Generic Specification of Digital Hardware

by

Jeffrey J. Joyce

Technical Report 90-27
September 6, 1990

Department of Computer Science
University of British Columbia
Vancouver, B.C. V6T 1W5

Generic Specification of Digital Hardware

Jeffrey J. Joyce
Department of Computer Science
University of British Columbia
Vancouver, B.C., CANADA V6T 1W5

phone: (604) 228-4327
e-mail: joyce@cs.ubc.ca

Abstract

This paper argues that generic description is a powerful concept in the context of formal verification, in particular, the formal verification of digital hardware. The paper also describes a technique for creating generic specifications in any language with (at least) the expressive power of higher-order logic. This technique is based on the use of higher-order predicates parameterized by function variables and type variables. We believe that this technique is a very direct (if not the most direct) way to specify hardware generically. Two examples of generic specification are given in the paper: a resettable counter and the programming level model of a very simple microprocessor.

Introduction

Generic description is already established as a powerful concept in many high-level programming languages. For instance, the 'generic mechanism' of Ada allows a subprogram or package to be parameterized by types and subprograms as well as values and objects. This feature supports modularity and abstraction and provides a convenient mechanism for the reliable re-use of software.

This paper argues that generic description is also a powerful concept in the context of formal verification. In addition to the well-known advantages of modularity, abstraction and re-usability, generic description can be used in a formal proof to filter out non-essential detail. The elimination of non-essential detail from a formal specification offers several potential benefits:

- it sharpens the distinction between what has and what has not been formally considered in a correctness proof.
- it supports a truely hierarchical approach to the formal verification of digital circuits where each level in a hierarchical specification is isolated from details only relevant to other levels.
- it reduces the amount of special-purpose infrastructure needed to reason about particular application areas, e.g., hardware-oriented data types.

It may be thought that the thesis of this paper - that formal specifications and correctness statements should be as general as possible - is inconvertible. But when one considers state-of-the-art examples such as the formal verifications of the Viper microprocessor [4, 5, 6] and the CLI 'verified stack' [1, 2, 15], it is easy to see that, in actual practice, formal specifications and correctness statements are not as general as possible. Instead, many examples of hardware verification are encumbered with non-essential details. These non-essential

details are likely to obscure correctness results, interfere with the advantages of a hierarchical approach, and depend on the development of special-purpose infrastructure such as hardware-oriented data types.

The formal verification of the 'major state machine' of the Viper microprocessor [4] is an example of when specifications and correctness statements are not as general as possible. The Viper specification uses a number of special-purpose data types (e.g., `:word4`, `:word32`) and constants (e.g., `VAL4`, `WORD32`) for reasoning about hardware. However, the correctness results derived from these specifications are only concerned with the flow of control in the Viper machine state machine.[1] Cohn [5] writes:

> There was no computation of values at the major state level - that is, additions, comparisons, shifts, and so on - so the essential correctness of Viper was not really addressed; the proof did not require any analysis of the function representing the arithmetic-logic unit, at either level.

The use of special-purpose hardware-oriented data types and constants in the Viper specification was consistent with earlier work on hardware verification at Cambridge [3, 9]. However, building non-essential details into a computational model such as the formal specification of the Viper major state machine risks the false impression that these details have been formally considered in the correctness proof. Furthermore, correctness results for the Viper major state machine are not directly re-usable when low-level details are varied, e.g., different machine word sizes. Finally, the effort of building up a computational model 'from scratch' for non-essential details, e.g., defining arithmetic operations on Viper machine words, is wasted in the case of correctness results that do not depend on these details.

Another example of when specifications and correctness results are not as general as possible is the vertically verified computing system developed by Computation Logic Inc. (i.e., the CLI 'verified stack') [1, 2, 15]. For instance, Moore [15] observes that specifications and correctness results for the Piton assembler are "unnecessarily restricted" to a word size of 32 even though other word sizes are possible. It is also likely that the formal specification of the Piton assembler involves defined symbols whose actual definition is not needed to establish correctness results for the Piton assembler.

There are at least several reasons why non-essential details are built into formal specifications. One reason may simply be that this is common practice or at least common practice in software production. Another reason, suggested by Eveking [7], is the "the-more-detailed-the-better" attitude carried over from experience with multi-level simulation of digital hardware. Yet another reason is the absence of explicit mechanisms (analogous to the generic mechanism of Ada) in most hardware specification languages to support and encourage the creation of specifications which are as general as possible.

To remedy the ill effects of building non-essential detail into formal specifications of digital hardware, we advocate the use of generic specifications. We also propose a technique for expressing genericity in any language with (at least) the expressive power of higher-order logic in a manner that avoids introduction of new constructs to explicitly support generic specifications. This technique is based on the use of higher-order predicates parameterized by function variables and type variables. We believe that this technique is a very direct (if not the most direct) way to specify hardware generically. Although the idea of parameterizing hardware specifications by functions variables has appeared previously [12, 16], singling

[1] While the first level of proof was only concerned with flow of control in the Viper major state machine, a second level of proof for the Viper microprocessor did take into account computational details [5].

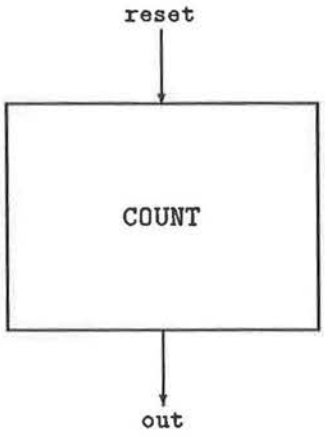


Figure 1: External View of the Resettable Counter

out a special technique (in particular, the use of 'representation variables') for expressing genericity is, to the best of our knowledge, a novel contribution of the research described in this paper.

We use the example of a resettable counter to describe a method for eliminating non-essential detail from a formal specification to eventually yield a generic specification. This example was used by Mike Gordon in an early discussion of hardware verification [9]. Although the resettable counter example is very simple, it is enough to illustrate the idea of using generic descriptions to formally specify digital hardware. Later in the paper, we give a more substantial example, namely, the generic specification of the programming level model of a simple microprocessor.

The Resettable Counter Example

The resettable counter is a sequential device which counts upwards until it is externally reset. Resetting the counter causes it to begin counting from zero. As shown in Figure 1, this device has a single input (the reset signal) and a single output (the current state of the counter).

This device can be implemented using a multiplexor, a register and an increment circuit. Figure 2 shows interconnection of these three components to implement the resettable counter.

Formal verification is a matter of showing that the interconnection of these three components yields a correct implementation of the resettable counter. This is not a guarantee of absolute correctness for a physical realization of the resettable counter. Formal verification only shows that models for the three components can be composed to yield a set of simultaneous constraints which satisfy the behavioural specification of the counter.

