Generic Interpreters
and
Microprocessor Verification

Phillip J. Windley

Department of Computer Science
University of Idaho

August, 1990

This work was sponsored under Boeing Contract NAS1-18586, Task Assignment No. 3, with NASA-Langley Research Center.
Outline

• Introduction

• Generic interpreters

• Microprocessor Verification

• Future Work
VIPER, the first commercially available, "verified" microprocessor, has never been formally verified.

The proof was not completed even though 2 years were spent on the verification.
Microprocessor Verification
(continued)

• Our research is aimed at making the verification of large microprocessors tractable.

• Our objective is to provide a framework in which a masters-level student can verify VIPER in 6 person-months.
Determining Correctness

In VIPER (and most other microprocessors), the correctness theorem was shown by proving that the electronic block model implies the macro-level specification.
The Problem
(continued)

• Microprocessor verification is done through case analysis on the instructions in the macro level.

• The goal is to show that when the conditions for an instruction's selection are right, the electronic block model implies that it operates correctly.

• A lemma that the EBM correctly implements each instruction can be used to prove the top-level correctness result.
The Problem

Unfortunately, the one-step method doesn't scale well because

- The number of cases gets large.
- The description of the electronic block model is very large.
Hierarchical Decomposition

- A microprocessor specification can be decomposed hierarchically.

- The abstract levels are represented explicitly.
Interpreters

An abstract model of the different layers in the hierarchy provides a methodological approach to microprocessor verification.

- The model drives the specification.

- The model drives the verification.
Interpreters
(top level)
Specifying an Interpreter
(overview)

We specify an interpreter by:

- Choosing a \( n \)-tuple to represent the state, \( S \).

- Defining a set of functions denoting individual interpreter instructions, \( J \).

- Defining a next state function, \( N \).

- Defining a predicate denoting the behavior of the interpreter, \( I \).
Verifying an Interpreter
(overview)

We verify an interpreter, \( I \) with respect to its implementation \( M \) by showing

\[
M \Rightarrow I.
\]

To do this, we will show that every instruction in \( J \) can be correctly implemented by \( M \):

\[
\forall j \in J.
M \Rightarrow (\forall t: \text{time.} \quad C(t) \Rightarrow s(t + n) = j(s(t)) \}
\]

where \( C \) represents the conditions for instruction \( j \)'s selection.
We have designed and are verifying a microcomputer with interrupts, supervisory modes and support for asynchronous memory.

- The datapath is loosely based on the AMD 2903 bit-sliced datapath.

- The instruction format is very simple.

- The control unit is microprogrammed.
AVM-1's Instruction Set
(subset)

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Mnemonic</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>000000</td>
<td>JMP</td>
<td>jump on 16 conditions</td>
</tr>
<tr>
<td>000001</td>
<td>CALL</td>
<td>call subroutine</td>
</tr>
<tr>
<td>000010</td>
<td>INT</td>
<td>user interrupt</td>
</tr>
<tr>
<td>000110</td>
<td>LD</td>
<td>load</td>
</tr>
<tr>
<td>000111</td>
<td>ST</td>
<td>store</td>
</tr>
<tr>
<td>010000</td>
<td>ADD</td>
<td>add (3-operands)</td>
</tr>
<tr>
<td>011011</td>
<td>SUBI</td>
<td>subtract immediate (2-operands)</td>
</tr>
<tr>
<td>011111</td>
<td>NOOP</td>
<td>no operation</td>
</tr>
</tbody>
</table>

- The architecture is load-store.

- The instruction set is RISC-like.

- There is a large register file.
Figure 5.2: The AVM-1 Datapath
The Phase–Level Specification

The $n$–tuple representing the state:

$$S_{phase} = (mir, mpc, reg,\)$$

$$alatch, blatch, mar, mbr,$$

$$clk, mem, urom, ireq, iack)$$
The Phase-Level Specification

A typical function specifying an instruction's behavior from $J_{\text{phase}}$:

\[ f_{\text{def}} \, \text{phase}_\text{two} \, \text{rep} \, (\text{mir}, \text{mpc}, \text{reg}, \text{alatch}, \text{blatch}, \text{mbr}, \text{mar}, \text{clk}, \text{mem}, \text{urom}, \text{ireq}, \text{iack}) = \]
\[ (\text{mir}, \text{mpc}, \text{reg}, \text{EL} \, (\text{bt5_val} \, (\text{SrcA} \, \text{mir})), \text{reg}, \text{EL} \, (\text{bt5_val} \, (\text{SrcB} \, \text{mir})), \text{reg}, \text{mbr}, \text{mar}, \text{TF}, \text{mem}, \text{urom}, \text{ireq}, \text{iack} \, \text{mir}) \]
The Electronic Block Model

The electronic block model is not specified as an interpreter.

