Technical Report

UCAM-CL-TR-147
ISSN 1476-2986

Number 147

UNIVERSITY OF CAMBRIDGE

Computer Laboratory

Formal specification and verification of microprocessor systems

Jeffrey Joyce

September 1988

15 JJ Thomson Avenue
Cambridge CB3 0FD
United Kingdom
phone +44 1223 763500

https://www.cl.cam.ac.uk/

Technical reports published by the University of Cambridge
Computer Laboratory are freely available via the Internet:

https://www.cl.cam.ac.uk/techreports/

ISSN 1476-2986

Formal Specification and Verification of Microprocessor Systems

Jeffrey J. Joyce

Computer Laboratory
New Museums Site
Pembroke Street
Cambridge CB2 3QG

Keywords: microprocessors, specification and verification, hardware description languages, formal methods, safety-critical applications, VLSI design methodology.

ABSTRACT

This paper describes the use of formal methods to verify a very simple microprocessor. The hierarchical structure of the microprocessor implementation is formally specified in higher-order logic. The behaviour of the microprocessor is then derived from a switch level model of MOS (*Metal Oxide Semiconductor*) behaviour using inference rules of higher-order logic with assistance from a mechanical theorem-proving system. The complexity of the formal proof is controlled by a multi-level approach based on increasingly abstract views of time and data. While traditional methods such as multi-level simulation may reveal errors or inconsistencies, formal verification can provide greater certainty about the correctness of a design. The main difference with formal verification, and its strength, is that behaviour at one level is formally derived from lower levels with a precise statement of the conditions under which one level accurately models lower levels.

1. INTRODUCTION

The use of formal methods in hardware design is an increasingly active area of research fueled by the widespread use of microprocessor systems in real-time control. [8] suggests that the use of formal methods may soon become a requirement in the certification of "safety-critical" devices in such control applications as nuclear reactors and avionics. The maturity of these methods is not widely known, yet, formal methods have been used to design or specify (aspects of) several commercially-available microprocessor systems including: the British Ministry of Defense VIPER microprocessor [7,8]; microcode for the floating point unit of the INMOS T800 transputer [26]; and the Motorola 6800 instruction set [3].

[1] An earlier version of this paper will appear in: EUROMICRO 88, Proceedings of the 14th Symposium on Microprocessing and Microprogramming, Zurich, Switzerland, 29 August - 1 September, 1988, S. Winter and H. Schumny, eds., North-Holland, 1988.

The very simple microprocessor described in this paper, called "Tamarack", was originally developed as a hardware verification example [12]. But unlike other examples involving commercially-available microprocessors where the use of formal methods has been limited to higher level aspects of the design, the Tamarack implementation has been completely specified down to the transistor level. We have also gained experience with the practical application of these methods by implementing an earlier version of this microprocessor as a 5,000 transistor CMOS microchip (Figure 1).

The microprocessor implementation is described by a relation on the externally visible input and output signals of the microprocessor. This relation is defined hierarchically in higher-order logic following the hierarchical structure of the design. The very lowest level of specification is defined in terms of primitive behaviours for CMOS transistors, capacitances and voltage sources. Starting with this structural specification, the semantics of the microprocessor instruction set are formally derived through a series of intermediate specifications based on increasingly abstract views of time and data. The highest level of specification describes the programmer's model of the microprocessor.

This paper describes the formal specification of hardware in higher-order logic and outlines the formal proof which derives the behavioural specification of the Tamarack microprocessor from its structural specification. Thus, the paper provides both an introduction to the formal specification and verification of microprocessor systems as well as a description of some steps towards building totally verified computer systems for safety-critical applications.

2. FORMAL SPECIFICATION

A variety of languages have been to used to write formal specifications of computer hardware and architecture. These languages usually fall into one or more of the following categories.

- A language specifically designed for the description of computer hardware and architecture.
- A language embedded in a general-purpose programming language possibly requiring extensions.
- A formalism with transformation rules and a well-defined semantics, *e.g.* mathematical logic.

ISPS (*Instruction Set Processor Specifications*) [1] is an example of a hardware description language which has been specifically designed to describe hardware

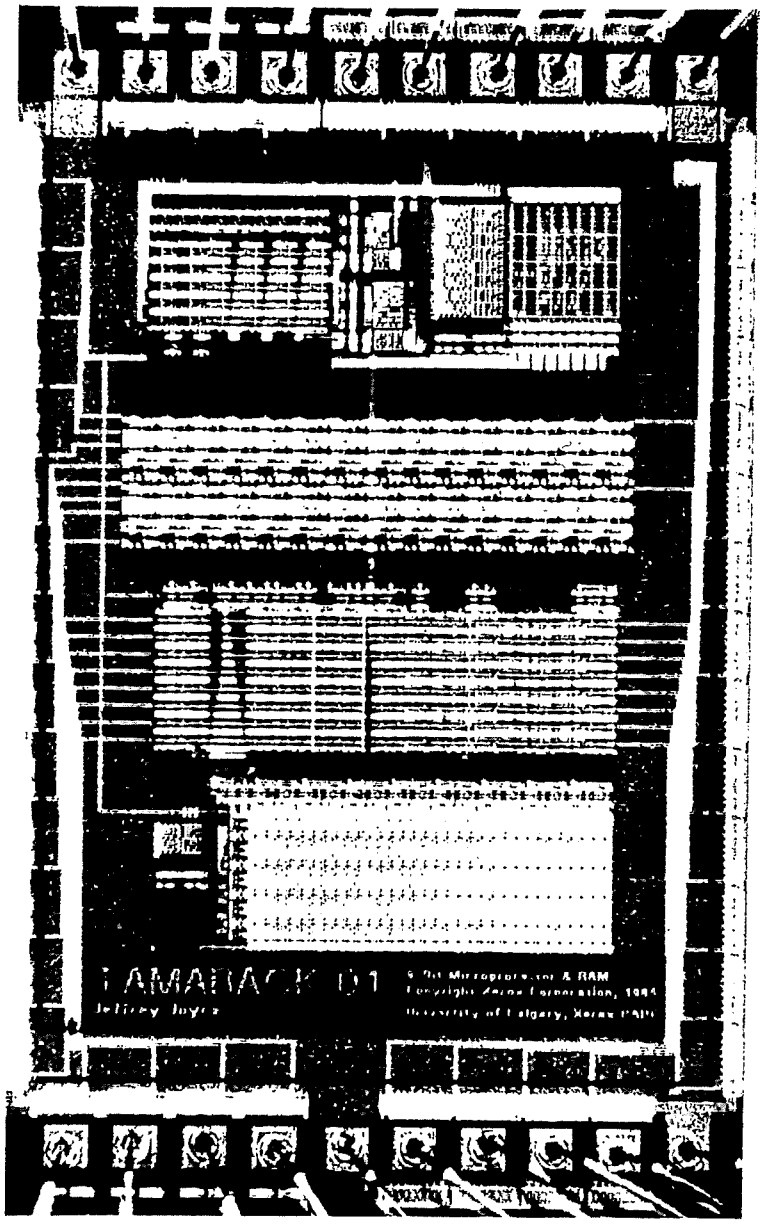

Figure 1: Tamarack Microprocessor, 3.8 by 2.8 mm.

and architecture. Languages in this category provide a design-capture tool as well as a documentation aid. While this approach offers the advantage of generality, it depends on *ad-hoc* simulators or language interfaces for existing simulation tools.

