficult task. Avra Cohn, in [6], describes her specification of VIPER's EBM 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 statement not only describes the difficulties of developing formal specifications from the kinds of informal descriptions commonly in use, but it also alludes to the inadequacies of those descriptions. After the specification is complete, verifying that the implementation meets the behavioral specification

is equally arduous, sometimes requiring the proof of hundreds of multipage theorems.

Every microprocessor verification done to date has been a virtuoso performance, carried out by experts in logic, specification, and mechanical reasoning. In contrast to this, we are striving to make microprocessor verification a viable tool for VLSI design engineers. To that end, this paper presents a methodology for verifying microprocessors. This methodology is embodied in a formalism for the HOL theorem prover, providing tool support for a step-by-step approach to system verification. In addition, we have produced several case studies and are working on additional examples of verified systems for use in instructing engineers in microprocessor verification. The latter part of this paper presents a case study of the specification and verification of a microprocessor using our methodology.

Organization of the Paper: This paper consists of two parts. In the first part, Section II presents a brief introduction to the HOL theorem proving system, Section III contains a mathematical model of interpreters, and Section IV contains a formalization of that model in the HOL theorem-proving system.

The second part of the paper demonstrates the use of the model in a case study involving the specification and verification of a microprocessor called AVM-1. Section 5 presents an introduction to AVM-1, Section VI contains the hierarchical specification of AVM-1 in HOL, Section VII presents the verification of AVM-1, and Section VIII presents our observations about the case study.

II. A Brief Introduction to HOL

To ensure the accuracy of our specifications and proofs, we used a mechanical verification system to develop them. The mechanical system performs syntax and type checking of the specifications and prevents the proofs from containing logical mistakes. The HOL system was selected for this project because it has higher order logic, generic specifications, and polymorphic type constructs. These features directly affect the expressibility of the specification language. Furthermore, HOL is widely available, robust, and has a growing international user base. However, nothing in our work requires that the HOL theorem-proving system be used.

HOL is a general theorem-proving system developed at the University of Cambridge [3, 8] that is based on Church's theory of simple types, or higher order logic [4]. Although Church developed higher order logic as a foundation for mathematics, it can be used for reasoning about computational systems of all kinds. Similar to predicate logic in allowing quantification over variables, higher order logic also allows quantification over predicates and functions, thus permitting more general systems to be described.

HOL is not a fully automated theorem prover, but, it is more than simply a proof checker; it serves as a proof assistant. HOL has several features that contribute to its use as a verification environment.

- Several built-in theories, including booleans, individuals, numbers, products, sums, lists, and trees. These theories build on the five axioms that form the basis of higher order logic to derive a large number of theorems that follow from them.
- 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 be done 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.
- A large collection of tactics to support goal-directed proof. Included in HOL are tactics that rewrite a goal according to some previously proven theorem or definition, remove unnecessary universally quantified variables from the front of a goal, and split equalities into two implicative subgoals.
- A proof management system that records the state of an interactive proof session.
- A metalanguage, ML, for programming and extending the theorem prover. Using the metalanguage, tactics can be combined 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.

For the most part, the notation of HOL is that of standard logic: $\forall, \exists, \wedge, \vee$, etc. have their usual meanings. A few constructs deserve special attention because that are used in this paper.

- HOL types are identified by a prefixed colon. Built-in types include `:bool` and `:num`. Function types are constructed using $\rightarrow$. HOL is polymorphic; type variables are indicated by a type names beginning with an asterisk.
- The HOL conditional statement, written `a` $\rightarrow$ b | c, means "if `a`, then `b`, else `c`".
- The HOL list containing elements `a`, `b`, `c`, and `d` is represented as `[a;b;c;d]`. A list that contains elements with type `x` has the type `: (x) list`, where `x` can be any valid type (including type variables since HOL is polymorphic).
- `EL` is a curried function that accepts two arguments, a number, n, and a list, and returns the nth member of the list.
- Tuples are formed using a comma. Parentheses are only required when the scope of the comma is ambiguous. The function `FST` returns the first member of a tuple and `SND` returns the second.
- The construct `let v1 = expr1 and v2 = expr2 and` $\cdots$ `in` simultaneously defines local variables `v1`, `v2`, etc. with values `expr1`, `expr2`, etc.

III. Formal Microprocessor Modeling

Numerous efforts have been made to formally model microprocessors. The best known of these include J. Joyce's Tamarack microprocessor [12], W. Hunt's FM8501 microprocessor [10], and A. Cohn's VIPER microprocessor [5]. Tamarack is a simple microprocessor with only 8 instructions. FM8501 is larger (roughly the size of a PDP-11) but has

not been implemented (a 32-bit version has been verified and implemented by Hunt *et al.* [11]). Perhaps the most interesting of these is VIPER, since even though VIPER is significantly simpler than today's general-purpose microprocessors, its verification provides a benchmark of the state-of-the-art in microprocessor verification.

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; it is commercially available. VIPER is the first microprocessor intended for commercial use where formal verification was used. However, the verification has not been completed because of the large number of instruction cases and the size of the proofs in each of the cases. This is not to say that the proof could not be completed, but that it could be carried out only at great expense. Recent work on hierarchical specification [18], coupled with the work presented here, has overcome the problems that faced the VIPER verification team, and microprocessors significantly more complicated than VIPER are now within the realm of formal treatment. The case study in Sections V–VII is one example.

The specifications for the microprocessors mentioned above appear very different on the surface; in fact, the specification for FM8501 is even in a different language than the specifications of Tamarack and VIPER. On closer inspection, however, we find that each of them (as well as many others) use the same implicit behavioral model. In general, the model uses a state transition system to describe the microprocessor. We call this model an *interpreter*.

The essence of verification is to relate mathematical models at different levels of abstraction. The rest of this section gives a mathematical definition of the interpreter model and shows how two interpreters are related. In the discussion that follows, and for the rest of the paper, we speak of the "abstract level" and the "concrete level," but these terms are relative; as we move up and down a hierarchy of interpreters, what we call "abstract" at one level will be termed "concrete" with respect to the level above it. As a matter of convention, we will decorate variables that represent the concrete level with primes.

A. Basic Types

The basic types for our model are shown in Table I. In addition to these basic types, we also use the following type constructors: **product**, written $(\alpha \times \beta)$; **coproduct**, (or **sum**) written $(\alpha + \beta)$; and **function**, written $(\alpha \rightarrow \beta)$. An n-tuple is indicated by $(\alpha_1 \times \alpha_2 \times \cdots \times \alpha_{n-1} \times \alpha_n)$.

B. State

At times it is convenient to treat state as an object of type $\mathbf{S}$, where $\mathbf{S}$ is uninterpreted. This allows us to treat state in an abstract manner, even though we may know nothing of its structure or content.

Eventually, we will provide interpretations for $\mathbf{S}$ to model a specific machine. To provide such an interpretation, we represent state using n-tuples. We let $\mathbf{S}_n$ be the domain of n-tuples representing state. These n-tuples have the type

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

where

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

Whether or not $\mathbf{S}$ is interpreted, we write $\mathbf{S} \leq \mathbf{S}'$ to indicate that $\mathbf{S}$ is an abstraction of $\mathbf{S}'$. When $\mathbf{S}$ is an abstraction of $\mathbf{S}'$ there exists a function, $\mathcal{S}$:$\mathbf{S}' \rightarrow \mathbf{S}$. The function $\mathcal{S}$ is called the state abstraction function.

TABLE I
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

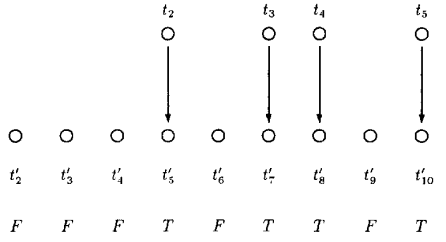


Fig. 1. The function, $\mathcal{F}$, which maps time at one level to another, can be defined in terms of a predicate, $\mathcal{G}$, which is true only when the mapping occurs.

