Technical Report

UCAM-CL-TR-150
ISSN 1476-2986

Number 150

UNIVERSITY OF CAMBRIDGE
Computer Laboratory

Simulation as an aid to verification using the HOL theorem prover

Albert John Camilleri

October 1988

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

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

ISSN 1476-2986

October 4, 1988

Simulation as an aid to Verification using the HOL Theorem Prover[†]

Albert John Camilleri

University of Cambridge[‡]
Computer Laboratory
New Museums Site
Pembroke Street
Cambridge CB2 3QG
England

Abstract
The HOL theorem proving system, developed by Mike Gordon at the University of Cambridge, is a mechanisation of higher order logic, primarily intended for conducting formal proofs of digital system designs. In this paper we show that hardware specifications written in the HOL logic can be executed to enable simulation as a means for supporting formal proof. Specifications of a small microprocessor are described, showing how HOL logic sentences can be transformed into executable code with minimum risk of introducing inconsistencies. A clean and effective optimisation strategy is recommended to make the executable specifications practical.

[†]To appear in the proceedings of the IFIP TC-10 International Working Conference on "Design Methodologies for VLSI and Computer Architecture", Pisa, 19–21 September 1988 (North-Holland), edited by D. Edwards.

[‡]The author's new address is: Hewlett-Packard Limited, Information Systems Centre, Filton Road, Stoke Gifford, Bristol BS12 6QZ, England.

Contents

List of Figures

1 Introduction

Recent advances in digital design technology have brought on an increasing concern for the reliability of digital systems. Conventional approaches for testing hardware such as simulation and prototyping have become increasingly inadequate with the result that computer scientists have begun to explore more reliable techniques for demonstrating correctness of digital designs.

Recently, formal verification has become the preferred approach—a process which involves the use of mechanical theorem provers to verify that a given system actually behaves in the desired way.

The verification process, however, is complicated and expensive; even proofs of simple circuits can involve thousands of logical and mathematical steps. Often it can be extremely difficult to find correct device specifications, and it is therefore desirable that one sets off to prove a correct specification from the start, rather than repeatedly backtrack from the verification process to modify the original definitions after discovering they were incorrect or inaccurate.

A recent idea has been to amalgamate the techniques of simulation and verification, rather than have the latter replace the former. The result is that behavioural definitions can be simulated until it is reasonably sure that the specification is correct. Furthermore, proving the correctness with respect to these simulated specifications avoids the inadequacies of simulation, where it may not be computationally feasible to demonstrate correctness by exhaustive testing. In other words, simulation here is given a dual purpose: (1) to discover obvious design bugs, and (2) to get specifications correct as early as possible in the verification process. Its purpose is no longer to demonstrate the correctness of the implementation—this is done in the verification stage.

By combining simulation and verification, the process of designing and manufacturing digital circuits becomes more cost-effective. Simulation helps to track down inaccuracies in the specifications and thereby reduces the chances of attempting to verify a design with incorrect specifications, involving a considerable waste of time and effort. The use of simulation also helps in understanding specifications better, thus shedding some light on how the specifications can be proven correct.

In [3] it is shown how the facility for conducting simulation can be added to the HOL theorem proving framework [9] by showing how higher order logic specifications can be automatically translated into a syntactically similar but executable language. In this paper we illustrate some of the concepts presented in [3] by considering the specifications of a small microprocessor, showing how they can be transformed into executable programs and used for simulation. We do not discuss the correctness of the microprocessor here. A description of the verification of the computer at the register-transfer level using higher order logic can be found in [11], whereas the verification of the computer at a lower level of description is given in [13].

The organisation of the paper is as follows. First we give a brief description of the HOL logic, and we discuss the various factors that influence the particular choice of executable language. Next we present a description of a simple microprocessor, followed by a description of its specification and implementation definitions, comparing the non-executable HOL logic specifications with corresponding executable programs. We go on to discuss certain efficiency problems encountered when executing the transformed specifications, and show how they can be cleanly optimised for faster simulations. Finally we describe how the entire translation process from HOL logic specifications to efficient executable programs can be automated, and propose some possible areas for future research.

2 The HOL Logic

The HOL logic is a variety of higher order logic based on Church's Type Theory [4]. In the HOL logic, one uses standard predicate logic notation such as the propositional logic connectives denoting negation ($\neg$), conjunction ($\wedge$), disjunction ($\vee$), implication ($\supset$) and equivalence ($\equiv$) for constructing propositions, and the universal ($\forall$) and existential ($\exists$) quantifiers for binding free variables. Since the logic is higher order, variables are allowed to range over functions and the arguments of functions can themselves be functions. Furthermore, the logic is strongly typed, i.e. every term in the logic must have a type.

The specification of a two-input NOR-gate shown below gives an example of how hardware specifications can be written in the HOL logic.

$$\mathsf{NOR}(i_1, i_2, o) \equiv \forall t{:}time.\,(o\,t = \neg(i_1\,t \vee i_2\,t))$$

The behaviour of the NOR-gate is specified using a higher order predicate NOR which takes three arguments i_1, i_2, and o, themselves functions that map time values (represented by a type *time*) to booleans. The predicate $\mathsf{NOR}(i_1, i_2, o)$ holds if and only if for all time instants t, the value on o at time t is equal to the negation of the disjunction of the values on i_1 and i_2 at time t.

We do not give any further introduction to the HOL logic in this paper; for this see [9]. We move on straightaway to discuss the choice of executable language into which HOL specifications can be translated.

3 Choosing an Executable Language

In general, it is not possible to execute specifications written in the HOL logic and so, for the purposes of simulation, hardware specifications have to be translated into an executable language. Perhaps the most natural choice of language would be one used by a special purpose simulator. The main problem here, however, is that the notations used in the logic and in special purpose simulators are often very different, and this gives rise to the danger of introducing inconsistencies during the translation process. Of course, the greater the difference in notations, the greater the risk of introducing errors and thus obtaining executable definitions which do not model the original specifications. This is undesirable because we require the simulation process to be an aid to formal verification, and so it is important that both the non-executable and the executable specifications model the same design.

Traditionally in HOL, it has been common to model behaviour in a *relational* way because it is relatively easy and natural to express the behaviour derived from structure relationally [2]. With the use of predicates, one merely states boolean conditions which define the intended behaviour of a device and so has the advantage of:

- only stating the conditions describing the features of a device which are of interest, thus forming a *partial specification*, and
- dealing with *bidirectional* devices by merely defining relations between ports without distinguishing inputs from outputs.

Another possibility for executing HOL specifications, therefore, would be to translate HOL relations to an executable relational language, such as PROLOG. William Clocksin has shown that PROLOG can be used to simulate the behaviour of digital circuits by executing relations [5]. Once again, however, HOL relations and PROLOG clauses do not bear a strong syntactic and semantic resemblance. Transformation from HOL to PROLOG would either involve the translation of higher order relations to first order relations, or the restriction of writing HOL relations in first order logic to facilitate translation. In either case this is undesirable: in the former case direct translation will not always be possible, and in the latter case one loses the expressive power of higher order logic when writing specifications. For example, the explicit representation of time and the higher order parameterisation of functions that map from time to values, as shown above, will not be possible.

It is desirable, therefore, that the higher order functionality present in HOL specifications can also be expressed in the executable language chosen. The general purpose programming language ML [6] has this desired property, and in this paper we show that it can be used to interpret HOL specifications in a way that greatly reduces the problems and dangers mentioned so far. The syntax and semantics of both the HOL logic and ML are based on the lambda-calculus, and so the specifications used for modelling hardware in HOL and in ML are very similar.

Since ML is a functional programming language, however, we shall be concerned with executing *functional* specifications, where parameters are passed as inputs and values are calculated and returned as outputs. It is therefore useful to translate HOL relational specifications into HOL functional specifications, as an intermediate form to deriving ML programs. Many concepts of hardware design at the register-transfer level can be modelled in HOL by using both relations and functions, and corresponding ML programs can be written such that they are almost identical to the HOL functional specifications. The transformation from HOL relations to HOL functions is safe because it can be checked by formal proof.

In the following sections we present the specifications of a microprocessor to illustrate the relationship between HOL relations, HOL functions and ML functions, and to demonstrate that the use of a special purpose simulator at this level is not necessary to aid verification. Since the transformation from HOL specifications to ML programs is clear and straightforward, the translation process has been automated [3], thus reducing the possibility of introducing errors in the translations, and making it easier to obtain executable specifications. The choice of ML as a simulation language for HOL is also influenced by the fact that ML happens to be the meta-language of HOL, and so this enables all the simulations and proofs to take place within the same system.