This paper focuses on writing formal specifications as a first step towards proving the correctness of a design. For the resettable counter, formal specifications need to be written for each of the three components together with a structural specification of the counter implementation, and finally, a behavioural specification of the counter.

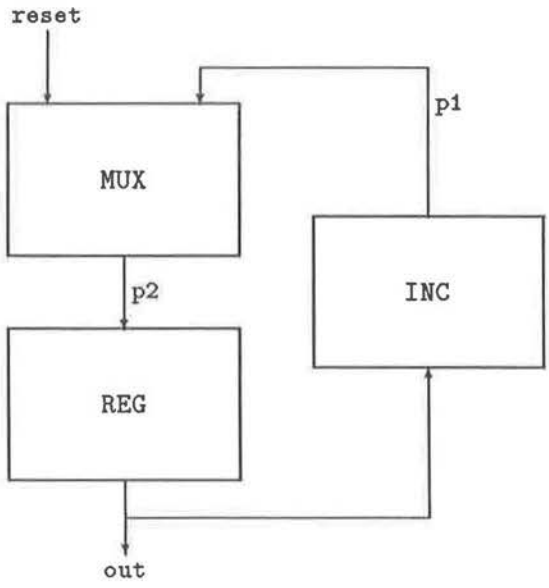


Figure 2: Internal View of the Resettable Counter

Basic Data Types and Primitive Operations

Translating an informal description into a formal specification is often the most interesting and creative aspect of using formal methods to verify hardware. A preliminary step is to decide upon a set of basic data types and a set of primitive operations involving these data types. For instance, in the formal specification of a microprocessor, this set of basic data types would probably include representations for bits, bytes and words. The corresponding set of primitive operations would probably include operations such as addition, subtraction, shift-left, and so on.

For the counter example, we need to decide upon basic data types for the reset signal and the internal state (which is also the output signal). We also need to decide upon a set of primitive operations to describe the functions performed by the multiplexor and the increment circuit.

It is reasonably easy to decide on a representation for values of the reset signal, namely, Boolean values, `T` and `F`. However, it is more difficult to decide upon a representation for the internal state of the counter.

A first attempt at a representation for the internal state of the counter is given in the next section: this is a very simple representation where the internal state of the counter is represented by a natural number. After identifying a problem with the accuracy of this simple (and idealized) representation - and how adding more detail is not an ideal solution - we describe several revisions to the counter specification which eventually lead to a generic specification of the resettable counter.

A First Attempt

In this first version of the counter specification, natural numbers are used to represent the internal state of the counter. The operation performed by the increment circuit is described in terms of natural number arithmetic, in particular, the 'plus one' function. Using the HOL formulation of higher-order logic [11], this version of the counter specification is given by the following set of predicate definitions.

```
⊢def MUX (reset,i,out) = ∀t. out t = (reset t ⇒ 0 | (i t))

⊢def REG (i,out) = ∀t. out (t+1) = i t

⊢def INC (i,out) = ∀t. out t = ((i t) + 1)

⊢def COUNT_IMP (reset,out) =
       ∃p1 p2.
         MUX (reset,p1,p2) ∧
         REG (p2,out) ∧
         INC (out,p1)

⊢def COUNT (reset,out) =
       ∀t. out (t+1) = (reset t ⇒ 0 | ((out t) + 1))
```

We digress briefly to describe how the above predicate definitions are used to formally describe hardware behaviour and structure.

Hardware is described either behaviourally or structurally by constraints on a set of signals. These signals represent the externally visible behaviour of the device. Because the resettable counter is a sequential device, signals are described by a sequence of values. It is convenient to represent this sequence of values by a function which maps discrete time (i.e. positions in the sequence) to signal values. For instance, the reset signal,

```
reset:time→bool
```

is modelled by a function which maps discrete time to Boolean values. Discrete time is a set of values denoted by the type `:time` which is isomorphic to the natural numbers.

Behavioural models for the three components of the implementation are formally specified by the definitions of `MUX`, `REG` and `INC`. For instance, the predicate `INC` specifies that, at all times, the current output of the increment circuit is the result of adding one to its current input value. The definition of `MUX` uses a conditional expression of the form `b ⇒ t1 | t2` ("if `b` then `t1` else `t2`") to select between the two data inputs. The predicate `REG` uses the current input and current output value to determine the next output value.

The predicate `COUNT_IMP` is a structural specification of the counter implementation. Interconnections are specified by common names, e.g., `p2` is an internal connection between the multiplexor and the register. Logical conjunction is used to compose terms corresponding to each component of the implementation. Existential quantification is used to 'hide' the internal signals, namely, `p1` and `p2`.

Finally, the predicate `COUNT` is a behavioural specification for the resettable counter. It is significant, in this particular example, that the 'plus one' function used in the definition of `INC` to denote the operation performed by increment circuit re-appears in the definition

of COUNT when specifying the behaviour of the counter. (As we will see, this makes the resettable counter example a good candidate for generic specification.)

This style of using higher-order logic to specify the structure and behaviour of hardware is described more fully in a report by Gordon [10].

The essence of the verification problem, in this case, is to show that behavioural models of the three components of the counter implementation can be composed to satisfy the behavioural specification of the counter. From the above set of specifications, the following correctness result can be formally derived as a theorem of higher-order logic. This theorem states that the constraints imposed by COUNT_IMP (defined in terms of MUX, REG and INC) satisfy the constraints expressed by COUNT.

$\vdash_{thm}$ `COUNT_IMP (reset,out)` $\Longrightarrow$ `COUNT (reset,out)`

Representing the internal state of the counter as a natural number is a very simple approach. But this approach has the disadvantage of being an idealized model of physically-realizable hardware. This idealized model of a counter will count continuously upwards until it is reset. However, in reality, the internal state of the counter will have a finite number of bits and therefore, it will eventually overflow unless it is reset at some point.

The next section of this paper considers the approach of adding more detail into the formal specification of the resettable counter to correct the problem of modelling overflow. However, we will eventually reject this approach in favour of the very opposite approach of eliminating non-essential detail.

The More Detailed, The Better ?

The following is a revised set of specifications for the resettable counter which uses modular arithmetic to model the finite limitations of physically-realizable hardware.

$\vdash_{def}$ `MUX (n) (reset,i,out) = ∀t. out t = (reset t ⇒ 0 | (i t))`

$\vdash_{def}$ `REG (n) (i,out) = ∀t. out (t+1) = i t`

$\vdash_{def}$ `INC (n) (i,out) = ∀t. out t = (((i t) + 1) MOD` 2^{n}`)`

```
⊢def COUNT_IMP (n) (reset,out) =
       ∃p1 p2.
         MUX (n) (reset,p1,p2) ∧
         REG (n) (p2,out) ∧
         INC (n) (out,p1)
```

$\vdash_{def}$ `COUNT (n) (reset,out) = ∀t. out (t+1) = (reset t ⇒ 0 | (((out t) + 1) MOD` 2^{n}`))`

This revised set of specifications is a simple example of parameterized hardware description. Each predicate definition is parameterized by an additional variable, n, giving the number of bits used to represent the internal state of the counter. For reasons of (personal) style, this additional parameter is separated from the list of parameters representing the input and output signals of the counter. However, this separation has no logical significance. Technically speaking, this is an instance of a 'curried' function, that is, a function which can evaluate its arguments "one at a time".