Hardware description within a general-purpose programming language offers the advantage of an executable specification. Designers of the SCHEME-79 chip used LISP to specify microcode; by defining functions to simulate primitive machine operations, the operation of the chip could be simulated by running the microcode as a LISP program [35]. Hardware description languages in this category usually exploit distinctive features of the programming language. [6] exploits the relational nature of PROLOG to model bi-directional signal flow. [18] describes an APL approach observing that the flow of data from source to destination acted upon sequentially by various devices in a datapath corresponds well with the way in which APL statements are executed from right to left.

Programming language notations with well-defined semantics have led to approaches which combine the advantages of an executable specification with the advantages of a manipulable formalism. [34] extended Backus's FP language with a notion of "state" to describe regular array circuits. Algebraic laws based on the semantics of μFP can be used to transform a behavioural specification into an implementation. Similarly, algebraic laws for OCCAM have been used to transform a high-level specification into microcode for the floating unit of the INMOS T800 transputer [26]. Parallelism in the OCCAM language is used to simulate the inherent parallelism of hardware.

Several approaches have adapted existing formalisms to the purpose of describing hardware behaviour and structure. The LCF_LSM system extended Scott and Milner's PPLAMDA logic with terms denoting the behaviour of sequential machines; this was used to produce the original proof of correctness for the Tamarack microprocessor at the register-transfer level [12]. This register-transfer level implementation was also proven correct in a PROLOG based approach using a special-purpose logic similar to LCF_LSM [2]. Another formalism designed specifically to describe aspects of hardware (and software) behaviour is the CIRCAL calculus [28].

More general formalisms have also been used as hardware description languages. [29] used temporal logic to specify the behaviour of the AM2901 bit slice and proved correctness results about the interconnection of bit slices. An executable subset of this logic has been implemented as an imperative programming language and used to execute hardware specifications [30]. Hunt [19] uses the notation of pure LISP to describe the structure and behaviour of his design for a 16-bit microprocessor. Recursive functions model sequential logic where registers and memory appear as function parameters. Each recursive call updates the contents of

registers with combinational logic simulated by the body of the function definition. The underlying formalism is Boyer-Moore logic, a quantifier-free, first-order logic with equality and induction schemes. Using the Boyer-Moore theorem-prover, Hunt has proved the correctness of his implementation addressing issues such as multiple levels of data representation and asynchronous communication.

The approach described in this paper, inspired by [13], is based entirely on the formalism of higher-order logic. Higher-order logic can be used to specify aspects of hardware structure and behaviour which are cumbersome when expressed in less powerful formalisms. This increased expressiveness is partly due to terms such as existential quantification which are not executable. The importance of simulation as a means of exercising a specification is recognized in our approach; [5] describes the implementation of an executable subset of higher-order logic. However, the main emphasis in this paper is the formal verification of hardware using inference rules of higher-order logic to derive theorems expressing the correctness of an implementation with respect to a specification of its intended behaviour.

3. FORMAL VERIFICATION

Hardware is usually simulated at several levels of detail, *e.g.* instruction set execution, register-transfer level, switch level and circuit level. General-purpose programming languages are used to simulate higher level aspects of behaviour while special-purpose simulation tools such as MOSSIM [4] and SPICE [31] are used at lower levels. The apparent absence of errors at one level is often taken as an indication that the simulation model at the next higher level is an accurate model of the hardware.

While multi-level simulation is an important design tool which may reveal errors or inconsistencies, it is not a proof of correctness. It is usually impractical to simulate an entire design at the detailed levels of switch and circuit level simulation; only sub-systems or worst-case models are investigated at this level of detail. Even with massive gains in computational power, *e.g.* parallel processing, the meaningful interpretation of results from large scale simulations is not a simple task. Furthermore, exhaustive testing is rarely possible and it is a difficult problem to design a finite but "complete" set of tests. Finally, the relationship between simulation levels is not formally defined and the exact conditions under which one level accurately models a lower level are not always clear.

Formal verification provides a solution for each of these problems. The incremental and hierarchical nature of formal proof allows a large design to be investigated at a detailed level of behaviour. The structure of the Tamarack microprocessor is described hierarchically by a network of primitives based on switch level

models for transistors, capacitance and voltage sources. Initial steps in the verification procedure prove the correctness of simple components such as logic gates. For example, the inputs and output of a NAND gate are shown to be related by the logical NAND function under certain conditions. These results are then used to prove the correctness of larger components such as an n-bit adder. Once a component has been verified, its structural specification is replaced by its behavioural specification. For instance, the structural specification of the Tamarack ALU (*Arithmetic Logic Unit*) is replaced by its behavioural specification defined in terms of arithmetic functions. At higher levels, the correctness of the microcode is verified by showing that target level operations, *e.g.* the execution of instructions in a program, are correctly implemented by the corresponding sequence of microinstructions. Microcode verification resembles simulation; however, the state of the microprocessor system is represented symbolically allowing all possible states to be tested.

Another advantage of this approach is that the verification methodology avoids the "de-contextualization" of design errors. For example, a design error in the ALU will prevent derivation of the ALU behaviour before any other aspect of the design is considered. This contrasts with the simulation approach where the manifestation of the error may appear in some unrelated part of the simulation model and often many simulation steps later. A similar point is made in [37].

The complexity of the verification task is controlled by structuring the formal proof with several intermediate levels of specification based on increasingly abstract views of time and data. In order of increasing abstraction, these are:

switch level implementation
register-transfer level model
abstract state machine
target machine

Each level of abstraction simplifies one or more aspects of the low-level behaviour of the microprocessor implementation. The main simplifications introduced in the register-transfer level view are sequential behaviour and Boolean logic. The next level of abstraction is principally a structural simplification which replaces the register-transfer level architecture with an abstract state machine. Correctness of the microcode is established by symbolic simulation of the abstract state machine using inference rules of higher-order logic. The highest level of abstraction, the target machine, specifies the net effect of microinstruction sequences; the main simplification at this level is the contraction of behaviour over time into a single state transition for each target level operation.