4 Simulating a Computer

The design of the general-purpose computer described in the rest of this paper was invented by Mike Gordon in [7] where it was specified and verified using a formalism based on denotational semantics. Commonly referred to as 'Gordon's computer', it became a classic example in hardware specification and verification due to its appeal as a simple yet sufficiently realistic circuit. It has been specified and verified in LCF-LSM by Mike Gordon [8], in VERIFY by Harry Barrow [1] and in HOL by Jeffrey Joyce [11]. Martin Richards has written specifications of Gordon's computer in BSPL [18] while Daniel Weise has written specifications in a LISP-like language of a modified version of the computer [19]. Gordon's computer became the first formally verified computer to be fabricated when an 8-bit version was implemented as a 5000 transistor CMOS microchip as part of a project conducted by Jeffrey Joyce [12] at Xerox Parc and Calgary.

The computer example described in this paper had previously been specified in HOL without intentions of simulating the definitions. In the rest of this section, we present relational and functional definitions based on the HOL specifications presented in [11]. The ML programs shown can be derived automatically using an algorithm described in [3].

The reason for choosing to show the translations and simulations of an existing example, therefore, is to offer evidence of the extent and generality of the automatic translating techniques. Inventing a new example would have introduced the danger of writing specifications in a style which suits the automatic translation, resulting in a biased idea of how versatile and effective the techniques are. What we venture to show in the rest of this section, therefore, is the similarity between HOL and ML specifications, and the type of specifications that can be transformed for simulation using automated techniques.

4.1 Description of Gordon's Computer

A detailed description of Gordon's computer is given in [11]. A brief outline is presented below, however, to enable a full understanding of the formal specifications and their translations presented in the rest of the paper.

At the register-transfer level, the target computer contains a random access memory which is addressed by 13-bit words each pointing to a 16-bit location, and two registers: a 13-bit program counter and a 16-bit accumulator.

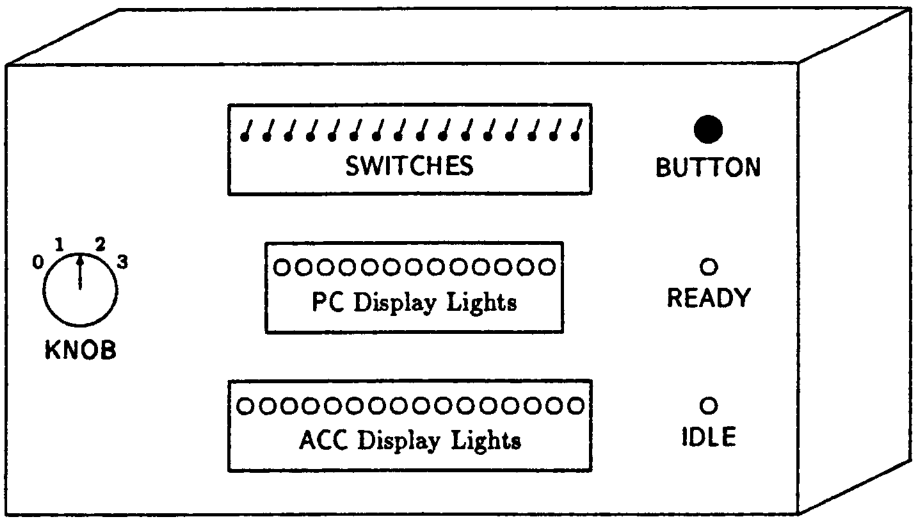

Figure 1: Front Panel of Gordon's Computer

Externally it has four sets of lights used to display output, and a set of buttons and switches which are used for input. Figure 1 is an illustration of the front panel of the computer which shows the four sets of output lights, namely:

- a set of 13 lights to display the contents of the program counter (PC),
- a set of 16 lights to display the contents of the accumulator (ACC),
- a light which goes on to indicate when the computer is idle (IDLE), and
- a light which goes on to indicate the completion of a major state transition (READY),

and the three input mechanisms:

- a set of 16 switches for loading data into the program counter or the accumulator (SWITCHES),

- a knob to select the type of instruction to execute (KNOB), and
- a button used to interrupt the computer during program execution and make it idle, or if the computer is already idle, to execute the instruction selected by the knob (BUTTON).

Each switch or button can be either on or off; these two states are represented by the booleans *true* and *false* respectively. Thus, a sequence of switches or lights in some combination of on and off states is used to model sequences of binary digits.

The types of instructions to be executed are determined by the KNOB being in one of positions 0, 1, 2 or 3, where:

- position 0 = load PC
- position 1 = load ACC
- position 2 = store ACC at PC
- position 3 = start execution at PC.

When the button is pushed and the computer is idle, if the knob is in position 0, the rightmost thirteen bits indicated by the switches are loaded into the program counter. If the knob is in position 1 then all sixteen bits are loaded into the accumulator instead. When the knob is in position 2, no input is read but the current contents of the accumulator are stored in memory at the location indicated by the contents of the program counter. The knob in position 3 starts the execution of a program (loaded in memory) at the location indicated by the contents of the program counter. During the execution of a program, the idle light remains off indicating that the computer is busy; the light going back on when execution of the program is terminated. If the button is pressed during the execution of the program then the program is interrupted and the idle light comes on again.

Programs are written using the following eight microinstructions: HALT (terminates execution), JMP x (jump to address x), JZR x (jump to address x if ACC=0), ADD x (add contents of address to ACC), SUB x (subtract contents of address from ACC), LD x (load contents of address into ACC), ST x (store contents of ACC in memory at address), and SKIP (no operation). Each instruction consists of sixteen bits: the leftmost three denote the opcode, and the rightmost thirteen denote the address. For example, 001 000 000 000 0111 denotes the instruction JMP 7 where 001 is the opcode for JMP and the address field has value 7. Further details of all the instructions are presented in [11].

Before showing the formal HOL specifications that represent the above behaviour, along with their corresponding executable translations, it is necessary to mention the various data types used to write the definitions describing Gordon's computer.

4.2 Data Types Required

Signals in HOL and ML can be represented by functions from time (represented by non-negative integers) to values. Such functions with a domain of type *time* are often referred to as history functions.

Values denoting sequences of bits can be represented in HOL in several ways. A basic way is to represent an n-bit sequence by functions mapping from bit positions to boolean values [2,13]. It is more natural, however, to use specialised data types to represent such bit sequences, and in the HOL specifications presented in this paper, bit sequences are represented as values of type *wordn*, where n is the number of bits represented by the particular type. For example, values stored in the program counter are represented by the type *word*13 and values stored in the accumulator are represented by the type *word*16.

The *wordn* types are defined in HOL as primitive data types and a number of axioms and theorems are defined and proved which describe their properties. A description of the formal axiomatisation of *wordn* types in the HOL logic is described in [15].

The HOL definitions modelling Gordon's computer use the following data types to represent n-bit words: *word*2, *word*3, *word*5, *word*13, *word*16 and *word*30. There is also a *tri_wordn* data type defined for every *wordn* which allows the values on data lines to be floating, and two data types: *mem*5_30 and *mem*13_16 used to represent memory. The data type *memx_y* represents a memory addressed by x-bit words, each pointing to a y-bit word location.

There are several ways for defining data types in ML. One way is to use abstract data type definitions using the declarator abstype. The data type for sixteen bit words, for example,, can be defined as follows:

```
abstype word16 = bool list
with VAL16 w = val 16 (rep_word16 w)
and WORD16 n = abs_word16 (int_to_list 16 n)
and BITS16 = rep_word16
and NOT16 w = abs_word16 (map not (rep_word16 w))
and OR16 v w = abs_word16 (word_or (rep_word16 v) (rep_word16 w))
and AND16 v w = abs_word16 (word_and (rep_word16 v) (rep_word16 w))
```

This type declaration introduces a new type *word*16 represented by the type *bool list* denoting lists of booleans. It makes use of the two locally available functions *abs_word*16 and *rep_word*16 [6] to define a set of primitive functions for manipulating the new data type, namely VAL16, WORD16, BITS16, NOT16, OR16 and AND16. Further explanation of the above definition or the definitions of the other data types is not given in this paper. Detailed descriptions of ML definitions for all the above different data types are presented in [3].

With a systematic formalism as presented in [15] it is promising that an algorithm can be found to automatically translate the HOL data type specifications into ML. Keeping the representations of data types consistent in the two formalisms enables a cleaner and easier translation of specifications.