C. Time

In general, different levels in the interpreter hierarchy have different views of time. A temporal abstraction function maps time at the abstract level to time at the concrete level [9], [12], [15]. Fig. 1 shows a temporal abstraction function, $\mathcal{F}$. The circles represent clock ticks. Notice that the number of clock ticks required at the concrete level to produce one clock tick at the abstract level is irregular.

The temporal projection, $\mathcal{F}$, can be defined recursively on time. We define $\mathcal{F}$ in terms of a predicate, $\mathcal{G}$, which is true whenever there is a valid abstraction from the concrete level to the abstract level. In a microprocessor specification, $\mathcal{G}$ is usually a predicate that indicates when the lower level interpreter is at the beginning of its cycle—a condition that is easy to test.

The function $\mathcal{F}$ is defined recursively so that $\mathcal{F}(\mathcal{G}, 0)$ is the first time that $\mathcal{G}$ is true and $\mathcal{F}(\mathcal{G}, (n+1))$ is the next time after time n when $\mathcal{G}$ is true. The resulting function is monotonically increasing. We use $\mathbf{N}$ to represent time. Thus, we define $\mathcal{F}$:$(\mathbf{N} \rightarrow \mathbf{B}) \times \mathbf{N} \rightarrow \mathbf{N}$ such that

$$\forall n\ m \cdot (n > m) \Rightarrow (\mathcal{F}(\mathcal{G}, n) > \mathcal{F}(\mathcal{G}, m))$$

We refer the interested reader to the references given above and to [17] for the details of the temporal abstraction function.

D. State Streams

A state stream, $\mathbf{U}$, is a function from time to state, $\mathbf{N} \rightarrow \mathbf{S}$. We have chosen n-tuples of booleans, bit-vectors, and stores to represent state. 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.

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

where

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

An important part of our theory is the abstraction between state streams at different levels. State stream u is an abstraction of state stream u' (written $(u \preceq u')$ if and only if

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

There are two distinct kinds of abstraction here: the first is a data abstraction; the second is a temporal abstraction.

Using the state abstraction function, $\mathcal{S}$, and the temporal abstraction function, $\mathcal{F}$, we define stream abstraction as follows

$$u \preceq u' \equiv \exists(\mathcal{S}{:}\mathbf{S}' \rightarrow \mathbf{S}) \cdot \exists(\mathcal{F}{:}\mathbf{N} \rightarrow \mathbf{N}) \cdot \mathcal{S} \circ u' \circ \mathcal{F} = u$$

where $\circ$ denotes function composition.

E. Environments

The environment represents the external world; 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 and environment.

At the abstract level, we treat the environment as an uninterpreted type. 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 $\mathcal{E}{:}\mathbf{E}' \rightarrow \mathbf{E}$. When we provide an interpretation for $\mathcal{E}$, we represent the environment using n-tuples of booleans and bit-vectors.

We perform the same kinds of abstraction on the environment as on states. Temporal abstraction is performed as it was for states. We define abstraction for environment streams in the same manner as we defined it for state streams. Thus we write $e \preceq e'$ when e is an stream abstraction of e' and define stream abstraction for environment streams as follows:

$$e \preceq e' \equiv \exists(\mathcal{E}{:}\mathbf{E}' \rightarrow \mathbf{E}) \cdot \exists(\mathcal{F}{:}\mathbf{N} \rightarrow \mathbf{N}) \cdot \mathcal{E} \circ e' \circ \mathcal{F} = e$$

F. The Interpreter Specification

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

Interpreters are state transition systems. An interpreter, $\mathcal{I}$, is a predicate defined in terms of a 3-tuple, $\mathcal{J}, \mathcal{K}, \mathcal{C}$, where $\mathcal{J}, \mathcal{K}$, and $\mathcal{C}$ are defined as follows.

- Let $\mathbf{J}$ be the set of all functions with domain $(\mathbf{S} \times \mathbf{E})$ and codomain $\mathbf{S}$. 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 $\mathcal{J}$. The functions in $\mathcal{J}$ provide a denotational semantics for the instructions that they represent.
- In order to uniquely identify each instruction in $\mathcal{J}$, we associate it with a unique key. At the abstract level, we take keys from the uninterpreted domain, $\mathbf{K}$. At the concrete level, keys can have various representations, as we will see in the example in Section 6. We must be able to choose instructions from $\mathcal{J}$ according to some predefined selection criteria. The selection is based on the current state and environment. We define $\mathcal{K}$ to be a function with domain $(\mathbf{S} \times \mathbf{E})$ and codomain $\mathbf{K}$.
- We define $\mathcal{C}_{\mathcal{J}}$ to be a choice function that has domain $\mathbf{K}$ and codomain $(\mathbf{S} \times \mathbf{E} \rightarrow \mathbf{S})$. $\mathcal{C}_{\mathcal{J}}$ selects the state transition function from $\mathbf{J}$ that is associated with key $\mathbf{K}$.

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

$$\mathcal{I}[s, e] \equiv \forall t{:}\mathbf{N} \cdot s_{t+1} = \mathcal{N}(s_t, e_t)$$

where

$$\mathcal{N} = \mathcal{C}_{\mathcal{J}}(k_t)$$

and

$$k_t = \mathcal{K}(s_t, e_t)$$

In this equation, $s_t(e_t)$ is the state (environment) in the state (environment) stream $s(e)$ at time t. The predicate, $\mathcal{I}$, constrains the state of the interpreter at time $t+1$ to be a function, $\mathcal{N}$, of the state and environment at time t. The function is determined by applying the choice function, $\mathcal{C}_{\mathcal{J}}$, to the key returned by $\mathcal{K}$ for the state and environement at time t.

G. Interpreter Verification

Our goal is to prove a correctness relation between the interpreters at different levels of a microprocessor abstraction. In particular, for two interpreters, $\mathcal{I}_m$ and $\mathcal{I}_\ell$, we wish to show that

$$\mathcal{I}_m[s_m, e_m] \Rightarrow \mathcal{I}_\ell[s_\ell, e_\ell]$$

where $s_m(e_m)$ is the state (environment) stream at level m and s_ℓ (e_ℓ) is the state (environment) stream at level ℓ. By definition, we require that $s_\ell \preceq s_m$ and $e_\ell \preceq e_m$.

When this implication is true, $\mathcal{I}_\ell$ is an abstraction of $\mathcal{I}_m$ and $\mathcal{I}_m$ is said to *implement* $\mathcal{I}_\ell$.

The correctness theorem above follows from the following lemma:

$$\forall j \in \mathcal{J} \cdot \mathcal{I}_m(s_m, e_m) \wedge (j = \mathcal{C}_{\mathcal{J}}(k_t)) \Rightarrow \exists c \cdot (\mathcal{S} \circ s_m)_{(t+c)} = j((\mathcal{S} \circ s_m)_t, (\mathcal{E} \circ e_m)_t)$$

This lemma, which we call the instruction correctness lemma, states that every instruction follows from the concrete interpreter, $\mathcal{I}_m$. Specifically, it says that for every instruction, j in $\mathcal{J}$, if j is selected, then applying j to the current abstract state and environment, $(\mathcal{S} \circ s_m)_t$ and $(\mathcal{E} \circ e_m)_t$, yields the same abstract state that results from letting the implementing interpreter $\mathcal{I}_m$ run for c cycles.

The instruction correctness lemma suggests a case analysis on the instruction set and ignores temporal abstraction, stating only that there exists a time in the future when the states correspond. This lemma plays an important role in the work we describe next.

TABLE II
THE ABSTRACT FUNCTIONS AND THEIR TYPES FOR THE GENERIC INTERPRETER MODEL

Operation	*Type*
`instructions`	`:(*key×(*state→*env→*state))list`
`select`	`:*state→*env→*key`
`key`	`:*key→num`
`substate`	`:*state'→*state`
`subenv`	`:*env'→*env`
`Impl`	`:(time'→*state')→(time'→*env')→bool`
`count`	`:*state'→*env'→*key'`
`begin`	`:*key'`

IV. A MODEL OF INTERPRETERS IN HOL

The similarities in past microprocessor verifications can be exploited to form a methodology for microprocessor verification in general. To make this information usable, we have formalized the microprocessor specification model in the HOL mechanical theorem-proving system. The formalization does several things.