As with multi-level simulation, this structured approach to verification models

hardware behaviour at various levels of accuracy and complexity. Indeed, each level of specification corresponds to a simulation model. The main difference with formal verification, and its strength, is that behaviour at one level is formally derived from lower levels with a precise statement of the conditions under which one level accurately models lower levels.

The use of formal methods in hardware design is not a new idea; Boolean algebra has always been used to show that networks of logic gates implement specific functions. However, the example described in this paper shows that formal methods can be applied to a much greater range of digital behaviours and moreover, that this is possible within a single formalism.

4. HIGHER-ORDER LOGIC

Functions which accept other functions as arguments or return functions as results are called "higher-order" functions. Such functions often lead to function definitions which are both simple and powerful [16]. Higher-order logic is a formalism for reasoning about functions; this includes higher-order functions as well as relations, *i.e.* functions that return Boolean values. Following the higher-order logic approach presented in [13], hardware is described by relations on input and output signals where signals are modelled by functions from discrete time to sampled values. Because signals are modelled by functions, these will be higher-order relations. Thus, higher-order logic provides a formal basis for reasoning about functions and relations modelling hardware.

Higher-order logic extends first-order logic by allowing variables to range over functions and predicates. Most of the notation such as "$\forall$" ("*for all*") and "$\exists$" ("*there exists*") should be familiar to computer scientists. Every term in higher-order logic has a "type" which is either a primitive type, *e.g.* Booleans, natural numbers, or built up from other types using type constructors such as Cartesian product. Every compound term must correctly type-check; for example, a function from numbers to Booleans can only be applied to a number.

The derivation of non-trivial theorems by hand is usually a tedious and complex undertaking. However, much of the work can be reduced to simple syntactic checks that primitive inference rules have been used correctly. This task has been automated in the HOL theorem-proving system [14]. This system, a descendant of Edinburgh LCF [11], is an interactive computer program written in a mixture of LISP and a functional programming language called ML. Derived inference rules and powerful proof strategies can be programmed as ML functions to automate most of the minor steps leaving the user to supply only the main steps in a proof [32].

In addition to standard logical connectives, the HOL language supports conditional expressions, "`b ⇒ t1 | t2`", which may be read as "*if* `b` *then* `t1` *else* `t2`", let-expressions, "`let x = t1 in t2`", denoting the result of replacing all free occurrences of `x` in `t2` by `t1`, and n-tuples of the form "`(t1, ... tn)`". Certain features of the HOL language have a special syntactic status for improved readability, *e.g.* the definition of infix functions. The language also includes basic arithmetic functions and relations.

5. CMOS SWITCH MODEL

A single wire in a CMOS circuit is modelled by a function from discrete time to a fixed set of MOS values, `Hi` (high), `Lo` (low), `Zz` (high impedance) and `Er` (unknown or undefined). The behaviour of a simple logic gate is represented by the sequence of MOS values sampled at each of its inputs and outputs. For example, the sequence of input and output values in Table 1 illustrates the behaviour of a CMOS NAND gate. Interpreting `Hi` and `Lo` as the Boolean values `T` and `F` respectively, the output value is always the logical NAND of the input values when the inputs are "well-defined", *i.e.* `Hi` or `Lo`. When one of the inputs is not well-defined, *i.e.* `Zz` or `Er`, the output is `Er`.

	t	$t+1$	$t+2$	$t+3$	$t+4$	$t+5$
a:	Hi	Hi	Lo	Lo	Zz	Lo
b:	Hi	Lo	Hi	Lo	Hi	Er
out:	Lo	Hi	Hi	Hi	Er	Er

Table 1: NAND Gate Behaviour.

The behaviour of a NAND gate can be derived from a switch level model of CMOS behaviour. This model, motivated by [9], is related to the simulation model presented in [4]. P-type and N-type transistors are modelled as switches, but unlike [4] these models are uni-directional and delay-less. Even though fabricated transistors are bi-directional and have delay, these primitives provide a reasonable model of behaviour for the restricted circuit style and clocking scheme used to implement the Tamarack microprocessor. The validity of the model in this design style could be established by traditional analysis techniques such as circuit simulation, timing verification and signal flow analysis or formally derived from more accurate circuit models such as those described in [17,24,38].

The predicates `Ptran` and `Ntran` are defined as relations on input and output signals. These signals are explicitly declared as functions from discrete time to MOS values by the type abbreviation "`wire`". Function application is denoted

by juxtaposition; for example, "g t" is the MOS value of the transistor gate at time t. The right-hand sides of the output equations use conditional expressions to determine the output value. Universal quantification is used to state that the output equations hold for all times t.

```
⊢ Ptran (g:wire,i:wire,o:wire) =
    ∀t. o t = (g t = Lo) ⇒ i t | (g t = Hi) ⇒ Zz | Er

⊢ Ntran (g:wire,i:wire,o:wire) =
    ∀t. o t = (g t = Hi) ⇒ i t | (g t = Lo) ⇒ Zz | Er
```

Capacitance in a CMOS circuit is modelled explicitly by the Cap primitive which is the only primitive expressing a temporal relationship. This primitive provides a model of charge storage used to derive the behaviour of clocked registers in the Tamarack implementation.

```
⊢ Cap (i:wire,o:wire) = ∀t. o(t+1) = (i(t+1) = Zz) ⇒ o t | i(t+1)
```

To model the result of joining two wires, the four MOS values are ordered to form a lattice with a top, Er, and a bottom, Zz, as shown in Figure 2. The infix function ⊔ computes the least upper bound of two MOS values based on this ordering.