Modular arithmetic models the fact that the counter will overflow when counting past the highest representable value, namely, 2^n - 1. Undoubtedly, this revised set of specification is a more accurate model of physical hardware. One might think that a more accurate model entails a more comprehensive proof of correctness. But this is not necessarily true. In this case, proving a correctness result of the form,

$$\vdash_{thm} \texttt{COUNT\_IMP (n) (reset,out)} \Longrightarrow \texttt{COUNT (n) (reset,out)}$$

does not involve any properties of either + or MOD. In fact, the above correctness result can be established even if + and MOD were replaced by undefined symbols in the formal specifications. Hence, + and MOD are no more than place-holders in this particular proof of correctness.

The fact than + and MOD are just place-holders underlines the difference between formal verification and conventional simulation. In the case of conventional simulation, + and MOD would have to be defined symbols to use this specification as input to a conventional simulator. But in the case of formal verification, a meaningful correctness result can be obtained without necessarily having to develop a full-scale model of the computation.

Eliminating Non-essential Detail

Because building more detail into the counter specification, in particular, the use of modular arithmetic, may give the false impression that a particular correctness result is more comprehensive than it really is, we argue in a favor of the very opposite approach. Instead of building more detail into the specification to remedy the inaccurate modelling of overflow in the original specification of the counter, we filter out details about the computation performed by the counter. Very importantly, this can be done without changing the essence of the verification problem.

The next few sections of this paper illustrate a method for eliminating non-essential detail by describing a series of incremental revisions to the original specification of the resettable counter. The first step involves parameterizing the formal specification by function variables. The second step involves parameterizing the formal specification by type variables. The third and final step is to 'package' function variables into a single 'representation variable'.

Parameterizing with Function Variables

The following set of specifications is obtained by replacing the 'plus one' operation in the original specification of the resettable counter by a function variable inc. Each of the specifications is parameterized by this function variable (and hence, they are higher-order predicates). For the sake of uniformity, the predicates MUX and REG are parameterized by inc even though they do not make use of this function.

```
⊢def MUX (inc) (reset,i,out) = ∀t. out t = (reset t ⇒ 0 | (i t))

⊢def REG (inc) (i,out) = ∀t. out (t+1) = i t

⊢def INC (inc) (i,out) = ∀t. out t = inc (i t)

⊢def COUNT_IMP (inc) (reset,out) =
       ∃p1 p2.
         MUX (inc) (reset,p1,p2) ∧
         REG (inc) (p2,out) ∧
         INC (inc) (out,p1)

⊢def COUNT (inc) (reset,out) =
       ∀t. out (t+1) = (reset t ⇒ 0 | (inc (out t)))
```

The parameterization of the above specifications by the function variable `inc` eliminates detail about the operation performed by the increment circuit which is not relevant to the verification problem. For this revised set of specifications, this verification problem is expressed by the following correctness result:

$$\vdash_{thm} \texttt{COUNT\_IMP (inc) (reset,out)} \Longrightarrow \texttt{COUNT (inc) (reset,out)}$$

Parameterizing with Type Variables

The next step is to revise the above specifications by using a type variable, namely, `:*word`, to represent the internal state of the counter. (In the HOL formulation of higher-order logic, a type variable always begins with an asterisk.)

In the previous set of specifications, the type associated with the function variable `inc` was a function from natural numbers to natural numbers. But now, the type of `inc` is a function from `:*word` to `:*word`.

Since we are now using the type variable `:*word` to replace natural numbers, we must also introduce a variable called `zero` to replace the natrual number constant `0`. This variable stands for a value of type `:*word` (it is a function variable for a 0-place function). Each of the predicates in the revised specification of the counter, as shown below, will be parameterized by both `inc` and `zero` in addition to the input and output variables of the counter.

```
⊢def MUX (inc,zero) (reset,i,out) = ∀t. out t = (reset t ⇒ zero | (i t))

⊢def REG (inc,zero) (i,out) = ∀t. out (t+1) = i t

⊢def INC (inc,zero) (i,out) = ∀t. out t = inc (i t)

⊢def COUNT_IMP (inc,zero) (reset,out) =
       ∃p1 p2.
         MUX (inc,zero) (reset,p1,p2) ∧
         REG (inc,zero) (p2,out) ∧
         INC (inc,zero) (out,p1)

⊢def COUNT (inc,zero) (reset,out) =
       ∀t. out (t+1) = (reset t ⇒ zero | (inc (out t)))
```

These changes do not significantly alter the proof of correctness; likewise, the revised correctness result is not significantly different than before.

$\vdash_{thm}$ `COUNT_IMP (inc,zero) (reset,out)` $\Longrightarrow$ `COUNT (inc,zero) (reset,out)`

Representation Variables

Our approach to generic specification is based on the parameterization of formal specifications by function variables and type variables. Scaling this approach upwards for more complex specifications (with more function variables) could result in the unwieldy parameterization of predicates. However, this can be avoided by 'packaging' functions variables into a single representation variable.

```
⊢def inc rep = FST rep

⊢def zero rep = SND rep

⊢def MUX (rep) (reset,i,out) = ∀t. out t = (reset t ⇒ (zero rep) | (i t))

⊢def REG (rep) (i,out) = ∀t. out (t+1) = i t

⊢def INC (rep) (i,out) = ∀t. out t = (inc rep) (i t)

⊢def COUNT_IMP (rep) (reset,out) =
       ∃p1 p2.
         MUX (rep) (reset,p1,p2) ∧
         REG (rep) (p2,out) ∧
         INC (rep) (out,p1)

⊢def COUNT (rep) (reset,out) =
       ∀t. out (t+1) = (reset t ⇒ (zero rep) | ((inc rep) (out t)))
```

The above specifications are parameterized by a single representation variable called `rep`. This variable,

`rep: ((*word→*word) × *word)`

is a pair of values. The first element of this pair represents the function performed by the increment circuit. The second element of this pair is the representation of zero.

In this version of the counter specification, `inc` and `zero` are defined as 'selector functions' which extract the first and second elements of a representation. These two selectors functions are meaningful synonyms for the pre-defined selectors functions `FST` and `SND`. That is:

`(inc rep)` - "the increment operation"
`(zero rep)` - "the representation of zero"

Once again, the essence of the verification problem is unchanged from before. The following correctness result shows that behavioural models of the three components used to implement the resettable counter can be composed to satisfy the behavioural specification of the counter.