1) The formalization provides a step-by-step approach to microprocessor specification by enumerating the important definitions that need to be made for any microprocessor specification.
2) Using the formalization, the verification tool can derive the lemmas that need to be verified from the specification.
3) After these lemmas have been established, the verification tool can use the formalization to automatically derive the final result from the lemmas.

To formalize the model developed in the last section, we use abstract theories. The next section discusses abstract theories and how they can be used to formalize mathematical models. The remaining sections discuss the parts of the generic model.

A. Abstract Theories

A theory is a set of types, definitions, constants, axioms and parent theories. Logics are extended by defining new theories. An abstract theory is parameterized so that some of the types and constants 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 an abstract theory: the multiplication operator is undefined except for its syntax (a binary operator on an uninterpreted type) and a semantics given by the axioms of group theory.

Abstract theories are useful because they provide proofs about abstract structures which can then be used to reason about specific instances of those structures. 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.

There are two key components of an abstract theory: 1) the *abstract representation* and 2) the *theory obligations*. The abstract representation is a set of abstract objects and a set of abstract operations. The operations are unspecified; that is, we don't know (inside the theory) what the objects and operations mean. Their partial meaning is specified through the theory obligations—a set of predicates that define relationships among members of the abstract representation. The abstract theory models any structure with objects and operations that satisfy the predicates.

The theory obligations axiomatize the theory. Using the obligations as axioms, we prove theorems of interest about the abstract objects and operations. The goal is to use the abstract theory to reason about specific objects by instantiating the abstract theory with a concrete representation that has been shown to meet the obligations. The instantiation specializes the abstract theorems, producing a set of theorems about the concrete representation. The concrete representation is an instance of the abstract theory and represents a *member* of the class of abstract objects that the abstract theory describes.

HOL, the verification environment used in the research reported here, does not explicitly support abstract theories; however, HOL's metalanguage, ML, combined with higher order logic, provides a framework sufficient for implementing abstract theories [21]. Several specification and verification systems, such as EHDM [16], offer explicit support for abstract theories.

B. The Abstract Representation

We specify the abstract representation for the generic interpreter model by defining a list of abstract types and operations. Table II shows the operations and their types.

When compared with the mathematical description given in the last section, the formalization in HOL is more operational, largely for efficiency reasons. As an example, we will use lists, rather than sets, to describe the instruction set. HOL can express choice on sets, but the resulting proofs are considerably more difficult than similar proofs about lists. Certainly the readability of specifications is an important problem; most microprocessor specifications are difficult enough to read without unneeded details. Current work by the author and others is addressing these notational problems. Our ultimate goal is specifications that are readily readable as well as practically verifiable.

Before describing the abstract representation, we must emphasize that the representation *is* abstract, and thus the types 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.

The following abstract types are used in the representation.

- `:*state` represents the state and corresponds to **S** from the last section.
- `:*env` represents the environment and corresponds to **E** from the last section.
- `:*key` is type containing all of the keys and corresponds to **K** from the last section.

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 concrete rather than the abstract level of the hierarchy.

The abstract representation is divided into three parts. The first contains those operations concerned with the interpreter proper.

- `instructions` represents the instruction set, which is represented by a list. Each member of the list is an instruction that associates a key with a state transition function. Throughout the rest of the paper, we make use of two selector functions, `KEY` and `IFUNC`, which respectively select the key and state transition function from an instruction. `instructions` corresponds to $\mathcal{J}$ from the last section.
- `select` represents the function that selects a key based on the present state and environment. `select` corresponds to $\mathcal{K}$ from the last section.
- `key` maps an object of type `:*key` to a number. This number is used to index the list containing the instructions. `key` is used in conjunction with the `EL` function (which selects the nth member of a list) to implement $\mathcal{C}$ from the last section.

The second part contains the abstraction functions that relate the state and environment at the concrete level to the state and environment at the abstract level.

- `substate` is the state abstraction function for the interpreter. The domain of `substate` is primed indicating that it is from the concrete level. `substate` corresponds to $\mathcal{S}$ from the last section.
- `subenv` is the environment abstraction function similar to `substate`. `subenv` corresponds to $\mathcal{E}$ from the last section.

Because we want to prove correctness results about the interpreter, we must have an implementation to verify the interpreter against. The third part of the abstract representation contains three functions that 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, but doing so would require that every implementation be an interpreter or at least have some pre-chosen structure. As we see in the example, the implementation need not be modeled as an interpreter at all. Thus we say nothing about it except to define its type; its structure and operation are completely unknown.
- `count` is analogous to `select` except it operates at the concrete level. Notice that it uses the state and environment at the concrete level to produce a key for the concrete level.
- `begin` denotes the beginning of the implementation clock cycle.

The functions `count` and `begin` are used to implement the predicate **G** that indicates when time at the concrete and abstract levels correspond.

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 model other than the relationships that are defined in the theory obligations.

C. The Theory Obligations

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

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 given 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. We call this the *instruction correctness lemma.*

The predicate `INSTRUCTION_CORRECT` expresses the conditions that we require in the instruction correctness lemma. `INSTRUCTION_CORRECT` is a good example of the kind of information that is captured in the generic model. Previous microprocessor verifications created this lemma, or one similar to it, in a largely adhoc manner. `INSTRUCTION_CORRECT` expresses the correctness condition for a single instruction, namely that under certain conditions it follows from the definition of the more concrete implementation.

The complete definition of `INSTRUCTION_CORRECT` is given below:

```
⊢def INSTRUCTION_CORRECT s' e' inst =
     let s = (λ t · (substate (s' t))) in
     let e = (λ t · (subenv (e' t))) in
     let g = (λ t · (count (s' t) (e' t)
           = begin)) in (
     (Impl s' e') ⇒
      (∀ t: time'.
         (select(s t) (e t)= (KEY inst)) ∧
         (count(s' t) (e' t)= begin) ⇒
           ∃ c. Next g (t,t+c) ∧
           ((IFUNC inst)(s t)(e t)
                 = (s (t + c))))).
```

In the definition, `s'(e')` represents the state (environment) stream at the more concrete level; `s (e)` is the state stream (environment) at the abstract level and is derived from `s' (e')` using the function `substate (subenv)`. The predicate `g` is the predicate $\mathcal{G}$ of Fig. 1. Recall that $\mathcal{G}$ was true whenever the states in the concrete and abstract levels corresponded. In our model this happens whenever the counter

in the implementation (denoted by `count (s' t) (e' t)`) is at the beginning (denoted by `begin`). Recall that `KEY` and `IFUNC` are the selectors for the key and instruction function respectively.

`INSTRUCTION_CORRECT` states that the implementation implies that for every time, `t`, if `inst` is selected and the implementation's counter is at the beginning, then there exists a time `c` cycles in the future such that applying the instruction to the current state yields the same state change that the implementation does in `c` cycles.

Using `INSTRUCTION_CORRECT` we can define the theory obligations:

```
EVERY (INSTRUCTION_CORRECT s' e')
  instructions
∀ k:*key · (key k) < (LENGTH
  instructions)
∀ k:*key · k = (KEY (EL (key k)
  instructions)).
```

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

The obligations are used in two ways. First, they are used axiomatically in proving the correctness result; we do this in the next section. Second, the obligations form a set of necessary and sufficient conditions for showing that the implementation meets its specification. In the second case, they are the properties that users of the model must prove about an instantiation; we show this in Section VI. At first the theory obligations may seem like an additional proof burden, but in fact, they are typical of the lemmas that have to be proven in any microprocessor proof. More to the point, the theory obligations provide a method for deriving the proof obligations from the specification. Without the theory obligations, these lemmas would be arrived at in an ad hoc fashion.

D. The Correctness Statement

One of the important parts of the collection of abstract theorems is the definition of a generic interpreter. The definition is based on functions from the abstract representation.

```
⊢def INTERP s e =
      ∀ t: time
       let inst = EL (key (select (s t)
                          (e t))) instructions in
       s(t + 1) = (IFUNC inst) (s t) (e t)
```

The specification of an interpreter is a predicate that relates 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.