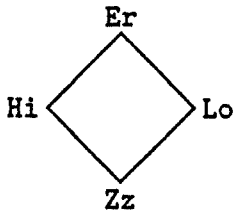

Figure 2: Lattice of MOS Values.

```
⊢ x ⊔ y = ((x = y) ⇒ x | (x = Zz) ⇒ y | (y = Zz) ⇒ x | Er)
```

The Join primitive determines the result of joining two wires by taking the least upper bound of their current values. This models the dominating effect of a non-floating value over Zz and the even greater dominance of Er over the other three MOS values. Primitives for voltage sources are also shown below.

```
⊢ Join (i1:wire,i2:wire,o:wire) = ∀t. o t = (i1 t ⊔ i2 t)

⊢ Vdd (o:wire) = ∀t. o t = Hi

⊢ Gnd (o:wire) = ∀t. o t = Lo
```

These switch level primitives are used to specify implementations of CMOS circuits. When the formal parameters of a behaviour predicate are replaced by the actual names of wires and busses in an implementation, the result is a behaviour term describing the effect of the component in the implementation. The behaviour of an implementation can be expressed by the logical conjunction of behaviour terms for each of the components in the implementation. Existential quantification is used to hide internal wires, *i.e.* signals not connected to the external ports of the implementation. The following specification illustrates the use of logical conjunction and existential quantification to describe the implementation of a CMOS NAND gate (Figure 3).

```
⊢ NAND (a:wire,b:wire,out:wire) =
    ∃ w1 w2 w3 w4 w5 w6 w7.
      Vdd w1 ∧
      Ptran (a,w1,w2) ∧
      Ptran (b,w1,w3) ∧
      Join (w2,w3,w4) ∧
      Join (w4,w5,out) ∧
      Ntran (a,w6,w5) ∧
      Ntran (b,w7,w6) ∧
      Gnd w7
```

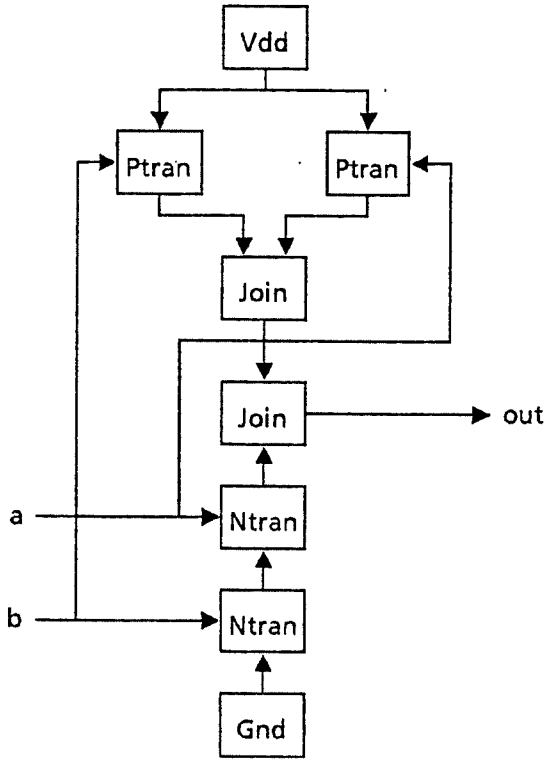

Figure 3: CMOS NAND Gate Implementation.

Behaviour composition and signal hiding provide a basis for the hierarchical specification of hardware structure. Circuits are defined structurally at one level of specification and then used as components at higher levels. Another specification technique is iteration, for example, of stages in an n-bit adder. This can be expressed in higher-order logic either by primitive recursion or by bounded quantification. A third method of controlling structural complexity in a hardware specification is based on the use of generic structures [22].

The low-level behaviours of the switch level primitives are expressed directly in terms of relations on MOS values. Using techniques to exploit hierarchy, iteration and generic structure, the structural specification of a circuit in higher-order logic results in a system of simultaneous constraints on inputs and outputs of the circuit. The behaviour defined by these constraints is also a relation on MOS values but this will almost always be too complex to express directly in terms of MOS values. Instead, behaviour is expressed at higher algebraic levels such as Boolean logic and arithmetic functions by defining formal relationships between MOS values and higher levels of data representation. This higher level view of behaviour is then formally derived from lower levels. For example, the following theorem states the correctness of the CMOS NAND gate in terms of an equation on Boolean values using the abstraction function `BoolAbs` to interpret MOS values as Boolean values and the predicate `WellDef` to describe the conditions under which this abstraction holds.

```
⊢ ∀ a b out.
    NAND (a,b,out)
    ⟹
    ∀t.
      WellDef (a t) ∧ WellDef (b t)
      ⟹
      BoolAbs (out t) = ~(BoolAbs (a t) ∧ BoolAbs (b t))
```

The above theorem states that the behavioural specification of a NAND gate is implied by its structural specification. The function `BoolAbs` interprets MOS values as Boolean values by mapping `Hi` to `T` and `Lo` to `F`. Because all functions are total in higher-order logic, `BoolAbs` maps the other two MOS values, `Zz` and `Er`, to arbitrary but fixed values. Since the output equation only holds when the input values are either `Hi` or `Lo`, the predicate `WellDef` is used to express this validity condition.

```
⊢ WellDef m = (m = Hi) ∨ (m = Lo)
```

Simple logic gates can be used to implement n-bit wide structures such as an n-bit adder. Functions from bit positions to MOS values are used to model n-bit words.

For example, the binary value 011 is represented by the function f where $f(0)$ is Hi, $f(1)$ is Hi and $f(2)$ is Lo. An n-bit bus is a function from discrete time to n-bit words. The following definition shows the formal specification of an n-bit adder using the higher-order relation Iterate to specify n iterations of a 1-bit adder. Iterate has a primitive recursive definition which uses existential quantification to hide internal carry signals.

```
⊢ ADDERn n (a:bus,b:bus,out:bus) = Iterate n (a,b,out) ADDER1
```

The correctness of the n-bit adder can be stated in terms of ADDn which denotes an addition operation for n-bit words. The n-bit adder is then combined with other n-bit structures to implement the 3-function ALU used in the Tamarack datapath. The following theorem states the correctness of the ALU with respect to its register-transfer level behaviour.

```
⊢ ∀ n f0 f1 a b out.
    ALUn n (f0,f1,a,b,out) ∧
    BusWidth n out
    ⟹
    ∀t.
      WellDef (f0 t) ∧ WellDef (f1 t) ∧
      WellDefn n (a t) ∧ WellDefn n (b t)
      ⟹
      ∃w.
        out t =
          (((f0 t,f1 t) = (Hi,Hi)) ⇒ w |
           ((f0 t,f1 t) = (Lo,Hi)) ⇒ INCn n (b t) |
           ((f0 t,f1 t) = (Lo,Lo)) ⇒ ADDn n (a t,b t) |
                                      SUBn n (a t,b t))
```

Both the structural and behavioural specifications of the ALU are parameterized by n, the word size of the microprocessor. In addition to the constraints imposed by ALUn, the predicate BusWidth is used to declare the width of the output bus. For well-defined control signals and well-defined inputs, the output of the ALU is determined by nested conditional expressions describing the output for each of the four possible pairs of well-defined control values. One of these pairs, (Hi,Hi), is not used in the Tamarack microcode; the unspecified value computed by the ALU for this pair of control values is hidden by existential quantification. The other three operations are described by INCn, ADDn and SUBn. These functions describe simple arithmetic operations on n-bit words.