$\vdash_{thm}$ `COUNT_IMP (rep) (reset,out)` $\Longrightarrow$ `COUNT (rep) (reset,out)`

Re-usable Correctness Results

Eliminating detail from a specification does not result in a less comprehensive correctness proof. In fact, the opposite situation is true: a generic specification yields a correctness result which covers a wider range of possible implementation. The above correctness result (for the generic specification of the resettable counter) can be instantiated for various implementations, e.g., a 2-bit counter, an 8-bit counter, a 16-bit counter, etc. The correctness result is instantiated by assigning a particular representation to the representation variable `rep`.

For example, the original specification of the counter is described by the representation,[2]

```
REP_num = (λx. x + 1, 0)
```

where the 'plus one' function is used to represent the operation performed by the increment circuit and the natural number constant `0` is the representation of zero. Instantiating the generic correctness result for this particular value of the representation variable,

$$\vdash_{thm} \texttt{COUNT\_IMP (REP\_num) (reset,out)} \Longrightarrow \texttt{COUNT (REP\_num) (reset,out)}$$

yields a correctness result which is logically equivalent to the first correctness result given for the counter specification.

Another example is based on the built-in HOL data types for bit strings and machine words used, for instance, in the formal verification of the Viper microprocessor. An 8-bit version of the resettable counter is described by the representation,

```
REP8 = (λx. WORD8 ((VAL8 x) + 1), WORD8 0)
```

where `WORD8` is a pre-defined function for converting a natural number into a 8-bit word and `VAL8` is a pre-defined function for converting a 8-bit word into a natural number. The corresponding correctness result is an instance of the generic correctness result:

$$\vdash_{thm} \texttt{COUNT\_IMP (REP8) (reset,out)} \Longrightarrow \texttt{COUNT (REP8) (reset,out)}$$

Hierarchical Verification

Correctness proofs for digital hardware are typically organized into several levels. For instance, another level of verification could be used to verify that each of the three components used in the counter implementation is correctly implemented by logic gates. An even lower level of proof would establish that logic gates are correctly implemented by networks of transistors.

To sharpen the distinction between what has and what has not been considered at each level, we believe that a truely hierarchical approach to formal verification depends on the use of generic specifications to eliminate non-essential detail from each proof level. In other words, each level is a highly localized concern which should be isolated as much as possible from details relevant only to other levels. This is achieved by using generic specifications at higher levels parameterized by data types and functions which only need to be "fleshed out" at lower levels in the proof hierarchy. This contrasts with a 'closed-world approach' to formal verification where every data type and every operator is completely defined at each level. Eveking [7] elaborates on the distinction between an open (or interpreted) approach and a closed-world approach.

[2]The term `λx. x + 1` is a lambda-expression which denotes the 'plus one' function.

Generic Specification of a Microprocessor

We have used the techniques described in this paper to write generic specifications for a very simple microprocessor called TAMARACK-3 [14]. A design for the register-transfer level implementation of this microprocessor has been proven correct with respect to a programming level model of its operation. These correctness results can be re-used for various realizations of the microprocessor, e.g., an 8-bit version, a 16-bit version, etc. Moreover, the generic specification of this microprocessor does not depend on any special-purpose infrastructure for reasoning about hardware, e.g., data types for bit strings or machine words. This should make it much easier to re-produce this correctness result in a variety of verification systems (as long as these systems support higher-order predicates).

To give the reader an impression of how the generic specification techniques described in this paper can be applied to a more substantial example, this section of the paper presents the generic specification of the TAMARACK-3 programming level model.

The TAMARACK-3 microprocessor was designed as a verification example and is not seriously intended for practical applications.[3] It has just eight different programming level instructions and only one addressing mode. The only kind of hardware exception is a single level, non-vectored hardware interrupt. The microprocessor can be interfaced to external memory (or some other perpherial device) to operate in one of three possible modes: fully synchronous, fully asynchronous, and extended cycle mode. All I/O is memory-mapped. Figure 3 shows a functional diagram for the externally visible signals of TAMARACK-3.

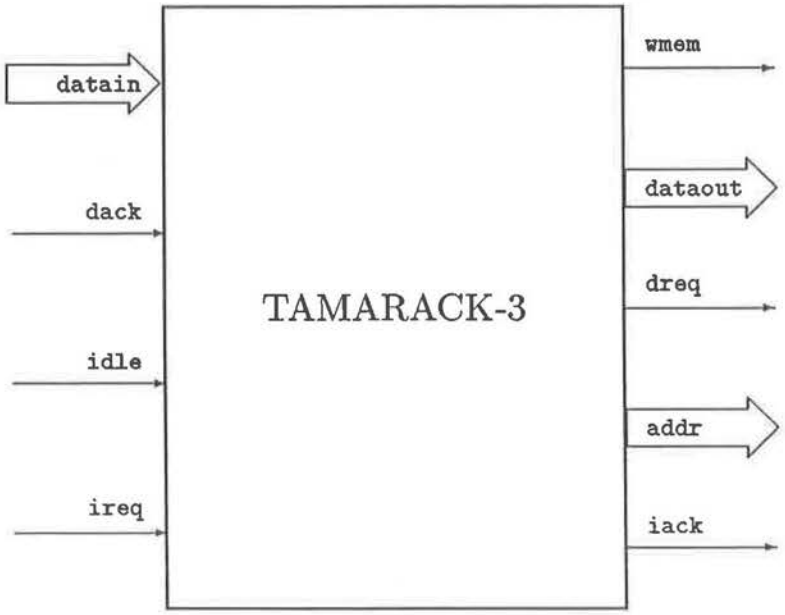
The programming level model, or external architecture, of TAMARACK-3 is a description of its operation as seen by a programmer. This model hides all aspects of the internal architecture which the programmer does not need to know about when writing programs for this microprocessor. The programming level model can be viewed as an interpreter for manipulating a set of variables which corresponds to the externally visible state of the microprocessor. It consists of five main parts:

- Basic data types and primitive operations.
- Variables manipulated by the interpreter.
- Format of instructions.
- Instruction semantics.
- Instruction Cycle.

Basic Data Types and Primitive Operations

A total of seven different data types are used to specify the programming level model of TAMARACK-3. The data type `:bool` is used to represent voltage values or logical conditions. The data type `:num` is used when some lower level form of data is interpreted as the representation of a natural number. The remaining five data types are used to represent machine words (three different sizes), a particular field of bits within a machine word, and memory states.

[3] Although TAMARACK-3 is a very simple microprocessor, some aspects of its operation (support for interrupts and asynchronous interaction with external memory using handshaking signals) are more complex behaviours than found in the formal specification of the Viper microprocessor.



