Technical Report

UCAM-CL-TR-91
ISSN 1476-2986

Number 91

UNIVERSITY OF CAMBRIDGE
Computer Laboratory

Hardware verification using higher-order logic

Albert Camilleri, Mike Gordon, Tom Melham

September 1986

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

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

© 1986 Albert Camilleri, Mike Gordon, Tom Melham

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

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

ISSN 1476-2986

September 18, 1986

Hardware Verification using Higher-Order Logic [1]

Albert Camilleri   Mike Gordon   Tom Melham

Computer Laboratory

Corn Exchange Street

Cambridge CB2 3QG

Abstract

The Hardware Verification Group at the University of Cambridge is investigating how various kinds of digital systems can be verified by mechanised formal proof. This paper explains our approach to representing behaviour and structure using higher-order logic. Several examples are described, including a ripple-carry adder and a sequential device for computing the factorial function. The dangers of inaccurate models are illustrated with a CMOS exclusive-or gate.

[1] To appear in the proceedings of the IFIP International Working Conference: From H.D.L. Descriptions to Guaranteed Correct Circuit Designs, Grenoble, September 9-11, 1986.

Contents

1 Introduction 2

2 Representing behaviour with predicates 3

3 Representing structure with predicates 4

4 Eliminating existential quantifiers 5

5 A CMOS inverter 6

6 A CMOS exclusive-or (XOR) gate 8

7 A 1-bit full-adder 10

8 An n-bit adder 13

9 A sequential device 15

9.1 The specification 15

9.2 The implementation 16

9.3 The correctness proof 17

9.4 Verification of condition 1 18

9.5 Verification of condition 2 18

9.6 Verification of condition 3 19

10 The 'false implies everything' problem 22

11 Related work 23

12 Conclusions and current research 24

List of Figures

1 Incorrect Design of an Exclusive-Or Gate 8

2 Correct Exclusive-Or Gate Design 9

3 Full Adder 10

4 Implementations of Carry (left) and Sum (right). 11

5 Implementation of a Binary Adder. 13

6 The Factorial Device Implementation 16

1 Introduction

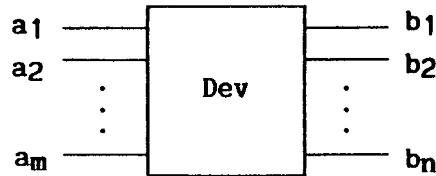
Formal methods have always played a role in hardware design and verification. Logic circuits are routinely specified with logical operators such as $\neg$ (not), $\wedge$ (and) and $\vee$ (or) and then shown to be correct by formal proof using the laws of boolean algebra. Although boolean algebra is adequate for representing and reasoning about the logic-level behaviour of simple combinational circuits, it cannot represent sequential behaviour, or higher-level data-types such as integers. This is not surprising; it is well known that propositional calculus (a formalism equivalent to boolean algebra) is not capable of representing much of mathematics. The formalisation of mathematics requires a logical system at least as powerful as set theory or type theory [9]. In this paper, we describe an approach to specifying and verifying hardware based on a version of type theory called *higher-order logic* [3]. The attraction of this logic is that it provides a directly usable theory of functions. Our approach to specifying hardware uses predicates whose arguments are functions. Higher-order logic can directly represent such predicates and provides efficient inference rules for reasoning about them. In set theory, predicates and functions are introduced as certain kinds of sets and the various inference rules (*e.g.* the rules of λ-conversion) must be laboriously derived. In the long term it is not clear that this is a disadvantage, but it does make it harder to get off the ground with set-theory than with type theory.

The choice of a formalism for hardware verification involves a compromise between expressive power and ease of proof. A simple and restricted formalism will make the proofs of *some* devices easy but may make it hard to specify complex devices simply and concisely. A powerful formalism will be expressive enough for a wide class of devices but may involve difficult proofs. Higher-order logic makes available the results of general mathematics, allowing (in principle) the construction of whatever mathematical tools are needed to deal with the verification task at hand. To make proofs tractable, however, we do not develop special methods for each circuit to be verified, but rather look for general constructions and techniques applicable to wide classes of circuits.