The correctness result can be proven from the definition of the interpreter and the theory obligations as follows:

```
⊢ let g = (λ t: time(count (s' t) (e' t)
        = begin)) in
   let abs = (Temp_Abs g) in
   let s = (substate o s' o abs) and
       e = (subenv o e' o abs) (
   (Impl s' e') ∧
   (∃ t · g t) ⇒
   INTERP s e)
```

The correctness statement says that if the implementation is valid on its state and environment streams and there is a time when the concrete clock is at the beginning of its cycle, then the interpreter is valid on its state and environment streams.

This theorem is remarkably similar to the requirement for correctness developed in Section III-G. The function `abs` is analogous to $\mathcal{F}$ and is defined in terms of a general temporal abstraction function `Temp_Abs` using the predicate `g`.

In the correctness statement, `s' (e')` is the state (environment) stream in the implementation. The term `(substate o s' o abs) ((subenv o e' o abs))` is the state (environment) stream for the interpreter defined in the model. Thus by definition, `s ( e)` is a data and temporal abstraction of `s' (e')` and `s` $\preceq$ `s' (e` $\preceq$ `e')`.

There is one major difference between this correctness statement and the one given in Section III–G, the additional prerequisite that there exists at least one time when the predicate `g` is true. This requirement is a reset requirement that ensure liveness. The fact that it shows up here and not in the less formal statement is an example of the utility of mechanical verification: there are no hidden assumptions.

The result in this section is a correctness statement for a generic interpreter model. The model defines a class of computational objects. The correctness result is a verification of *every* microprocessor that matches the semantics defined in the model; that is, once a microprocessor is shown to meet the theory obligations of our model, this correctness result applies to it without further work.

The most important benefit of the generic model is that it structures 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 great improvement over previous microprocessor verifications in which these decisions were made on an ad hoc basis.

V. AVM-1

We have designed and verified a computer designated AVM-1 (*A Verified Microprocessor*) to serve as a test-bed for microprocessor verification. For a more detailed look at the architecture and organization of AVM-1, see [17].

Our design is the result of 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 [18], the experience of others [5], [12], 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 components of the system include a memory

TABLE III
THE PROGRAM STATUS WORD

Bit	Mnemonic	Meaning when set
0	zflag	Last ALU result was zero
1	cflag	Last ALU operation caused a carry
2	nflag	Last ALU result was negative
3	vflag	Last ALU operation caused a overflow
4	ie	Interrupts enabled
5	sm	In supervisory mode

management unit, a floating point unit, an interrupt controller, and a direct memory access chip.

A. The 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.

The Registers: AVM-1 has a load-store architecture based on a large register file. The register file is divided into three portions:

1) register 0, which is read-only and contains the constant 0;
2) seven supervisor-mode registers, including a distinguished register for use as the supervisor stack pointer (SSP). The supervisor-mode registers are read-only unless the CPU is in supervisor-mode (determined by the state of the 6th 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 (PC) is used to sequence the computer—it indicates which instruction to execute next. The program status word (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 III. shows the meaning of the 6 bits in the program status word.

The Instruction Set: The instruction set contains 30 instructions. The instruction set for AVM-1 was inspired by the RISC I instruction set found in Katevenis [14]; it is a load-store architecture, meaning that most instructions are not allowed to access memory for their operands. The instruction formats are simple and regular.

The 30 programming level instructions include the following:

- 8, 3—argument arithmetic instructions
- 8, 2—argument arithmetic instructions that use a 16-bit immediate value
- 4 instructions for loading and storing registers
- 10 instructions for performing user interrupts, jumps, subroutine calls, and shifts.

The Instruction Format: The instruction formats are simple and regular. Fig. 2 shows the four instruction formats. All the formats use the same opcode field.

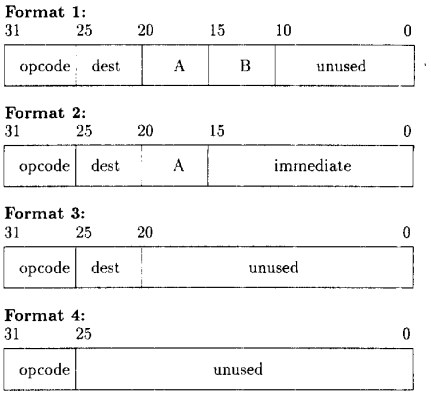


Fig. 2. The instruction formats in AVM-1.

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 composed 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 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 must be dealt with in the proof.

B. The Organizational View

The implementation of AVM-1 can be divided into two major parts: the datapath and the control unit. We will briefly describe the datapath and discuss the timing issues that affect AVM-1's control unit.

The AVM-1 *Datapath:* The AVM-1 datapath is loosely based on the AMD 2903 bit-sliced datapath [1] shown in Fig. 3. 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 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

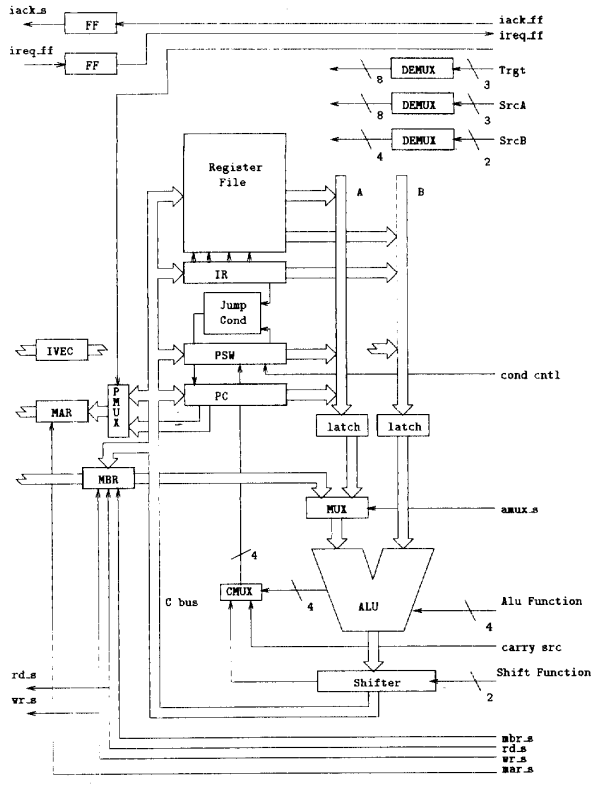


Fig. 3. The AVM-1 datapath.

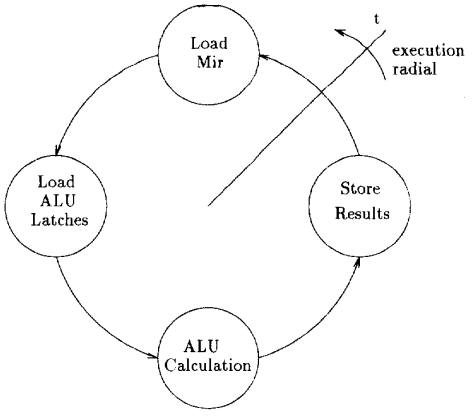


Fig. 4. A phase diagram for AVM-1.

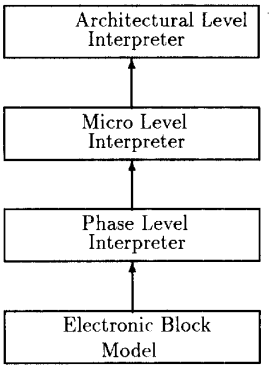


Fig. 5. A microprocessor specification can be decomposed as a series ofinterpreters.

on the 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) that are directly saved in the program status word.

Timing: Rather than describe the control unit of AVM-1 in detail, we concentrate on the timing behavior since that is the most important feature for understanding what is to follow. The control unit establishes this timing. The timing of AVM-1 is based on a four-phase clock as shown in Fig. 4. During the four phases, the machine performs the following state transitions.

1) In phase 1, the microinstruction register is loaded from the micro-rom.
2) In phase 2, the latches that feed the 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 in the register file and system registers.