`datain`	- data from memory	`wmem`	- read/write select
`dack`	- data acknowledge	`dataout`	- data to memory
`idle`	- extended cycle mode	`dreq`	- data request
`ireq`	- interrupt request	`addr`	- address to memory
		`iack`	- interrupt acknowledge

Figure 3: Functional View of the TAMARACK-3 Microprocessor

`:bool`	- Boolean values {T,F}
`:num`	- natural numbers {0,1,2,...}
`:*wordn`	- full-size machine words
`:*word3`	- instruction opcodes
`:*word4`	- 4-bit words
`:*address`	- memory addresses
`:*memory`	- memory states

A conventional description of a microprocessor would typically be very specific about details such as the number of bits in a machine word and the size of memory. However, we avoid specifying these details by regarding `:*wordn`, `:*word3`, `:*word4`, `:*address` and `:*memory` as *uninterpreted types*. The actual representation of these basic data types may be thought of as implementation dependent details. The prefix * indicates our intention to use type variables for these data types.

Functional elements such as the ALU (*Arithmetic Logic Unit*) at the lowest level of architectural description perform various operations on data. These operations are regarded as *uninterpreted primitives*. The TAMARACK-3 programming level model is formally specified in terms of thirteen different primitives operations. Employing the generic specification technique outlined earlier in this paper, these thirteen different operations will be packaged into a single representation variable, `rep`. Each primitive will be selected (or extracted) from the representation variable by a unique selector function. These thirteen selector functions are listed below. Although the operations selected by these selectors are formally regarded as uninterpreted primitives, the following list also gives a suggested interpretation for each primitive operation.

`(iszero rep)`	- "test if zero"
`(inc rep)`	- "increment"
`(add rep)`	- "addition"
`(sub rep)`	- "subtraction"
`(wordn rep)`	- "full-size word representation of a number"
`(valn rep)`	- "value of a full-size word"
`(opcode rep)`	- "extract opcode field"
`(val3 rep)`	- "value of an opcode"
`(address rep)`	- "extract address field"
`(fetch rep)`	- "read memory"
`(store rep)`	- "write memory"
`(word4 rep)`	- "4-bit word representation of a number"
`(val4 rep)`	- "value of a 4-bit word"

Like every term in higher-order logic, the variable `rep` has a type. The type of `rep` is denoted by the following type abbreviation.[4]

[4]The built-in HOL system utility for creating type abbreviations does not allow this particular abbreviation since it contains type variables. However, there is an alternative way to introduce names to stand for fully expanded type expressions (using ML variables and ML antiquotation) - but these details are beyond the scope of this paper.

```
rep_ty =
  :(*wordn→bool)×                                  % iszero %
   (*wordn→*wordn)×                                % inc %
   (*wordn×*wordn→*wordn)×                         % add %
   (*wordn×*wordn→*wordn)×                         % sub %
   (num→*wordn) ×                                  % wordn %
   (*wordn→num) ×                                  % valn %
   (*wordn→*word3)×                                % opcode %
   (*word3→num)×                                   % val3 %
   (*wordn→*address)×                              % address %
   (*memory×*address→*wordn)×                      % fetch %
   (*memory×*address×*wordn→*memory)×              % store %
   (num→*word4)×                                   % word4 %
   (*word4→num)                                    % val4 %
```

The selector functions are defined in the formal specification by composing various sequences of the two primitive selectors FST and SND. For instance, the first three selectors, iszero, inc, and add, have the following definitions.

```
⊢def iszero (rep:rep_ty) = FST rep

⊢def inc (rep:rep_ty) = FST(SND rep)

⊢def add (rep:rep_ty) = FST(SND(SND rep))
```

The rest of the selectors are defined in a similar manner such that the following theorem is true:

```
⊢thm rep =
       ((iszero rep),
        (inc rep),
        (add rep),
        (sub rep),
        (wordn rep),
        (valn rep),
        (opcode rep),
        (val3 rep),
        (address rep),
        (fetch rep),
        (store rep),
        (word4 rep),
        (val4 rep))
```

Externally Visible State

The set of variables manipulated by the programming level model corresponds to the externally visible state of the microprocessor. In TAMARACK-3, these variables are:

Instruction	Opcode Value	Effect
JZR	0	jump if zero
JMP	1	jump
ADD	2	add accumulator
SUB	3	subtract accumulator
LDA	4	load accumulator
STA	5	store accumulator
RFI	6	return from interrupt
NOP	7	no operation

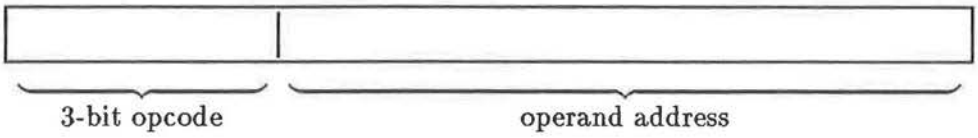
Table 1: TAMARACK-3 Instruction Set

`mem` - memory
`pc` - program counter
`acc` - accumulator
`rtn` - return address register
`iack` - interrupt acknowledge flag

The memory stores memory states, represented by the data type `:*memory`. Each of the registers stores full-size memory words, represented by the data type `:*wordn`. The interrupt acknowledge flag is stored internally by a flipflop whose value belongs to the data type `:bool`.

Instruction Word Format

Instructions are exactly one full-size machine word. Although specific details about word size and instruction word format are not given in this description, we can assume that the instruction word consists of a 3-bit opcode (since there are eight different instructions) with the remaining bits used as an operand address. The operand address is the absolute address of a memory word which may be used as the address of either data or an instruction.



Opcodes and operand addresses are represented by the uninterpreted types `:*word3` and `:*address`. They are extracted from an instruction word by the uninterpreted primitives `opcode` and `address`.

Instruction Set Semantics

The eight TAMARACK-3 programming level instructions are in Table 1. Their opcode values and a brief explanation of each instruction are also given in the table. The opcode is extracted from the current instruction word by `opcode` and its numerical value is then obtained by applying `val3` to the extracted opcode.

Formally, the semantics of each instruction is given individually by the definition of a function which returns the next (externally visible) state of the microprocessor, i.e., the

next values of the memory state **mem**, program counter pc, accumulator acc, return address register rtn and interrupt acknowledge flag iack. The following definitions specify how these values are computed from the current state of the microprocessor. In addition to a formal definition, the informal notation,

<destination> ← <expression>

is used to denote when a value computed from the current machine state is loaded into a register, flipflop or memory to form a component of the next machine state.