The organisation of this paper is as follows: first we describe how behaviour and structure can be directly represented by logical formulas, next we illustrate how hardware can be verified by formal proof using just the laws of logic, we then discuss an example that shows some of the dangers of using models that are too simple, and finally we give two more substantial examples to show that the approach is applicable to less trivial systems. The first of these examples is an n-bit adder; it illustrates how a logic circuit can be proved to correctly implement a specification expressed in terms of a higher level data-type (numbers). The second example illustrates our approach to modelling sequential behaviour. An implementation operating at a 'fine grain' low-level time-scale is shown to implement a specification expressed at a 'coarser grained' higher level time-scale. Both these two examples illustrate the use of *abstraction* in specification and verification. In the first example we have data abstraction and in the second one temporal abstraction.

2 Representing behaviour with predicates

Consider a device Dev with external lines a1, a2, ... , am, b1, b2, ... , bn.



The behaviour of this device can be specified by defining a predicate Dev with $m+n$ arguments such that $\mathsf{Dev}(a_1, a_2, \ldots, a_m, b_1, b_2, \ldots, b_n)$ holds if and only if a_1, a_2, ... , a_m, b_1, b_2, ... , b_n are allowable values on the corresponding lines of Dev.

We will use predicate calculus to formally specify behaviour. '$P(x)$' means 'x has property P'. The unary boolean operator $\neg$ denotes negation. The infixed binary boolean operators $\vee$, $\wedge$, $\supset$ and $\equiv$ denote disjunction ('or'), conjunction ('and'), implication ('implies') and logical equivalence ('if and only if') respectively. The quantified terms $\forall x.\ t[x]$ and $\exists x.\ t[x]$ mean 'for all x it is the case that $t[x]$' and 'for some x it is the case that $t[x]$' respectively. The conditional term $(t \rightarrow t_1 \mid t_2)$ means 'if t is true then t_1 else t_2'. The two truth-values representing truth and falsity are represented by the boolean constant symbols T and F respectively.

We allow variables to range over functions and the arguments of predicates to be functions. As we will see below, the 'values' needed to model sequential devices will sometimes be functions from time to data-values. Predicates on such values will thus be predicates whose arguments are functions; such predicates are called *higher-order*. For example, the behaviour of a unit-delay device (*i.e.* a register) can be specified with a predicate Del defined by:

$$\mathsf{Del}(i, o) \;\equiv\; \forall t.\ o(t+1) = i(t)$$

Here i and o are functions that map times (represented by non-negative integers) to values (represented by booleans). These functions are in the Del relation if and only if for all times t, the value of o at time $t+1$ (*i.e.* the 'next' time) equals the value of i at time t.

Notice that Del is a *partial* specification; its definition leaves the value of o at time zero unspecified. Using predicates to represent behaviour makes it simple to write such partial specifications, without having to deal with partially defined 'output functions'.

Higher-order predicates are not needed to model combinational devices. For example, a delayless switch can be represented by the predicate Switch defined by:

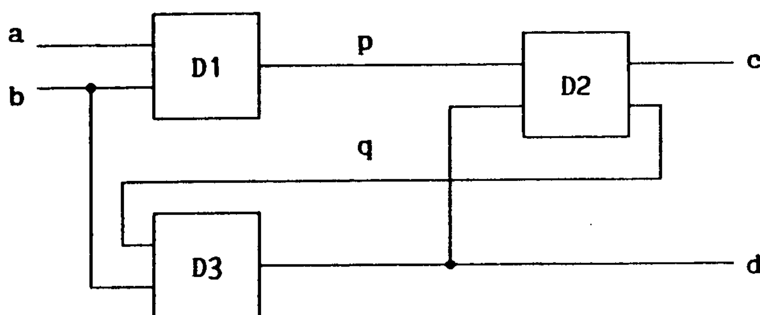
$$\mathsf{Switch}(g, a, b) \;\equiv\; (g \supset (a = b))$$

