2

A Proof of Correctness of the Viper Microprocessor: The First Level

Avra Cohn

University of Cambridge
Computer Laboratory
Corn Exchange Street
Cambridge, CB2 3QG
England.

Abstract: The Viper microprocessor designed at the Royal Signals and Radar Establishment (RSRE) is one of the first commercially produced computers to have been developed using modern formal methods. Viper is specified in a sequence of decreasingly abstract levels. In this paper a mechanical proof of the equivalence of the first two of these levels is described. The proof was generated using a version of Robin Milner's LCF system.

1 Introduction

The Viper microprocessor designed at the Royal Signals and Radar Establishment (RSRE) is one of the first commercially produced computers to have been developed using modern formal methods. Viper is specified in a sequence of decreasingly abstract levels. In this paper a mechanical proof of the equivalence of the first two of these levels is described. The approach used for this proof is based on HOL, a version of Robin Milner's LCF proof generating system.

There are two reasons for verifying at successive levels of abstraction. First, though it is ultimately a circuit that is being built, and the correctness of this circuit is the most important concern, successive levels of description represent successive stages in the development of the implementation, and one would like to know if these levels are correct. It is possible for the circuit to be correct even if a more abstract specification is incorrect, but the error is still worth knowing about. In fact, some minor errors were discovered, during the machine-checked proof, in the two specifications of Viper. As far as we know, the actual implementation does not reflect these difficulties. (See Section 7 for details.)

Second, working along a sequence of levels makes the verification of the circuit a more tractable problem; there is a logical transformation and an introduction of detail at each stage, and these can then be treated separately. We intend to continue the machine-checked proof of Viper down to more circuit-like levels of abstraction in the future.

G. Birtwistle et al. (eds.), *VLSI Specification, Verification and Synthesis*

This paper is intended to be self contained, but is necessarily brief on background material. A description of Viper can be found in Kershaw [13]; the two specifications can be found in Cullyer [4] and Cullyer [5]. (See also Cullyer [6].) The machine proof is based on an informal proof outline given in [5]. A description of the HOL system is given in Gordon [10]. More generally, the LCF approach to proof is described in the successive manuals [7] and [15]. [2] is a study by Cohn and Gordon of a similar but very much simpler machine-checked proof (of the correctness of a counter); it is based on Cullyer and Pygott [3].

We know of three other machine-checked proofs of computers: Gordon [8] verified a PDP-8 style machine in a precursor of HOL called LCF_LSM. This was redone and improved in HOL by Joyce [12]. In the Boyer-Moore paradigm, Hunt [11] has verified a PDP-11 like machine. Neither of these machines was intended for serious use, as is Viper.

The design of Viper, the high-level specification and host machine are due to Cullyer, Kershaw and Pygott. The only differences in the specifications as they appear in this paper are that they have been put into HOL format, and have been corrected for errors. The idea of viewing the host machine as a transition graph is also due to Cullyer [5], as is the informal proof outline [5]. We have formalised the notion of traversing the graph, made the notion of time explicit, formulated the correctness statements, and generated a machine-checked proof.

1.1 The HOL System

1.1.1 Proof in HOL

HOL, like LCF, is a system for generating formal proofs. In HOL, a logic in which problems can be expressed is interfaced to a programming language called ML in which proof strategies can be encoded. The logic is conventional higher-logic (Church, [1]). It is oriented towards hardware verification only in that it provides types, constants and axioms for representing bit strings. New types, constants and axioms can be introduced by the user, and organised into hierarchies of logical *theories*.

The type discipline of the programming language ensures that the only way to create theorems is by performing a proof; theorems have the ML type *thm*, objects of which type can only be constructed by the application of inference rules to other theorems or axioms. (Theorems are written with a turnstile, |-, in front of them.)

Formal proofs (sequences of elements each of which is either an axiom or a theorem which follows from earlier elements of the sequence by a rule of inference) are *generated* in HOL in the sense that for each element of the sequence which is not an axiom, an ML procedure representing the rule of inference is *executed* to produce that element. That the final element, the fact proved, is actually a theorem is guaranteed by the type discipline of the programming language. The sequences themselves are not retained. LCF-style proof is discussed

further in Sections 5.1 and 5.4.

1.1.2 The Logic

The HOL system uses the ASCII characters ~, \/ and /\, ==>, ! and \ to represent the logical symbols $\neg$, $\vee$, $\wedge$, $\supset$, $\forall$ and λ respectively.

For the purposes of this paper, a *term* of higher-order logic can be one of eleven kinds.

- A *variable*
- A *constant* such as `T` or `F` (which represent the truth-values *true* and *false* respectively)
- A *function application* of the form $\mathtt{t}_1\mathtt{t}_2$ where the term $\mathtt{t}_1$ is called the *operator* and the term $\mathtt{t}_2$ the *operand*
- An *abstraction* of the form `\x.t` where the variable `x` is called the *bound variable* and the term `t` the *body*
- A *negation* of the form `~t` where `t` is a term
- A *conjunction* of the form $\mathtt{t}_1$`/\`$\mathtt{t}_2$ where $\mathtt{t}_1$ and $\mathtt{t}_2$ are terms
- A *disjunction* of the form $\mathtt{t}_1$`\/`$\mathtt{t}_2$ where $\mathtt{t}_1$ and $\mathtt{t}_2$ are terms
- An *implication* of the form $\mathtt{t}_1$`==>`$\mathtt{t}_2$ where $\mathtt{t}_1$ and $\mathtt{t}_2$ are terms
- A *universal quantification* of the form `!x.t` where the variable `x` is the bound variable and the term `t` is the body
- A *conditional* of the form `t=>`$\mathtt{t}_1$`|`$\mathtt{t}_2$ where `t`, $\mathtt{t}_1$ and $\mathtt{t}_2$ are terms; this has *if-part* `t`, *then-part* $\mathtt{t}_1$ and *else-part* $\mathtt{t}_2$
- A *"local declaration"* of the form `let x=`$\mathtt{t}_1$ `in` $\mathtt{t}_2$, where `x` is a variable and $\mathtt{t}_1$ and $\mathtt{t}_2$ are terms; this is provably equivalent to `(\x.`$\mathtt{t}_2$`)`$\mathtt{t}_1$ (see Section 5.1)

All terms in HOL have a *type*. The expression `t:ty` means `t` has type `ty`; for example, the expressions `T:bool` and `F:bool` indicate that the truth-values `T` and `F` have type `bool` for boolean, and `3:num` indicates that `3` is a number.

If `ty` is a type then `(ty)list` (also written `ty list`) is the type of lists whose components have type `ty`. If $\mathtt{ty}_1$ and $\mathtt{ty}_2$ are types, then $\mathtt{ty}_1$`->`$\mathtt{ty}_2$ is the type of functions whose arguments have type $\mathtt{ty}_1$ and results of type $\mathtt{ty}_2$. The cartesian product operator is represented by `#`, so that $\mathtt{ty}_1$`#`$\mathtt{ty}_2$ is the type of pairs whose first components have type $\mathtt{ty}_1$ and second, $\mathtt{ty}_2$.

The HOL system provides a number of predefined types and constants for reasoning about hardware. The types include `word`n, the type of n-bit words and `mem`n_1`_`n_2 for memories of n_2-bit words addressed by n_1-bit words. `#b`$_{n-1}\cdots$`b`$_0$ (where `b`$_i$ is either `0` or `1`) denotes an n-bit word in which `b`$_0$ is the least significant bit.

The predefined constants used in this paper are shown below.

- `V:(bool)list->num` converts a list of truth-values to a number
- `VAL`n`:word`n`->num` converts an n-bit word to a number
- `BITS`n`:word`n`->(bool)list` converts an n-bit word to a list of booleans
- `WORD`n`:num->word`n converts a number to an n-bit word
- `FETCH`n`:mem`n_1`_`n_2 `->(word`n_1`->word`n_2`)` looks up a word in memory

List functions include `HD` for taking the head of a list and `CONS` for constructing lists. $[t_1;\cdots;t_n]$ denotes the list containing $t_1,\cdots,t_n$.

To make terms more readable, HOL uses certain conventions. One is that a term $t_1t_2\cdots t_n$ abbreviates $(\cdots(t_1t_2)\cdots t_n)$; function application associates to the left. The product operator `#` associates to the right and binds more tightly than the operator `->`. For example, the function used to model Viper's `ALU` (see Section 3.1) has type

```
word4#word2#word3#word32#word32#bool->word32#bool#bool
```

which abbreviates

```
(word4#(word2#(word3#(word32#(word32#bool)))))->(word32#(bool#bool))
```

From the axioms defining the various constants we can prove **Theorem 0**:

```
|- !b. HD(BITS1(WORD1(V[b]))) = b
```

1.2 What is Proved

At the most abstract level, Viper can be viewed as a function from a state of the machine to a new state, where a state reflects the configuration of the memory, registers and program counter. This is called the high-level or functional specification. The functional specification contains an *operational* semantics of the Viper instruction set. It carries an implicit notion of time in the concepts of *current* and *next* state.

Each state transition at the top level is implemented by a sequence of events at the lower level (the host or major state machine), each of which may affect the *internal* state of the machine. The sequence is determined as the events are performed, according to the internal state and the current event. The possible sequences define a graph of events, each event pointing to one or more others, and one designated *initial*. The lower level can thus be viewed as a function from internal states and nodes in the graph to new internal states and new nodes in the graph.

An internal state is more detailed than a high-level state; it includes the high-level state and also the state of some internal registers. The latter are collectively called the *transient*. The graph is traversed by starting at a fixed node at a fixed time, and moving from node to node until the initial node is

reached again. The host machine is modelled with an explicit notion of time in the sequencing and accumulation of the effects. (This level of description is a first step toward a description of the Viper hardware.)

The host and target machine time scales are different, but coincide at intervals. In fact, the host machine provides a specific event (at the beginning of each sequence) during which registers may be examined. Because the notions of time are related in this way, the proof that the two levels of description correspond is not logically difficult. It involves examining each possible sequence of events at the lower level and relating its cumulative effect (the visible part of it) to the corresponding state transition at the higher level. As it happens, there are twenty-four such sequences, hence twenty-four cases to consider in the proof.

To give an idea of what is being proved, and how the proof is generated mechanically, one of the twenty-four cases is examined in some detail: the execution of a procedure call to a literal address.

1.3 The Top Level Correctness Statement

We have formulated and proved the following statement of correctness for Viper. All concepts are explained in detail later in the paper.

Theorem 1: The Top Level Correctness of Viper

```
|- HOST_NEXT_SIG(state_sig,transient_sig,node_sig) /\
   AT node_sig #10000 n ==>
   let d = NUMBER_OF_STEPS(state_sig n) in
   NEXT_TIME(n,n+d)(AT node_sig #10000) /\
   (state_sig(n+d) = NEXT(state_sig n))
```

The terms ending with `_sig` are signals: functions from time (a number) to something else: `state_sig` is a function from time to states, and `state_sig n` means the state at time `n`; similarly for `transient_sig` and `node_sig`. `HOST_NEXT_SIG` is an abbreviation for:

```
!n.(state_sig(n+1),transient_sig(n+1)),node_sig(n+1) =
   HOST_NEXT((state_sig n,transient_sig n),node_sig n)
```

`HOST_NEXT` represents the host machine. `HOST_NEXT_SIG` holds of a state-, transient- and node-signal if the host machine always (for every `n`) takes the state, transient and node at the current time and returns the state, transient and node at the next time. `AT` is defined by:

```
AT f x n = (f n = x)
```

It means: `f` at time `n` is `x`. `NEXT` is the high-level or functional specification. `NEXT_TIME(n1,n2) P` means that `n2` is the next time after `n1` that the predicate

P is true. NUMBER_OF_STEPS takes a state s and returns the total number of node transitions required to return to the initial node starting in state s. The initial node happens to be called #10000.

Therefore the correctness statement means that if:

- HOST_NEXT applied to the state, transient and node at any time gives the state, transient and node at the next time, and
- the node at some particular time n is #10000

then

- d is the number of steps it takes to return to node #10000, and
- after d steps, the state attained by the host (HOST_NEXT) is the same as the state specified functionally (by NEXT).

(The transient does not matter in this comparison; it is just used by HOST_NEXT in its graph traversal.)

The correctness statement is deceptively compact. To prove it requires computing for each possible path through the graph the number of steps that path comprises and the final state accumulated. Each final state must be compared to the state specified at the higher level, under the conditions that caused that particular path to be chosen. The proof depends on the definitions of NEXT and HOST_NEXT, as well as on various properties of numbers.

In this paper, one such path (the execution of a literal *call* instruction) is examined in detail to illustrate the nature of the proof. Two main theorems are proved for this path; the first is discussed in Section 5.2.3, and the second in Sections 5.3 and 5.5.