VI. Specifying AVM-1

Presenting the complete specification and verification of AVM-1 is beyond the scope of this paper. This section shows parts of the specification to demonstrate how writing the specification is aided by the generic interpreter model.

The specification of AVM-1 is hierarchical in nature and uses four levels, as shown in Fig. 5. The bottom level, or EBM, is a structural specification; we do not present it here. By *structural specification* we mean a specification that describes how the major components of the microprocessor, such as the register file and ALU, are connected together. The structural specification in HOL corresponds to the netlists commonly used to describe circuits textually and is similar in form to the structural descriptions of circuits written in VHDL or other hardware description languages.

The specifications above the electronic block model are *behavioral* specifications. Based on state transition functions, they specify *what* happens without describing *how* it happens. The use of a single behavioral model (that is, the generic interpreter model) to describe three different levels in the microprocessor design may seem to be an instance of making the problem fit the solution; however, modeling the various levels of a microprocessor design in a uniform way is not new (see [2]). Describing the behavior of each level in terms of state transitions modeled as instructions is quite natural.

The descriptions of the three behavioral levels all follow the pattern imposed by the generic model. The generic model requires that we define each of the abstract operations in the representation; the following abstract operations are defined in each section: `instructions`, `select`, `key`, `substate`, and `subenv`. The definitions of `Impl`, `count`, and `begin` are defined as part of the specification of the lower level. We divide each section into parts and define each of these abstract operations[1].

A. Specifying the Architectural Level

The architectural level is the topmost specification in our hierarchy- and is thus the most abstract. The architectural level specification is a formal specification of what one would generally find in a programmer's manual for a microprocessor. The specification describes the effect of each of the architectural 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 unambiguous and concise.

This section provides definitions for thefollowing components of the architectural level specification: the state-tuple; three sample architectural level instructions (JMP, ADD, and the external interrupt); the architecural operations corresponding to the operations in the generic interpreter theory (select, key, etc); and the architectural level interpreter.

Defining the Architectural Level State: We use a state-tuple to describe the architectural level interpreter state.

```
(reg, psw, pc, mem, ivec)
```

The only state visible to the architectural level programmer is the register file (`reg`), the program status word (`psw`), the program counter (`pc`), the memory (`mem`), and the interrupt vector (`ivec`).

Defining the Instruction List: The complete specification for the architectural level is contained in [19]; here we highlight three sample instructions.

The JMP *Instruction:* The JMP instruction uses instruction format 2 (see Table II). In the JMP instruction, the first 5-bit field is used to specify the jump condition (only the least significant 4-bits are used), the second 5-bit field specifies the index of the register to use as a base address, and the 16-bit immediate field is used as an offset from the base value.

Behaviorally, 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 according to the conditions in Table IV. 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 (reg, psw, pc, mem, ivec) =
    let a = EL (GetSrcA pc mem) reg and
        i = GetImm pc mem and
        d = GetDest pc mem in
    let jump_cond = JUMP_COND d psw in
    let new_pc = (jump_cond → (add(a, i))
                  | inc pc) in
    (reg, psw, new_pc, mem, ivec)
```

[1] Note that the presentation that follows is not intended to be an engineering document that presents the specification and analysis, but rather an expository document that shows how microprocessors can be specified and verified. An engineering presentation would differ considerably since the purpose of the document would be to demonstrate the correctness of the design rather than the utility of the model.

TABLE IV
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 function `GetSrcA` retrieves the value of the A source field from the word in memory that represents the current instruction (as determined by the program counter). `GetDest` and `GetImm` similarly retrieve the value of the destination and immediate fields from the current instruction. The definitions of these auxiliary functions are precise and available to readers of the specification. The definition of JUMP_COND describes the conditions under which a jump occurs and returns a boolean value used in the calculation of the new value for the program counter.

The ADD *Instruction:* The ADD instruction uses instruction format 1. 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. The HOL specification is given below:

```
⊢def ADD (reg, psw, pc, mem, ivec) =
    let a = EL (GetSrcA pc mem) reg and
        b = EL (GetSrcB pc mem) reg and
        d = GetDest pc mem in
    let result  = add (a, b) in
    let cflag   = addp (a, b, result) and
        vflag   = aovfl (a, b, result) and
        nflag   = negp result and
        zflag   = zerop result and
        sm      = get_sm psw and
        ie      = get_ie psw in
    let new_reg = UPDATE_REG psw d reg
                  result and
```

```
        new_psw = mk_psw (sm, ie, vflag,
                  nflag, cflag, zflag) and
        new_pc  = inc pc in
    (new_reg, new_psw, new_pc, mem, ivec)
```

Clearly there is more to an add instruction than just calculating the new value and storing it in the right place, although that is certainly the high point. The specification unambiguously defines what state changes take place and the definitions of auxiliary functions tell precisely how the new state is calculated. For example, `addp` returns a boolean result, indicating whether there was carry out, `aovfl` returns a boolean result that is true when overflow has occurred, `negp` determines if the result is negative, and `zerop` determines whether the result was zero. Unlike some reference manuals, the definitions of these functions are available for inspection by readers of the specification. One can look at the definitions of these functions and know exactly how, for example, carry or overflow are calculated.

The `EINT` *Instruction:* The `EINT` instruction describes the behavior of the microprocessor upon an external interrupt. We include the specification of the external interrupt in this paper because it has interest both in its own right and in showing how events not typically thought of as instructions can be specified in our model.

The selection criterion 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 criterion differs substantially from that of the other instructions (and because an assembly language programmer would not really consider it an instruction) we term `EINT` a "pseudoinstruction."

Every state variable in the architectural 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 (reg, psw, pc, mem, ivec) =
    let cd = SSP_REG reg and
        d = ssp_reg in
    let cflag = get_cf psw and
        vflag = get_vf psw and
        nflag = get_nf psw and
        zflag = get_zf psw and
        sm    = T and
        ie    = F in
    let new_psw =
          mk_psw (sm, ie, vflag, nflag,
           cflag, zflag) in
    let new_reg = UPDATE_REG new_psw d reg
                  (inc cd) and
        new_pc = band (lower_8_bits, ivec)
                 and
        new_mem = store (mem, address cd,
                  pc) in
    (new_reg, new_psw, new_pc, new_mem,
     ivec).
```

In the specification, `SSP_REG` selects the `SSP` register from the register file; `ssp_reg` is a constant value used to avoid arbitrary numbers in the specification (much as one tries to avoid them in programming). Several other functions also bear explanation: `band` is bitwise logical conjunction, `address` coerces an n-bit word into an address, and `store` updates a value in memory at a particular address. The constant `lower_8_bits` is an n-bit word with integer value 256.

The Instruction List: Before defining the instruction list and the selection function for the architectural level, we must decide on 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, one instruction, `EINT`, 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, not to mention making the architectural level model unverifiable.

We chose to represent the keys at the architectural 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.

We now define the architectural 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 arch_instructions =
      [(INL(F,F,F,F,F),JMP);
                       ⋮
      (INL(T,F,F,F,F),ADD);
                       ⋮
      (INR(one),       EINT);
      ].
```

Defining `select`*:* The instruction selection function `Opcode` uses the environment and the state to determine which instruction to execute.

```
⊢def Opcode (reg, psw, pc, mem, ivec)
     (int_e, reset_e) =
       (int_e ∧ (get_ie psw)) →
           INR(one) |
           INL(opcode (fetch (mem, address
            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-bit opcode portion of the word in memory pointed to by the program counter is returned.

Defining key*:* To instantiate the generic interpreter model, 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.
```

The function determines whether its argument is a left or right injection and then, for a left injection, uses bt5_val (which returns the integer value of a boolean 5-tuple) to return the value. Because there is only one possible right injection, we return 32 without any further work.

Defining substate*:* Micro_Substate is the function used to transform a micro-level state-tuple into the architectural level state tuple shown above.