4.3 The Host Machine

The implementation of Gordon's computer at the register-transfer level, referred to as the 'host machine', is shown in Figure 2. The implementation is composed of a random access memory, a number of registers, a bus, several bus drivers, a read-only memory, an arithmetic and logic unit, and a decoder. We consider below each of these components, and show how each of them can be modelled.

4.3.1 The Bus Drivers

We begin by the five tri-state bus drivers G0, G1, G2, G3 and G4 used to control the data that goes onto the sixteen bit wide bus. To model these devices, two bus drivers are first defined as primitives, one to model *word*13 input drivers and one to model *word*16 input drivers. These devices, GATE13 and GATE16, convert thirteen and sixteen bit words to the corresponding sixteen bit triwords.

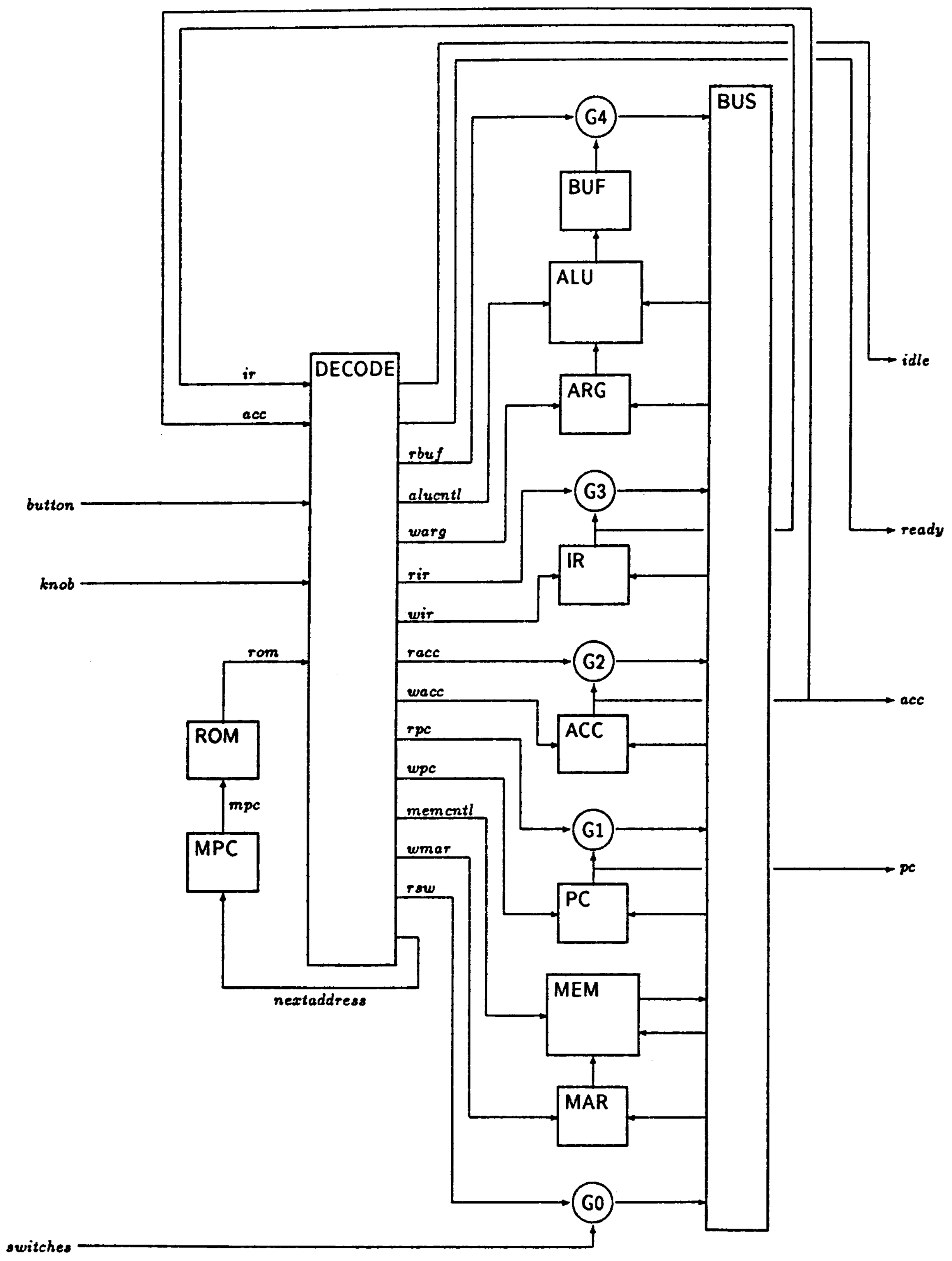

Figure 2: Implementation of Gordon's Computer

The traditional way for writing the two definitions in HOL is to use relations as follows:

$$\mathsf{GATE13}_{rel}(i, cntl, o) \equiv \forall t.\ o(t) = cntl(t) \Rightarrow \mathsf{MK\_TRI16}\ (\mathsf{PAD13\_16}\ i(t))\ |\ \mathsf{FLOAT16}$$

$$\mathsf{GATE16}_{rel}(i, cntl, o) \equiv \forall t.\ o(t) = cntl(t) \Rightarrow \mathsf{MK\_TRI16}\ i(t)\ |\ \mathsf{FLOAT16}$$

where the notation $a \Rightarrow b \mid c$ is an abbreviation for the conditional expression if$a$then$b$else$c$. In the case of GATE13, the *word*13 input is padded up to a 16-bit word using the function PAD13_16. In both cases, the 16-bit word is converted to a triword and output if the value on *cntl* is true. If the value on *cntl* is false, an undefined value FLOAT16 is output. There is no delay modelled in these devices.

The definitions of these bus drivers can alternatively be written as functions in HOL as follows:

$$\mathsf{GATE13}_{fun}\ i\ cntl = \mathsf{let}\ o(t) = (cntl(t) \Rightarrow \mathsf{MK\_TRI16}\ (\mathsf{PAD13\_16}\ i(t))\ |\ \mathsf{FLOAT16})\ \mathsf{in}\ o$$

$$\mathsf{GATE16}_{fun}\ i\ cntl = \mathsf{let}\ o(t) = (cntl(t) \Rightarrow \mathsf{MK\_TRI16}\ i(t)\ |\ \mathsf{FLOAT16})\ \mathsf{in}\ o$$

In these functional models, the style is to use let expressions to show how outputs are evaluated, instead of merely stating relationships between the ports.

The relational and functional definitions are indeed similar, but differences do exist. For example, in the relational models, all the ports to the devices are parameters to the specifications, and no distinction is made between inputs and outputs. With the functional models, however, it is necessary to know which ports of a device are inputs, and which are outputs, because one is interested in evaluating the values on the output ports as functions of the inputs. Only the input lines are needed as parameters.

It is possible to write ML programs to model the bus driver primitives in a way which is almost identical to the functional approach above. As can be seen from the ML programs below, the only syntactical difference to the HOL functions is the presence of the external ML let expression.

$$\mathsf{let}\ \mathsf{GATE13}_{ml}\ i\ cntl = \mathsf{let}\ o(t) = (cntl(t) \Rightarrow \mathsf{MK\_TRI16}\ (\mathsf{PAD13\_16}\ i(t))\ |\ \mathsf{FLOAT16})\ \mathsf{in}\ o$$

$$\mathsf{let}\ \mathsf{GATE16}_{ml}\ i\ cntl = \mathsf{let}\ o(t) = (cntl(t) \Rightarrow \mathsf{MK\_TRI16}\ i(t)\ |\ \mathsf{FLOAT16})\ \mathsf{in}\ o$$

The gates G0, G2, G3 and G4 are defined as instances of GATE16 above while G1 is defined as the only instance of GATE13. The relational definitions and their functional translations are trivial and similar for all the bus drivers.

4.3.2 The Registers

The simplest register in the computer is the register BUF which merely acts as a delay mechanism used to store the output of the ALU for one unit of time. The HOL and ML definitions to model the register are shown below:

$$\mathsf{BUF}_{rel}(alu, buf) \equiv \forall t.\ buf(t+1) = alu(t)$$

$$\mathsf{BUF}_{fun}\ alu\ buf\_val = \mathsf{let}\ buf\ t = (t=0) \Rightarrow buf\_val\ |\ alu(t-1)\ \mathsf{in}\ buf$$

$$\mathsf{let}\ \mathsf{BUF}_{ml}\ alu\ buf\_val = \mathsf{let}\ buf\ t = (t=0) \Rightarrow buf\_val\ |\ alu(t-1)\ \mathsf{in}\ buf$$

Three important points come through from these specifications. The first point is that in the HOL relational model, unit delay is modelled by setting the value of the output at time $t+1$ to some function of the input at time t, the previous time cycle. The model is only partial since the output at time 0 is not defined. Functions in HOL and ML, however, need to be total so functional definitions model delay by setting the value of the output at time t to some function of the input at time $t-1$ instead.