In this case the first theorem has the form:

Theorem 2: The Number of Steps for the Call Path

```
C(state_sig n) /\
HOST_NEXT_SIG(state_sig,transient_sig,node_sig) /\
(node_sig n = #10000) ==>
NEXT_TIME(n,n+4)(AT node_sig #10000)
```

where C is some property of the state which causes the literal *call* path through the graph to be chosen. The theorem says: given that condition C applies to the state at some time n, and given that the state, transient and node at any time are computed by HOST_NEXT applied to the state, transient and node at the previous time, and given that the node at time n is #10000, then n+4 happens to be the next time at which the node is again #10000.

The second theorem has the form:

Theorem 3: Equivalence of Host and Specification for the Call Case

```
C(state_sig n) /\
HOST_NEXT_SIG(state_sig,transient_sig,node_sig) /\
(node_sig n = #10000) ==>
(state_sig(n+4) = NEXT(state_sig n))
```

This states the correctness of Viper for the one kind of instruction. It says that under condition `C`, and given that `HOST_NEXT` computes the successive states, transients and nodes, and starting at node `#10000` at time `n`, the state attained by the host machine at time `n+4` agrees with the high-level transformation of the time-n state.

A similar pair of theorems is proved for each of the twenty-four possible paths through the graph. To tie these together into the main theorem, it has to be shown that at each node, the conditions for choosing the next node cover all possibilities. Then the main theorem follows by analysis of all logical cases.

2 The Design of Viper

Viper has a 32-bit memory. Addresses are 20-bit words, but the memory is addressed by 21-bit words, where the most significant bit distinguishes main from peripheral memory; peripheral memory is for input-output operations.

Instructions are interpreted in the context of a memory (`ram`) of 32-bit words, a 20-bit program counter (`p`), three 32-bit registers (accumulator `a` and index registers `x` and `y`), a 1-bit register (`b`, to hold the results of comparisons, *etc*) and a flag for stopping the machine (`stop`) should an anomolous situation arise. These seven components comprise the configurations (or states) described by the functional specification.

Instructions are 32 bit words, segmented as follows:

2 bits	2 bits	3 bits	1 bit	4 bits	20 bits
source register field	memory address control field	destination control field	compare flag	function field	address

Each field is a bit string. The first five are represented by the variables `rsf`, `msf`, `dsf`, `csf` and `fsf`, respectively.

The top twelve bits encode the register source, the memory source, the destination, and the function part of an instruction. The bottom twenty bits are the address. The source register field can hold the values 0, 1, 2 or 3. These values respectively indicate that the source register for the operation is the `a`, `x`, or `y` register, or the program counter. The memory address control field can also

hold the values 0, 1, 2 or 3; these indicate that the memory source is the literal address of the instruction, the contents of the address, or the contents of the address offset by the value in the x or y register. The destination control field can hold the values 0,...,7, which indicate the destination of the operation. The a, x and y registers are indicated by values 0, 1 and 2, respectively; the program counter by 3; the program counter *if* the b-flag is set, by 4; the program counter *if* the b-flag is not set, by 5; and the location given by the 20-bit address field (in peripheral or main memory) by 6 and 7, respectively. The 1-bit compare flag indicates a compare instruction if it holds the value 1, and a non-compare if 0. The function field can hold values 0,...,15. A *call* instruction is indicated by the value 1; a peripheral memory operation by 2; various other functions which require a memory source by 0 and 3,...,11; and functions which do not need a memory source by 12. The values 13, 14 and 15 are spare instructions whose attempted use indicates an error.

3 The Specification in HOL

To specify the behaviour of Viper in HOL, a hierarchy of logical theories is constructed in which the new types, constants and definitions required can be neatly organized. All of the definitions in Sections 3.1, 3.2 and 4.1 are corrected HOL versions of those in [4] and [5].

3.1 Basic Definitions

A high-level state `(ram,p,a,x,y,b,stop)` is represented in HOL as an object with the following type:

```
mem21_32#word20#word32#word32#word32#bool#bool
```

The following logical constants are used in [4]:

- `AND:bool->(bool->bool)` for the logical $\wedge$
- `OR:bool->(bool->bool)` for the logical $\vee$
- `NOT:bool->bool` for the logical $\neg$

In this paper, we use HOL's /\ and \/ for `AND` and `OR`, respectively, to avoid having two equivalent symbols in different places; in the actual proof `AND`, `OR` and `NOT` are used in the places they occur in [4], and simple substitutions are made so that HOL's inference rules for /\, and \/ and ~ apply. We retain `NOT` in this paper, however, as HOL's ~ is unfortunately rather unreadable.

There are some function constants for acting on bit strings:

- `PAD20TO32:word20->word32` for extending a 20-bit word to 32 bits
- `TRIM32TO20:word32->word20` for truncating a 32-bit word to 20 bits

- `INCP32:word20->word32` for padding a 20-bit word, then incrementing

The first two are constrained by the axiom

Axiom 1: Trim-Pad Axiom

```
|- !w. TRIM32TO20(PAD20TO32 w) = w
```

The function `REG` for selecting a source register according to the source register field `rsf` has the type

```
REG:word2#word32#word32#word32#word20->word32
```

and is defined by

```
|- REG(rsf,a,x,y,p) =
   let r = VAL2 rsf in
   ((r=0) => a | (r=1) => x | (r=2) => y | PAD20TO32 p)
```

The function `INSTFETCH` to fetch from main memory according to the program counter has the type

```
INSTFETCH:mem21_32#word20->word32
```

and is defined below. A 21-bit address is formed from `p` and `F`.

```
|- INSTFETCH(ram,p) = FETCH21 ram (WORD21(V(CONS F(BITS20 p))))
```

The boolean value (`F` in this case) distinguishes main from peripheral memory. There are functions to extract the various fields of an instruction:

- `R:word32->word2` to extract the source register field
- `M:word32->word2` to extract the memory address field
- `D:word32->word3` to extract the destination field
- `C:word32->word1` to extract the compare field
- `FF:word32->word4` to extract the function field
- `A:word32->word20` to extract the address.

(The name `FF` is used instead of `F`, as in [4], to avoid confusion with the truth value `F`.) There are constants for recognizing certain illegal instructions:

- `INVALID:word32->bool` for detecting invalid addresses
- `ILLEGALCALL:word3#word1#word4->bool` for detecting illegal call instructions
- `ILLEGALPDEST:word3#word1#word4->bool` for detecting illegal uses of the program counter as a destination
- `ILLEGALWRITE:word3#word1#word2->bool` for detecting illegal write instructions

- SPAREFUNC:word3#word1#word4->bool for detecting attempted uses of the spare ALU functions fields.

defined, for example, by:

```
|- INVALID value = NOT(value = PAD20TO32(TRIM32TO20 value))

|- ILLEGALCALL(dsf,csf,fsf) =
   (let df = VAL3 dsf in
    let cf = VAL1 csf in
    let ff = VAL4 fsf in
    (cf=0) /\ ((ff=1) /\ ((df=0) \/ ((df=1) \/ (df=2)))))
```

INVALID tests whether the top twelve bits of a word are actually being used; a valid address can only use the bottom twenty bits. From the definition of INVALID and Axiom 1, it follows that

Theorem 4: Validity of Padded Addresses

```
|- !w. NOT(INVALID(PAD20TO32 w))
```

which means that a 20-bit address padded to 32 bits is always valid.

ILLEGALCALL tests whether an instruction is a *call* whose destination is the a, x or y register. If so it is illegal; the program counter must be the destination of the ALU result, so that the jump to the procedure can occur.

There is a function NOOP for recognizing instructions that are non-operations; it has the type

```
NOOP:word3#word1#bool->bool
```

and is defined by

```
|- NOOP(dsf,csf,b) =
   let df = VAL3 dsf in
   let cf = VAL1 csf in
   (cf=0) /\ (((df=5) /\ b) \/ ((df=4) /\ (NOT b)))
```

An instruction is a non-operation if it is not a comparison and if its destination field holds the value 5 while the b flag is set, or 4 while it's not. (This allows for conditional *call* and *jump* instructions whose conditions fail.)

Next, there is the important function that represents the behaviour of the ALU:

```
ALU:word4#word2#word3#word32#word32#bool->word32#bool#bool
```

ALU takes a function field, a memory address field and a destination field, a register source, a memory source and the b register, and returns a 32-bit result (a memory source), along with values for the b register and the stop flag. At the moment, we are only interested in the behaviour of ALU for *call* functions, in which case only the destination field and memory source matter. The computed result is the memory value given, the b-value is the value given, and the stop-value is true only if the destination is not the program counter, or if the memory source is invalid. (For *call*s, the memory source returned represents the location of the procedure being called.) "..." abbreviates parts of the definition not relevant to *call* instructions.

```
|- ALU(fsf,msf,dsf,r,m,b) =
   let ff = VAL4 fsf in
   let mf = VAL2 msf in
   let df = VAL3 dsf in
   let pwrite = (df=3) \/ ((df=4) \/ (df=5)) in
   ((ff=0) => ... |
    (ff=1) => (m,b,(NOT pwrite) \/ (INVALID m)) |
     (ff=2) => ... |
      (ff=3) => ... |
       (ff=4) => ... |
        (ff=5) => ... |
         (ff=6) => ... |
          (ff=7) => ... |
           (ff=8) => ... |
            (ff=9) => ... |
             (ff=10) => ... |
              (ff=11) => ... | (ff=12) => ... |
               (ff=13) => ... | (ff=14) => ... | ... )
```

The functions VALUE, BVAL and SVAL extract the respective components of the 3-tuple returned by ALU. They are defined simply by

```
|- VALUE(result,b,stop) = result
|- BVAL(result,b,stop) = b
|- SVAL(result,b,stop) = stop
```

It is easy to unfold the definition of ALU and prove:

Theorem 5: The ALU Result for the Call Case

```
|- (VAL4 fsf = 1) ==>
   (ALU(fsf,msf,dsf,r,m,b) =
    m,b,
    (NOT((VAL3 dsf = 3) \/ ((VAL3 dsf = 4) \/ (VAL3 dsf = 5)))) \/
    (INVALID m))
```

Finally, there are three more function constants:

- OFFSET:word2#word20#word32#word32->word32
- NILM:word3#word1#word4->bool
- MEMREAD:mem21_32#word2#word20#word32#word32#bool#bool->word32

defined below. (The definition of ADD32, which adds the contents of two 32-bit registers, is not given here; its importance for now is only that the addition may generate an invalid address. ADD32 returns a 32-bit result and two booleans, so VALUE can be used to extract the result.)

```
|- OFFSET(msf,addr,x,y) =
   let mf = VAL2 msf in
   let addr32 = PAD20TO32 addr in
   ((mf=0) => addr32 | (mf=1) => addr32 |
    (mf=2) => VALUE(ADD32(addr32,x)) | VALUE(ADD32(addr32,y)))

|- NILM(dsf,csf,fsf) =
   let df = VAL3 dsf in let cf = VAL1 csf in
   let ff = VAL4 fsf in
   (cf=0) /\ ((NOT((df=7) \/ (df=6))) /\ (ff=12))

|- MEMREAD(ram,msf,addr,x,y,io,nilm) =
   let m = VAL2 msf in
   (nilm => ... |
    (m=0) => PAD20TO32 addr |
     FETCH21 ram
     (WORD21(V(CONS io(BITS20(TRIM32TO20(OFFSET(msf,addr,x,y))))))))
```

OFFSET returns a memory value according to the memory address control field, an address, and the x and y registers. It either pads the address to 32 bits (if the memory address field indicates that the address is a literal or an indirect address) or adds the x or y register to the address (if it is an offset address).

NILM is a predicate which is true of the parts of an instruction if they indicate that no memory source is required to interpret the instruction. It holds of non-comparisons with non-memory destinations whose ALU operations do not require a memory source.

MEMREAD reads from memory; it takes a memory and a memory address control field, an address, two registers, a flag to distinguish main from peripheral memory, and a flag to indicate whether a memory source is required at all. For instructions which *do* require a memory source: if the memory address control field holds value 0, the literal address is returned (padded to 32 bits); otherwise the contents of the address of the (possibly offset) address is fetched from the appropriate part of memory. (For instructions which do not require a memory source, MEMREAD returns some arbitrary value.)