- EBM is a *structural* specification.

- The specification
  - is in terms of smaller blocks.
  - uses existential quantification to hide internal lines.
Objects

There are several abstract classes of objects that we will use to define and verify an abstract interpreter.

:*state An object representing system state.
:*key The identifying tokens for instructions.
:*time A stream of natural numbers.

We will prime class names to indicate that the objects are from the implementing level.
# Operations

<table>
<thead>
<tr>
<th>Operation</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>inst_list</td>
<td>$(\ast key \times (\ast state \rightarrow \ast state))\text{list}$</td>
</tr>
<tr>
<td>key</td>
<td>$\ast key \rightarrow \text{num}$</td>
</tr>
<tr>
<td>select</td>
<td>$\ast state \rightarrow \ast key$</td>
</tr>
<tr>
<td>cycles</td>
<td>$\ast key \rightarrow \text{num}$</td>
</tr>
<tr>
<td>substate</td>
<td>$\ast state' \rightarrow \ast state$</td>
</tr>
<tr>
<td>Impl</td>
<td>$(\text{time} \rightarrow \ast state') \rightarrow \text{bool}$</td>
</tr>
<tr>
<td>clock</td>
<td>$\ast state' \rightarrow \ast key'$</td>
</tr>
<tr>
<td>begin</td>
<td>$\ast key'$</td>
</tr>
</tbody>
</table>
The instruction correctness lemma is important in the generic interpreter verification.

Here is the generic version of that lemma for a single instruction:

\[ \vdash_{\text{def}} \text{INST-CORRECT } s' \; \text{inst} = \]

\[ (\text{Impl } s') \Rightarrow \]

\[ \forall t': \text{time}' . \]

let \( s = (\lambda t. \text{substate}(s' \; t')) \) in

let \( c = (\text{cycles}(\text{select}(s \; t'))) \) in

(\( \text{select}(s \; t') = (\text{FST inst}) \)) \( \wedge \)

(\( \text{clock}(s' \; t') = \text{begin} \)) \( \Rightarrow \)

(\( (\text{SND inst}) \; (s \; t') = (s(t' + c)) \)) \( \wedge \)

(\( \text{clock}(s'(t' + c)) = \text{begin} \))

\]
Interpreter Theory
(obligations)

Using the predicate INST_CORRECT, we can define the theory obligations:

1. The instruction correctness lemma:

   \[
   \text{EVERY } (\text{INST\_CORRECT } s') \text{ inst\_list}
   \]

2. Every key selects an instruction:

   \[
   \forall k : \ast\text{key}. \ (\text{key } k) < (\text{LENGTH inst\_list})
   \]

3. The instruction list is ordered correctly:

   \[
   \forall k : \ast\text{key}. \ k = (\text{FST } (\text{EL (key } k) \text{ inst\_list}))
   \]
Generic Interpreters

Instantiation

- Generic Interpreter + Macro Level Definitions → Macro Level Interpreter
- Generic Interpreter + Micro Level Definitions → Micro Level Interpreter
- Generic Interpreter + Phase Level Definitions → Phase Level Interpreter

Electronic Block Model
Interpreter Theory
(temporal abstraction)

We need to show a relationship between the state stream at the implementation level and the state stream at the top level.

The function $f$ is a temporal abstraction function for streams.
Interpreter Theory
(definition)

An interpreter's behavior is specified as a predicate over a state stream.

\[ \vdash_{\text{def}} \text{INTERP } s = \]
\[ \forall t : \text{time}. \]
\[ \text{let } n = (\text{key}(\text{select}(s \ t))) \text{ in} \]
\[ s(t + 1) = (\text{SND} \ (\text{EL} \ n \ \text{inst\_list}))(s \ t) \]
Interpreter Theory
(correctness result)

Our goal is to verify an interpreter, \( I \) with respect to its implementation \( M \) by showing

\[ M \Rightarrow I. \]

Here is the abstract result:

\[ \vdash \text{Impl } s' \land (\text{clock}(s' 0) = \text{begin}) \Rightarrow \text{INTERP } (s \circ f) \]

where

\[ s = (\lambda t : \text{time. } \text{substate} (s' t)) \quad \text{and} \]

\[ f = (\text{time_abs } (\text{cycles } \circ \text{select}) s) \]
Instantiating a Theory

Instantiating the abstract interpreter theory requires:

- Defining the abstract constants.
- Proving the theory obligations.
- Running a tool in the formal theorem prover.
Definitions