The second point is that since time is represented using the non-negative subset of the integers (the natural numbers), one must ensure that statements of the form $t-1$ do not result in a negative time representation (i.e. when t=0). In the functional models above, a conditional statement is used to check for the special case when t=0; in such a case it assigns an initial value, *buf_val*, to the output at time t=0. One can think of *buf_val* as the value present on the output line *buf* when the register is in its initial state. The initial value *buf_val* is passed as a parameter to the external definition, thus enabling one to simulate the specification with different (or perhaps all) values of *buf_val*.

The final point is that for the functional models, the HOL and ML type checkers infer that t is of type *time* from the subterm t=0, but are unable to determine the types of $buf(t)$, $alu(t-1)$ or *buf_val*. The variables *alu* and *buf* are, therefore, assigned types $time \rightarrow \alpha$ where α is a type variable. The advantage of not stating the types of *alu* and *buf* explicitly in the definitions of BUF is that, by forcing the type checker to use type variables, one is able to define general specifications of a register which can be used for storing values of different types. For example, in the ML model, if i_1 is a variable of type $time \rightarrow bool$ and i_2 is a variable of type $time \rightarrow int$, then ($\mathsf{BUF}_{ml}\, i_1\, false$) denotes a register storing boolean values with an initial state of *false*, while ($\mathsf{BUF}_{ml}\, i_2\, 0$) denotes a register storing integers with an initial state of 0.

The similarity between the HOL functional definitions and the ML programs should by now be obvious. In presenting the specifications of the rest of the computer components, therefore, only the HOL relational definitions and the ML programs will be presented. HOL functions corresponding to the ML programs can in each case be written in an almost identical manner as already demonstrated, but will be omitted to avoid repetition.

To model the remaining registers of the computer two registers are first defined as primitives, one to store thirteen bit words and one to store sixteen bit words. Their HOL definitions are as follows:

$$\mathsf{REG13}_{rel}(i, ld, o) \equiv \forall t.\ o(t{+}1) = (ld(t) \Rightarrow \mathsf{CUT16\_13}(i(t)) \mid o(t))$$

$$\mathsf{REG16}_{rel}(i : num \rightarrow word16, ld, o) \equiv \forall t.\ o(t{+}1) = (ld(t) \Rightarrow i(t) \mid o(t))$$

These two specifications are partial and recursive. In both cases, the *word*16 value on i is sampled at time t. In the case of the thirteen bit register, the least significant thirteen bits are extracted using the function CUT16_13 and output on o at time t+1 if the value on ld at time t is true. In the case of the sixteen bit register, the entire input word is passed on to the output o at time t+1 if the value on ld at time t is true. In both cases, the output at time t+1 is set to its previous value at time t when the value on ld at time t is false.

The translations of the above two relations need to cope with feedback as well as delay. This is done by using the recursive letrec declaration as shown in the definitions below to recursively compute values on the output functions. The technique of parameterising initial values is once again used to convert the partial specifications to total functions.

```
let REG13$_{ml}$ $i$ $ld$ $o\_val$ =
    letrec $o(t) = (t{=}0) \Rightarrow o\_val \mid ld(t-1) \Rightarrow \mathsf{CUT16\_13}(i(t-1)) \mid o(t-1)$ in $o$

let REG16$_{ml}$ $i$ $ld$ $o\_val$ =
    letrec $o(t) = (t{=}0) \Rightarrow o\_val \mid ld(t-1) \Rightarrow i(t-1) \mid o(t-1)$ in $o$
```

The implementation of the computer has two thirteen bit registers: the program counter PC and the memory address register MAR, and three sixteen bit registers: the accumulator ACC, the instruction register IR and a register ARG used for storing arguments to be processed by the arithmetic and logic unit. These registers are defined as instances of the thirteen and sixteen bit registers defined above. The relational definitions and their functional translations are trivial and similar for all the registers.

4.3.3 The Memory

The memory device is modelled by a device MEM which takes three inputs: a two bit wide control signal *memcntl*, the output of the memory address register *mar*, and the value on the bus *bus*. MEM returns one output line *mout*, which writes the fetched contents from memory directly to the bus. In addition, the actual memory representation *mem* is parameterised and is also treated as an output. The relational definition of MEM is shown below.

$$
\begin{array}{l}
\mathsf{MEM}_{rel}(mem, mar, bus, memcntl, mout) \equiv \\
\quad \forall t.\ (mout(t) = \\
\qquad\quad (\mathsf{VAL2}\ memcntl(t)) = 1 \Rightarrow \mathsf{MK\_TRI16}\ (\mathsf{FETCH13}\ mem(t)\ mar(t))\ | \\
\qquad\qquad\qquad\qquad\qquad\qquad \mathsf{FLOAT16}) \wedge \\
\qquad (mem(t+1) = \\
\qquad\quad (\mathsf{VAL2}\ memcntl(t)) = 2 \Rightarrow (\mathsf{STORE13}\ mar(t)\ bus(t)\ mem(t))\ | \\
\qquad\qquad\qquad\qquad\qquad\qquad mem(t))
\end{array}
$$

Once again, the definition is recursive. The contents of the memory are changed only when the value on *memcntl* is equivalent to 2. In this case, the value on the bus is stored in memory at the address specified by the memory address register. In other states, the memory stays the same. When the value on *memcntl* is equivalent to 1, the value stored in memory at the address specified by *mar* is fetched and written to the bus as a triword via *mout*. In other cases, FLOAT16 is written to the bus. The derived program modelling MEM is shown below:

```
let MEM_ml mar bus memcntl mem_val =
  letrec mem(t) =
    (t=0) ⇒ mem_val |
    (VAL2 memcntl(t−1)) = 2 ⇒ (STORE13 mar(t−1) bus(t−1) mem(t−1)) |
    mem(t−1)
  and mout(t) =
    (VAL2 memcntl(t)) = 1 ⇒ MK_TRI16 (FETCH13 mem(t) mar(t)) | FLOAT16
  in (mem, mout)
```

4.3.4 The Arithmetic and Logic Unit

The arithmetic and logic unit is also defined as a primitive at this level of description.[1] Three arithmetic functions are performed by the ALU on sixteen bit words, namely incrementation, addition, and subtraction. The two bit input control line *alucntl* determines which function is performed on the data present on lines *arg* and *bus*. The result of an ALU computation is output on line *alu* at the same instant of time. The relational definition is shown below:

$$
\begin{array}{l}
\mathsf{ALU}_{rel}(arg, bus, alucntl, alu) \equiv \\
\quad \forall t.\ alu(t) = (\mathsf{VAL2}\ alucntl(t)) = 0 \Rightarrow bus(t)\ | \\
\qquad\qquad\qquad (\mathsf{VAL2}\ alucntl(t)) = 1 \Rightarrow \mathsf{INC16}\ bus(t)\ | \\
\qquad\qquad\qquad (\mathsf{VAL2}\ alucntl(t)) = 2 \Rightarrow \mathsf{ADD16}\ arg(t)\ bus(t)\ | \\
\qquad\qquad\qquad\qquad\qquad\qquad \mathsf{SUB16}\ arg(t)\ bus(t)
\end{array}
$$

The specification is total and not recursive. The functional translation is therefore straightforward.

[1]A switch-level description of the arithmetic and logic unit can be found in [13].

$$
\begin{array}{l}
\text{let } \mathsf{ALU}_{ml}\ arg\ bus\ alucntl = \\
\quad \text{let } alu(t) = (\mathsf{VAL2}\ alucntl(t)) = 0 \Rightarrow bus(t) \mid \\
\qquad\qquad\qquad (\mathsf{VAL2}\ alucntl(t)) = 1 \Rightarrow \mathsf{INC16}\ bus(t) \mid \\
\qquad\qquad\qquad (\mathsf{VAL2}\ alucntl(t)) = 2 \Rightarrow \mathsf{ADD16}\ arg(t)\ bus(t) \mid \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathsf{SUB16}\ arg(t)\ bus(t) \\
\quad \text{in } alu
\end{array}
$$

4.3.5 The Bus

All the devices described so far, except BUF, either read from or write to the BUS. The five selectively loadable registers and the ALU read sixteen bit words from the BUS, the five tri-state bus drivers write sixteen bit triwords to the BUS, and the memory both reads from and writes to the BUS. Hence, the BUS device takes six triwords of type *triword*16 as inputs and returns a word of type *word*16 as output.