The condition $\mathsf{Switch}(g, a, b)$ holds if and only if whenever g is true then a and b are equal. For example, $\mathsf{Switch}(\mathsf{T}, \mathsf{F}, \mathsf{F})$ holds because $\mathsf{T} \supset (\mathsf{F}{=}\mathsf{F})$ is true, and $\mathsf{Switch}(\mathsf{F}, \mathsf{T}, \mathsf{F})$ holds because $\mathsf{F} \supset (\mathsf{T}{=}\mathsf{F})$ is true, but $\mathsf{Switch}(\mathsf{T}, \mathsf{T}, \mathsf{F})$ does not hold because $\mathsf{T} \supset (\mathsf{T}{=}\mathsf{F})$ is false. We will see later how such switches can be used as a simple but nevertheless useful model of transistors.

The predicate Switch specifies a *bidirectional* device; there is no input/output distinction between a and b. Being able to deal with bidirectional devices in this way is one of the reasons for representing behaviour with predicates, rather than with functions from inputs to outputs.

3 Representing structure with predicates

Consider the following structure:



This is a device, D say, that is built by connecting together three component devices D1, D2 and D3. The external lines a, b, c and d are external and specify D's interface to the outside world; the lines p and q are internal.

Suppose the behaviours of D1, D2 and D3 are specified by predicates D_1, D_2 and D_3 respectively. Each device constrains the values on its lines. For example, if a, b and p denote the values on the lines a, b and p, then D1 constrains these values so that $D_1(a,b,p)$ holds. To get the constraint imposed by the whole device D we just $\wedge$-together the constraints imposed by D1, D2 and D3. The combined constraint is thus:

$$D_1(a,b,p) \;\wedge\; D_2(p,d,c) \;\wedge\; D_3(q,b,d)$$

This expression constrains the values on the external lines a, b, c and d as well as the internal lines p and q. If we regard D as a 'black box' with the internal lines invisible, then we are really only interested in what constraints are imposed on its external lines. The variables a, b, c and d will denote possible values at the external lines a, b, c and d if and only if the conjunction above holds *for some* values p and q. This can be expressed formally using the existential quantifier $\exists$ by specifying that a, b, c, d are possible values on the output lines of D if and only if:

$$\exists p\; q.\; D_1(a,b,p) \;\wedge\; D_2(p,d,c,q) \;\wedge\; D_3(q,b,d)$$

We can therefore define a predicate D representing the behaviour of D by:

$$D(a,b,c,d) \;\equiv\; \exists p\; q.\; D_1(a,b,p) \;\wedge\; D_2(p,d,c,q) \;\wedge\; D_3(q,b,d)$$

Thus we see that the behaviour corresponding to a circuit is got by conjoining the constraints corresponding to the components and then existentially quantifying the variables corresponding to the internal lines. This technique of representing circuit diagrams in logic

is fairly well known [10]. In the terminology of CCS [13] we are using conjunction for parallel composition and existential quantification for hiding. Other ways of representing structure in logic are also possible [4].

4 Eliminating existential quantifiers

The existentially quantified variables used to hide internal signals can often be eliminated by a sequence of simple logical deductions. A common form of specification defines the output values of a device as a function of the inputs. For example, the definitions of the predicates D_1, D_2 and D_3, mentioned above, could take the form:

$$\begin{array}{lcl} \mathsf{D}_1(a,b,p) & \equiv & p = \mathsf{F}_1(a,b) \\ \mathsf{D}_2(p,d,c,q) & \equiv & c = \mathsf{F}_2(p,d) \wedge q = \mathsf{F}'_2(p,d) \\ \mathsf{D}_3(q,b,d) & \equiv & d = \mathsf{F}_3(q,b) \end{array}$$

In this case, the existential quantifiers in the above definition of D can be eliminated as follows:

1. Expanding D using the definitions of D_1, D_2 and D_3 gives:

$$\begin{array}{lcl} \mathsf{D}(a,b,c,d) & \equiv & \exists p\ q.\ p = \mathsf{F}_1(a,b) \wedge c = \mathsf{F}_2(p,d) \wedge \\ & & \qquad q = \mathsf{F}'_2(p,d) \wedge d = \mathsf{F}_3(q,b) \end{array}$$

2. Substituting with the equations for p and q yields:

$$\begin{array}{lcl} \mathsf{D}(a,b,c,d) & \equiv & \exists p\ q.\ p = \mathsf{F}_1(a,b) \wedge c = \mathsf{F}_2(\mathsf{F}_1(a,b),d) \wedge \\ & & \qquad q = \mathsf{F}'_2(\mathsf{F}_1(a,b),d) \wedge d = \mathsf{F}_3(\mathsf{F}'_2(\mathsf{F}_1(a,b),d),b) \end{array}$$

3. The existential quantifiers can now be moved 'inwards' using rules such as:

$$(\exists x.\, t_1 \wedge t_2) \equiv (\exists x.\, t_1)\ \wedge t_2 \qquad (x \text{ not free in } t_1)$$

   giving

$$\begin{array}{lcl} \mathsf{D}(a,b,c,d) & \equiv & (\exists p.\, p = \mathsf{F}_1(a,b)) \wedge c = \mathsf{F}_2(\mathsf{F}_1(a,b),d) \wedge \\ & & (\exists q.\, q = \mathsf{F}'_2(\mathsf{F}_1(a,b),d)) \wedge d = \mathsf{F}_3(\mathsf{F}'_2(\mathsf{F}_1(a,b),d),b) \end{array}$$

4. $(\exists x.x{=}tm)$ is true for all tm (if x is not free in tm) by $\exists$-introduction applied to the left hand side of $tm{=}tm$. Thus, the expression for D can be reduced to:

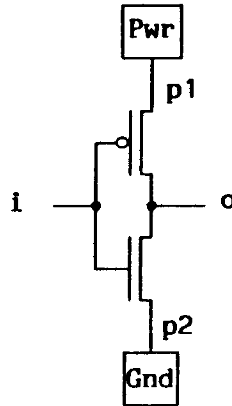
$$\mathsf{D}(a,b,c,d) \ \equiv\ c = \mathsf{F}_2(\mathsf{F}_1(a,b),d) \wedge d = \mathsf{F}_3(\mathsf{F}'_2(\mathsf{F}_1(a,b),d),b)$$

   which completes the elimination of the existential quantifiers in the definition of D.

This method of eliminating existential quantifiers — which uses only logical inference — is widely applicable and easily automated. In some cases, however, it may be inconvenient or impossible to eliminate the existential quantifiers using this method and other means of dealing with them must be used. An example involving such a case will be considered later.

5 A CMOS inverter

This first example is trivial, but it illustrates the general idea of verification by formal proof. The standard CMOS implementation of an inverter is:

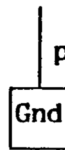


The inverter shown above can be viewed as a structure built out of four components: a power source, a ground, an n-transistor and a p-transistor.

A power source

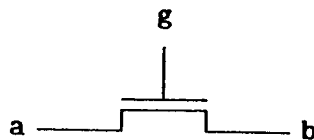
can be modelled by a predicate Pwr that constrains the value on the line p always to be T: $\mathsf{Pwr}(p) \equiv (p = \mathsf{T})$.

Dual to Pwr is 'ground'



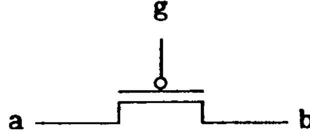
which can be modelled by a predicate Gnd that constrains the value on the line p always to be F: $\mathsf{Gnd}(p) \equiv (p = \mathsf{F})$.

In CMOS there are two kinds of transistors: n-transistors and p-transistors. An n-transistor



can be modelled as a switch: $\mathsf{Ntran}(g, a, b) \equiv (g \supset (a = b))$.