```
⊢def Micro_Substate (reg, psw, pc, mem,
     ivec, ir, mar, mbr, mpc) =
                        (reg, psw, pc, mem,
                         ivec).
```

The instruction register (ir), memory address register (mar), memory buffer register (mbr), and microprogram counter (mpc), which are all visible at the micro-level, are deleted from the micro-level state-tuple to produce the architectural level state-tuple.

+*Defining* subenv*:* The environment is identical at the architectural level and the micro-level; therefore, the subenv function is represented using the built-in identity function, I.

Defining Impl, clock, *and* begin*:* The definitions of Impl, clock, and begin are taken directly from the specification of the micro-level. Impl is the definition of the micro-level interpreter. clock is the microprogram counter and begin is the starting location for the microprogram, FETCH_ADDR.

Defining the Architectural Level Interpreter: In Section IV, we defined the generic interpreter, INTERP. The first argument to INTERP is the representation. The representation tuple contains the concrete functions that instantiate the abstract operations from the abstract representation. We use the definitions from the previous sections to instantiate INTERP and produce a top-level specification of the interpreter at the architectural level. The instantiation is given in Table V.

```
⊢ Arch_Int s e =
    (∀ t ·
      s(t + 1) =
      IFUNC
      (EL (Opc_Val(Opcode (s t)(e t)))
       arch_instructions)
      (s t)
      (e t)).
```

B. Specifying the Micro-Level

We do not present the details of the specification of the micro-level interpreter because it is similar to the architectural level interpreter. The final product of the specification is a definition that looks much like the definition of the architectural level interpreter:

```
⊢Micro_Int s e =
 (∀ t ·
   s(t + 1) =
     IFUNC (EL (bt6_val(GetMPC(s t)(e t)))
        micro_instructions)
       (s t)
       (e t))
```

TABLE V
THE FUNCTIONS USED TO INSTANTIATE THE ABSTRACT REPRESENTATION OF THE GENERIC INTERPRETER MODEL FOR THE ARCHITECTURAL LEVEL

Operation	*Instantiation*
instructions	arch_instructions
key	Opc_Val
select	Opcode
substate	Micro_Substate
subenv	I
Impl	Micro_Int
clock	GetMPC
begin	FETCH_ADDR

C. Specifying the Phase-Level

The phase-level model is the behavioral representation of AVM-1 from the standpoint of AVM-1's polyphase clock. AVM-1 has a 4-phase clock that describes how each microinstruction is executed. The term *instruction* does not really fit at this level, but the idea of a state transition function does and the description of the phase-level behavior by means of the generic interpreter model is both natural and useful.

This section provides definitions for the following components of the phase-level specification: the state-tuple, one sample phase-level instruction (for phase two), the phase-level operations corresponding to the operations in the generic interpreter theory (select, key, etc.), and the phase-level interpreter.

Defining the Phase-Level State: The state-tuple that describes the phase-level interpreter state is shown below.

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

The variables correspond to the registers, flip-flops, and memories in the datapath shown in Fig. 3.

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 there. During the four phases, the machine performs the following state transitions shown in Fig. 4 and described in Section V-B-2. The formal definitions for these phases describe in detail what happens in each phase. Due to space constraints, we present only the definition of the second phase.

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 (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 (srca ir)) reg) |
     ((SrcA mir) = (F,F,T)) →
         (EL (reg_len (dest ir)) reg) |
     ((SrcA mir) = (F,T,F)) →
         (SSP_REG reg) |
     ((SrcA mir) = (F,T,T)) → psw |
     ((SrcA mir) = (T,F,F)) → (wordn 255)
                            | pc) in
  let new_blatch = (
     ((SrcB mir) = (F,F)) →
         (EL (reg_len (srcb ir)) reg) |
     ((SrcB mir) = (F,T)) →
         (int_fetch ivec) | (imm 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).
```

The state transition function takes a state tuple and an environment tuple as its arguments and returns a new state tuple.

Phase-One, Phase-Three, and Phase-Four: To complete the specification of this level, we would write formal descriptions of the state transitions that take place in the first, third, and fourth phases.

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 phase-clock.

```
⊢def GetPhaseClock
       (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 `Impl`, `count`, *and* `begin`: The implementation for the phase-level is the electronic block model. The specification of the electronic block model is a structural description; fully expanded, it is about 6 pages long. The top-level of the specification is a predicate called `EBM` operating over a state and environment stream.

TABLE VI
THE FUNCTIONS USED TO INSTANTIATE THE ABSTRACT REPRESENTATION OF THE GENERIC INTERPRETER MODEL FOR THE PHASE-LEVEL

Operation	*Instantiation*
`instructions`	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`

The definitions of `count` and `begin` are trivial since there is no temporal abstraction between the electronic block model and the phase-level. They describe a single phase-clock, so `GetEBMClock` returns an arbitrary constant value, `EBM_Begin`.

Defining the Phase-Level Interpreter: Table VI shows the functions used to instantiate the abstract representation. The result is a specification of the phase-level interpreter:

```
⊢ Phase_Int s e =
 (∀ t ·
   s(t + 1) = IFUNC (EL (bt2_val
              (GetPhaseClock (s t)(e t)))
      [(F,F),phase_one;
       (F,T),phase_two;
       (T,F),phase_three;
       (T,T),phase_four ]) (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 nth member of the instruction list where n is calculated from the phase-level clock.

VII. VERIFYING AVM-1

In this section, we instantiate the generic interpreter model to provide the desired correctness lemmas for each level of the AVM-1 specification. These correctness lemmas are later combined to provide an overall correctness theorem for AVM-1.

For each level, we carryout the following steps.

1) Instantiate the generic correctness predicate so that it can be used in the proofs of the theory obligations.
2) Prove the three theory obligations for the instantiation.
3) Using the proofs of the theory obligations, instantiate the correctness result from the generic model.

A. Verifying the Architectural Level

The goal of the architectural level verification is to show that the micro-level implements the architectural level. At this level, the micro-level specification becomes the implementation and the architectural 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 architectural level specification.

The Instruction Correctness Predicate: The correctness predicate represents one of the most important parts of the theory obligations. The main advantage of using the generic interpreter model is that once the specification is completed, the theorem prover can instantiate the generic model to produce the goals that need to be established to prove the final result. This is much better than determining these goals by trial and error.

The instruction correctness predicate, once instantiated, says exactly what must be proven about the instructions at the architectural level to meet the theory obligations and instantiate the generic model.

```
Arch_Inst_Correct =
⊢Arch_Inst_Correct s' e' p =
   Micro_Int s' e' ⇒
   (∀ t ·
     (Opcode (Micro_Substate (s' t))(e' t)
       ▫ KEY 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) ∧
       (IFUNC p(Micro_Substate (s' t))
          (e' t) =
       Micro_Substate (s'(t + c)))))
```

It is interesting to compare this version of the instruction correctness predicate with the generic one given in Section IV–C. The structure is the same, but the names have changed.

The Theory Obligations: We are required to meet three theory obligations before we can instantiate the generic model.

1) We must show that each instruction in the architectural level specification is correct with respect to the micro-level specification. Specifically, we must prove that the instruction correctness predicate, `Arch_Inst_Correct`, is true for every instruction in the architectural 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: We can prove the instruction correctness lemma using symbolic execution for each instruction at the architectural level. For example, here is the instruction correctness lemma for the first instruction in the list, `JMP`.

```
⊢ Arch_Inst_Correct
      (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),JMP).
```

Because of the regularity imposed by the generic interpreter model, we are able to develop a single HOL tactic that proves the instruction correctness lemma for every instruction in the architectural level instruction set.This relieves much of the burden of proving the instruction correctness lemma. Using the individual results for each instruction in the list, we can prove the instruction correctness lemma for the architectural level.

```
Arch_Int_CORRECT_LEMMA =
⊢  EVERY (Arch_Inst_Correct
            (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))
          arch_instructions.
```

The Length Lemma: In the length lemma at the architectural level, 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 it is not substantially more difficult.

```
Arch_Int_LENGTH_LEMMA =
⊢ ∀ opc · Opc_Val opc < (LENGTH
                          arch_instructions).
```

The Order Lemma: The proof of the order lemma for the architectural level is also different from the proof of the order lemma for the other levels due to the coproduct representation of the keys. Again, the result is not difficult to prove.

```
Arch_Int_ORDER_LEMMA =
⊢ ∀ opc · opc = (KEY (EL (Opc_Val opc)
                   arch_instructions)).
```

Instantiating the Correctness Theorem: After the theory obligations for the architectural level have been established, we can instantiate the generic model to provide a correctness result for this level. After the instantiation is complete, some minor rewriting and beta reduction lead to the final result for this level:

```
ARCH_LEVEL_CORRECT_LEMMA =
⊢ Micro_Int
      (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) ⇒
  Arch_Int
      ((λ t · (reg t,psw t,pc t,mem t,
       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))).
```

According to this result, that the architectural level interpreter is correct with respect to the micro-level interpreter. The expression

```
(Temp_Abs(λ t · mpc t = F,F,F,F,F,F))
```

is the temporal abstraction function that relates time at the architectural level to time at the micro-level.