Relationally, BUS is modelled as:

$$
\begin{array}{l}
\mathsf{BUS}_{rel}(mout, g_0, g_1, g_2, g_3, g_4, bus) \equiv \\
\quad \forall t.\ bus(t) = \mathsf{DEST\_TRI16}\ (mout(t)\ \mathsf{U16}\ g_0(t)\ \mathsf{U16}\ g_1(t)\ \mathsf{U16}\ g_2(t)\ \mathsf{U16}\ g_3(t)\ \mathsf{U16}\ g_4(t))
\end{array}
$$

where U16 is the infix function for merging two triwords, and DEST_TRI16 is the function which converts sixteen bit triwords to sixteen bit words. The derived program is, once again, almost identical:

$$
\begin{array}{l}
\text{let } \mathsf{BUS}_{ml}\ mout\ g_0\ g_1\ g_2\ g_3\ g_4\ = \\
\quad \text{let } bus(t) = \mathsf{DEST\_TRI16}\ (mout(t)\ \mathsf{U16}\ g_0(t)\ \mathsf{U16}\ g_1(t)\ \mathsf{U16}\ g_2(t)\ \mathsf{U16}\ g_3(t)\ \mathsf{U16}\ g_4(t)) \\
\quad \text{in } bus
\end{array}
$$

4.3.6 The Data Path

The specifications we have presented so far are models of the behaviour of the devices that specify the data path of the computer implementation. This data path, therefore, can be specified by structuring all the devices described so far as shown in Figure 2. The relational definition which models the data path is shown below, together with its functional translation.

$$
\begin{array}{l}
\mathsf{DATA}_{rel}(switches, rsw, wmar, memcntl, wpc, rpc, wacc, racc, wir, \\
\qquad rir, warg, alucntl, rbuf, mem, mar, pc, acc, ir, arg, buf) \equiv \\
\quad \exists\ g_0\ g_1\ g_2\ g_3\ g_4\ mout\ alu\ bus. \\
\qquad \mathsf{MEM}_{rel}(mem, mar, bus, memcntl, mout) \wedge \\
\qquad \mathsf{MAR}_{rel}(bus, wmar, mar) \wedge \\
\qquad \mathsf{PC}_{rel}(bus, wpc, pc) \wedge \\
\qquad \mathsf{ACC}_{rel}(bus, wacc, acc) \wedge \\
\qquad \mathsf{IR}_{rel}(bus, wir, ir) \wedge \\
\qquad \mathsf{ARG}_{rel}(bus, warg, arg) \wedge \\
\qquad \mathsf{BUF}_{rel}(alu, buf) \wedge \\
\qquad \mathsf{G0}_{rel}(switches, rsw, g_0) \wedge \\
\qquad \mathsf{G1}_{rel}(pc, rpc, g_1) \wedge \\
\qquad \mathsf{G2}_{rel}(acc, racc, g_2) \wedge \\
\qquad \mathsf{G3}_{rel}(ir, rir, g_3) \wedge \\
\qquad \mathsf{G4}_{rel}(buf, rbuf, g_4) \wedge \\
\qquad \mathsf{ALU}_{rel}(arg, bus, alucntl, alu) \wedge \\
\qquad \mathsf{BUS}_{rel}(mout, g_0, g_1, g_2, g_3, g_4, bus)
\end{array}
$$

```
let DATA_ml switches rsw wmar memcntl wpc
            rpc wacc racc wir rir warg alucntl rbuf
            memval marval pcval accval irval argval bufval =
  let g_0 = (G0_ml switches rsw) in
  letrec mem(t) = (fst (MEM_ml mar bus memcntl memval)) t
  and mout(t) = (snd (MEM_ml mar bus memcntl memval)) t
  and mar(t) = (MAR_ml marval bus wmar) t
  and pc(t) = (PC_ml pcval bus wpc) t
  and acc(t) = (ACC_ml accval bus wacc) t
  and ir(t) = (IR_ml irval bus wir) t
  and arg(t) = (ARG_ml argval bus warg) t
  and buf(t) = (BUF_ml bufval alu) t
  and g_1(t) = (G1_ml pc rpc) t
  and g_2(t) = (G2_ml acc racc) t
  and g_3(t) = (G3_ml ir rir) t
  and g_4(t) = (G4_ml buf rbuf) t
  and alu(t) = (ALU_ml arg bus alucntl) t
  and bus(t) = (BUS_ml mout g_0 g_1 g_2 g_3 g_4) t
  in (mem, mar, pc, acc, ir, arg, buf)
```

In the relational model, composition of devices is represented by conjunction. In the functional definition this is represented by and and in constructs of let expressions. The and construct in a let declaration is used to model a parallel connection of devices, and the in construct is used to model a serial connection.

There is also a contrast in the way hiding is modelled in the definitions above. In the relational definition, existential quantification is used to quantify over variables representing hidden lines, whereas local declarations are used to the same effect in the functional definition.

4.3.7 The Control Unit

The three devices in the implementation of Gordon's computer shown in Figure 2 which have not yet been specified are MPC, ROM and DECODE.

The microcode program counter MPC is another non selectively-loadable register and is defined in almost the same way as BUF. The only difference is that the inputs and outputs to MPC are represented by functions of type *time→word5*. The register is used to store a 5-bit address for the read-only memory.

The read-only memory ROM outputs the 30-bit word stored in the microcode at the address specified by the MPC. The microcode is addressed by 5-bit words, each pointing to a 30-bit instruction. The HOL representation of the microcode is generated by a function as a set of axioms which relate the contents of the memory to their corresponding address [11]. In ML the microcode is defined as a function which takes an argument of type *word5* (the address) and returns a value of type *word30* (the contents). The function is parameterised in the definition of ROM which makes use of the primitive function FETCH5 to fetch the contents from the microcode. The definition of ROM and its translation are as follows:

$$\mathsf{ROM}_{rel}\ mcode\ (mpc, rom) \equiv \forall t.\ rom(t) = \mathsf{FETCH5}\ mcode\ mpc(t)$$

$$\mathsf{let\ ROM}_{ml}\ mcode\ mpc = \mathsf{let}\ rom(t) = \mathsf{FETCH5}\ mcode\ mpc(t)\ \mathsf{in}\ rom$$

The decode unit DECODE reads in the instruction from ROM and decodes it into the relevant signals which control the operation of the data path DATA. It also computes the next address to index the ROM by examining the values on *knob*, *button*, *acc* and *ir*.

The specification of DECODE and its translation are rather long and are therefore omitted here. As can be seen from [11], though, the definition is purely combinational and so its translation is straightforward. In the rest of this section, the relational and functional specifications will be referred to as DECODE_{rel} and DECODE_{ml} respectively.

The three devices MPC, ROM and DECODE can now be grouped together to form a top-level component of the computer implementation. This component is called the control unit and is relationally specified using the predicate $\mathsf{CONTROL}_{rel}$ below:

$$
\begin{array}{l}
\mathsf{CONTROL}_{rel} \\
\quad mcode \\
\quad (knob, button, acc, ir, rsw, wmar, memcntl, wpc, rpc, \\
\quad\ \ wacc, racc, wir, rir, warg, alucntl, rbuf, ready, idle) \equiv \\
\ \ \exists\, mpc\; rom\; nextaddress. \\
\quad \mathsf{ROM}_{rel}\; mcode\; (mpc, rom) \wedge \mathsf{MPC}_{rel}\; (nextaddress, mpc) \wedge \\
\quad \mathsf{DECODE}_{rel}(rom, knob, button, acc, ir, nextaddress, rsw, wmar, memcntl, \\
\qquad\qquad wpc, rpc, wacc, racc, wir, rir, warg, alucntl, rbuf, ready, idle)
\end{array}
$$

4.3.8 The Host Specification

The overall specification of the host machine is obtained by joining together the control unit and the data path. The usual techniques for modelling structure are used, namely conjunction and existential quantification. The relational specification of the computer implementation is defined using the predicate HOST_{rel}.

$$
\begin{array}{l}
\mathsf{HOST}_{rel}(knob, button, switches, mem, pc, acc, ready, idle) \equiv \\
\ \ \exists\, ir\; rsw\; wmar\; memcntl\; wpc\; rpc\; wacc\; racc \\
\quad wir\; rir\; warg\; alucntl\; rbuf\; mar\; arg\; buf. \\
\quad \mathsf{CONTROL}_{rel} \\
\quad\ \ microcode \\
\quad\ \ (knob, button, acc, ir, rsw, wmar, memcntl, wpc, rpc, wacc, racc, \\
\quad\ \ \ \ wir, rir, warg, alucntl, rbuf, ready, idle) \wedge \\
\quad \mathsf{DATA}_{rel}(switches, rsw, wmar, memcntl, wpc, rpc, wacc, racc, wir, \\
\qquad\qquad rir, warg, alucntl, rbuf, mem, mar, pc, acc, ir, arg, buf)
\end{array}
$$

The value *microcode* in the definition of HOST_{rel} is a constant which represents the microcode.