A p-transistor



can be modelled as a switch: $\mathsf{Ptran}(g, a, b) \equiv (\neg g \supset (a = b))$ which conducts when its gate(*i.e.* line g) is low.

Conjoining together the constraints from the four components of the inverter and existentially quantifying the internal line variables yields the following definition of a predicate Inv:

$$\mathsf{Inv}(i, o) \equiv \exists p_1\ p_2.\ \mathsf{Pwr}(p_1) \wedge \mathsf{Ptran}(i, p_1, o) \wedge \mathsf{Ntran}(i, o, p_2) \wedge \mathsf{Gnd}(p_2)$$

If $\mathsf{Inv}(i, o)$ holds then the values i and o are constrained to be in the relation determined by the inverter circuit above. It follows by standard logical reasoning that if Inv is defined as above, then the constraint on i and o imposed by the inverter circuit is exactly what we want, namely $o = \neg i$. An outline of the formal proof of this is as follows:

1. By the definition of Inv:

$$\mathsf{Inv}(i, o) \equiv \exists p_1\ p_2.\ \mathsf{Pwr}(p_1) \wedge \mathsf{Ptran}(i, p_1, o) \wedge \mathsf{Ntran}(i, o, p_2) \wedge \mathsf{Gnd}(p_2)$$

2. Expanding using the definitions of Pwr and Gnd yields:

$$\mathsf{Inv}(i, o) \equiv \exists p_1\ p_2.\ (p_1 = \mathsf{T}) \wedge \mathsf{Ptran}(i, p_1, o) \wedge \mathsf{Ntran}(i, o, p_2) \wedge (p_2 = \mathsf{F})$$

3. Eliminating the existential quantifiers by the method outlined above yields:

$$\mathsf{Inv}(i, o) \equiv \mathsf{Ptran}(i, \mathsf{T}, o) \wedge \mathsf{Ntran}(i, o, \mathsf{F})$$

4. Expanding using the definitions of Ptran and Ntran results in:

$$\mathsf{Inv}(i, o) \equiv ((i = \mathsf{F}) \supset (\mathsf{T} = o)) \wedge ((i = \mathsf{T}) \supset (o = \mathsf{F}))$$

5. From which we can derive:

$$\mathsf{Inv}(\mathsf{T}, o) \equiv (o = \mathsf{F}) \qquad \wedge \qquad \mathsf{Inv}(\mathsf{F}, o) \equiv (o = \mathsf{T})$$

and finally, using the excluded middle axiom $\forall i.\, i{=}\mathsf{T} \vee i{=}\mathsf{F}$, we can derive:

$$\mathsf{Inv}(i, o) \equiv (o = \neg i)$$

6 A CMOS exclusive-or (XOR) gate

The next example illustrates some of the dangers of using an inaccurate model. Consider the circuit shown in Figure 1.

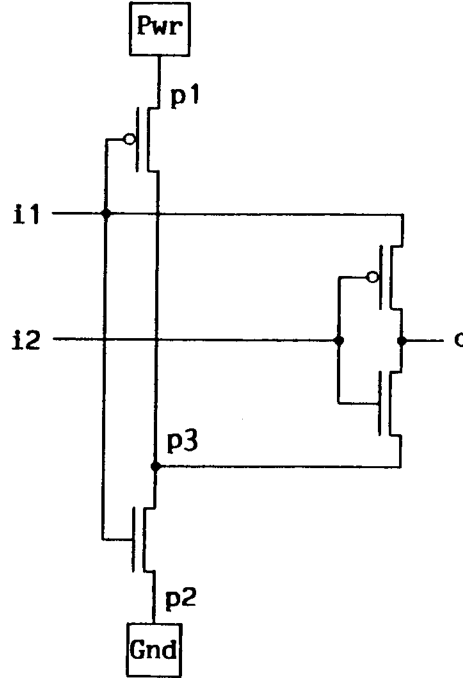


Figure 1: Incorrect Design of an Exclusive-Or Gate

If transistors are modelled by switches *then* it is easy to show that this circuit implements an exclusive-or gate. The proof proceeds as follows.