B. Verifying the Micro-Level

We do not present the verification of the micro-level because of space constriants. The verification of the micro-level is much like the verification of the architectural level. The following theorem is the final result of the verification:

```
MICRO_LEVEL_CORRECT_LEMMA =
⊢ Phase_Int (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
   ((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, or in other words, a state stream. This expression represents a data abstraction of the phase-level state stream and is not a micro-level state stream until it is composed with the temporal abstraction function

```
(Temp_Abs(λ t · clk t = F,F))
```

that maps micro-level time onto phase-level time.

The correctness result also contains the following 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.

C. Verifying the Phase-Level

The verification of the phase-level differs from the verification of the architectural level and the micro-level in a significant way: the implementation is a structural representation of the electronic block model rather than a behavioral representation. This makes some steps in the verification less uniform, but the overall process is essentially the same.

The Instruction Correctness Predicate: Each of the phase-level instructions must satisfy the instruction correctness predicate if we are to meet the theory obligations. We use the same procedure that produced the phase-level interpreter specification to instantiate the generic correctness predicate.

```
Phase_Int_Inst_Correct =
⊢Phase_Int_Inst_Correct s' e' p =
 EBM s' e' ⇒
 (∀ t ·
  (GetPhaseClock (s' t)(e' t) = KEY p) ∧
  (GetEBMClock (s' t)(e' t) = EBM_Start) ⇒
  (∃ c ·
   Next(λ t' · GetEBMClock (s' t')
   (e' t') = EBM_Start)(t,t + c) ∧
   (IFUNC p (s' t) (e' t) = s'(t + c)))
```

Because the instruction correctness predicate is derived from the specification rather than being developed in an ad hoc manner, it has the same form as the instruction correctness predicate for the architectural level.

The Theory Obligations: Just as at the architectural level, the theory obligations to be proven are automatically derived from the abstract theory obligations by HOL.

The Instruction Correctness Lemma: To establish the first theory obligation for the generic interpreter model, we first 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.

The following theorem holds that the instruction correctness predicate applied to the first instruction, phase_one, is a tautology.

```
PHASE_ONE_EBM_LEMMA =
⊢ Phase_Int_Inst_Correct
    (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).
```

We also have to prove a similar lemma about each of the other instructions in the phase-level specification. The proofs in each case are long but fairly straightforward. They are not, however, uniform and each must be dealt with individually.

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
      (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;(F,T),phase_two;(T,F),
    phase_three;(T,T),phase_four].
```

The Length Lemma: The second theory obligation is easy to show. The theorem holds that the numeric value of a boolean 2-tuple is always less than the length of a four-element list.

```
Phase_Int_LENGTH_LEMMA =
⊢     ∀ clk · bt2_val clk <
 (LENGTH [(F,F), phase_one;(F,T),phase_two;
    (T,F),phase_three;(T,T), phase_four]).
```

The Order Lemma: The third theory obligation holds 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., the list is correctly ordered). This lemma is also quite easy to show by case analysis.

```
Phase_Int_ORDER_LEMMA =
⊢ ∀ clk · clk =
  KEY (EL (bt2_val clk)
      [(F,F),phase_one;(F,T),phase_two;
      (T,F),phase_three;(T,T),phase_four]).
```

Instantiating the Correctness Theorem: Having proven the theory obligations, we can now instantiate the generic inter-

preter model. The result of the instantiation can be simplified through minor rewriting and beta reduction.

```
PHASE_LEVEL_CORRECT_LEMMA =
⊢EBM (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
    (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)).
```

The result states that the electronic block model implies the phase-level for the concrete state and environment in our model.

D. AVM-1 is Correct

We have successfully instantiated the generic interpreter theory for each of the levels in our hierarchical decomposition.

We establish

$$\mathcal{I}_{\text{EBM}} \Rightarrow \mathcal{I}_{\text{arch}}$$

in stages by showing

$$\mathcal{I}_{\text{EBM}} \Rightarrow \mathcal{I}_{\text{phase}} \Rightarrow \mathcal{I}_{\text{micro}} \Rightarrow \mathcal{I}_{\text{arch}}.$$