The translations of $\mathsf{CONTROL}_{rel}$ and HOST_{rel} are straightforward and involve the same techniques used to translate models which describe the structure of other devices, such as the definition of DATA already presented. The programs $\mathsf{CONTROL}_{ml}$ and HOST_{ml} are therefore omitted here.

4.4 The Target Machine

The behaviour of Gordon's computer described in Subsection 4.1 is formalised by the definition of the predicate $\mathsf{COMPUTER}_{rel}$ shown below. The parameters of the specification denote the memory, the input ports, and the output ports of the computer; all represented by history functions of the appropriate type. The definition recursively equates the values at time $t{+}1$ of the memory, the program counter, the accumulator, and the idle light with an expression involving the values at time t of *knob*, *button*, *switches*, *mem*, *pc*, *acc*, and *idle*.

This definition is a straightforward model of the behaviour of the computer. It uses a conditional statement to define the different actions of the computer for the different situations determined by the values of *idle*, *button* and *knob*. Each terminal branch of the conditional represents a single target machine operation.

```
∀ knob button switches mem pc acc idle.
  COMPUTER_rel(knob, button, switches, mem, pc, acc, idle) ≡
    (mem(t+1), pc(t+1), acc(t+1), idle(t+1)) =
      (idle(t) ⇒
        (button(t) ⇒
          ((VAL2 (knob(t)) = 0) ⇒
            (mem(t), CUT16_13(switches(t)), acc(t), T) |
           (VAL2 (knob(t)) = 1) ⇒
            (mem(t), pc(t), switches(t), T) |
           (VAL2 (knob(t)) = 2) ⇒
            (STORE13 pc(t) acc(t) mem(t)), pc(t), acc(t), T) |
            (mem(t), pc(t), acc(t), F)) |
          (mem(t), pc(t), acc(t), T)) |
        (button(t) ⇒ (mem(t), pc(t), acc(t), T) |
                     EXECUTE(mem(t), pc(t), acc(t))))
```

The function VAL2 returns the integer value of a two-bit word and is used to check whether the value on *knob* at time *t* is set to 0, 1, 2 or 3. The function CUT16_13 returns a thirteen bit word consisting of the thirteen least significant bits of a sixteen bit word. It is used to load the thirteen rightmost bits set up on the switches into the program counter when the knob is set to zero.

When the value of *idle* at time *t* is F (i.e. the computer is executing a program) and the value of *button* is F (i.e. the computer is not interrupted) then the next values of *mem*, *pc*, *acc* and *idle* are determined by a function EXECUTE which describes the execution of a single target level instruction.

The definition of EXECUTE is shown below, where the function FETCH13 is used to fetch the contents of the memory stored at the location specified by the 13-bit program counter. The opcode *op* and the address *addr* are extracted from the 16-bit word fetched from memory and are used to evaluate the next values of memory, program counter, accumulator and idle. An eight branch conditional is used, one branch for every possible value of the opcode. Each conditional evaluates the instruction corresponding to the particular opcode. For example, when the value of *op* is 0 the instruction to be executed is the HALT command, and so the current values in the memory, the program counter, and the accumulator are retained while the value on the *idle* line is changed to T to indicate the computer has finished executing the program.

```
∀ mval pcval accval.
  EXECUTE (mval, pcval, accval) =
  let op = VAL3 (OPCODE (FETCH13 mval pcval)) in
  let addr = CUT16_13 (FETCH13 mval pcval) in
  (op=0) ⇒ (mval, pcval, accval, T) |
  (op=1) ⇒ (mval, addr, accval, F) |
  (op=2) ⇒ ((VAL16 accval)=0 ⇒ (mval, addr, accval, F) |
                                (mval, INC13 pcval, accval, F)) |
  (op=3) ⇒ (mval, INC13 pcval, ADD16 accval (FETCH13 mval addr), F) |
  (op=4) ⇒ (mval, INC13 pcval, SUB16 accval (FETCH13 mval addr), F) |
  (op=5) ⇒ (mval, INC13 pcval, FETCH13 mval addr, F) |
  (op=6) ⇒ (STORE13 addr accval mval, INC13 pcval, accval, F) |
           (mval, INC13 pcval, accval, F))
```

The two definitions above fully model the behaviour of the target machine at the register-transfer level. Since EXECUTE is already defined as a HOL function (rather than a relation), translating it to the corresponding ML function is trivial.

The function $\mathsf{COMPUTER}_{ml}$ recursively computes the output functions which represent the output mechanisms and the random access memory. Although the memory is an internal mechanism, it is included among the outputs of the definition, since this makes it possible for the contents of the memory to be examined when simulating the functional definitions.

5 Optimising Executable Specifications

Before moving on to present an example simulation of the computer, there are certain features in the executable models described in the previous section which must be explained further to show how they can be modified to improve the performance of simulation.

5.1 Inefficiencies in Basic Method

The application of ML to hardware simulation outlined so far seems satisfactory. Many aspects of register-transfer level descriptions of hardware circuits have been modelled naturally using standard functional programming techniques. When attempting to simulate large and complex circuits, however, a major problem is encountered: certain simulations are far too slow. While many circuits can be simulated successfully at a reasonable speed, others are totally impractical.

The problem is due to circuits with models that have the following properties: too many recursive calls on history functions (generally resulting from feedback), and too many repeated computations (generally resulting from fanout). In general, therefore, the problem arises to varying extents in most sequential devices.

The functional translation of DATA_{rel} is a good example for demonstrating this effect of inefficiency due to the many repetitions of recursive calculations it involves. For example, the evaluation of the value on the bus at time t, $bus(t)$, requires $mout(t)$ and $g_0(t), \ldots, g_4(t)$. Now the computations of g_1, g_2, g_3 and g_4 at time t require the computations of $pc(t)$, $acc(t)$, $ir(t)$ and $buf(t)$ respectively, each requiring, among other data, the value for $bus(t-1)$. Furthermore, $mout(t)$ requires $mem(t)$ and $mar(t)$, which in turn also require $bus(t-1)$. Thus, to compute $bus(t)$, $bus(t-1)$ is calculated six times. At time t, therefore, $bus(t-n)$ is calculated a maximum of 6^n times, i.e. $bus(t-10)$ could be calculated well over 60.5 million times! Each time a value is computed at a time t, the computation is recursive all the way down to time 0. For a large t, the effect of recalculating the same functions over the entire time scale is disastrous.

The nature of the problem suggests the solution. For every value required in a computation, either it is evaluated and stored for possible later use, or evaluation is delayed until the last possible moment when the value is evaluated once and assigned to all instances where the value is required in the course of the computation.

Both optimisations are standard techniques, called *memoisation* and *lazy evaluation* respectively. Below we explain how memoisation can be used to optimise naively coded functions in a way that is clean and simple.

5.2 Memoisation

The technique of memoisation was invented by Donald Michie and was used as an aid for improving the performance of programs [16]. Memoisation can make a vast difference in the performance of a program. The idea is to define higher order functions called memo-functions which take an inefficient function as a parameter and return an optimised function.

A memo-function remembers all arguments to which it has been applied as well as the results computed from them. This is done by maintaining a table in which values of previous calls to the function are stored using the arguments that produced the values as keys.

When a memoised function is applied to a set of arguments to compute a value, it first checks the table to see if the function has already been applied to these arguments, and if so, it merely returns the previously computed value stored in the table. If the function has not already been applied to the arguments, however, then the new value is computed in the ordinary way, stored in the table with the function's arguments as keys, and returned as the result of the function.