1. From the circuit diagram:

$$\begin{aligned}\mathsf{Xor_Imp}(i_1,i_2,o) \;\equiv\; \exists p_1\ p_2\ p_3.\ & \mathsf{Pwr}(p_1) \ \wedge\ \mathsf{Gnd}(p_2)\ \wedge \\ & \mathsf{Ptran}(i_1,p_1,p_3)\ \wedge\ \mathsf{Ntran}(i_1,p_3,p_2)\ \wedge \\ & \mathsf{Ptran}(i_2,i_1,o)\ \wedge\ \mathsf{Ntran}(i_2,o,p_3)\end{aligned}$$

2. Expanding with the definitions of Pwr and Gnd and eliminating the existentially quantified variables p_1 and p_2 gives:

$$\begin{aligned}\mathsf{Xor_Imp}(i_1,i_2,o) \;\equiv\; \exists p_3.\ & \mathsf{Ptran}(i_1,\mathsf{T},p_3)\ \wedge\ \mathsf{Ntran}(i_1,p_3,\mathsf{F})\ \wedge \\ & \mathsf{Ptran}(i_2,i_1,o)\ \wedge\ \mathsf{Ntran}(i_2,o,p_3)\end{aligned}$$

3. By the definitions of Ntran and Ptran:

$$\begin{aligned}\mathsf{Xor_Imp}(i_1,i_2,o) \;\equiv\; \exists p_3.\ & (\neg i_1 \supset (p_3{=}\mathsf{T}))\ \wedge\ (i_1 \supset (p_3{=}\mathsf{F}))\ \wedge \\ & (\neg i_2 \supset (o{=}i_1))\ \wedge\ (i_2 \supset (o{=}p_3))\end{aligned}$$

4. Boolean algebra and laws for equality give:

$$\mathsf{Xor_Imp}(i_1,i_2,o) \;\equiv\; \exists p_3.\ (p_3 = \neg i_1)\ \wedge\ (o = \neg(i_1{=}i_2))$$

5. Eliminating the existential quantifier gives:

$$\mathsf{Xor_Imp}(i_1, i_2, o) \;\equiv\; (o = \neg(i_1 = i_2))$$

Despite this correctness proof, the circuit may not work in practice. Transistors don't always behave likes switches in the way represented by the definitions of Ptran and Ntran. The actual behaviour of an n-transistor is that it conducts (is 'on') if its gate voltage is higher, by some threshold, than the voltage on one of its other lines, otherwise it doesn't conduct (is 'off'). Dually, a p-transistor is 'on' if its gate voltage is lower, by some threshold, than the voltage on one of its other lines. For definiteness, suppose T is $5v$, F is $0v$ and the thresholds are $1v$. Thus, for example, a p-transistor is off (*i.e.* non-conducting) unless its gate voltage is at least $1v$ less than the voltage on one of its other lines. Consider now the situation with i1 at $0v$ and i2 at $5v$. The p-transistor connected to Pwr will be on and so p3 will be at $5v$. Clearly the p-transistor whose gate is connected to i2 will be off, but what about the n-transistor below it whose gate is also connected to i2? This has $5v$ on its gate and on one of its other lines (namely the one connected to p3). If the voltage at o is $5v$ then this n-transistor will be off and o will float. As soon as some charge is removed from o (*e.g.* by the circuitry it is connected to), its voltage will begin to drop. When o's voltage gets below $4v$ the n-transistor connected to it will start to conduct and so o will be charged up again by Pwr. After a short time the voltage at o will rise above $4v$ and the transistor will switch off again, leaving o at about $4v$. The upshot of this is that if the circuit is used to drive anything that requires any power, then the n-transistor connected to o will repeatedly switch on and off to maintain the voltage at o at about $4v$.