3.2 The Specification

The high-level specification of Viper can now be stated. It is called NEXT since it takes a state and returns the next state. The HOL definition (with parts not immediately relevant abbreviated as "...") is:

```
|- NEXT(ram,p,a,x,y,b,stop) =
   let fetched = INSTFETCH(ram,p) in
   let newp = TRIM32TO20(INCP32 p) in
   let rsf = R fetched in
   let msf = M fetched in
   let dsf = D fetched in
   let csf = C fetched in
   let fsf = FF fetched in
   let addr = A fetched in
   let df = VAL3 dsf in
   let cf = VAL1 csf in
   let ff = VAL4 fsf in
   let comp = (cf=1) in
   let call = ((cf=0) /\ (ff=1)) in
   let output = ((cf=0) /\ (df=6)) in
   let input = ((cf=0) /\ NOT((df=7) \/ (df=6)) /\ (ff=2)) in
   let io = (output \/ input) in
   let writeop = ((cf=0) /\ ((df=7) \/ (df=6))) in
   let skip = NOOP(dsf,csf,b) in
   let noinc = INVALID(INCP32 p) in
   let illegaladdr = (NOT(NILM(dsf,csf,fsf))) /\
                     ((INVALID(OFFSET(msf,addr,x,y))) /\
                      (NOT skip)) in
   let illegalcl = ILLEGALCALL(dsf,csf,fsf) in
   let illegalsp = SPAREFUNC(dsf,csf,fsf) in
   let illegalonp = ILLEGALPDEST(dsf,csf,fsf) in
   let illegalwr = ILLEGALWRITE(dsf,csf,msf) in
   let source = REG(rsf,a,x,y,newp) in
   (stop =>
    (ram,p,a,x,y,b,T) |
    (noinc \/ illegaladdr) \/
     ((illegalcl \/ illegalsp) \/ (illegalonp \/ illegalwr)) =>
     (ram,newp,a,x,y,b,T) |
     (comp => ... |
      (writeop => ... |
       (skip =>
        (ram,newp,a,x,y,b,F) |
        let m = MEMREAD(ram,msf,addr,x,y,io,NILM(dsf,csf,fsf)) in
        let aluout = ALU(fsf,msf,dsf,source,m,b) in
```

```
(df=0) =>
 (ram,newp,VALUE aluout,x,y,BVAL aluout,SVAL aluout) |
 (df=1) =>
  (ram,newp,a,VALUE aluout,y,BVAL aluout,SVAL aluout) |
  (df=2) =>
   (ram,newp,a,x,VALUE aluout,BVAL aluout,SVAL aluout) |
   (call =>
    (ram,TRIM32TO20(VALUE aluout),a,x,INCP32 p,BVAL aluout,
     SVAL aluout) | ... )))))
```

NEXT first tests whether the stop flag is set, and if so, returns the original state unchanged. Otherwise, it fetches a new instruction from memory according to the program counter, and examines its various fields. It decodes the instruction with a series of tests. The new instruction is either an illegal instruction, a comparison, a write instruction, a non-operation, an ALU operation with the a, x or y register as its destination, a call instruction, or a jump. In each of these nine cases, a new state is determined, representing the state *after* the new instruction has been executed. The new state may have the memory changed, the program counter incremented or otherwise changed, and so on.

Illegal instructions include the five sorts mentioned in 3.1, as well as instructions with illegal addresses. These latter are instructions for which the ALU requires a memory source, the address is invalid, and the operation indicated is *not* a non-operation.

Not all of the possible new states are of interest at the moment; we are really only interested in the conditional branch for call. For a *call*, the memory source for the ALU is provided by MEMREAD (and the register source is selected by REG). The value computed by the ALU, trimmed to 20 bits, is the program counter of the new state, and the incremented original program counter is the y register of the new state. In this way the new state "points to" the address of the procedure being called, and carries information for an eventual return to the original location (plus 1) in the y register. The b register and stop flag of the new state are also set according to the result of the ALU operation.

To arrive at the *call* branch of the conditional, certain conditions must obviously apply. For one thing, the stop flag must be false. Also, the six *illegal* situations must be avoided: first, the incremented program counter must be valid. Second, the address of the new fetched instruction must be legal. (For the example case, the address is literal, *i.e.* the value of the memory address control field is 0, so OFFSET just pads the address; by Theorem 4 a padded address is necessarily valid. Thus the address *is* legal.) Third, the instruction must be a legal *call*; the new destination cannot be the a, x or y register. This excludes the values 0, 1 and 2 for the new destination field. The other three *illegal* conditions must also be avoided. The instruction cannot be a comparison (the compare field does not hold 1) or a write operation (the values 6 and 7 are also excluded for the destination field) or a non-operation. Finally, for the *call*

branch to be selected, the function field must hold the value 1. The complete list of conditions is as follows:

```
1. NOT stop
2. NOT(INVALID(INCP32 p))
3. VAL2(M(INSTFETCH(ram,p))) = 0
4. NOT(ILLEGALCALL(D(INSTFETCH(ram,p)),C(INSTFETCH(ram,p)),
                   FF(INSTFETCH(ram,p))))
5. NOT(SPAREFUNC(D(INSTFETCH(ram,p)),C(INSTFETCH(ram,p)),
                 FF(INSTFETCH(ram,p))))
6. NOT(ILLEGALPDEST(D(INSTFETCH(ram,p)),C(INSTFETCH(ram,p)),
                    FF(INSTFETCH(ram,p))))
7. NOT(ILLEGALWRITE(D(INSTFETCH(ram,p)),C(INSTFETCH(ram,p)),
                    M(INSTFETCH(ram,p))))
8. VAL1(C(INSTFETCH(ram,p))) = 0
9. NOT(VAL3(D(INSTFETCH(ram,p))) = 7) /\
   NOT(VAL3(D(INSTFETCH(ram,p))) = 6)
10. NOT(NOOP(D(INSTFETCH(ram,p)),C(INSTFETCH(ram,p)),b))
11. VAL4(FF(INSTFETCH(ram,p))) = 1
```

4 The Host Machine in HOL

4.1 Event Sequences

The functional specification of Viper gives the new state (after a new instruction is executed) directly from the current state. (The new instruction is implicit in the state because the state includes a memory and a program counter.) At the host machine level, however, there are several stages of computation for each state transition at the higher level. A new instruction is fetched and placed in internal registers. The state and these registers are then transformed in stages until the instruction is completely executed. Only then is a single transition at the time scale of functional specification completed.

For the interpretation of instructions by the host machine, five internal registers (of appropriate size) are used to hold the first five fields of an instruction; these (as mentioned in Section 2) are called `rsf`, `msf`, `dsf`, `csf` and `fsf`. In addition, a multi-purpose 32-bit register (`t`) is used to hold a (padded) address or various other information. These six registers together comprise what is called the *transient*, and are invisible to the functional specification. The type of the transient `(t,rsf,msf,dsf,csf,fsf)` is:

```
word32#word2#word2#word3#word1#word4
```

The host machine interprets an instruction by executing a sequence of (from three to seven) events. Each event in a sequence can affect the state and/or the transient. The event, in the context of the state and transient, determines whether there is a next event to be executed and if so what it is.

For example, a stop is a simple event during which the stop flag is set. It is always the last event in the sequence in which it occurs.

An instruction fetch is an event during which a new instruction is found in memory according to the program counter, and its various fields placed in the appropriate registers of the transient. The new address is placed in the t register. The state is affected only insofar as the program counter is imcremented, and possibly the stop flag set, depending on the new instruction and new program counter. Whether there is a next event in the sequence, and if so what it is, is determined by inspection of the new state and transient. The next event may be one of several, including the preparation for a call, or a stop event.

The preparation for a call causes the y register of the state to contain the program counter, and the stop flag to be set to false since nothing new can go wrong at this point; the transient is not affected. The next event must be an ALU operation. (The purpose is to save the program counter so that it can be restored after the called procedure is finished.)

Performing an operation can mean either performing a comparison or performing an ALU operation. Performing an ALU operation is an event during which either the a, x, or y register or the program counter receives a value computed by the ALU. The transient does not change. The next event is based on the result computed by the ALU; it is either a stop event, or the end of the sequence in which the ALU event occurred is signalled.

Thus the following sequence of events is possible:

- Fetch a new instruction
- Prepare for a call
- Perform an ALU operation

For allowing the stop flag to be noticed, a dummy event is placed at the beginning of the sequence, during which the state and transient remain unchanged. The event following a dummy event is an instruction fetch, unless the stop flag is set, so the sequence of events is:

- Dummy event
- Fetch a new instruction
- Prepare for a call
- Perform ALU operation

Viper is modelled at the host level as continuously running. Its infinite sequence of events consists of repetitions of twenty-four possible finite sequences each beginning with a dummy event and ending as soon as the end of the sequence is indicated (*i.e.* a dummy event is the necessary next event).

Events are associated with numbers corresponding to nodes in the transition graph. (The numbers look random here because not all the possible events have been revealed.) The numbers are represented by 5-bit strings.

16. DUMMY, 8. STOP, 1. FETCH, 3. PRECALL, 4. PERFORM

In HOL, the functions DUMMY, STOP, FETCH, PRECALL and PERFORM formalize the five events mentioned so far. The auxiliary functions FMOVE and PMOVE compute the next event after a FETCH and PERFORM event respectively. As usual, "..." abbreviates parts of the definition not relevant to the *call* sequence.

```
|- DUMMY((ram,p,a,x,y,b,stop),t,rsf,msf,dsf,csf,fsf) =
   let stopstate = WORD5 8 in
   let fetch = WORD5 1 in
   (stop =>
    (((ram,p,a,x,y,b,T),t,rsf,msf,dsf,csf,fsf),stopstate) |
    (((ram,p,a,x,y,b,F),t,rsf,msf,dsf,csf,fsf),fetch))


|- STOP((ram,p,a,x,y,b,stop),t,rsf,msf,dsf,csf,fsf) =
   let dummy = WORD5 16 in
   ((ram,p,a,x,y,b,T),t,rsf,msf,dsf,csf,fsf),dummy


|- FMOVE(msf,dsf,csf,fsf,b) =
   let mf = VAL2 msf in
   let df = VAL3 dsf in
   let cf = VAL1 csf in
   let ff = VAL4 fsf in
   let b' = HD(BITS1 b) in
   let noop = NOOP(dsf,csf,b') in
   let precall = WORD5 3 in
   let stopstate = WORD5 8 in
   let dummy = WORD5 16 in
   let ... = ... in let ... = ... in let ... = ... in
   let ... = ... in let ... = ... in let ... = ... in
   ((cf=1) =>  ((mf=0) => ... | ... )|
   ((df=7) =>
    ((mf=0) => ... | ((mf=1) => ... | ...)) |
    ((df=6) =>
     ((mf=0) => ... | ((mf=1) => ... | ... )) |
     (noop => dummy |
      ((mf=0) =>
       (((cf=0) /\ (ff=1)) => precall | ... ) |
       ((mf=1) =>
        (((cf=0) /\ (ff=2)) => ... |
         (((cf=0) /\ (ff=12)) => ... | ... )) |
         (((cf=0) /\ (ff=12)) => ... | ... )))))))
```

```
|- FETCH((ram,p,a,x,y,b,stop),t,rsf,msf,dsf,csf,fsf) =
   let fetched = INSTFETCH(ram,p) in
   let newp = TRIM32TO20(INCP32 p) in
   let newr = R fetched in let newm = M fetched in
   let newd = D fetched in let newc = C fetched in
   let newf = FF fetched in let newt = PAD20TO32(A fetched) in
   let notinc = INVALID(INCP32 p) in
   let illegalcl = ILLEGALCALL(newd,newc,newf) in
   let illegalsp = SPAREFUNC(newd,newc,newf) in
   let illegalonp = ILLEGALPDEST(newd,newc,newf) in
   let illegalwr = ILLEGALWRITE(newd,newc,newm) in
   let stopstate = WORD5 8 in let b' = WORD1(V[b]) in
   (notinc \/
    (illegalcl \/ (illegalsp \/ (illegalonp \/ illegalwr))) =>
    (((ram,newp,a,x,y,b,T),newt,newr,newm,newd,newc,newf),stopstate) |
    (((ram,newp,a,x,y,b,F),newt,newr,newm,newd,newc,newf),
     FMOVE(newm,newd,newc,newf,b')))

|- PRECALL((ram,p,a,x,y,b,stop),t,rsf,msf,dsf,csf,fsf) =
   let perform = WORD5 4 in
   ((ram,p,a,x,PAD20TO32 p,b,F),t,rsf,msf,dsf,csf,fsf),perform

|- PMOVE halt =
   let stopstate = WORD5 8 in
   let dummy = WORD5 16 in (halt => stopstate | dummy)

|- PERFORM((ram,p,a,x,y,b,stop),t,rsf,msf,dsf,csf,fsf) =
   let comp = (VAL1 csf = 1) in
   let df = VAL3 dsf in
   let source = REG(rsf,a,x,y,p) in
   let dummy = WORD5 16 in
   (comp => (((...,...,...,...,...,...,F),
             ...,...,...,...,...,...), ... ) |
    let result = ALU(fsf,msf,dsf,source,t,b) in
    let ans = VALUE result in let newb = BVAL result in
    let halt = SVAL result in
    ((df=0) =>
     (((ram,p,ans,x,y,newb,halt),t,rsf,msf,dsf,csf,fsf),PMOVE halt) |
     (df=1) =>
      (((ram,p,a,ans,y,newb,halt),t,rsf,msf,dsf,csf,fsf),PMOVE halt) |
      (df=2) =>
       (((ram,p,a,x,ans,newb,halt),t,rsf,msf,dsf,csf,fsf),PMOVE halt) |
       (((ram,TRIM32TO20 ans,a,x,y,newb,halt),t,rsf,msf,dsf,csf,fsf),
        PMOVE halt)))
```