Definitions of memoisation functions in ML are given in [3], along with representations of tables and definitions of lookup and storing functions. We merely present, below, the optimised function DATA_{ml} to demonstrate the clarity of the memoisation transformation.

5.3 Memoisation of the Data Path Definition

The function DATA_{ml} below is the memoised definition of the data path in which the parameters ending in *val* represent initial values, and the rest of the parameters are history functions representing the various input data lines. The functions $memo_{ty}$ are memoisation functions that store and retrieve values of type *ty*.

```
let DATA_ml switches rsw wmar memcntl wpc
            rpc wacc racc wir rir warg alucntl rbuf
            memval marval pcval accval irval argval bufval =
  let g_0 = (G0_ml switches rsw) in
  letrec mem(t) = memo_mem (fst (MEM_ml mar bus memcntl memval)) t
  and mout(t) = memo_tri16 (snd (MEM_ml mar bus memcntl memval)) t
  and mar(t) = memo_wd13 (MAR_ml marval bus wmar) t
  and pc(t) = memo_wd13 (PC_ml pcval bus wpc) t
  and acc(t) = memo_wd16 (ACC_ml accval bus wacc) t
  and ir(t) = memo_wd16 (IR_ml irval bus wir) t
  and arg(t) = memo_wd16 (ARG_ml argval bus warg) t
  and buf(t) = memo_wd16 (BUF_ml bufval alu) t
  and g_1(t) = memo_tri16 (G1_ml pc rpc) t
  and g_2(t) = memo_tri16 (G2_ml acc racc) t
  and g_3(t) = memo_tri16 (G3_ml ir rir) t
  and g_4(t) = memo_tri16 (G4_ml buf rbuf) t
  and alu(t) = memo_wd16 (ALU_ml arg bus alucntl) t
  and bus(t) = memo_tri16 (BUS_ml mout g_0 g_1 g_2 g_3 g_4) t
  in (mem, mar, pc, acc, ir, arg, buf)
```

The optimisations performed on the naive DATA_{ml} functions are clean and simple. The structure of the naive function can still be seen clearly embedded in the efficient one.

6 Executing Programs on Gordon's Computer

In the previous two sections we showed functional specifications to model the computer at the host level and the target level. The optimised program which models the target machine, $\mathsf{COMPUTER}_{ml}$, takes seven arguments: three history functions modelling the input mechanisms, and four initial values for the memory, the program counter, the accumulator and the idle button. It computes four history functions modelling the values displayed in the memory, the program counter, the accumulator and the idle button.

At the host level, the specification HOST_{ml} takes eleven arguments: three history functions modelling the input mechanisms, and eight initial values needed to initialise the various registers as well as the memory. In addition to the four history outputs modelling the memory, the program counter, the accumulator and the idle light, another history function is also returned to model the ready light. This last output is not included in the target level description of the computer.

These specifications can be used to simulate the execution of programs by the computer. For example, consider the following interaction to add two integers:

> Starting with an empty memory, store an integer a in location 0 of the memory and store an integer b in location 1. From location 3 of the memory onwards, store the instructions which compute $a+b$ and store the result in location 2 of the memory.

Figure 3 illustrates the layout of the contents of the memory: the data in the first two locations, the program starting at location 3 and the result in location 2.

0	a
1	b
2	$a+b$
3	program ↓

Figure 3: Representation of Memory for Program to Add Two Integers

The simulation of the computer running this interaction is performed by setting up the appropriate values on the input mechanisms to load the program and the data into memory. The program is expressed using the microinstructions of the computer, described in Subsection 4.1. Figure 4 uses an assembler style notation to code the sequence of events loading the program and data, and executing the program. In fact, one can define a small assembler language for the computer which would set the values on the input history functions automatically.

In the program of Figure 4, the integers 40960, 24577, 49154 and 0 on line numbers 3, 6, 9 and 12 respectively are the integer representations of the 13-bit instruction codes explained in the adjacent comments.

Both the specifications $\mathsf{COMPUTER}_{ml}$ and HOST_{ml} were used to simulate this interaction running on Gordon's computer. Figure 5 shows a table of the input values for the first 25 time cycles used to execute the target level specification $\mathsf{COMPUTER}_{ml}$. The figure also shows the output values for the corresponding 25 time cycles. The initial values on the program counter and the accumulator are set to zero, the initial value of the idle light is set to T and the memory is cleared using the primitive constant EMPTY13_16. In this example, the values of the numbers a and b added together are set to 54 and 85 respectively.

```
1                          ; Load Program
2   LD  PC   3
3   LD  ACC  40960         ; Instruction code for "Load ACC with a"
4   ST  ACC  PC
;
5   LD  PC   4
6   LD  ACC  24577         ; Instruction code for "Add b to ACC"
7   ST  ACC  PC
;
8   LD  PC   5
9   LD  ACC  49154         ; Instruction code for "Store ACC at Location 2"
10  ST  ACC  PC
;
11  LD  PC   6
12  LD  ACC  0             ; Instruction code for "HALT"
13  ST  ACC  PC
;
14                         ; Load Data
15  LD  PC   0
16  LD  ACC  a
17  ST  ACC  PC
;
18  LD  PC   1
19  LD  ACC  b
20  ST  ACC  PC
;
21                         ; Execute
22  LD  PC   3
23  EX  PC
```

Figure 4: Representation of Program Executed on Gordon's Computer

The table shows the different stages in the execution of the program. The outputs displayed at time t+1 are the results of the instruction executed at time t. For example, at time 0, all outputs are set to their initial values. Then at time 1, the value 3 is loaded in the program counter and at time 2, the number 40960, which codes the instruction for loading the accumulator with the contents of location 0 in memory, is loaded in the accumulator. At time 3 no change is apparent in the tabulated outputs because at this stage the contents of the accumulator are stored in memory at the location specified by the program counter. The loading of the program and the data in memory proceeds until time 19. Execution of the stored program begins at time 20 when the idle light goes off. The result of a+b is shown in the accumulator at time 22, when it is stored in memory. The halt command is executed at time 23 and the idle light goes back on at time 24.

The program HOST_{ml} produces a similar but much longer table of outputs when simulated over the same example. In fact, 96 time cycles are necessary to show the entire output history. This is because each instruction is split into several microinstructions at this level of abstraction and so an instruction executed in one time unit at the target level takes several time units at the host level. The tabulated results of the host machine simulation are not presented here.

The results shown over 24 time units in Figure 5 are computed by the optimised function COMPUTER$_{ml}$ in an overall time of 288 seconds cpu time. The results over 96 time units (including those for the extra line *ready*) computed by the optimised function HOST$_{ml}$ for the same example took 623 seconds cpu time. The simulations were carried out on an 8-megabyte SUN 3 machine running UNIX.

time	*knob*	*button*	*switches*	*pc*	*acc*	*idle*	Comment
0	0	T	3	0	0	T	
1	1	T	40960	3	0	T	
2	2	T	40960	3	40960	T	
3	0	T	4	3	40960	T	
4	1	T	24577	4	40960	T	
5	2	T	24577	4	24577	T	Program
6	0	T	5	4	24577	T	is loaded
7	1	T	49154	5	24577	T	
8	2	T	49154	5	49154	T	
9	0	T	6	5	49154	T	
10	1	T	0	6	49154	T	
11	2	T	0	6	0	T	
12	0	T	0	6	0	T	
13	1	T	54	0	0	T	
14	2	T	54	0	54	T	Data
15	0	T	1	0	54	T	is loaded
16	1	T	85	1	54	T	
17	2	T	85	1	85	T	
18	0	T	3	1	85	T	Go to location 3
19	3	T	3	3	85	T	Execute program
20	3	F	3	3	85	F	
21	3	F	3	4	54	F	
22	3	F	3	5	139	F	
23	3	F	3	6	139	F	
24	3	F	3	6	139	T	Computer is idle

Figure 5: Table Displaying Stages of Simulation

The times are certainly acceptable, especially when compared with the performance of the non-optimised versions of the specifications. Executing the unoptimised version of COMPUTER$_{ml}$ took over two hours of cpu time to terminate, and that of HOST$_{ml}$ was allowed to run for over 24 hours on an ATLAS 10 mainframe computer—after which it was still not finished. The bulk of the inefficiency was traced to the large amount of repetitive recursive calculations involved in the definition of the data path DATA. Memoisation avoids recalculation and transforms highly inefficient functions to relatively efficient ones by enabling recursive functions to remember previously computed values.