The above theorem illustrates how a complex relation on MOS signals can be succinctly expressed in terms of simple arithmetic relations. The functions INCn,

ADDn and SUBn are defined in terms of arithmetic functions on the natural numbers using abstraction functions which convert n-bit words to numbers and numbers to n-bit words. For instance, the definition of ADDn converts the two n-bit operands into numbers and then converts their sum back into an n-bit word. The abstraction functions Wordn and Valn have straightforward definitions based on the unsigned binary representation of a number and the representation of n-bit words in higher-order logic by functions from bit positions to MOS values.

```
⊢ ADDn n (w1,w2) = Wordn n ((Valn n w1) + (Valn n w2))
```

Following the verification strategy outlined in Section 3, the derived behaviour of the ALU is then used to reason about the function of the ALU in higher level aspects of the Tamarack design.

6. THE TAMARACK MICROPROCESSOR

Figure 4 shows the register-transfer level architecture of the Tamarack microprocessor. Even though this microprocessor was designed as a hardware verification example, it has features found in real designs such as microcoded control. Earlier work on this example is presented in [12,20,21]. Gordon's design has also been used by several other researchers to describe their own approaches to hardware specification and verification [2,33,37].

The externally visible state of the microprocessor consists of the program counter PC, accumulator ACC and external memory. Unseen by the user, the internal architecture involves several more registers used by the microcoded control to implement target level operations. In addition to registers and external memory, the datapath includes a 3-function ALU implementing addition, subtraction and incrementation. Datapath components are interfaced to a single bus using tri-state bus drivers. The control unit is implemented by the microcode ROM, a 5-bit latch MPC for the current microinstruction address and combinational logic to compute the address of the next microinstruction. The implementation also includes latches for external inputs from the function-select knob and data switches.

The microprocessor is always in one of two possible modes of operation. In "idle" mode, the user can use the button, knob and switches on the front panel of the computer to load values into the PC, ACC and external memory or start execution of a program. In "run" mode, program execution continues until a HALT instruction is encountered or the user interrupts by pushing the front panel button.

The simple instruction set precludes the possibility of illegal operations and conditions such as arithmetic overflow cannot be detected. As shown in Table 2, the

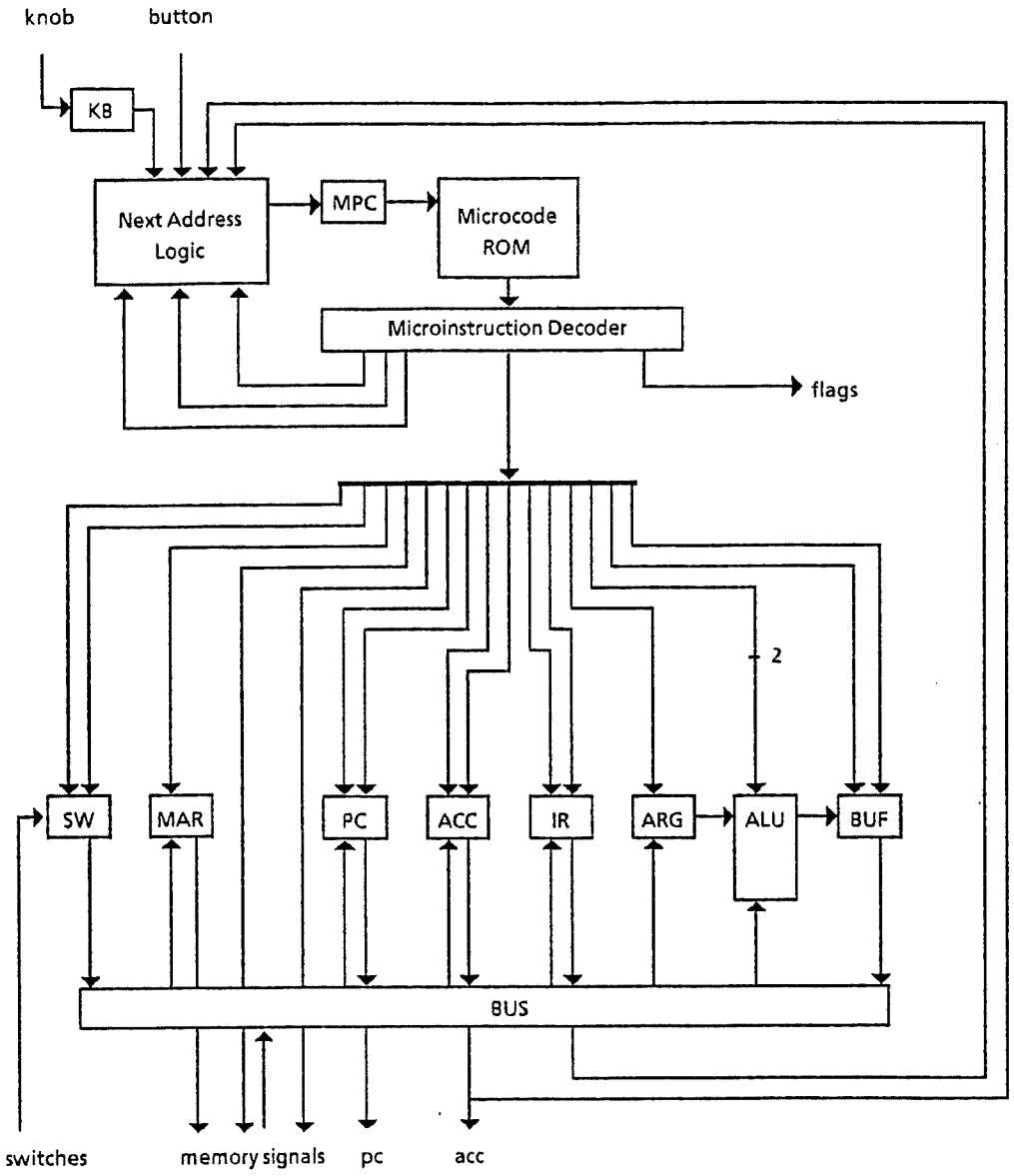

Figure 4: Register-Transfer Level Architecture.

microprocessor has eight instructions each consisting of a 3-bit opcode and an address field. In Gordon's original specification, the address field was thirteen bits wide but we consider a more general specification where the width of the address field is parameterized by a variable.

000—address—	HALT	Stop execution
001—address—	JMP	Jump
010—address—	JZR	Jump if ACC = 0
011—address—	ADD	Add contents to ACC
100—address—	SUB	Subtract contents from ACC
101—address—	LD	Load contents into ACC
110—address—	ST	Store ACC
111—address—	SKIP	Jump to next instruction

Table 2: Tamarack Instruction Set.