4.2 The Transition Graph

Each of the events defined in Section 4.1 determines one or more subsequent events. Together they determine a transition graph of events, part of which is already known:

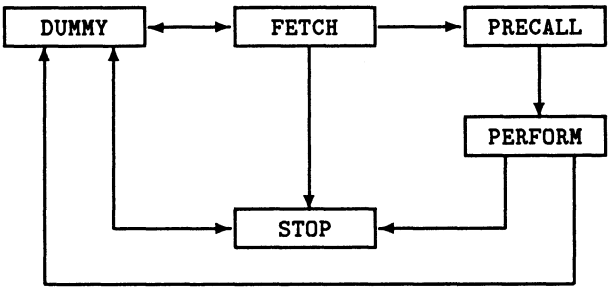


Figure 1: The Graph of Events

Each possible path through the graph, starting and ending at the DUMMY node, can be shown in a tree:

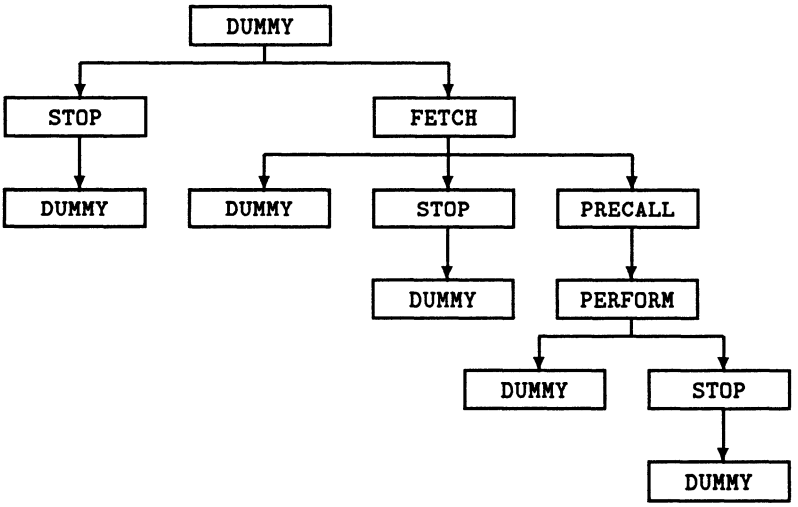


Figure 2: Paths through the Graph

Of the five possible sequences, we only consider in this paper the one containing PRECALL which does not stop. It is clear from the event definitions that where there is a choice of next event, certain conditions must hold in order that the desired next event be chosen. As DUMMY is executed, FETCH will be chosen if

the stop flag is set to false. As FETCH is executed, STOP can be avoided if none of the five *illegal* conditions apply to the new fetched instruction. Furthermore, PRECALL will be chosen if the new value of the compare field is not 1 (*i.e.* is 0); the new value of the destination field neither 7 nor 6; NOOP is false of the destination field, compare field and b flag; the new value of the memory address control field is 0; and the new value of the function field is 1. As PERFORM is executed, STOP is again avoided if *either* the current value of the compare field is 1 *or* the stop value returned by performing the ALU operation is false; that is, if NOT(SVAL(ALU(fsf,msf,dsf,REG(rsf,a,x,y,p),t,b))) holds for the values in the various registers when PERFORM is executed. PRECALL gives no choice of next node.

To formalize these conditions, the predicates c1, c3 and c17, from states to boolean values, are introduced. (The conjunction of these was called C in Section 1.3. The twenty four paths determine in all thirty-four different conditions of which these are three.)

The examinations of the fields that take place must take into account that in the particular sequence DUMMY, FETCH, PRECALL, PERFORM, after FETCH is executed, it is the fields of the *new* instruction that occupy the transient registers. To make the desired choice of next node during DUMMY, we require:

```
c1(ram,p,a,x,y,b,stop) = NOT stop
```

To make the correct choice during FETCH:

```
c3(ram,p,a,x,y,b,stop) =
(NOT
 ((INVALID(INCP32 p)) \/
  ((ILLEGALCALL(D(INSTFETCH(ram,p)),C(INSTFETCH(ram,p)),
                FF(INSTFETCH(ram,p)))) \/
   ((SPAREFUNC(D(INSTFETCH(ram,p)),C(INSTFETCH(ram,p)),
               FF(INSTFETCH(ram,p)))) \/
    ((ILLEGALPDEST(D(INSTFETCH(ram,p)),C(INSTFETCH(ram,p)),
                   FF(INSTFETCH(ram,p)))) \/
     (ILLEGALWRITE(D(INSTFETCH(ram,p)),C(INSTFETCH(ram,p)),
                   M(INSTFETCH(ram,p)))))))) /\
((NOT(VAL1(C(INSTFETCH(ram,p))) = 1)) /\
 ((NOT((VAL3(D(INSTFETCH(ram,p))) = 7) \/
       (VAL3(D(INSTFETCH(ram,p))) = 6))) /\
  ((NOT(NOOP(D(INSTFETCH(ram,p)),C(INSTFETCH(ram,p)),b))) /\
   ((VAL2(M(INSTFETCH(ram,p))) = 0) /\
    ((VAL1(C(INSTFETCH(ram,p))) = 0) /\
     (VAL4(FF(INSTFETCH(ram,p))) = 1))))))
```

and to make the correct choice during PERFORM:

```
c17(ram,p,a,x,y,b,stop) =
(VAL1(C(INSTFETCH(ram,p))) = 1) \/
(NOT(SVAL
     (ALU
      (FF(INSTFETCH(ram,p)),M(INSTFETCH(ram,p)),D(INSTFETCH(ram,p)),
       REG(R(INSTFETCH(ram,p)),a,x,PAD20TO32(TRIM32TO20(INCP32 p)),
           TRIM32TO20(INCP32 p)),PAD20TO32(A(INSTFETCH(ram,p))),b))))
```

Some elementary inference can be done on the combined conditions: since the function field holds 1, the compare field holds 0 and the call is not illegal, it follows that the destination field holds neither 0, 1 nor 2. Indeed, since it does not hold 6 or 7 either, and its value has to be 0,...,7, it must hold 3, 4 or 5. That is:

```
|- NOT(ILLEGALCALL(dsf,csf,fsf)) /\
   (VAL1 csf = 0) /\ NOT(VAL3 dsf = 7) /\ NOT(VAL3 dsf = 6) /\
   (VAL4 fsf = 1)
   ==>
   ((VAL3 dsf = 3) \/ (VAL3 dsf = 4) \/ (VAL3 dsf = 5)) /\
   NOT(VAL3 dsf = 0) /\ NOT(VAL3 dsf = 1) /\ NOT(VAL3 dsf = 2)
```

In particular it follows that:

Theorem 6: Legal Call Destinations

```
|- NOT(ILLEGALCALL(D(INSTFETCH(ram,p)),C(INSTFETCH(ram,p)),
                   FF(INSTFETCH(ram,p)))) /\
   (VAL1(C(INSTFETCH(ram,p))) = 0) /\
   NOT(VAL3(D(INSTFETCH(ram,p))) = 7) /\
   NOT(VAL3(D(INSTFETCH(ram,p))) = 6) /\
   (VAL4(FF(INSTFETCH(ram,p))) = 1)
   ==>
   ((VAL3(D(INSTFETCH(ram,p))) = 3) \/
   (VAL3(D(INSTFETCH(ram,p))) = 4) \/
   (VAL3(D(INSTFETCH(ram,p))) = 5)) /\
   NOT(VAL3(D(INSTFETCH(ram,p))) = 0) /\
   NOT(VAL3(D(INSTFETCH(ram,p))) = 1) /\
   NOT(VAL3(D(INSTFETCH(ram,p))) = 2)
```

Because the value of the compare field is 0, it follows from c3 and c17 that:

```
NOT(SVAL
    (ALU
     (FF(INSTFETCH(ram,p)),M(INSTFETCH(ram,p)),D(INSTFETCH(ram,p)),
      REG(R(INSTFETCH(ram,p)),a,x,PAD20TO32(TRIM32TO20(INCP32 p)),
          TRIM32TO20(INCP32 p)),PAD20TO32(A(INSTFETCH(ram,p))),b)))
```

In fact, this last fact also follows from c3, Theorem 4, Theorem 5 and Theorem 6, so c17 is actually redundant for this path. We keep it as a reminder that there *is* a choice of next node during PERFORM.

The three conditions and their corollaries give all of the conditions for choosing the *call* branch of the conditional in the specification, NEXT, in Section 3.2. (The instruction does not have an illegal address for the same reason as before.)

5 The Equivalence Proof of Specification and Host Machine

What has to be proved is that *under the conditions* c1, c3 and c17, the functional specification agrees with the host machine on the visible state. The first thing to establish is what that state is; the second is whether NEXT gives that state.

The events comprising the host machine (Section 4.1) only say implicitly what the transformation is; they determine a sequence of events (a path through the graph shown in Section 4.2) which transforms the initial state and transient in stages. The transformation can be made *more* explicit by defining a "control" function (HOST_NEXT) linking the events. (Branches which are not needed at the moment are filled in by "...".) The type major abbreviates state#transient. HOST_NEXT has type major#node->major#node. Its partial definition is:

```
|- HOST_NEXT(maj,node) =
   (let nodenum = VAL5 node in
   (nodenum=0)  => ... |          (nodenum=1)  => FETCH maj |
   (nodenum=2)  => ... |          (nodenum=3)  => PRECALL maj |
   (nodenum=4)  => PERFORM maj |  (nodenum=5)  => ... |
   (nodenum=6)  => ... |          (nodenum=7)  => ... |
   (nodenum=8)  => STOP maj |     (nodenum=9)  => ... |
   (nodenum=10) => ... |          (nodenum=11) => ... |
   (nodenum=12) => ... |          (nodenum=13) => ... |
   (nodenum=14) => ... |          (nodenum=15) => ... |
   (nodenum=16) => DUMMY maj |    (nodenum=17) => ... |
   (nodenum=18) => ... |          (nodenum=19) => ... |
   (nodenum=20) => ... |          (nodenum=21) => ... |
   (nodenum=22) => ... |          (nodenum=23) => ... |
   (nodenum=24) => ... |          (nodenum=25) => ... |
   (nodenum=26) => ... |          (nodenum=27) => ... |
   (nodenum=28) => ... |          (nodenum=29) => ... |
   (nodenum=30) => ... |          (nodenum=31) => ... | ... )
```

HOST_NEXT says how to step through the graph. It defines the host machine, which takes a state, transient and node to a new state, transient and node. The new state, transient and node are still not fully explicit; they are computed

by functions such as PRECALL which in turn may either call other functions or return a new major and node. In Sections 5.1 and 5.2 the transformation is made completely explicit.

5.1 A Digression on Forward Proof in HOL

First, each transition in the graph is characterized by saying exactly how HOST_NEXT changes a state and transient and selects a next node during that transition. It is clearly true, for example, by the definition of HOST_NEXT and the fact that VAL5 #00011 = 3 that

```
|- HOST_NEXT(((ram,p,a,x,y,b,stop),t,rsf,msf,dsf,csf,fsf),#00011) =
   PRECALL((ram,p,a,x,y,b,stop),t,rsf,msf,dsf,csf,fsf)
```

and that by the definition of PRECALL

```
|- PRECALL((ram,p,a,x,y,b,stop),t,rsf,msf,dsf,csf,fsf) =
   let perform = WORD5 4 in
   ((ram,p,a,x,PAD20TO32 p,b,F),t,rsf,msf,dsf,csf,fsf),perform
```

By definition of let and the fact that WORD5 4 = #00100, it follows that

```
|- (let perform = WORD5 4 in
    ((ram,p,a,x,PAD20TO32 p,b,F),t,rsf,msf,dsf,csf,fsf),perform) =
   ((ram,p,a,x,PAD20TO32 p,b,F),t,rsf,msf,dsf,csf,fsf),#00100
```

hence

Theorem 7: HOST_NEXT for Node 3

```
|- HOST_NEXT(((ram,p,a,x,y,b,stop),t,rsf,msf,dsf,csf,fsf),#00011) =
   ((ram,p,a,x,PAD20TO32 p,b,F),t,rsf,msf,dsf,csf,fsf),#00100
```

Simple as this reasoning appears, it represents a long chain of primitive inferences. To unfold HOST_NEXT the facts NOT(3=0), NOT(3=1), NOT(3=2) and 3=3, as well as (T=>t1|t2) = t1 and (F=>t1|t2) = t2 must be used to rewrite the definition.