7 Automatic Generation of Programs

In the previous sections we have shown that ML can be used as a hardware simulator at the register-transfer level and that it is a good medium for executing specifications written in the HOL logic. Apart from reducing the chance of introducing inconsistencies, the similarities between the two languages also make it possible for the translation from HOL specifications to ML programs to be automated. Such an automatic tool has been designed to translate both HOL relations and HOL functions into ML programs, since hardware specifications in HOL can be written using both styles. Translating from HOL functions to ML programs is easy because the definitions are very similar. HOL relations, however, are a more common way of writing specifications and to facilitate their translation to ML programs, it is convenient to translate the HOL relations into HOL functions first, as a stepping stone to translating to ML programs.

The main syntactic difference between relational and functional specifications explained earlier on is that:

- in a relational model there is no distinction between inputs and outputs—all ports are parameterised, whereas
- in a functional model, only the inputs are parameterised from which the function computes the values on the output ports.

Given such information as to which parameters of the predicate are to be inputs of the function and what form the output should take, the translation from relations to functions can be automated. In [3] an algorithm for automatically translating relations in the HOL logic to ML programs is described in detail. The process of translating HOL relations into ML programs can be summarised using the following diagram:

$$\begin{tikzcd}
\boxed{\begin{array}{c}\text{HOL}\\\text{relations}\end{array}} \arrow[r, "\text{translate}"] & \boxed{\begin{array}{c}\text{HOL}\\\text{functions}\end{array}} \arrow[r, "\text{encode}"] & \boxed{\begin{array}{c}\text{ML}\\\text{programs}\end{array}} \arrow[r, "\text{optimise}"] & \boxed{\begin{array}{c}\text{optimised}\\\text{ML}\\\text{programs}\end{array}}
\end{tikzcd}$$

The first step in the above diagram represents the transformation of HOL relations to HOL functions, the second step in the diagram shows that the HOL functions are converted into a representation of an ML program, and the third step represents the final stage at which the naive ML programs are converted into optimised ones. The third stage of the translation involves the automatic generation of memo-functions and memo-tables whenever new memo-functions and memo-tables are required, and the automatic insertion of memoisation strategies wherever appropriate in the inefficient programs.

8 Conclusions and Future Research

The aim of this paper has been to demonstrate how simulation can be performed within the HOL theorem proving framework in order to help facilitate verification. It is believed that the combination of simulation and verification could be an effective and efficient way to obtain correct hardware designs. The verification process would involve writing specifications and implementation definitions, transforming them into executable definitions, simulating them until they are fully understood, and finally verifying them.

Few mechanical theorem provers combine the two notions of simulation and verification. The trend has been either to develop a special purpose hardware simulator with no infrastructure for conducting formal proof, or to develop a theorem prover which carries out formal proof by manipulating specifications but which does not do simulation. Examples of mechanical systems which can be used to conduct both simulation and verification are the BOYER-MOORE theorem prover [10] and CIRCAL [17].

More complex examples than the one presented in this paper have been successfully simulated using the automatic translation tools mentioned earlier to generate simulation specifications from HOL relations. These include a communications chip with a complexity of about 360 gates. The simulations of these complex examples, conducted by executing the derived optimised programs, performed at an acceptable speed. Without the optimisations, however, the programs were impractical to run. It is possible that the efficiency of the optimisations may be improved further by developing faster techniques for retrieving and storing computed information.

An interesting improvement on the automatic translation mechanisms would be to attempt to perform the translation from HOL functions to HOL relations by automatic formal proof using the HOL inference rules. Presently, the tool is merely an ML program which parses HOL relations and generates HOL functions. Transformation by formal proof would help in further reducing the chances of introducing inconsistencies in the translations.

Another extension to this research could entail the investigation of executing behavioural definitions expressed at lower levels of detail. It could be that the same tools can be successfully applied to translating most of the aspects of hardware specifications at lower levels, with little or no extensions to the algorithms. Certain features like bidirectionality, however, will pose problems, and research in tackling such issues will be valuable.

Acknowledgements

I would like to thank all the members of the Hardware Verification Group at the University of Cambridge for their support and motivating discussions. Special thanks to Mike Gordon for early discussions regarding this paper, and Jeffrey Joyce for later comments. I would also like to thank various members of Hewlett Packard Laboratory: Derek Coleman for encouraging me to write this paper, and Patrick Goldsack, Ajay Gupta and Bob Welham for making several constructive comments. Thanks are also due to the Commonwealth Scholarship Commission in the United Kingdom for funding my research.

References

[1] Barrow H. G., 'VERIFY: A Program for Proving Correctness of Digital Hardware Designs', *Artificial Intelligence*, 1984, Vol. 24, pp. 437–491.

[2] Camilleri A., Gordon M., Melham T., 'Hardware Verification using Higher-Order Logic', in *From H.D.L. Descriptions to Guaranteed Correct Circuit Designs*, Borrione D. (editor), North-Holland, Amsterdam, 1987, pp. 43–67, Proceedings of the IFIP WG 10.2 International Working Conference, Grenoble, France, 9–11 September 1986.

[3] Camilleri A. J., 'Executing Behavioural Definitions in Higher Order Logic', Technical Report No. 140, Ph.D. Thesis, University of Cambridge, Computer Laboratory, July 1988.

[4] Church A., 'A Formulation of the Simple Theory of Types', *The Journal of Symbolic Logic*, 1940, Vol. 5, pp. 56–58.

[5] Clocksin W. F., 'Logic Programming and Digital Circuit Analysis', *The Journal of Logic Programming*, 1987, Vol. 4, pp. 59–82.

[6] Cousineau G., Huet G., Paulson L., 'The ML Handbook', INRIA, 1986.

[7] Gordon M. J. C., 'A Model of Register Transfer Systems with Applications to Microcode and VLSI Correctness', Internal Report CSR-82-81, University of Edinburgh, Department of Computer Science, March 1981.

[8] Gordon M. J. C., 'Proving a Computer Correct', Technical Report No. 42, University of Cambridge, Computer Laboratory.

[9] Gordon M. J. C., 'HOL—A Proof Generating System for Higher-Order Logic', in *VLSI Specification, Verification and Synthesis*, Birtwistle G. and Subrahmanyam P. A. (editors), Kluwer Academic Publishers, Boston, 1988, pp. 73–128, Proceedings of the Hardware Verification Workshop, Calgary, Canada, 12–16 January 1987.

[10] Hunt W. A., Jr., 'FM8501: A Verified Microprocessor', Ph.D. Thesis, Technical Report No. 47, Institute for Computing Science, University of Texas at Austin, December 1985.

[11] Joyce J., Birtwistle G., Gordon M., 'Proving a Computer Correct in Higher Order Logic', Technical Report No. 100, University of Cambridge, Computer Laboratory, December 1986.

[12] Joyce J., 'Formal Verification and Implementation of a Microprocessor', in *VLSI Specification, Verification and Synthesis*, Birtwistle G. and Subrahmanyam P. A. (editors), Kluwer Academic Publishers, Boston, 1988, pp. 129–157, Proceedings of the Hardware Verification Workshop, Calgary, Canada, 12–16 January 1987.

[13] Joyce J., 'Using Higher-Order Logic to Specify Computer Hardware and Architecture', in *Design Methodologies for VLSI and Computer Architecture*, Edwards D. (editor), North-Holland, Amsterdam, (this volume), Proceedings of the IFIP TC-10 International Working Conference, Pisa, Italy, 19–21 September 1988.

[14] Melham T. F., 'Abstraction Mechanisms for Hardware Verification', in *VLSI Specification, Verification and Synthesis*, Birtwistle G. and Subrahmanyam P. A. (editors), Kluwer Academic Publishers, Boston, 1988, pp. 267–291, Proceedings of the Hardware Verification Workshop, Calgary, Canada, 12–16 January 1987.

[15] Melham T. F., Ph.D. Thesis, University of Cambridge, Computer Laboratory, Forthcoming 1988.

[16] Michie D., 'Memo Functions and Machine Learning', *Nature*, April 1968, Vol. 218, pp. 19–22.

[17] Milne G., 'Simulation and Verification Related Techniques for Hardware Analysis', Technical Report CSR-174-84, University of Edinburgh, Department of Computer Science, November 1984.

[18] Richards M., 'BSPL—A Language for Describing the Behaviour of Synchronous Hardware', Technical Report No. 84, University of Cambridge, Computer Laboratory, April 1986.

[19] Weise D. W., 'Formal Multilevel Hierarchical Verification of Synchronous MOS VLSI Circuits', Ph.D. Thesis, Massachusetts Institute of Technology, August 1986.