JZR - jump if zero

pc ← if (iszero rep) acc then inst else (inc pc)

```
⊢def JZR_SEM (rep:rep_ty)
       (mem:*memory,pc:*wordn,acc:*wordn,rtn:*wordn,iack:bool) =
       let inst = (fetch rep) (mem,(address rep) pc) in
       let nextpc = ((iszero rep) acc) ⇒ inst | ((inc rep) pc) in
         (mem,nextpc,acc,rtn,iack)
```

If the result of applying iszero to the current contents of the accumulator acc is T, then the current instruction word is loaded into the program counter pc. Otherwise, the instruction is completed by incrementing the program counter pc.[5]

JMP - jump

pc ← inst

```
⊢def JMP_SEM (rep:rep_ty)
       (mem:*memory,pc:*wordn,acc:*wordn,rtn:*wordn,iack:bool) =
       let inst = (fetch rep) (mem,(address rep) pc) in
         (mem,inst,acc,rtn,iack)
```

The current instruction word is unconditionally loaded into the program counter pc.

ADD - add accumulator

acc ← (add rep) (acc,operand)
pc ← (inc rep) pc

```
⊢def ADD_SEM (rep:rep_ty)
       (mem:*memory,pc:*wordn,acc:*wordn,rtn:*wordn,iack:bool) =
       let inst = (fetch rep) (mem,(address rep) pc) in
       let operand = (fetch rep) (mem,(address rep) inst) in
         (mem,(inc rep) pc,(add rep) (acc,operand),rtn,iack)
```

[5]Here we have relaxed our presentation style by referring to an uninterpreted primitive, namely, inc, in terms of its suggested interpretation.

The add operation is applied to the current contents of the accumulator acc and the memory word addressed by the operand address field of the current instruction. The result is loaded into the accumulator acc. The instruction is completed by incrementing the program counter pc.

SUB - subtract accumulator

```
acc ← (sub rep) (acc,operand)
pc ← (inc rep) pc
```

```
⊢def SUB_SEM (rep:rep_ty)
       (mem:*memory,pc:*wordn,acc:*wordn,rtn:*wordn,iack:bool) =
       let inst = (fetch rep) (mem,(address rep) pc) in
       let operand = (fetch rep) (mem,(address rep) inst) in
         (mem,(inc rep) pc,(sub rep) (acc,operand),rtn,iack)
```

The sub operation is applied to the current contents of the accumulator acc and the memory word addressed by the operand address field of the current instruction. The result is loaded into the accumulator acc. The instruction is completed by incrementing the program counter pc.

LDA - load accumulator

```
acc ← operand
pc ← (inc rep) pc
```

```
⊢def LDA_SEM (rep:rep_ty)
       (mem:*memory,pc:*wordn,acc:*wordn,rtn:*wordn,iack:bool) =
       let inst = (fetch rep) (mem,(address rep) pc) in
       let operand = (fetch rep) (mem,(address rep) inst) in
         (mem,(inc rep) pc,operand,rtn,iack)
```

The memory word addressed by the operand address field of the current instruction is loaded into the accumulator acc. The instruction is completed by incrementing the program counter pc.

STA - store accumulator

```
mem ← (store rep) (mem,(address rep) inst,acc)
pc ← (inc rep) pc
```

```
⊢def STA_SEM (rep:rep_ty)
       (mem:*memory,pc:*wordn,acc:*wordn,rtn:*wordn,iack:bool) =
       let inst = (fetch rep) (mem,(address rep) pc) in
       let newmem = (store rep) (mem,(address rep) inst,acc) in
         (newmem,(inc rep) pc,acc,rtn,iack)
```

The current contents of the accumulator acc are stored in external memory at the location specified by the operand address field of the current instruction. The instruction is completed by incrementing the program counter pc.

RFI - return from interrupt

```
pc ← rtn
iack ← F
```

```
⊢def RFI_SEM (rep:rep_ty)
       (mem:*memory,pc:*wordn,acc:*wordn,rtn:*wordn,iack:bool) =
       (mem,rtn,acc,rtn,F)
```

The current contents of the return address register `rtn` are loaded into the program counter `pc` and the interrupt acknowledge flag `iack` is reset to `F`. This instruction does not check whether the interrupt acknowledge flag `iack` is currently set.

NOP - no operation

```
pc ← (inc rep) pc
```

```
⊢def NOP_SEM (rep:rep_ty)
       (mem:*memory,pc:*wordn,acc:*wordn,rtn:*wordn,iack:bool) =
       (mem,(inc rep) pc,acc,rtn,iack)
```

Hardware Interrupts

The processing of a hardware interrupt is described in a similar way by the definition of a function which computes the next state of the microprocessor from its current state.

```
pc ← (wordn rep) 0
rtn ← pc
iack ← T
```

```
⊢def IRQ_SEM (rep:rep_ty)
       (mem:*memory,pc:*wordn,acc:*wordn,rtn:*wordn,iack:bool) =
       (mem,((wordn rep) 0),acc,pc,T)
```

An interrupt is processed by loading the hard-wired address of the interrupt routine (location 0) into the program counter, saving the current contents of the program counter in the return address register, and setting the interrupt acknowledge flag `iack` to `T`.

Instruction Cycle

The opcode of the current instruction word determines which instruction is executed during a particular instruction cycle. The following set of definitions specify the opcode value for each instruction.

```
⊢def JZR_OPC = 0
⊢def JMP_OPC = 1
⊢def ADD_OPC = 2
⊢def SUB_OPC = 3
⊢def LDA_OPC = 4
⊢def STA_OPC = 5
⊢def RFI_OPC = 6
⊢def NOP_OPC = 7
```

The opcode value of the current instruction is obtained by fetching the memory word addressed by the program counter, extracting the value of its opcode field and interpreting the opcode as a number between 0 and 7. This procedure is specified in the definition of `OpcVal`

```
⊢def OpcVal (rep:rep_ty) (mem,pc) =
       (val3 rep) ((opcode rep) ((fetch rep) (mem,(address rep) pc)))
```

Every instruction cycle results in the execution of a programming level instruction unless a hardware interrupt is detected at the beginning of this cycle. The following definition of **NextState** specifies the overall control mechanism for determining what happens during a particular instruction cycle.

```
⊢def NextState (rep:rep_ty) (ireq,mem,pc,acc,rtn,iack) =
       let opcval = OpcVal rep (mem,pc) in
        ((ireq ∧ ¬iack)    ⇒ IRQ_SEM rep (mem,pc,acc,rtn,iack) |
         (opcval = JZR_OPC) ⇒ JZR_SEM rep (mem,pc,acc,rtn,iack) |
         (opcval = JMP_OPC) ⇒ JMP_SEM rep (mem,pc,acc,rtn,iack) |
         (opcval = ADD_OPC) ⇒ ADD_SEM rep (mem,pc,acc,rtn,iack) |
         (opcval = SUB_OPC) ⇒ SUB_SEM rep (mem,pc,acc,rtn,iack) |
         (opcval = LDA_OPC) ⇒ LDA_SEM rep (mem,pc,acc,rtn,iack) |
         (opcval = STA_OPC) ⇒ STA_SEM rep (mem,pc,acc,rtn,iack) |
         (opcval = RFI_OPC) ⇒ RFI_SEM rep (mem,pc,acc,rtn,iack) |
                              NOP_SEM rep (mem,pc,acc,rtn,iack))
```