Furthermore, each expression of the form let z = t1 in t2 is logically equivalent to (\z.t2)t1. Taking that inference step on the definition of PRECALL gives:

```
|- PRECALL((ram,p,a,x,y,b,stop),t,rsf,msf,dsf,csf,fsf) =
   (\z.
     (\perform.
       ((ram,p,a,x,PAD20TO32 p,b,F),t,rsf,msf,dsf,csf,fsf),perform)
     z)
   (WORD5 4)
```

The next inference step is beta-conversion; expressions of the form `(\z.t2)t1` can be reduced to `t2[t1/z]`, the result of substituting `t1` for free occurrences of `z` in `t2` (subject to restrictions on free variable capture). Taking that step once gives:

```
|- PRECALL((ram,p,a,x,y,b,stop),t,rsf,msf,dsf,csf,fsf) =
   (\perform.
     ((ram,p,a,x,PAD20TO32 p,b,F),t,rsf,msf,dsf,csf,fsf),perform)
   (WORD5 4)
```

and taking it again

```
|- PRECALL((ram,p,a,x,y,b,stop),t,rsf,msf,dsf,csf,fsf) =
   ((ram,p,a,x,PAD20TO32 p,b,F),t,rsf,msf,dsf,csf,fsf),WORD5 4.
```

Transitivity is needed to complete the chain: if `t1 = t2` and `t2 = t3` then `t1 = t3`. The fact that `WORD5 4 = #00100` is also required.

This chain of primitive inferences is the proof of Theorem 7. All of the proofs discussed in this paper are proofs in that sense: a chain of inferences justified by axioms and inference rules of the logic. Obviously, it is not practical to construct proofs of any size manually, nor would they be very interesting to look at, but one likes to know that they exist and *could* be displayed. (The whole correctness proof described in this paper comprises over a million primitive inferences, for example.) In HOL, the general purpose programming language (ML) interfaced to the logic allows the user to write procedures to generate the actual chains of inferences at some level of abstraction above primitive inference steps. (The level of abstraction depends on the cleverness of the procedure.)

A general ML procedure which proves Theorem 7 unfolds (rewrites) the definitions of all new constants (`HOST_NEXT`, `PRECALL`, *etc*), unfolds the definition of `let`, and does beta-conversion repeatedly until there are no lambda expressions left. Wherever possible, the procedure uses facts about numbers, substitutes equals for equals, use the transitivity of equality, and so on. By executing this general procedure, the proof is generated with a single high-level command. In fact, over 7,000 primitive inferences are performed in the course of generating the proof using this procedure. The number is as large as that partly *because* the procedure is so general; a great deal of inference is done in the course of each unfolding (rewriting) in order to find the location of the replacement and build up a new theorem by inference. This saves the user the trouble of specifying exactly each replacement to be made and its location in the structure of a term, or indeed of thinking very hard about how to prove the fact; the procedure is a general way of unfolding *any* definition phrased in terms of previous definitions and `let` expressions. This sort of proof by rewriting is central to HOL (and LCF) methodology; it is at once very powerful and quite expensive. For further discussion of this trade-off, see Section 8. For more on LCF-style rewriting see the LCF manual (Paulson, [15]) and also Paulson [14].

This is an example of *forward* proof: the user supplies a general ML procedure with the definitions it will need (and the bit string #00011 in this case), and HOL unfolds definitions and lets and does routine inferences until there is no more to do. The user does not necessarily have to know in advance what the final term will be; the procedure computes it, and proves it equal to the initial term. This is one mode in which HOL may be used; the other is discussed in Section 5.4.

5.2 Stepping through the Graph of the Host Machine

5.2.1 Node to Node

Using the same procedure, the new state, transient and node that HOST_NEXT gives for DUMMY, STOP, FETCH and PERFORM can also be produced.

For DUMMY all choice can be limited (by properties of conditionals) to the node component.

Theorem 8: HOST_NEXT for Node 16

```
|- HOST_NEXT(((ram,p,a,x,y,b,stop),t,rsf,msf,dsf,csf,fsf),#10000) =
   ((ram,p,a,x,y,b,stop),t,rsf,msf,dsf,csf,fsf),(stop => #01000|#00001)
```

For STOP there is no choice.

Theorem 9: HOST_NEXT for Node 8

```
|- HOST_NEXT(((ram,p,a,x,y,b,stop),t,rsf,msf,dsf,csf,fsf),#01000) =
   ((ram,p,a,x,y,b,T),t,rsf,msf,dsf,csf,fsf),#10000
```

For FETCH, the new stop flag is set if an illegal condition has arisen, and the next node (if not STOP) is selected by FMOVE. The transient receives parts of the new instruction. The choice can be limited to the stop and node components. The procedure uses Theorem 0 as a rewrite rule at the appropriate point. (Theorem 10 is easier to read with some of the let-expressions not unfolded.)

Theorem 10: HOST_NEXT for Node 1

```
|- HOST_NEXT(((ram,p,a,x,y,b,stop),t,rsf,msf,dsf,csf,fsf),#00001) =
   let fetched = INSTFETCH(ram,p) in
   let new_msf = M fetched in
   let new_dsf = D fetched in
   let new_csf = C fetched in
   let new_fsf = FF fetched in
   ((ram,TRIM32TO20(INCP32 p),a,x,y,b,
     (INVALID(INCP32 p)) \/
     ((ILLEGALCALL(new_dsf,new_csf,new_fsf)) \/
      ((SPAREFUNC(new_dsf,new_csf,new_fsf)) \/
       ((ILLEGALPDEST(new_dsf,new_csf,new_fsf)) \/
```

```
    (ILLEGALWRITE(new_dsf,new_csf,new_msf)))))),
 PAD20TO32(A(INSTFETCH(ram,p))),R(INSTFETCH(ram,p)),new_msf,new_dsf,
 new_csf,new_fsf),
((INVALID(INCP32 p)) \/
 ((ILLEGALCALL(new_dsf,new_csf,new_fsf)) \/
  ((SPAREFUNC(new_dsf,new_csf,new_fsf)) \/
   ((ILLEGALPDEST(new_dsf,new_csf,new_fsf)) \/
    (ILLEGALWRITE(new_dsf,new_csf,new_msf))))) =>
 #01000 |
 ((VAL1 new_csf = 1) =>
  ((VAL2 new_msf = 0) => ... |
                         ((VAL2 new_msf = 1) => ... | ... )) |
  ((VAL3 new_dsf = 7) =>
   ((VAL2 new_msf = 0) => ... |
                          ((VAL2 new_msf = 1) => ... | ... )) |
   ((VAL3 new_dsf = 6) =>
    ((VAL2 new_msf = 0) => ... |
                           ((VAL2 new_msf = 1) => ... | ... )) |
    ((VAL1 new_csf = 0) /\
     (((VAL3 new_dsf = 5) /\ b) \/
      ((VAL3 new_dsf = 4) /\ (NOT b))) =>  #10000 |
      ((VAL2 new_msf = 0) =>
      ((VAL1 new_csf = 0) /\ (VAL4 new_fsf = 1) => #00011 | ... ) |
      ((VAL2 new_msf = 1) =>
       ((VAL1 new_csf = 0) /\ (VAL4 new_fsf = 2) => ... |
        ((VAL1 new_csf = 0) /\ (VAL4 new_fsf = 12) =>
         ... | ... )) |
       ((VAL1 new_csf = 0) /\ (VAL4 new_fsf = 12) =>
        ... | ... ))))))))
```

For PERFORM, only the memory component and the transient involve no choice.

Theorem 11: HOST_NEXT for Node 4

```
|- HOST_NEXT(((ram,p,a,x,y,b,stop),t,rsf,msf,dsf,csf,fsf),#00100) =
   ((ram,
     ((VAL1 csf = 1) => ... |
      ((VAL3 dsf = 0) =>  p |
       ((VAL3 dsf = 1) =>  p |
        ((VAL3 dsf = 2) =>  p |
         TRIM32TO20(VALUE(ALU(fsf,msf,dsf,REG(rsf,a,x,y,p),t,b)))))),
     ((VAL1 csf = 1) =>  ... |
      ((VAL3 dsf = 0) =>
       VALUE(ALU(fsf,msf,dsf,REG(rsf,a,x,y,p),t,b)) | a)),
     ((VAL1 csf = 1) =>  ... |
```

```
    ((VAL3 dsf = 0) =>  x |
     ((VAL3 dsf = 1) =>
      VALUE(ALU(fsf,msf,dsf,REG(rsf,a,x,y,p),t,b)) | x))),
   ((VAL1 csf = 1) => ... |
    ((VAL3 dsf = 0) => y |
     ((VAL3 dsf = 1) => y |
      ((VAL3 dsf = 2) =>
       VALUE(ALU(fsf,msf,dsf,REG(rsf,a,x,y,p),t,b)) | y)))),
   ((VAL1 csf = 1) =>
    ... | BVAL(ALU(fsf,msf,dsf,REG(rsf,a,x,y,p),t,b))),
   ((VAL1 csf = 1) =>
    F | SVAL(ALU(fsf,msf,dsf,REG(rsf,a,x,y,p),t,b)))),
  t,rsf,msf,dsf,csf,fsf),
 ((VAL1 csf = 1) =>  ... |
  (SVAL(ALU(fsf,msf,dsf,REG(rsf,a,x,y,p),t,b)) => #01000 | #10000))
```

5.2.2 Making Time Explicit

We now consider making time explicit. Times are represented as numbers. *Signals* are functions from times to registers:

```
ram_sig:num->mem21_32          p_sig:num->word20
  a_sig:num->word32            x_sig:num->word32
  y_sig:num->word32            b_sig:num->bool
stop_sig:num->bool             t_sig:num->word32
 rsf_sig:num->word2           msf_sig:num->word2
 dsf_sig:num->word3           csf_sig:num->word1
 fsf_sig:num->word4
```

`ram_sig n` means the value of `ram` at time `n`, and so on. A theorem of the following form expresses the behaviour of `HOST_NEXT` as it steps through a node `#.....`; a time unit is the time it takes for one event to happen at the host level.

```
|- (!n.
    ((ram_sig(n+1),p_sig(n+1),a_sig(n+1),x_sig(n+1),
      y_sig(n+1),b_sig(n+1),stop_sig(n+1)),t_sig(n+1),
     rsf_sig(n+1),msf_sig(n+1),dsf_sig(n+1),csf_sig(n+1),
     fsf_sig(n+1)),node_sig(n+1) =
    HOST_NEXT
    (((ram_sig n,p_sig n,a_sig n,x_sig n,y_sig n,b_sig n,stop_sig n),
      t_sig n,rsf_sig n,msf_sig n,dsf_sig n,csf_sig n,fsf_sig n),
     node_sig n))
    ==>
   (node_sig n = #.....)
    ==>
```

```
(((ram_sig(n+1) = ...n...) /\
  (p_sig(n+1) = ...n...) /\
  (a_sig(n+1) = ...n...) /\
  (x_sig(n+1) = ...n...) /\
  (y_sig(n+1) = ...n...) /\
  (b_sig(n+1) = ...n...) /\
  (stop_sig(n+1) = ...n...)) /\
 (t_sig(n+1) = ...n...) /\
 (rsf_sig(n+1) = ...n...) /\
 (msf_sig(n+1) = ...n...) /\
 (dsf_sig(n+1) = ...n...) /\
 (csf_sig(n+1) = ...n...) /\
 (fsf_sig(n+1) = ...n...)) /\
(node_sig(n+1) = ...)
```

The first antecedent could be abbreviated by use of HOST_NEXT_SIG as in Section 1.3. It gives a sequence of 14-tuples of signals such that HOST_NEXT applied to the fourteen signals at any time gives the fourteen signals at the next time. The second antecedent fixes a particular event (a node in the graph) at some time n. The fourteen equations then give the value of the signals at time n+1 in terms of the signals at time n, after that node is traversed. For PRECALL (event 3, node #00011), the theorem is:

Theorem 12: Timed HOST_NEXT for Node 3

```
|- (!n.
     ((ram_sig(n+1),p_sig(n+1),a_sig(n+1),x_sig(n+1),
       y_sig(n+1),b_sig(n+1),stop_sig(n+1)),t_sig(n+1),
      rsf_sig(n+1),msf_sig(n+1),dsf_sig(n+1),csf_sig(n+1),
      fsf_sig(n+1)),node_sig(n+1) =
     HOST_NEXT
     (((ram_sig n,p_sig n,a_sig n,x_sig n,y_sig n,b_sig n,stop_sig n),
       t_sig n,rsf_sig n,msf_sig n,dsf_sig n,csf_sig n,fsf_sig n),
      node_sig n)) ==>
    (node_sig n = #00011) ==>
    (((ram_sig(n+1) = ram_sig n) /\
      (p_sig(n+1) = p_sig n) /\
      (a_sig(n+1) = a_sig n) /\
      (x_sig(n+1) = x_sig n) /\
      (y_sig(n+1) = PAD20TO32(p_sig n)) /\
      (b_sig(n+1) = b_sig n) /\
      (stop_sig(n+1) = F)) /\
     (t_sig(n+1) = t_sig n) /\
     (rsf_sig(n+1) = rsf_sig n) /\
     (msf_sig(n+1) = msf_sig n) /\
```

```
(dsf_sig(n+1) = dsf_sig n) /\
(csf_sig(n+1) = csf_sig n) /\
(fsf_sig(n+1) = fsf_sig n)) /\
(node_sig(n+1) = #00100)
```

This gives the effect over one time unit of passing through the PRECALL node: if the node signal at time n is #00011, then the y register at time n+1 holds the (padded) value of the program counter at time n, and so on. The relation to Theorem 7 is clear. To prove Theorem 12, it is assumed for some n that

```
|- node_sig n = #00011
```

The first antecedent is assumed and instantiated to that n:

Assumption 1: Sequence Assumption

```
|- ((ram_sig(n+1),p_sig(n+1),a_sig(n+1),x_sig(n+1),
    y_sig(n+1),b_sig(n+1),stop_sig(n+1)),t_sig(n+1),
   rsf_sig(n+1),msf_sig(n+1),dsf_sig(n+1),csf_sig(n+1),
   fsf_sig(n+1)),node_sig(n+1) =
  HOST_NEXT
  (((ram_sig n,p_sig n,a_sig n,x_sig n,y_sig n,b_sig n,stop_sig n),
    t_sig n,rsf_sig n,msf_sig n,dsf_sig n,csf_sig n,fsf_sig n),
   node_sig n)
```

Assumption 1 is unfolded using the first assumption and then by using Theorem 7 (which says what HOST_NEXT does at node #00011). The result can be expressed (using properties of tuples) as fourteen pairwise equalities with all assumptions discharged. This gives Theorem 12.