The register-transfer level architecture of the Tamarack microprocessor is formally specified in higher-order logic with the same specification techniques used to specify lower levels, *i.e.* hierarchical specification and iteration. For example, separate specifications for the control unit and the datapath are composed using logical conjunction to describe the top level structure of the microprocessor. Existential quantification is used to hide the internal connections between the control unit and datapath. Implementations of the control unit and datapath are specified in terms of register-transfer components such as the ALU. The result is a complete specification of the microprocessor implementation down to the level of switch level primitives.

7. TARGET MACHINE SPECIFICATION

The target machine is a high level behavioural specification of the Tamarack microprocessor describing both the semantics of the instruction set and the control structure. The target machine is defined as an abstract state machine whose state is a 4-tuple consisting of the current memory state, contents of the program counter and accumulator and a flag indicating whether the microprocessor is currently in idle mode. Each target level operation such as the execution of an instruction corresponds to a single state transition by the target machine. The next state of the machine is computed directly from the current state and the current input values.

Following the relational style of specification used to specify lower levels of behaviour, the target machine is formally defined in higher-order logic as a relation

on the externally visible signals of the microprocessor. Four of these signals correspond to the components of the target machine state. The program counter, accumulator and idle flag are physically available as output signals. The memory signal is only a virtual signal but it is still considered to be externally visible. Memory states are modelled by functions from memory locations to n-bit words and a memory signal is a function from discrete time to memory states. In addition to the state signals, the target machine relation involves three input signals. The specification is parameterized by the number of bits n in the address field of an instruction. Since the remaining bits form the 3-bit opcode, the word size of the microprocessor is $n + 3$.

```
⊢ TARGET_MACHINE n (
    (button:wire,knob:bus,switches:bus),
    (mem:memsig,pc:bus,acc:bus,idle:wire)) =
    ∀t.
      (mem(t+1),pc(t+1),acc(t+1),idle(t+1)) =
        NEXT_STATE n (
          (button t,knob t,switches t),(mem t,pc t,acc t,idle t))
```

The state of the machine at time t+1 is computed by the function NEXT_STATE directly from the state and inputs at time t. The formal specification of the target machine is completed by defining NEXT_STATE just as one would define this function in a functional programming language. Using a structured programming style, NEXT_STATE could be defined in terms of a function which computes the next state during an idle cycle and another function which computes the next state during a run cycle. The latter could be defined in terms of individual next state functions for each of the eight instructions. For instance, the following function specifies the semantics of an ADD instruction.

```
⊢ ADD_INSTRUCTION n (mem:memstate,pc:word,acc:word) =
    let inst = FETCH mem (Valn n pc) in
    let addr = ADDRESS n inst in
    let operand = FETCH mem (Valn n addr) in
    let nextpc = INCn n pc in
    let nextacc = ADDn (n+3) (acc,operand) in
    (mem,nextpc,nextacc,Lo)
```

ADDRESS is a function which extracts the n-bit address field from the instruction word and FETCH is a function constant representing a memory fetch. When an ADD instruction is executed, the program counter is incremented by one and a value from memory is added to the accumulator. The above definition also specifies that the memory state is left unchanged by an ADD instruction. The fourth element of the 4-tuple indicates that the microprocessor remains in run

mode after executing an ADD instruction. The semantics of the other seven instructions are defined in a similar manner.

8. CLOSING THE STRUCTURE-BEHAVIOUR GAP

The behaviour of the target machine has been formally derived from the structural specification of the Tamarack microprocessor using inference rules of higher-order logic. The behaviour of the internal architecture is derived entirely from the switch level model presented in Section 5 while external devices such as the memory unit are specified in terms of behavioural models. The complete proof required over three million primitive inference steps but almost all of these steps were automatically generated by the HOL system.

Section 3 outlined the structured methodology used to control the complexity of the verification task. One of the main simplifications was illustrated with the behavioural specification of the ALU in terms of arithmetic functions. Abstraction functions such as `BoolAbs`, `Wordn` and `Valn` are used to relate low level forms of data to higher algebraic levels. Another powerful abstraction technique involves formal relationships between various levels of timing in the microprocessor.

Three distinct levels of timing are modelled in the formal specification of the Tamarack microprocessor. The lowest level of timing corresponds to the generation of two-phase, non-overlapping clock signals. At this level, the sequential behaviour of the clocked registers is derived from the behavioural specification of the externally-generated clock signals and the model of charge storage provided by the capacitor primitive. The next level of timing is used to describe register-transfer level behaviour. A single interval of time on the register-transfer level time scale corresponds to a complete two phase clock cycle. The most abstract level of timing in the Tamarack specification is the time scale used to define the behaviour of the target machine. A single interval on this time scale corresponds to one target level operation such as the execution of an instruction.

Both [17] and [27] describe the use of higher-order functions to define formal relationships between time scales. Once a relationship between two time scales has been formally defined, behaviour at the fine-grained time scale is related to behaviour at the coarser grain of time by symbolic simulation of the low-level behaviour through an interval of course-grained time. This proof technique is used to relate behaviour at the level of two-phase clocking to behaviour defined in terms of the register-transfer level time scale. Symbolic simulation is also used to relate register-transfer level behaviour to the behaviour of the target level machine. For example, the target level machine is shown to correctly implement the semantics of an ADD instruction by symbolic simulation of the ten microinstructions im-

plementing an ADD instruction. The simulation takes the form of a sequence of forward inferences in higher-order logic. It is symbolic becauses variables are used in place of real data; alternatively case analysis is used to consider a finite number of possibilities, *e.g.* eight possible values for the 3-bit opcode in a target level instruction. Further details on the symbolic simulation of the Tamarack microcode using higher-order logic may be found in [20]. The use of multiple specification levels and symbolic simulation within higher-order logic is also described in [7].

9. PRACTICAL APPLICATIONS

An important aspect of the work described in this paper is the practical application of formal methods in a VLSI design methodology. By specifying the structure of hardware in terms of switch level primitives, we hope to eventually write formal specifications which can be mechanically checked for consistency with the mask level layout used to fabricate the design.

The use of intermediate levels of specification to structure the verification procedure can also be exploited in a VLSI design methodology. Each level of specification defines a virtual machine whose behaviour can be investigated by simulation. [5] describes how a simulation can be generated from a hardware specification written in higher-order logic. This would allow a specification to be exercised before attempting to relate it to a higher level using formal proof. Formal specifications could also be used to generate to circuit descriptions for special-purpose simulation tools through language interfaces. In this regard, higher-order logic would be used as a universal hardware description language.