We wish to instantiate the abstract interpreter theory for the phase-level. The electronic block model will be the implementing level.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Instantiation</th>
</tr>
</thead>
<tbody>
<tr>
<td>inst_list</td>
<td>a list of instructions</td>
</tr>
<tr>
<td>key</td>
<td>bt2_val</td>
</tr>
<tr>
<td>select</td>
<td>GetPhaseClock</td>
</tr>
<tr>
<td>cycles</td>
<td>PhaseLevelCycles</td>
</tr>
<tr>
<td>substate</td>
<td>PhaseSubstate</td>
</tr>
<tr>
<td>Impl</td>
<td>EBM</td>
</tr>
<tr>
<td>clock</td>
<td>GetEBMClock</td>
</tr>
<tr>
<td>begin</td>
<td>EBM_Start</td>
</tr>
</tbody>
</table>
An Example

After proving the theory obligations, we can perform the instantiation.

let theorem_list =
    instantiate_abstract_theorems
    'gen_I'
    [Phase_I_EVERY_LEMMA;
    Phase_I_LENGTH_LEMMA;
    Phase_I_KEY_LEMMA]
    [
        "([(F,F),phase_one;
            (F,T),phase_two
            (T,F),phase_three
            (T,T),phase_four],
            bt2_val, GetPhaseClock,
            PhaseLevelCycles, PhaseSubstate,
            EBM, GetEBMClock, EBM_Start)");
        "(λ t:time. (mir t, mpc t, reg_list t,
            alatch t, blatch t,
            mbr_reg t, mar_reg t,
            clk t, mem t, urom))"
    ]
    'PHASE';;
The Electronic Block Model

\[ \text{EBM rep (} \lambda t. \ (\text{mir} \ t, \ \text{mpc} \ t, \ \text{reg} \ t, \ \text{alatch} \ t, \ \text{blatch} \ t, \ \text{mbr} \ t, \ \text{mar} \ t, \ \text{clk} \ t, \ \text{mem} \ t, \ \text{urom}, \ \text{ireq} \ t, \ \text{iack} \ t)) = \]

\[ \exists \ \text{opc} \ \text{ie}_s \ \text{sm}_s \ \text{iack}_s \]
\[ \text{amux}_s \ \text{alu}_s \ \text{sh}_s \ \text{mbr}_s \ \text{mar}_s \ \text{rd}_s \ \text{wr}_s \]
\[ \text{cselect} \ \text{bselect} \ \text{aselect} \]
\[ \text{neg}_f \ \text{zero}_f \ (\text{float:time} \to \text{bool}). \]

\[ \text{DATAPATH rep amux}_s \ \text{alu}_s \ \text{sh}_s \ \text{mbr}_s \ \text{mar}_s \ \text{rd}_s \ \text{wr}_s \]
\[ \text{cselect} \ \text{bselect} \ \text{aselect} \ \text{neg}_f \ \text{zero}_f \ \text{float} \]
\[ \text{float} \ \text{ireq} \ \text{iack}_s \ \text{iack} \ \text{opc} \ \text{ie}_s \ \text{sm}_s \]
\[ \text{clk} \ \text{mem} \ \text{reg} \ \text{alatch} \ \text{blatch} \ \text{mar}_\text{reg} \]
\[ \text{mbr}_\text{reg} \ \text{reset}_e \ \text{ireq}_e \land \]

\[ \text{CONTROL_UNIT rep mpc mir clk amux}_s \ \text{alu}_s \ \text{sh}_s \ \text{mbr}_s \]
\[ \text{mar}_s \ \text{rd}_s \ \text{wr}_s \ \text{cselect} \ \text{bselect} \ \text{aselect} \ \text{neg}_f \]
\[ \text{zero}_f \ \text{ireq} \ \text{iack}_s \ \text{opc} \ \text{ie}_s \ \text{sm}_s \ \text{urom} \]
\[ \text{reset}_e \ \text{ireq}_e \]

Fully expanded, the electronic block model specification fills about six pages.
Future Work

- New architectural features.
- Composing verified blocks.
- Verifying operating systems.
- Gate-level verification.
- Byte-code interpreter verification.
- Other classes of computer systems.
An Example
(continued)

After some minor manipulation, the final result becomes:

\[ \vdash \text{EBM} \]
\[ (\lambda t. \,
   (\text{mir } t, \text{mpc } t, \text{reg_list } t, \text{alatch } t, \text{blatch } t,
   \text{mbr_reg } t, \text{mar_reg } t, \text{clk } t, \text{mem } t, \text{urom})) \Rightarrow \\
\text{Phase}_I
\]
\[ (\lambda t. \,
   (\text{mir } t, \text{mpc } t, \text{reg_list } t, \text{alatch } t, \text{blatch } t,
   \text{mbr_reg } t, \text{mar_reg } t, \text{clk } t, \text{mem } t, \text{urom})) \]
Conclusions

The generic proof

- Cleared away all the irrelevant detail.

- Formalized the notion of interpreter proofs which has been used in several microprocessor verifications.

- Provided a structure for future microprocessor verifications.