The same ML procedure which generates the proof of Theorem 12 also generates theorems for DUMMY, STOP, FETCH and PERFORM, which are not shown here but bear a similar relation to Theorems 8-11 (respectively) as Theorem 12 does to Theorem 7.

This is another example of a general procedure being used to generate a forward proof (in the case of Theorem 12, a proof of about 2,200 inferences); one does not have to know in advance the theorem being proved, but only its form.

5.2.3 End Results of Sequences in the Host Machine

Finally, the result of the whole sequence DUMMY, FETCH, PRECALL, PERFORM can be produced. To do this, it is assumed again that for some n, node_sig n = #10000 (we start at DUMMY at time n). The assumption is made which asserts that the signals at any time are the result of HOST_NEXT at the previous time. The three conditions (c1, c3 and c17) are assumed to hold of the state signals at time n, to ensure that the correct choices are made at each node. Then the path

is traversed by referring at each node to the corresponding theorem (Theorem 12 for node 3, *etc*) which describes the effects over the one time unit taken to traverse that node. That causes certain changes in the fourteen components, which are compounded with the changes at the previous node (if there was one). At each stage, the assumed conditions are used to help make decisions about the node signal, *i.e.* the next event. For each node, a theorem follows, describing the accumulated changes at that time.

Again, an ML procedure is written to generate the proof. It is a recursive procedure which refers initially to the theorem (analagous to Theorem 12) for node #10000, dismisses the antecedents because these have already been assumed, and then uses the pairwise equalities to rewrite the components and select the next node. Successive theorems are generated by examining the node, and referring to the theorem about HOST_NEXT for that node. The process continues until the first appearance of node #10000 as the next node. At each stage, more complex simplifications must also be done; for example, where there is a choice of next node, the conditional expression denoting the next node has to be reduced to a particular bit string by using assumed facts such as the fact that c17 holds of the state at time n. At some stages, the procedure must also use lemmas; for example, in the final stage of the literal *call* path, Theorem 6 is required to avoid choices involving a destination of 0, 1 or 2. Under the assumptions

```
1. !n.
    ((ram_sig(n+1),p_sig(n+1),a_sig(n+1),x_sig(n+1),
      y_sig(n+1),b_sig(n+1),stop_sig(n+1)),t_sig(n+1),
     rsf_sig(n+1),msf_sig(n+1),dsf_sig(n+1),csf_sig(n+1),
     fsf_sig(n+1)),node_sig(n+1) =
    HOST_NEXT
    (((ram_sig n,p_sig n,a_sig n,x_sig n,y_sig n,b_sig n,stop_sig n),
      t_sig n,rsf_sig n,msf_sig n,dsf_sig n,csf_sig n,fsf_sig n),
     node_sig n)
2. node_sig n = #10000
3. c1(ram_sig n,p_sig n,a_sig n,x_sig n,y_sig n,b_sig n,stop_sig n)
4. c3(ram_sig n,p_sig n,a_sig n,x_sig n,y_sig n,b_sig n,stop_sig n)
5. c17(ram_sig n,p_sig n,a_sig n,x_sig n,y_sig n,b_sig n,stop_sig n)
```

four theorems are proved. These, conjoined, are called **Theorem 13.**

At time n+1, DUMMY has been executed without having to stop.

```
|- ((((ram_sig(n+1) = ram_sig n) /\
      (p_sig(n+1) = p_sig n) /\
      (a_sig(n+1) = a_sig n) /\
      (x_sig(n+1) = x_sig n) /\
      (y_sig(n+1) = y_sig n) /\
      (b_sig(n+1) = b_sig n) /\
      (stop_sig(n+1) = F)) /\
     (t_sig(n+1) = t_sig n) /\
     (rsf_sig(n+1) = rsf_sig n) /\
```

```
    (msf_sig(n+1) = msf_sig n) /\
    (dsf_sig(n+1) = dsf_sig n) /\
    (csf_sig(n+1) = csf_sig n) /\
    (fsf_sig(n+1) = fsf_sig n)) /\
   (node_sig(n+1) = #00001))
```

At time n+2 FETCH has been executed, so the program counter is incremented, the transient fields contains the fields of the new instruction, and the t register holds the new address.

```
|- ((((ram_sig(n+2) = ram_sig n) /\
     (p_sig(n+2) = TRIM32TO20(INCP32(p_sig n))) /\
     (a_sig(n+2) = a_sig n) /\
     (x_sig(n+2) = x_sig n) /\
     (y_sig(n+2) = y_sig n) /\
     (b_sig(n+2) = b_sig n) /\
     (stop_sig(n+2) = F)) /\
    (t_sig(n+2) = PAD20TO32(A(INSTFETCH(ram_sig n,p_sig n)))) /\
    (rsf_sig(n+2) = R(INSTFETCH(ram_sig n,p_sig n))) /\
    (msf_sig(n+2) = M(INSTFETCH(ram_sig n,p_sig n))) /\
    (dsf_sig(n+2) = D(INSTFETCH(ram_sig n,p_sig n))) /\
    (csf_sig(n+2) = C(INSTFETCH(ram_sig n,p_sig n))) /\
    (fsf_sig(n+2) = FF(INSTFETCH(ram_sig n,p_sig n)))) /\
   (node_sig(n+2) = #00011))
```

At time n+3 PRECALL has been executed, so the y register holds the last value of the program counter.

```
|- ((((ram_sig(n+3) = ram_sig n) /\
     (p_sig(n+3) = TRIM32TO20(INCP32(p_sig n))) /\
     (a_sig(n+3) = a_sig n) /\
     (x_sig(n+3) = x_sig n) /\
     (y_sig(n+3) = INCP32(p_sig n)) /\
     (b_sig(n+3) = b_sig n) /\
     (stop_sig(n+3) = F)) /\
    (t_sig(n+3) =
     PAD20TO32(A(INSTFETCH(ram_sig n,p_sig n)))) /\
    (rsf_sig(n+3) = R(INSTFETCH(ram_sig n,p_sig n))) /\
    (msf_sig(n+3) = M(INSTFETCH(ram_sig n,p_sig n))) /\
    (dsf_sig(n+3) = D(INSTFETCH(ram_sig n,p_sig n))) /\
    (csf_sig(n+3) = C(INSTFETCH(ram_sig n,p_sig n))) /\
    (fsf_sig(n+3) = FF(INSTFETCH(ram_sig n,p_sig n)))) /\
   (node_sig(n+3) = #00100))
```

At time n+4 PERFORM has been executed. By the definition of PERFORM, the program counter now holds the result (the first value) computed by the ALU *if*

the compare field of the new instruction does not hold 1 (which it doesn't) and *if* the destination field of the new instruction is not 0, 1 or 2 (which it is not). The `b` flag holds the `b` value (the second value) computed by the `ALU`, and the `stop` flag the `stop` value (the third). Based on Theorem 11, the memory source supplied to the `ALU` comes from the `t` register, which at time `n+3` on this path held the (padded) new address. The other arguments are likewise dictated by Theorem 11 and the state and transient at time `n+3`. The whole `ALU` expression is:

```
ALU
(FF(INSTFETCH(ram_sig n,p_sig n)),
 M(INSTFETCH(ram_sig n,p_sig n)),
 D(INSTFETCH(ram_sig n,p_sig n)),
 REG
 (R(INSTFETCH(ram_sig n,p_sig n)),a_sig n,x_sig n,
  PAD20TO32(TRIM32TO20(INCP32(p_sig n))),
  TRIM32TO20(INCP32(p_sig n))),
 PAD20TO32(A(INSTFETCH(ram_sig n,p_sig n))),b_sig n)
```

For convenience this is abbreviated by a new constant, `ALU_ABBR6` (one of eight needed for the twenty-four cases). For *any* `ram`, `p`, `a`, `x`, `y` and `b`:

```
|- ALU_ABBR6(ram,p,a,x,y,b) =
   ALU
   (FF(INSTFETCH(ram,p)),M(INSTFETCH(ram,p)),D(INSTFETCH(ram,p)),
    REG
    (R(INSTFETCH(ram,p)),a,x,PAD20TO32(TRIM32TO20(INCP32 p)),
     TRIM32TO20(INCP32 p)),PAD20TO32(A(INSTFETCH(ram,p))),b)
```

Since it has been assumed that `VAL4(FF(INSTFETCH(ram_sig n,p_sig n))) = 1`, Theorem 5 can be used to compute the result of the `ALU` operation:

```
PAD20TO32(A(INSTFETCH(ram_sig n,p_sig n))),b_sig n,
NOT
 ((VAL3(D(INSTFETCH(ram_sig n,p_sig n))) = 3) \/
  ((VAL3(D(INSTFETCH(ram_sig n,p_sig n))) = 4) \/
   (VAL3(D(INSTFETCH(ram_sig n,p_sig n))) = 5)))) \/
(INVALID(PAD20TO32(A(INSTFETCH(ram_sig n,p_sig n))))
```

The padded address cannot be invalid (Theorem 4), and the destination of the new instruction must be 3, 4 or 5 (Theorem 6), so the `ALU` returns:

```
PAD20TO32(A(INSTFETCH(ram_sig n,p_sig n))),b_sig n,F
```

Thus at time `n+4` `PERFORM` has been executed:

```
|- (((ram_sig(n+4) = ram_sig n) /\
     (p_sig(n+4) = A(INSTFETCH(ram_sig n,p_sig n))) /\
     (a_sig(n+4) = a_sig n) /\
     (x_sig(n+4) = x_sig n) /\
     (y_sig(n+4) = INCP32(p_sig n)) /\
     (b_sig(n+4) = b_sig n) /\
     (stop_sig(n+4) = F)) /\
    (t_sig(n+4) =
     PAD20TO32(A(INSTFETCH(ram_sig n,p_sig n)))) /\
    (rsf_sig(n+4) = R(INSTFETCH(ram_sig n,p_sig n))) /\
    (msf_sig(n+4) = M(INSTFETCH(ram_sig n,p_sig n))) /\
    (dsf_sig(n+4) = D(INSTFETCH(ram_sig n,p_sig n))) /\
    (csf_sig(n+4) = C(INSTFETCH(ram_sig n,p_sig n))) /\
    (fsf_sig(n+4) =
     FF(INSTFETCH(ram_sig n,p_sig n)))) /\
   (node_sig(n+4) = #10000)
```

The next node is #10000, so the sequence is ended. The program counter now holds the address of the procedure to be called, and the y register stores the return address (the original program counter's value plus one).

These four theorems provide the basis for proving Theorem 2. The fourth is part of the proof of Theorem 3. (See Section 1.3). (Here, the theorems are stated in terms of the fourteen components. Various properties of n-tuples are required to formulate them as in Section 1.3.)

5.3 The Equivalence Proof

Now the high-level specification, NEXT (Section 3.2) is considered in relation to the results of the graph traversal. We wish to prove:

Theorem 14: Result of Specification for Call

```
c1(ram,p,a,x,y,b,stop) /\
c3(ram,p,a,x,y,b,stop) /\
c17(ram,p,a,x,y,b,stop) ==>
(NEXT(ram,p,a,x,y,b,stop) = ram,A(INSTFETCH(ram,p)),a,x,INCP32 p,b,F)
```

That is, for any state, NEXT specifies the same changes as have accumulated in the registers at the end of the sequence produced by the host machine (disregarding time).