The type-checking facilities of the HOL system provide valuable assistance in checking the consistency of a hardware specification even before attempting formal proof. Simply getting the HOL system to successfully type-check a non-trivial specification usually requires considerable effort. However, this effort often reveals errors which would go unnoticed without the scrutinizing eye of the type-checker. Thus, type-checking is a third way, prior to simulation and formal verification, of validating a hardware specification.

Experience with the Tamarack microprocessor suggests that writing formal specifications may also play an active role in the design process. Formal specification motivates cleanly derived views of behaviours which may influence the design. [21] describes how untidy aspects of the behavioural specification for an earlier version of the Tamarack implementation led to improvements in the current design.

10. FUTURE WORK

The current design of the Tamarack microprocessor assumes that every read and write request to external memory is completed in a single clock cycle. This assumption could be validated by a detailed analysis of timing characteristics for both the microprocessor and the external memory. We are now designing a new version of this microprocessor which has wait states where the microprocessor waits one or more clock cycles after issuing a read or write request to the external memory until the request is acknowledged. Wait states are implemented by while-loops in the microcode. [23] describes the use of higher-order logic to model the asynchronous interaction of the microprocessor with external memory using a four-phase "handshaking" protocol. Our approach differs from [19] in the use of existential quantification to specify the unpredictable length of wait states. Our relational specification style also makes it easy to write independent specifications for the microprocessor and the external memory behaviour.

We have also used higher-order logic to relate the target level behaviour of the Tamarack microprocessor to the formal semantics of a very simple imperative language containing constructs such as assignment statements and while-loops. Both the semantics of this language and a compiler are defined in higher-order logic. We have proved that the semantics of the language corresponds to the execution of compiled programs. Thus, the semantics of the language serves as a high level specification of the microprocessor behaviour. This formal connection between software and hardware also provides a verified hardware model for software verification. We hope to eventually integrate this with work on mechanizing programming logics in higher-order logic described by [15].

11. SUMMARY AND CONCLUSIONS

Hardware verification is a new science but it has reached a maturity suitable for practical use in some safety-critical applications. Reliability is more important than functionality in many of these applications. For instance, in regard to railroad control systems, [36] states that "the most complex interlock arrangement requires a simple combination of Boolean and time-based sequential-state logic, all well within the capability of the arithmetic unit of a small microprocessor". The endorsement of these methods by industry is reflected in the product release announcements supplied by the manufacturers of the VIPER microprocessor [10,25].

The cost of formal verification is dominated by the expert's time rather than the speed of the theorem-proving system. However, the main overhead of producing the initial specification and writing the ML programs to do the necessary inferences need only be done once. Thereafter, the finished proof could be easily modified

to accommodate small changes in the hardware. The correctness of incremental changes to a design can be verified by editing the specification and verification procedure, usually in a very minor way, and re-running the proof as a batch job. This contrasts with the simulation approach which may require less initial effort but every modification requires the complete re-run and debugging of the entire simulation test suite. The validity of the simulation approach is seriously impaired when time constraints do not permit complete re-simulation after last-minute modifications. Therefore, the initial effort to produce a formal verification may be rewarded in the long term.

The end product of formal verification is a theorem stating that a behavioural specification is a logical consequence of a structural specification and assumptions about the behaviour of a small set of primitive components which can be implemented as physical devices on a microchip. While formal verification offers greater certainty than simulation, the mere generation of such a theorem is not completely satisfactory evidence that a hardware design is correct. First, the validity of the theorem depends on the behavioural models used for primitive components and external devices. In Section 5 we suggested that the switch level primitives are a reasonable model of CMOS behaviour for the restricted circuit style and clocking scheme used in the Tamarack implementation but this should be verified by a more accurate model of CMOS behaviour. Second, the behavioural specification may be misread or its full implications may not be fully understood. Complete solutions for this second problem are more difficult to propose. However, the problem is partially solved by relating behaviour to increasingly abstract levels such as the semantics of a programming language.

A skeptical view of formal verification suggest that this approach is impractical because the process cannot be completely automated. In our view, complete automation would undermine the real value of formal verification. As part of a VLSI design methodology, the actual process of verifying a design may be of more valuable than the end product. Constructing a formal proof forces a designer to consider exactly why a design is correct. As suggested in [21], an untidy step in the proof may indicate the need for a design modification. Furthermore, the designer is more likely to understand the full implications of a behavioural specification after the effort of producing the proof. Powerful tools such as the HOL system automate most of the tedious steps in a large proof. This allows a designer to reason efficiently about a hardware specification. The underlying formalism ensures that this reasoning process is absolutely correct.

ACKNOWLEDGMENTS

I am grateful for the fertile environment provided by members of the Hardware Verification Group at Cambridge; the work described in this paper has built upon the research efforts of others, in particular, work on transistors models by Mike Gordon and Inder Dhingra and on abstraction mechanisms by Tom Melham. This work continues research on a project which I began while a student at the University of Calgary where I received support from Professor Birtwistle and his research group. I would also like to thank Xerox PARC for considerable support which led to the physical realization of the microprocessor as a CMOS microchip. Helpful comments on many aspects of this paper from Avra Cohn, Mike Gordon, Miriam Leeser and Lorie Joyce were much appreciated.

This research was funded by the Cambridge Commonwealth Trust, the Canada Centennial Scholarship Fund, the Government of Alberta Heritage Fund, the Natural Sciences and Engineering Research Council of Canada, Pembroke College and the UK Overseas Research Student Awards Scheme.

References

[1] Barbacci, M., Instruction Set Processor Specification (ISPS): The Notation and its Application, *IEEE Trans. on Computers* C-30 (1) (1981) pp. 24-40.

[2] Barrow, H., VERIFY: A Program for Proving Correctness of Digital Hardware Designs, *Artificial Intelligence* 24 (1984) pp. 437-491.

[3] Bowen, J. The Formal Specification of a Microprocessor Instruction Set, Tech. Monograph PRG-60, Computing Lab., Oxford Univ., 1987.

[4] Bryant, R., A Switch-Level Simulation Model for Integrated Logic Circuits, Ph.D. Thesis, Dept. of Electrical Engineering and Computer Science, Tech. Report MIT/LCS/TR-259, Massachusetts Inst. of Technology, 1981.

[5] Camilleri, A., Executing Behavioural Definitions in Higher Order Logic, Ph.D. Thesis, Computer Lab., Cambridge Univ., 1988.

[6] Clocksin, W., Logic Programming and Digital Circuit Analysis, *The Journal of Logic Programming* 4 (1987) pp. 59-82.