Finally, we use the function **NextState** to define the predicate **TamarackBeh** which specifies the intended behaviour of the microprocessor as a relation on the time-dependent signals `mem`, `pc`, `acc`, `rtn` and `iack`.

```
⊢def TamarackBeh (rep:rep_ty) (ireq,mem,pc,acc,rtn,iack) =
       ∀u:time.
         (mem (u+1),pc (u+1),acc (u+1),rtn (u+1),iack (u+1)) =
         NextState rep (ireq u,mem u,pc u,acc u,rtn u,iack u)
```

The programming level model not only hides structural details of the internal architecture but also timing details about the number of microinstructions executed for each instruction. To be more precise, the programming level model describes the operation of

the microprocessor in terms of an abstract time scale where each instruction is uniformly executed in a single unit of time. This abstract time scale is different than the time scale used to specify the behaviour of register-transfer level components where a single unit of time corresponds to a single clock cycle. To emphasize this difference, we have used the explicit time variable u instead of t in the above definition of `TamarackBeh` (but there is no logical distinction between these two variable names). A major part of the task of formally verifying TAMARACK-3 is to establish a formal relationship between these two granularities of discrete time.

Minimal Assumptions about Uninterpreted Primitives

Formal verification of the register-transfer level implementation of TAMARACK-3 is not a trivial problem. For instance, this level of verification includes a rigorous analysis of how TAMARACK-3 interacts asynchronously with external memory using a handshaking protocol. However, these correctness results depend very little on computational aspects of the machine's operation.

In place of a full-scale computational model, the generic specification of TAMARACK-3 is supplemented by a minimal set of assumptions necessary to prove that the register-transfer level implementations is correct with respect to the programming level model. Just two assumptions are needed:

```
∀w. ((val3 rep) w) < 8
∀n. n < 16 ==> (((val4 rep) ((word4 rep) n)) = n)
```

The first assumption states that the value of any 3-bit word, when interpreted as the representation of a natural number, is less than eight. The second assumption states that the functions selected by `val4` and `word4` from the representation variable are inverses for numbers less than sixteen. These two assumptions appear explicitly in the correctness results for the register-transfer level implementation of TAMARACK-3.

What is Proved ?

The formal verification of TAMARACK-3 at the register-transfer level is mostly concerned with three main issues:

- showing that the right actions occur at the right time, e.g, for an ADD instruction, that the ALU operation for addition is applied to the accumulator currents and the operand (fetched from memory) and that the result is stored in the accumulator.
- demonstrating that the microprocessor satisfies the handshaking protocol used for asynchronous interaction with external memory.
- establishing that a precisely defined timing relationship holds between the programming level model time scale and the register-transfer level time scale.

With the exception of the two assumptions mentioned above, these correctness results do not depend on any computational details about the functional units used in the register-transfer level implementation of TAMARACK-3. Functional units such as the ALU are specified generically with the same selector functions used to specify the TAMARACK-3 programming model. For instance, the four selectors `inc`, `add`, `sub` and `wordn` are used to generically specify the four primitive operations performed by the ALU.

```
⊢def ALU (rep:rep_ty) (f0,f1,inp1,inp2,out) =
       ∀t:time.
         out t =
            (((f0 t,f1 t) = (T,T)) → ((inc rep) (inp2 t)) |
             ((f0 t,f1 t) = (T,F)) → ((add rep) (inp1 t,inp2 t)) |
             ((f0 t,f1 t) = (F,T)) → ((sub rep) (inp1 t,inp2 t)) |
                                     ((wordn rep) 0))
```

Therefore, it should be clear that correctness results for TAMARACK-3 at the register-transfer level are not concerned with the implementation of functional units, for instance, whether the addition operation has been correctly implemented in the ALU. Using uninterpreted primitives instead of defined operations sharpens the distinction between what has and what has not been formally considered at this level of proof.

Creating Instances of a Generic Specification

The representation variable rep which appears as an extra parameter in definitions throughout the formal specification of TAMARACK-3 is effectively a parameterization of the formal theory. It provides a means of relating this theory to both lower and higher level models of computation.

For example, by assigning an appropriate value to the representation variable **rep**, the formal specification of the TAMARACK-3 programming level model can be made to stack upon a lower level theory about the implementation of register-transfer level devices. This lowel level theory might also, in turn, be a generic specification parameterized by its own representation variable and stacked upon an even lower level of representation at the transistor level.

To illustrate this idea with a simple example, the constant REP16 is defined as a value for rep based on the built-in HOL data types described in [4, 5, 9]. In this case, we have created data types for a 16-bit version of TAMARACK-3.

```
⊢def ISZERO16 w = ((VAL16 w) = 0)

⊢def INC16 w = WORD16 ((VAL16 w) + 1)

⊢def ADD16 (w1,w2) = WORD16 ((VAL16 w1) + (VAL16 w2))

⊢def SUB16 (w1,w2) = WORD16 ((VAL16 w1) - (VAL16 w2))

⊢def OPCODE w = WORD3 (V (SEG (0,2) (BITS16 w)))

⊢def ADDRESS w = WORD13 (V (SEG (3,15) (BITS16 w)))
```

```
⊢def REP16 =
       ISZERO16,                          % iszero %
       INC16,                             % inc %
       ADD16,                             % add %
       SUB16,                             % sub %
       WORD16,                            % wordn %
       VAL16,                             % valn %
       OPCODE,                            % opcode %
       VAL3,                              % val3 %
       ADDRESS,                           % address %
       (λ(x,y). FETCH13 x y),             % fetch %
       (λ(x,y,z). STORE13 y z x),         % store %
       WORD4,                             % word4 %
       VAL4                               % val4 %
```

There are two main reasons for stacking correctness results upon lower level correctness results. One reason is to discharge assumptions introduced at one level by establishing them as theorems at lower levels. For instance, the two assumptions,

```
∀w. ((val3 rep) w) < 8
∀n. n < 16 ⟹ (((val4 rep) ((word4 rep) n)) = n)
```

needed to obtain correctness results at the register-transfer level could be established as theorems when the representation variable `rep` is instantiated to be `REP16`:[6]

```
∀w. (VAL3 w) < 8
∀n. n < 16 ⟹ ((VAL4 (WORD4 n)) = n)
```

The other main reason for stacking correctness results is to link correctness results at one level together with correctness results at lower levels to obtain a single correctness result spanning multiple levels of a hierarchical specification. This hierarchy can even extend upwards above the level of digital hardware. For instance, correctness results for a formally verified compiler [13] can be stacked upon correctness results for the TAMARACK-3 microprocessor to formally relate the semantics of a programming language to the execution of a compiled program by digital hardware. In this manner, a hierarchy of widely separated concerns can be treated in a truely hierarchical fashion - each level isolated from details only relevant to other levels.