The proof consists in reducing the conditional in the definition of NEXT. Obviously, stop is false (c1 holds). The five *illegal* conditions mentioned in c3 are all false. Furthermore, the new address cannot be illegal: the new memory address control field holds 0, so OFFSET gives a padded 20-bit address. Since the new compare field also holds 0, comp is false; writeop and skip are also obviously false. The new destination field is neither 0, 1 or 2 (by Theorem 6) and so the

call branch is arrived at. The program counter, the b flag and the stop flag of the new state in this case depend on the result (aluout) computed by ALU. The new state is:

```
(ram,TRIM32TO20(VALUE aluout),a,x,INCP32 p,BVAL aluout,SVAL aluout)
```

In the fully unfolded definition of NEXT, aluout is

```
ALU
(FF(INSTFETCH(ram,p)),M(INSTFETCH(ram,p)),D(INSTFETCH(ram,p)),
 REG(R(INSTFETCH(ram,p)),a,x,y,TRIM32TO20(INCP32 p)),
 MEMREAD
 (ram,M(INSTFETCH(ram,p)),A(INSTFETCH(ram,p)),x,y,
  ((VAL1(C(INSTFETCH(ram,p))) = 0) /\
   (VAL3(D(INSTFETCH(ram,p))) = 6)) \/
  ((VAL1(C(INSTFETCH(ram,p))) = 0) /\
   NOT((VAL3(D(INSTFETCH(ram,p))) = 7) \/
     (VAL3(D(INSTFETCH(ram,p))) = 6)) /\
   (VAL4(FF(INSTFETCH(ram,p))) = 2)),
  ((VAL1(C(INSTFETCH(ram,p))) = 0) /\
   (NOT(VAL3(D(INSTFETCH(ram,p))) = 7) /\
    NOT(VAL3(D(INSTFETCH(ram,p))) = 6)) /\
   (VAL4(FF(INSTFETCH(ram,p))) = 12))),b)
```

which is abbreviated by a new constant ALU_ABBR2, so that the new state specified by NEXT is

```
(ram,TRIM32TO20(VALUE(ALU_ABBR2(ram,p,a,x,y,b))),a,x,
 INCP32 p,BVAL(ALU_ABBR2(ram,p,a,x,y,b)),
 SVAL(ALU_ABBR2(ram,p,a,x,y,b)))
```

Since VAL4(FF(INSTFETCH(ram,p))) = 1, the reasoning is as for ALU_ABBR6 in Section 5.2.3: Theorem 5 is used to conclude that ALU in this case returns MEMREAD(...) as its result, b for the b flag, and

```
NOT((VAL3(D(INSTFETCH(ram,p))) = 3) \/
   ((VAL3(D(INSTFETCH(ram,p))) = 4) \/
    (VAL3(D(INSTFETCH(ram,p))) = 5))) \/
(INVALID(MEMREAD(...)))
```

for the stop flag, where "..." stands for the arguments to MEMREAD.

The value of the new destination field is 3 or 4 or 5, so the stop flag becomes F \/ (INVALID(MEMREAD(...))). It remains to work out the value of MEMREAD(...). Its sixth argument is a boolean value which distinguishes main from peripheral memory, and the seventh, a boolean which indicates whether no memory source is required. The seventh works out to be false, since the value of the function

field is not 12 (but rather 1). The sixth also works out to be false, since the function field is not 2, nor is the destination field 6. Thus, since the memory address control field is 0, MEMREAD returns just PAD20TO32(A(INSTFETCH(ram,p))). This cannot be invalid (Theorem 4). Therefore the ALU returns:

```
PAD20TO32(A(INSTFETCH(ram,p))),b,F
```

so the state returned by NEXT is

```
(ram,TRIM32TO20(PAD20TO32(A(INSTFETCH(ram,p)))),a,x,INCP32 p,b,F)
```

which by Axiom 1 is just

```
(ram,A(INSTFETCH(ram,p)),a,x,INCP32 p,b,F)
```

which is what was wanted.

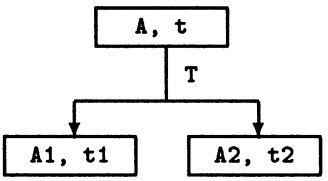
In this case, the memory component (ram) and the a, x and y components of the state given by NEXT agree immediately with the end results of HOST_NEXT. That the other three registers agree depends on an argument about the ALU. The essence of the argument can be expressed by a theorem relating the two values computed by the ALU: the one referring to MEMREAD, in the case of the specification, and the one determined one by Theorem 11 for the PERFORM node (at the correct point in the sequence), in the case of the host machine.

Theorem 15: Equivalence of ALU in Specification and Host

```
|- ((VAL2(M(INSTFETCH(ram,p))) = 0) /\
    (VAL1(C(INSTFETCH(ram,p))) = 0) /\
    NOT(VAL3(D(INSTFETCH(ram,p))) = 7) /\
    NOT(VAL3(D(INSTFETCH(ram,p))) = 6) /\
    (VAL4(FF(INSTFETCH(ram,p))) = 1)) ==>
   (ALU_ABBR2(ram,p,a,x,y,b) = ALU_ABBR6(ram,p,a,x,y,b))
```

5.4 A Digression on Goal-Oriented Proof

For the HOL proof of Theorem 14, we are starting for a change with a *goal*. We know that NEXT should unfold to the state specified by the host machine, in the circumstances; but do not know whether it really does, nor how to attempt to prove that it does. Here, a *goal oriented proof* is called for. One starts with a goal (a term with an associated list of assumptions), and applies strategies or *tactics* to the goal to produce subgoals; then tactics are applied to the subgoals, and so on, until subgoals are reached which are known to be true (that is, which correspond to previously proved theorems or axioms). The tactics justify each decomposition into subgoals so that a proof can be assembled in the end. The proof is built up by inferences, starting with the theorems corresponding to the final set of subgoals. For example, if a tactic T is applied to a goal (A,t) (term t and assumptions A) to produce two subgoals



then the ML procedure which justifies that decomposition expects two theorems

- `|-t1` with no assumptions beyond `A1`, and
- `|-t2` with no assumptions beyond `A2`

and infers a theorem `|-t` (with no assumptions beyond `A`). Tactics can be sequenced and combined in various ways to form compound tactics. In this way, arbitrarily complex proof strategies can be expressed as procedures which generate proofs.

This style of proof revolves around the list of current assumptions in a natural way. Tactics use the same rewriting mechanism as forward proof. One frequently-used tactic generates a subgoal by rewriting the goal using all current assumptions of that goal as rewrite rules. Each rewrite is justified by a chain of inferences which is used used later in assembling the proof.

Sometimes groups of assumptions can logically imply a new assumption; a useful tactic finds new conclusions and adds them to the assumption list. This provides a simple kind of resolution in HOL. For example, it follows from the definition of invalidity (Section 3.1) that

```
|- !p1. NOT(INVALID p1) ==> (PAD20TO32(TRIM32TO20 p1) = p1)
```

This has an instance

```
|- NOT(INVALID(INCP32 p)) ==> (PAD20TO32(TRIM32TO20(INCP32 p))=INCP32 p)
```

which can be used to simplify the expression

```
PAD20TO32(TRIM32TO20(INCP32 p))
```

during the proof of Theorem 14. The instance can be identified and its conclusion added as an assumption as soon as its antecedent, `NOT(INVALID(INCP32 p))`, joins the assumptions. The conclusion is not a *new* assumption because it can always be dismissed by a series of simple inference steps.

5.5 The Equivalence Proof in HOL

To prove Theorem 14 in HOL, first the definitions of `c1`, `c3` and `c17` are unfolded. The definition of `NEXT` is unfolded, using a version of the definition without the `let` and lambda expressions (analagous to the version of the definition of `PRECALL`

in Section 5.1). (Part of this unfolding has been seen in the form of the ALU expression, Section 5.3.) The assumptions are "resolved" with Theorem 6, all of whose antecedents are by this point among the assumptions. This adds new members to the list of current assumptions of the subgoal, including:

- `NOT(VAL3(D(INSTFETCH(ram,p))) = 0)`
- `NOT(VAL3(D(INSTFETCH(ram,p))) = 1)`
- `NOT(VAL3(D(INSTFETCH(ram,p))) = 2)`

The subgoal can now be rewritten using all of its current assumptions as well as facts such as Axiom 1 and Theorem 5 (to reduce the ALU expressions). By this process the NEXT expression (a conditional with ten branches) reduces to the correct case, and the state in that case to the desired state: both sides of the subgoal are equal.

A subgoal with a term `t=t` is known to correspond to an axiom: namely, reflexivity. From this point HOL starts with the axiom of reflexivity and builds up the proof by using the justification of each decomposition into subgoals. The book-keeping is all done by HOL, the user only supplying the tactics. The whole proof can be generated with one command, given the correct (compound) tactic. The point about forward proof is that it would be very difficult to structure a forward proof effort starting from `|-t=t` and applying inference rules to produce a theorem as complex as Theorem 14.

The only real tricks in this process are supplying the lemmas needed, and causing the rewriting to happen in an efficient way so the proof can be done in a reasonable amount of time. The former requires the proof effort to be planned and structured sensibly. The latter is discussed further in Section 8.

6 The Rest of the Proof

So far, the specification and host machine have only been revealed insofar as they concern the literal *call* instruction. This in fact gives a good idea of what is claimed and proved in the other cases, and how it is stated and proved in HOL. The literal *call* proofs are about average in length and difficulty among the twenty-four cases. The other four paths suggested in Figure 2 are for states in which the `stop` flag is set; *call* instructions for which the ALU sets the `stop` flag; states for which fetching produces an illegal instruction; and states for which the new fetched instruction is a non-operation. The other nineteen paths are for *call* instructions with indirect or offset addresses; ALU operations with the `a`, `x` or `y` register as destination; write operations to memory or output; read operations from memory or input; and variations of the above with differents sorts of addressing and in which illegal situations arise and the machine stops. The tree of all possible paths can be seen in [5].

In fact, the definitions of NEXT and the host machines in the example proofs have been unfolded to their maximum depth, but that is not necessary for doing

the proof. As soon as they are unfolded sufficiently, the host and specification are seen to be equal (by means of Theorem 15 and so on), and the proof is completed.

A large number of other lemmas and intermediate results were proved *en route* to the main theorems and not mentioned here; no one is very complicated, but they add up to a certain amount of unseen effort. Other results have been mentioned only in passing (see tables in Section 8).

7 Errors Found in the Viper Specifications

Aside from certain HOL conventions, such as consistently currying or uncurrying a function, and some corrections of typographical errors, the HOL versions of the functional specification and the host differ from the definitions in [4] at a few places where we found errors in the host and high-level specification. There was a type error in the function FMOVE which was passed a boolean b by FETCH when FMOVE was expecting a 1-bit string. More seriously, there were some other errors in FETCH, and one in NEXT. The function for FETCH had read:

```
|- FETCH((ram,p,a,x,y,b,stop),t,rsf,msf,dsf,csf,fsf) =
   let fetched = INSTFETCH(ram,p) in
   let newp = TRIM32TO20(INCP32 p) in
   let newr = R fetched in
   let newm = M fetched in
   let newd = D fetched in
   let newc = C fetched in
   let newf = FF fetched in
   let newt = PAD20TO32(A fetched) in
   let notinc = INVALID(INCP32 newp) in
   let illegalcl = ILLEGALCALL(dsf,csf,fsf) in
   let illegalsp = SPAREFUNC(dsf,csf,fsf) in
   let illegalonp = ILLEGALPDEST(dsf,csf,fsf)in
   let illegalwr = ILLEGALWRITE(dsf,csf,msf) in
   let stopstate = WORD5 8 in
   let b' = WORD1(V[b]) in
   (notinc \/
    (illegalcl \/ (illegalsp \/ (illegalonp \/ illegalwr))) =>
    (((ram,newp,a,x,y,b,T),newt,newr,newm,newd,newc,newf),stopstate) |
    (((ram,newp,a,x,y,b,F),newt,newr,newm,newd,newc,newf),
     FMOVE(newm,newd,newc,newf,b')))
```

so that the *phase* of fetch cycle was confused: it is actually the incremented program counter one wants to check for validity, not the twice incremented counter, and it is the *new* instruction one wants to check for legality, not the instruction that rested in the transient before the FETCH operation. This error

became apparent in the course of doing the proof because the host's results disagreed with the specification's results.

The original specification had an incomplete check for illegal instructions; the possibility of the instruction being a non-operation was neglected. The relevant part read:

```
... in let illegaladdr = (NOT(NILM(dsf,csf,fsf)) /\
                          (INVALID(OFFSET(msf,addr,x,y)))) in ...
```

This naturally caused a problem with interpreting non-operations whose fields met the two conditions above but which were meant to be legal instuctions.