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
 EBM (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) ⇒
 Arch_Int ((λ t · (reg t,psw t,pc t,
 mem t,ivec t)) o abs)
          ((λ t · (ireq_e t)) o abs)).
```

This result is the same result that we would have proven had we not used hierarchical decomposition and the generic interpreter model. However, the process by which we arrived at this result was methodical—the generic interpreter theory guided the specification and verification at every level.

VIII. Observations

Having completed the formal specification and verification of AVM-1, we make several observations.

- We have shown how a variety of architectural and organizational features can be modeled using the generic interpreter model. One should not assume that we claim that every architectural feature will map onto the model presented in Section IV. Indeed, many may not.
  What does this say, then, for the utility of abstract theories? Certainly, many interesting features, such as interrupts, can be mapped onto the model given in this paper. 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 abstract theories—structuring the proof—is not diminished.

  We are currently exploring the application of the generic interpreter theory to the verification of pipelined architectures with feedback. In [23]. we show why the model presented in this paper will not work for pipelined microprocessors, describe what it means for a microprocessor with an instruction pipeline to be correct, and provide an example verification of a microprocessor with a 5-stage instruction pipeline.
- 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, in our representation, as the key to select an instruction. The architectural 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 in the model. A program that is thousands of lines long at the architectural level would still only use the 30 instructions given here. For longer microprograms, a different representation of keys would have to be chosen.
- Another interesting point concerning keys is their use at the architectural 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 generic interpreter model 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.
- In order to deal with detailed 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 more difficult.

We believe that a better approach is to use heterogeneous verification environments that make use of several tools such as standard simulators, symbolic simulators, and theorem-proving tools. We are currently working on the integration of the HOL theorem-proving environment and a set of VLSI design tools including a low-level simulator [7]. Jeffrey Joyce and Carl Seger are working on integrating the HOL theorem proving environment and the Voss symbolic simulator [13]. Other work involving the integration of BDD's, model checking and other theorem-proving tools is underway as well.

The combination of theorem-proving with other means of verifying design correctness provides a way to use the right tool for the right job. Symbolic simulation, model checking, and BDD's are most useful at less abstract levels of the system design (BDD specifications, for example, are given in terms of boolean formulae) and theorem provers are most useful at more abstract levels of the system design (reasoning about mathematical operations, for example).

- 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 control 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 book 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 proof of the instruction correctness lemma was done using one tactic at the architectural level and another tactic at the micro-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 that 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 represents the greatest part of the effort involved in instantiating the theory.

We can also make some observations regarding the actual proof process.

- It took a person who was intimately familiar with the HOL theorem prover and experienced in hardware specification and verification about 2 person-months to complete the specification and verification of AVM-1. One unexpected observation is that most of that time was spent on the specification. The verification was tedious at times, yet relatively straightforward. Specification seems to be the difficult task.

  Writing a *correct* specification of the total functionality of a large system is usually an iterative process. One writes the specification in pieces, performs some verification, and then uses the feedback from the verification to extend and correct the specification. Once this process is completed and the specification is right, the verification of that fact is relatively simple. This is just another way of saying that the final result is not what is of most importance; rather, the process is what is important.

  Since writing the specification is the most difficult part, we choose to use a theorem prover that provides a more expressive specification language, rather than one that provides more automation for the proof process. Using a logic that forces the specification into unnatural mappings onto unfamiliar concepts only increases the conceptual burden on the verifier.

  As a result of this observation, we question the comparison of various theorem-proving environments as being easier to use than others, particularly when the comparisons are based on *redoing* a completed proof. Once the proof of correctness has been completed in one theorem proving environment, the process of reverifying it in another *should* be easy!
- The proof for AVM-1 contains more than 25 theories. Fig. 6 shows how the main theories of the proof of AVM-1 are related. This hierarchy shows `AVM` as the child theory of a long ancestry that follows the hierarchical decomposition discussed in the body of this paper. The picture is not complete; many theories are not shown. For example, a theory containing auxiliary definitions is the ancestor of almost every theory in the proof. A complete text of the proof is contained in [19].
- Table VII presents the run-times for the various theories in the proof on a SPARCStation with 16 megabytes 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

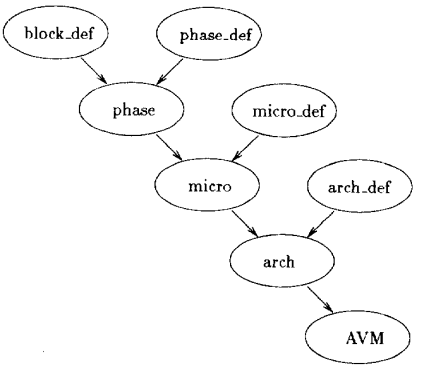


Fig. 6. The theory hierarchy for the proof of AVM-1.

TABLE VII
SCRIPT RUN-TIMES ON A SPARCSTATION WITH 16 M 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_arch.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_arch.ml	7247.9	200120
mk_avm.ml	790.9	10031
	208029.1	5167063

1.11 of HOL, which was built using the Austin Kyoto Common Lisp compiler.

The total time to run the proof was 208 029 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 megabytes) and caused the operating system to thrash when garbage collecting (due to a bug in the memory management unit on the original SPARC Station).

IX. CONCLUSION

This paper has shown that the verification of realistic microprocessors can be made practical by use of a model of generic interpreters. The correctness theorem, definitions, and abstractions that make up the model are important, for several reasons.

1) The model shows exactly what is required to verify that an interpreter is correct. No superfluous detail clutters up the definitions and theorems.
2) The abstract proof is *easier* than the specific proof. In proving theorems about specific interpreters, some amount of detail is always 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 model. This frees the user of the model from proving theorems about the temporal abstraction; it is only done once—when the model is built.
4) Similarly, data abstraction between the state and environment streams at the two levels in the model is clearly defined and consistently performed. The user's contributions are to define the abstractions, the model uses the abstractions to effect the proof.
5) The abstract proof can be instantiated which allows the theorems to be reused and saves the effort required to reverify them.

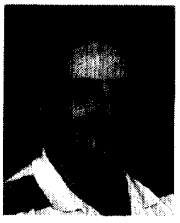
We believe that the structure provided by the generic interpreter model, coupled with the savings afforded by the hierarchical decomposition strategy, make the verification of usable microprocessors a viable engineering activity. We are currently in the process of validating this belief by having graduate students with only an introductory knowledge of HOL verify microprocessors as a semester project in a hardware verification class.

The use of a generic interpreter model for specifying and verifying microprocessors provides a methodological approach. Making specification and verification methodological is an important step in turning what has been primarily a research activity into an engineering activity.

REFERENCES

[1] Advanced Micro Devices, *Bipolar Microprocessor Logic and Interface Data Book*, AMD Inc., 1983.
[2] F. Anceau, *The Architecture of Microprocessors*. Reading, MA: Addison-Wesley, 1986.
[3] A. Camilleri, M. Gordon, and T. Melham, "Hardware verification using higher order logic," in *From HDL Descriptions to Guaranteed Correct Circuit Designs*, D. Borrione, Ed. New York: Elsevier, 1987.
[4] A. Church, "A formulation of the simple theory of types," *J. Symbolic Logic*, vol. 5, pp. 56–115, 1940.
[5] A. Cohn, "Correctness properties of the VIPER block model: The second level," Tech. Rep. 134, Univ. of Cambridge Comput. Lab., May 1988.
[6] ——, "The notion of proof in hardware verification," *J. Automated Reasoning*, vol. 5, pp. 127–139, 1989.
[7] J. W. Gambles and P. J. Windley, "Integrating formal verification with CAD tool environments," in *Proc. Fourth Annu. IEEE/NASA Symp. VLSI Design*, Oct. 1992, pp. 6.3.1–6.3.12.
[8] M. J. Gordon, "HOL: A proof generating system for higher order logic," in *VLSI Specification, Verification, and Synthesis*, G. Birtwhistle and P. Subrahmanyam, Eds. New York: Kluwer Academic Press, 1988.
[9] J. Herbert, "Temporal abstraction of digital designs," in *The Fusion of Hardware Design and Verification, Proceedings of the IFIP WG 10.2 International Working Conference, Glasgow, Scotland*, G. Milne, Ed. Amsterdam, The Netherlands: North-Holland, 1988.
[10] W. A. Hunt, "The mechanical verification of a microprocessor design," in *From HDL Descriptions to Guaranteed Correct Circuit Designs*, D. Borrione, Ed. New York: Elsevier, 1987.
[11] ——, "Microprocessor design verification," *J. Automated Reasoning*, vol. 5, pp. 429–460, 1989.
[12] J. J. Joyce, "Multi-level verification of microprocessor-based systems," Ph.D. thesis, Cambridge Univ., Dec. 1989.
[13] J. J. Joyce and C. Seger, private communication, Univ. of British Columbia, Oct. 1992.
[14] M. G. H. Katevenis, *Reduced Instruction Set Computer Architectures for VLSI*. Cambridge, MA: MIT Press, 1985.

[15] T. Melham, "Abstraction mechanisms for hardware verification," in *VLSI Specification, Verification and Synthesis*, G. Birtwhistle and P. A. Subrahmanyam, Eds. New York: Kluwer Academic Publishers, 1988.
[16] SRI Int. Comput. Sci. Lab., *EHDM Specification and Verification System: User's Guide*, Version 4.1, 1988.
[17] P. J. Windley, "The Formal verification of generic interpreters," Ph.D. thesis, Univ. of California, Davis, Div. of Comput. Sci., June 1990.
[18] ——, "A hierarchical methodology for the verification of microprogrammed microprocessors," in *Proc. IEEE Symp. Security and Privacy*, May 1990, pp. 345–359.
[19] ——, "The verification of AVM-1, Tech. Rep. CSE-90-21, Univ. of California, Davis, Div. of Comput. Sci., 1990.
[20] ——, "Using correctness results to verify behavioral properties of microprocessors," in *Proc. IEEE Comput. Assurance Conf.*, June 1991, pp. 99–106.
[21] ——, "Abstract theories in HOL," in *Proceedings of the 1992 International Workshop on the HOL Theorem Prover and its Applications*, L. Claesen and M. J. C. Gordon, Eds. New York: North-Holland, Nov. 1992.
[22] ——, "Instruction set commutivity," in *Proc. Fourth Annu. IEEE/NASA Symp. VLSI Design*, Oct. 1992, pp. 6.5.1–6.5.11.
[23] P. J. Windley and M. Coe, "The formal verification of instruction pipelines," in *Proceedings of the 1994 Conference on Theorem Provers in Circuit Design, Lecture Notes in Computer Science*, R. Kumar and T. Kropf, Eds. New York: Springer-Verlag, Sept. 1994.



Phillip J. Windley received the Ph.D. degree in computer science from the University of California, Davis, in 1990.

He was a member of the faculty at the University of Idaho and a member of the technical staff at the Department of Energy's Division of Naval Reactors. He is an Assistant Professor with the Department of Computer Science at Brigham Young University, Provo, UT, where he directs the Laboratory for Applied Logic. His research interests center on the use of formal methods in computer science; particularly the specification and verification of hardware. He has taught numerous courses on the use of higher order logic and the HOL theorem prover in computer system verification. Dr. Windley is a member of the Association for Computing Machinery.