[7] Cohn, A. 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 Tech. Report No. 104, Computer Lab., Cambridge Univ., 1987.

[8] Cullyer, W. Implementing Safety-Critical Systems: The VIPER Microprocessor, in: G. Birtwistle and P. Subrahmanyam, eds., *VLSI Specification, Verification and Synthesis* (Kluwer Academic Publishers, Boston, 1988) pp. 1-25.

[9] Dhingra, I., Formal Validation of an Integrated Circuit Design Style, in: G. Birtwistle and P. Subrahmanyam, eds., *VLSI Specification, Verification and Synthesis* (Kluwer Academic Publishers, Boston, 1988) pp. 293-321. Also Tech. Report No. 115, Computer Lab., Cambridge Univ., 1987.

[10] Ferranti Semiconductors, VIP 1, Produce Release Announcement, VIP 19/86, Ferranti Electronics Limited, Oldham, England, 1986.

[11] Gordon, M., R. Milner and C. Wadsworth. *Edinburgh LCF: An Mechanised Logic of Computation* (Springer-Verlag, Berlin, 1979).

[12] Gordon, M., Proving a Computer Correct using the LCF_LSM Hardware Verification System, Tech. Report No. 42, Computer Lab., Cambridge Univ., 1983.

[13] Gordon, M., Why Higher-Order Logic is a Good Formalism for Specifying and Verifying Hardware, in: G.J. Milne and P. Subrahmanyam, eds., *Formal Aspects of VLSI Design* (North-Holland, Amsterdam, 1986) pp. 153-177. Also Tech. Report No. 77, Computer Lab., Cambridge Univ., 1985.

[14] Gordon, M., A Proof Generating System for Higher-Order Logic, in: G. Birtwistle and P. Subrahmanyam, eds., *VLSI Specification, Verification and Synthesis* (Kluwer Academic Publishers, Boston, 1988) pp. 73-128. Also Tech. Report No. 103, Computer Lab., Cambridge Univ., 1987.

[15] Gordon, M., Mechanizing Programming Logics in Higher Order Logic, to be published in: G. Birtwistle, ed., Proceedings of the Banff Hardware Verification Workshop, Banff, Canada, June 12-18, 1988. Also Tech. Report No. 145, Computer Lab., Cambridge Univ., 1988.

[16] Henderson, P., *Functional Programming*, (Prentice-Hall, Englewood Cliffs, 1980).

[17] Herbert, J., Application of Formal Methods to Digital System Design, Ph.D. Thesis, Computer Lab., Cambridge Univ., 1986.

[18] Hobson, F., High-Level Microprogramming Support Embedded in Silicon, *IEE Procs.* 135 (E-2) (1981) pp. 73-81.

[19] Hunt, W.A., FM8501: A Verified Microprocessor, Ph.D. Thesis, Inst. for Computer Science, Univ. of Texas at Austin, 1986.

[20] Joyce, J., G. Birtwistle and M. Gordon, Proving a Computer Correct in Higher Order Logic, Tech. Report No. 100, Computer Lab., Cambridge Univ., 1986.

[21] Joyce, J., Formal Verification and Implementation of a Microprocessor, in: G. Birtwistle and P. Subrahmanyam, eds., *VLSI Specification, Verification and Synthesis* (Kluwer Academic Publishers, Boston, 1988) pp. 129-157.

[22] Joyce, J., Generic Structures in the Formal Specification and Verification of Digital Circuits, in: G. Milne, ed., *IFIP WG 10.2 International Workshop on Design for Behavioural Verification* (North-Holland, Amsterdam, 1988).

[23] Joyce, J., Formal Specification and Verification of Asynchronous Processes in Higher-Order Logic, Workshop on Specification and Verification of Concurrent Processes, Univ. of Stirling, Scotland, July 6 - 8, 1988. Also Tech. Report No. 136, Computer Lab., Cambridge Univ., 1988.

[24] Leeser, M. Reasoning about the Function and Timing of Integrated Circuits with Prolog and Temporal Logic, Ph.D. Thesis, Computer Lab., Cambridge Univ., 1987.

[25] Marconi Electronic Devices, VIPER 32 Bit Microprocessor, Publication No. C102 (1), Lincoln, England, 1986.

[26] May, D. and D. Shepherd, Formal Verification of the IMS T800 Microprocessor, Internal Report, INMOS Limited, 1987.

[27] Melham, T., Abstraction Mechanisms for Hardware Verification, in: G. Birtwistle and P. Subrahmanyam, eds., *VLSI Specification, Verification and Synthesis* (Kluwer Academic Publishers, Boston, 1988) pp. 267-291. Also Tech. Report No. 106, Computer Lab., Cambridge Univ., 1987.

[28] Milne, G., Behavioural Description and VLSI Verification, *IEE Procs.* 133 (E-3) (1986) pp. 127-137.

[29] Moszkowski, B., Reasoning about Digital Circuits, Ph.D. Thesis, Dept. of Computer Science, Stanford Univ., 1983.

[30] Moszkowski, B., *Executing Temporal Logic Programs*, (Cambridge University Press, Cambridge, England, 1986).

[31] Nagel, L., SPICE2: A Computer Program to Simulate Semiconductor Circuits, Memo ERL-M520, Univ. of California, Berkeley, 1975.

[32] Paulson, L., *Logic and Computation*, (Cambridge Univ. Press, Cambridge, England, 1987).

[33] Richards, M., BSPL: A Language for Describing the Behaviour of Synchronous Hardware, Tech. Report No. 84, Computer Lab., Cambridge Univ., 1986.

[34] Sheeran, M., Design and Verification of Regular Synchronous Circuits, *IEE Procs.* 133, (E-5) (1986) pp. 295-304.

[35] Sussman G., J. Holloway, G. Steel and A. Bell, Scheme-79: Lisp on a Chip, *IEEE Computer* 14 (7) (1981) pp. 10-21.

[36] Turner, D., R. Burns and H. Hecht, Designing Micro-Based Systems for Fail-Safe Travel, *IEEE Spectrum* 24 (2) (1987) pp. 58-63.

[37] Weise, D., Formal Multilevel Hierarchical Verification of Synchronous MOS VLSI Circuits, Ph.D. Thesis, Massachusetts Inst. of Technology, 1986.

[38] Winskel, G., Models and Logic of MOS Circuits, in: G. Birtwistle and P. Subrahmanyam, eds., *VLSI Specification, Verification and Synthesis* (Kluwer Academic Publishers, Boston, 1988) pp. 323-347. Also Tech. Report No. 105, Computer Lab., Cambridge Univ., 1987.