Finally, we added definitions to form a complete problem statement: the simple definition of HOST_NEXT to traverse the graph and the formal definitions of the conditions c1 and so on. There were more conditions than forseen in the informal proof ([5]), similar to c17; these conditions always decide whether to stop because of an abnormality after an ALU operation, or whether to end the sequence naturally, depending on the stop value returned by the ALU. However, the inputs to the ALU are different in different paths through the graph, depending on which nodes have come before. In the literal *call* path the ALU expression was:

```
ALU
(FF(INSTFETCH(ram_sig n,p_sig n)),
 M(INSTFETCH(ram_sig n,p_sig n)),
 D(INSTFETCH(ram_sig n,p_sig n)),
 REG
 (R(INSTFETCH(ram_sig n,p_sig n)),a_sig n,x_sig n,
  PAD20TO32(TRIM32TO20(INCP32(p_sig n))),
  TRIM32TO20(INCP32(p_sig n))),
 PAD20TO32(A(INSTFETCH(ram_sig n,p_sig n))),b_sig n)
```

but this instance was a result of having been through the DUMMY, FETCH and in particular the PRECALL node, and no others. In fact, there are seven other similar conditions for the twenty-four total paths.

Furthermore, the conditions suggested did not make a distinction between the contents of the transient before and after a FETCH event. For example, in [5], c3 was defined to be something like

```
c3(p,dsf,csf,fsf,msf,b) =
(NOT
 ((INVALID(INCP32 p)) \/
  ((ILLEGALCALL(dsf,csf,fsf)) \/
   ((SPAREFUNC(dsf,csf,fsf)) \/
    ((ILLEGALPDEST(dsf,csf,fsf)) \/
     (ILLEGALWRITE(dsf,csf,msf)))))) /\
```

```
((NOT(VAL1 csf = 1)) /\
 ((NOT((VAL3 dsf = 7) \/ (VAL3 dsf = 6))) /\
  ((NOT(NOOP(dsf,csf,b))) /\ ((VAL2 msf = 0) /\
    ((VAL1 csf = 0) /\ (VAL4 fsf = 1))))))
```

This is not what is required for traversing the graph (see Section 4.2) and in fact does not make sense; it is not a function of the state and so has no meaning at the higher level.

8 Performing the Proof

We have tried to present the proof as simply as possible in this paper, but the actual proof effort deserves some discussion.

The process of typing the definitions in [4] and [5] into HOL took about a week; it was straightforward since the definitions were written in a notation called LCF_LSM, a predecessor of HOL.

The generation of the proof took about six person-months of work. (It was done on a Sun 3 workstation with eight megabytes of memory.) This may seem a long time for a proof which is not conceptually difficult, but it should be borne in mind what was actually generated: a sequence of over a million inferences which ensures that the correctness statement is really true (see Section 1.1). The hand proof done by Cullyer ([5]) took only about three weeks but did not detect the errors described in Section 7.

The generation of the theorems about `HOST_NEXT` (Sections 5.1, 5.2.1, 5.2.2 and 5.2.3) was relatively quick and easy, as they were done by forward proof, *i.e.* letting HOL continuously unfold and simplify a definition to compute an unforseen theorem by a known method.

Most of the time was spent in proving the theorems about `NEXT`, for which the desired result was known in advance but the way to prove it was not. The latter is a more typical sort of HOL or LCF proof effort, and it is a very interactive process. One typically tries a tactic, examines the subgoals, backs up, tries another, and so on, until a successful structure of tactics is discovered. The proof effort has to be planned carefully and the proof well understood in advance. Lemmas have to be anticipated and supplied in a useful form. The difficulty is partly because HOL (and LCF) are really just *frameworks* for performing proofs; the only proof tool which is in any sense automatic is rewriting (either forward or goal oriented) and the mechanism which justifies it. Any proof tool or strategy one thinks of can be expressed in the programming language ML and then applied, but designing these is a research problem. One hopes that large proof efforts such as the present one suggest new possible proof tools.

The proofs are difficult also because of the sheer size of the theorems (for example, the versions of Theorem 13 before it is fully simplified are several pages long when printed in the style shown). A certain amount of the proof

effort was directed toward structuring theorems into smaller ones, and defining abbreviations to control the expansion of theorems (such as ALU_ABBR6 and ALU_ABBR2).

The proofs are also large in terms of the computation time; rewriting in particular can be costly of time, and can be avoided in various ways by investing more of the user's time in doing explicit local replacements or other *ad hoc* methods. Large theorems are even time-consuming to fetch from files or to pretty-print. This problem can probably be attacked both by increased computing resources, and research into efficient proof strategies. Computation time can also be reduced by identifying large inference steps which are computed more than once and proving them as lemmas. For example, in computing the end results of the host machine sequences, certain conditional subexpressions recur for choosing the next node. The reductions of these are proved once as lemmas rather than being computed several times in the course of the proof. This is also often more convenient than doing a proof within a larger proof.

To give a very rough idea of the magnitude of by the proof, the following tables give the number of primitive inference steps and the CPU time in seconds for the main theorems of the proof. (Garbage collection time has been ignored.) The remarks in Section 5.1 about HOL proofs all apply here; in particular, the apparent sizes and times of proofs can be greatly (and rather misleadingly) inflated by reliance on powerful, general rewriting strategies where more specific and local strategies could be found. Theorem 1, for example, has a large proof which is generated by a strategy which simply rewrites according to previously proved theorems; Theorem 14 has a relatively small proof which is generated by laborious instantiations and carefully specified substitutions. We mention this so that these figures are not taken too seriously.

The last six theorems in the first table are referred to but not shown in this paper. "Unfolded NEXT" is the fully unfolded definition of NEXT with no lets; likewise for ALU. "Covering" refers to the theorem stating that at each node, the conditions for choosing the next node cover all possibilities. "Numbers" refers collectively to all the properties of numbers involved in reasoning about the numbers of steps in paths. "Tuples" refers collectively to properties of n-tuples required to transform theorems in phrased in terms of the fourteen components into theorems phrased in terms of a state, transient and node.

Theorems used for All 24 Paths

Theorem	Steps	CPU Seconds
Section 3.1 Theorem 4	215	3
Section 5.1 Theorem 7	7,664	230
Section 5.2.1 Theorem 8	8,474	315
Section 5.2.1 Theorem 9	7,649	280
Section 5.2.1 Theorem 10	19,587	456
Section 5.2.1 Theorem 11	24,428	649
Section 5.2.2 Theorem 12	2,195	64
Similar for DUMMY	2,259	66
Similar for STOP	2,179	57
Similar for FETCH	10,291	232
Similar for PERFORM	8,019	213
Section 5.5 Unfolded NEXT	7,486	292
Unfolded ALU	4,492	82
Total	104,938	2,939

Theorems Just for the Literal Call Path

Theorem	Steps	CPU Seconds
Section 3.1 Theorem 5	233	43
Section 4.2 Theorem 6	1,836	35
Section 5.3 Theorem 15	2,545	81
Section 5.2.3 Theorem 13	14,037	312
Section 5.3Theorem 14	8,626	237
Section 1.3 Theorem 2	8,586	213
Section 1.3 Theorem 3	8,620	230
Total	44,483	1,151

Theorems tying the 24 Cases Together

Theorem	Steps	CPU Seconds
Section 1.3 Covering	19,772	364
Section 1.3 Numbers	20,173	222
Section 5.2.3 Tuples	5,116	120
Section 1.3 Theorem 1	35,554	384
Total	80,615	1,090

9 Conclusions

As the reader can tell, the machine-checked proof described in this paper was a very large and somewhat tedious project. Proofs of this sort, while perfectly feasible, require experts in theorem-proving rather than electrical engineers, and such experts are at present scarce. It is the experts' time rather than computation time which makes verification expensive[1]. At present, formal verification is only appropriate in selected applications where the cost can be justified.

One clear conclusion of this work is that very much more basic research must be done before formal verification becomes practical and commonplace in real-life applications. Work to date shows that verification of real examples is still too time-consuming and tedious, and requires too much user guidance, for routine use. We believe that HOL is an excellent framework in which to research the problems: to design more automatic proof tactics, better ways to abstract and describe proofs, and more efficient and flexible rewriting strategies. Experimental proofs of real hardware like the Viper chip suggest many research areas which must be addressed before larger applications can be attempted.

A second conclusion is that for theorem-proving experts to undertake hardware proofs, documentation is vital; the experts will not be knowledgeable in engineering matters, and informal explanations of the function of the systems will save a lot of time spent working out connections which are perhaps obvious to the engineer.

Viper is to be manufactured and sold by two companies, for both civil and military applications. As we have shown, the HOL methodology can detect design errors *at a certain level of abstraction* from the actual hardware. We feel it necessary to warn against a false sense of security afforded by an HOL proof; there are many classes of errors not even *visible* in the models used. In particular, we would caution against a false sense of security in such hazardous applications as nuclear reactor control systems. The art of formal verification is in its early days. We believe that *years* of basic research are needed before the techniques can be reliably applied to life-critical systems; even then they can only be applied with an understanding of their limitations.

Further, a proof that a design meets a specification is only as good as the specification, which, for a complex system, can be very difficult to produce. A perfectly correct proof may relate to a specification which misses some essential behaviour of the design, and so may not give much security. For an example, see [12] regarding the problems of specifying a simple computer design.

Finally, for verification to be effective, we must first move to a situation in which the same sources are used by designers, verifiers and manufacturers. For example, as mentioned in Section 1, the errors we found in Viper's specification

[1] The first level of the Viper proof, for example, takes several hours of CPU time to run, and that is using an inefficient prototype system. It took six months, however, to organize the proof and carry it out. We expect that subsequent lower-level stages will be much more complex.

and host machine are apparently not present in the actual chip; hence the manufacturers cannot have used the specification which we have started to verify. Uniting these various communities is the aim of the research at RSRE, but there is a long way to go.

Acknowledgements

Theorems 1, 2 and 3, which tie together the body of the proof (the twenty-four cases), and the basic lemmas about numbers and tuples were generated in HOL by Mike Gordon. The author is grateful to Elsa Gunter for assistance with typesetting and suggestions about the paper. The work described here was supported by a grant from RSRE, Research Agreement 2029/205(RSRE), and has been placed under Crown Copyright © HMSO London 1987. This paper has also appeared under the same title as a University of Cambridge Computer Laboratory Technical Report, No. 104, 1987.

References

[1] A. Church, "A Formulation of the Simple Theory of Types", Journal of Symbolic Logic 5, 1940

[2] A. Cohn and M. Gordon, "A Mechanized Proof of Correctness of a Simple Counter", University of Cambridge, Computer Laboratory, Tech. Report No. 94, 1986

[3] W. J. Cullyer and C. H. Pygott, "Hardware Proofs using LCF_LSM and ELLA", RSRE Memo. 3832, Sept. 1985

[4] W. J. Cullyer, "Viper Microprocessor: Formal Specification", RSRE Report 85013, Oct. 1985

[5] W. J. Cullyer, "Viper — Correspondence between the Specification and the 'Major State Machine' ", RSRE report No. 86004, Jan. 1986

[6] W. J. Cullyer, "Implementing Safety-Critical Systems: The Viper Microprocessor", In: VLSI Specification, Verification and Synthesis, Edited by G. Birtwistle and P. A. Subrahmanyam, (this volume)

[7] M. Gordon, R. Milner and C. P. Wadsworth, "Edinburgh LCF, Lecture Notes in Computer Science", Springer-Verlag, 1979

[8] M. Gordon, "Proving a Computer Correct", University of Cambridge, Computer Laboratory, Tech. Report No. 42, 1983

[9] M. Gordon, "HOL: A Machine Oriented Formulation of Higher-Order Logic", University of Cambridge, Computer Laboratory, Tech. Report No. 68, 1985

[10] M. Gordon, "HOL: A Proof Generating System for Higher-Order Logic", In: VLSI Specification, Verification and Synthesis, Edited by G. Birtwistle and P. A. Subrahmanyam (this volume), Also: University of Cambridge, Computer Laboratory, Tech. Report No. 103, 1987

[11] W. A. Hunt Jr., "FM8501: A Verified Microprocessor", University of Texas, Austin, Tech. Report 47, 1985

[12] J. J. Joyce, "Verification and Implementation of a Microprocessor", In: VLSI Specification, Verification and Synthesis, Edited by G. Birtwistle and P. A. Subrahmanyam, (this volume)

[13] J. Kershaw, "Viper: A Microprocessor for Safety-Critical Applications", RSRE Memo. No. 3754, Dec. 1985

[14] L. Paulson, "A Higher-Order Implementation of Rewriting", Science of Computer Programming 3, 119-149, 1983

[15] L. Paulson, "Interactive Theorem Proving with Cambridge LCF", Cambridge University Press, To Appear 1987