Summary

This paper has argued that generic specification is a powerful concept in the context of formally verifying digital hardware. In addition to the well-known advantages of modularity, abstraction and reliable re-usability, the use of generic specification to eliminate non-essential detail from a formal specification sharpens the distinction between what has and what has not been formally considered. In a hierarchical proof effort, the elimination

[6] The current version of the built-in HOL data types (as given by the eval library in the HOL88 system) is not fully axiomatized or secure, but with a complete axiomatization it would be possible to derive these two assumptions as theorems.

of non-essential detail isolates each level from details only relevant to other levels. Finally, there is the practical benefit of reducing the need for special-purpose proof infrastructure, e.g., hardware-oriented data types.

As mentioned earlier in the introduction of this paper, most of the languages used for hardware specification do not provide explicit mechanisms for supporting and encouraging generic specifications. Some of the few exceptions include EHDM [17] and OBJ [8]. However, this paper shows that it is unnecessary to have explicit mechanisms for generic specification in the case of languages with (at least) the expressive power of higher-order logic. We have described a technique for expressing genericity based on the use of higher-order predicates parameterized by function variables and type variables. We believe that the use of higher-order predicates in this manner this is a very direct (if not the most direct) way to specify hardware generically. Therefore, we claim that the ability to express genericity is a very strong argument for why it is necessary, for all practical purposes, to use a formalism with (at least) the expressive power of higher-order logic.

Acknowledgments

The ideas presented in this paper are based on my Ph.D. research while a member of the Hardware Verification Group at Cambridge University. I am particularly indebted to my supervisor, Mike Gordon, who helped to refine and clarify many of these ideas. I am also grateful to a number of people who commented on drafts of my Ph.D. dissertation especially John Herbert and John Van Tassel. Pioneering work on microprocessor verification by Avra Cohn and Warren Hunt provided a very good starting point for this research. The idea of generic specification was prompted by discussions with John Rushby and his colleagues at SRI International. This research is currently supported by an NSERC (*Natural Sciences and Engineering Research Council*) Operating Grant.

References

[1] William R. Bevier, Warren A. Hunt, Jr., and William D. Young, in: Towards Verified Execution Environments, in: Proceedings of the 1987 IEEE Symposium on Security and Privacy, 27-29 April 1987, Oakland, California Computer Society Press, Washington, D.C., 1987 pp. 106-115. Also Report No. 5, Computational Logic, Inc., Austin, Texas, February 1987.

[2] W. Bevier, W. Hunt, J Moore, and W. Young, An Approach to Systems Verification, Journal of Automated Reasoning, Vol. 5, No. 4, November 1989. Also Report No. 41, Computational Logic, Inc., Austin, Texas, April 1989.

[3] Avra Cohn and Mike Gordon, A Mechanized Proof of Correctness of a Simple Counter, Report No. 94, Computer Laboratory, Cambridge University, July 1986.

[4] Avra Cohn, A Proof of Correctness of the Viper Microprocessor: The First Level, in: G. Birtwistle and P. Subrahmanyam, eds., VLSI Specification, Verification and Synthesis, Kluwer Academic Publishers, Boston, 1988, pp. 27-71. Also Report No. 104, Computer Laboratory, Cambridge University, January 1987.

[5] Avra Cohn, Correctness Properties of the Viper Block Model: The Second Level, in: G. Birtwistle and P. Subrahmanyam, eds., Current Trends in Hardware Verification

and Automated Theorem Proving, Springer-Verlag, 1989, pp. 1-91. Also Report No. 134, Computer Laboratory, Cambridge University, May 1988.

[6] Avra Cohn, The Notion of Proof in Hardware Verification, Journal of Automated Reasoning, Vol. 5, May 1989, pp. 127-139.

[7] H. Eveking, How to Design Correct Hardware and Know It. G. Milne, ed., The Fusion of Hardware Design and Verification, Proceedings of the IFIP WG 10.2 International Working Conference, Glasgow, Scotland, 3-6 July 1988, North-Holland, 1988, pp. 250-262.

[8] Joseph A. Goguen, OBJ as a Theorem Prover with Applications to Hardware Verification, in: G. Birtwistle and P. Subrahmanyam, eds., Current Trends in Hardware Verification and Automated Theorem Proving, Springer-Verlag, 1989, pp. 219-267. Also Report No. SRI-CSL-4R2, Computer Science Laboratory, SRI International, Menlo Park, August 1988.

[9] M. Gordon, LCF_LSM, Report No. 41, Computer Laboratory, Cambridge University, 1983.

[10] M. J. C. Gordon, Why Higher-Order Logic is a Good Formalism for Specifying and Verifying Hardware, in: G. Milne and P. Subrahmanyam, eds., Formal Aspects of VLSI Design, Proceedings of the 1985 Edinburgh Conference on VLSI, North-Holland, 1986, pp. 153-177.

[11] Michael J. C. Gordon et al., The HOL System Description, Cambridge Research Centre, SRI International, Suite 23, Miller's Yard, Cambridge CB2 1RQ, England.

[12] Jeffrey J. Joyce, Generic Structures in the Formal Specification and Verification of Digital Circuits, in: G. Milne, ed., Proceedings of the IFIP WG 10.2 Working Conference on The Fusion of Hardware Design and Verification, 4-6 July 1988, Glasgow, Scotland, pp. 51-75.

[13] Jeffrey J. Joyce, Totally Verified Systems: Linking Verified Software to Verified Hardware, in: M. Leeser and G. Brown, eds., Specification, Verification and Synthesis: Mathematical Aspects, Proceedings of a Workshop, 5-7 July 1989, Ithaca, N.Y., Springer-Verlag, 1989. Also Report No. 178, Computer Laboratory, Cambridge University, September 1989.

[14] Jeffrey J. Joyce, Multi-Level Verification of Microprocessor-Based Systems, Ph.D. Thesis, Computer Laboratory, Cambridge University, December 1989. Report No. 195, Computer Laboratory, Cambridge University, May 1990.

[15] J Strother Moore, Piton: A Verified Assembly Level Language, Report No. 22, Computational Logic Inc., Austin, Texas, September 1988.

[16] W. Luk and G. Jones, From Specifications to Parmeterised Architectures, in: G. Milne, ed., Proceedings of the IFIP WG 10.2 Working Conference on The Fusion of Hardware Design and Verification, 4-6 July 1988, Glasgow, Scotland, pp. 267-288.

[17] F. W. von Henke, J. S. Crow, R. Lee, J. M. Rushby and R. A. Whitehurst, The EHDM Verification Environment: An Overview, Proceedings of the 11th National Computer Security Conference, Baltimore, October 1988, pp. 147-155.