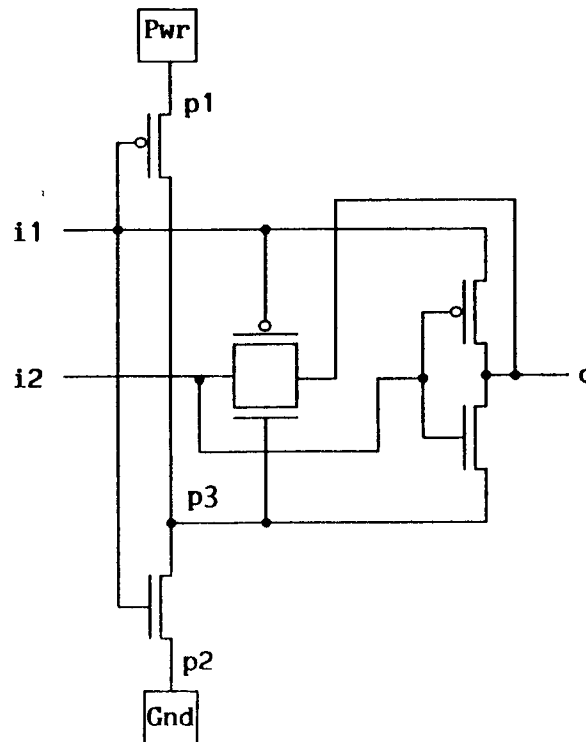


Figure 2: Correct Exclusive-Or Gate Design

Unfortunately, this rather weak and unstable output voltage may be inadequate to drive other parts of the circuit. It is possible that things could be made to work by adjusting the size of transistors, but doing this is a bit dodgy and certainly isn't clean CMOS design. In fact, the above circuit is not normally used. The standard exclusive-or gate given in textbooks is shown in Figure 2. This has an extra pair of transistors that ensure that if i1 is low then o is connected to i2, thereby preventing the nasty behaviour just described.

This circuit is formally equivalent to the simpler one if transistors are modelled by Ptran and Ntran. This shows how important it is to use accurate models. It is no good verifying an implementation using a model that doesn't represent the actual behaviour of the parts involved.

7 A 1-bit full-adder

This example will illustrate the proof of a 1-bit full-adder in preparation for the verification of the n-bit adder discussed in the next example.

A full-adder generates a sum and carry-out from two inputs and a carry-in. Together, the components Sum and Carry (shown in Figure 3) compute the binary addition of three bits. For example, if i1, i2 and cin had values 1, 1, and 0 respectively, then Sum would compute the binary sum 0 and Carry would compute the carry-over value of 1.

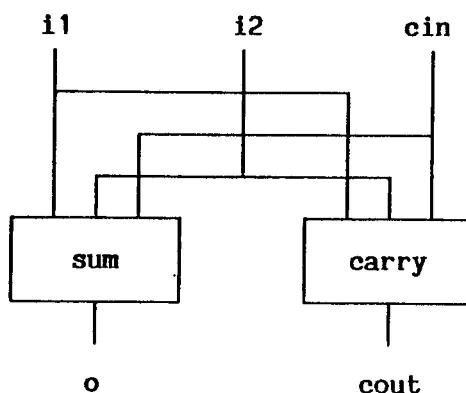


Figure 3: Full Adder

The desired behaviours of the components Sum and Carry are specified by the predicates Sum and Carry:

$$\mathsf{Sum}(i_1, i_2, cin, o) \equiv (o = (\neg i_1 \wedge \neg i_2 \wedge cin) \vee (\neg i_1 \wedge i_2 \wedge \neg cin) \vee (i_1 \wedge \neg i_2 \wedge \neg cin) \vee (i_1 \wedge i_2 \wedge cin))$$

$$\mathsf{Carry}(i_1, i_2, cin, cout) \equiv (cout = (i_1 \wedge i_2) \vee (i_1 \wedge cin) \vee (i_2 \wedge cin))$$

We could take Sum and Carry as primitives with the above behaviour (derived from their truth-tables) but instead we will derive these behaviours from a lower-level implementation using simpler components (as shown in Figure 4).

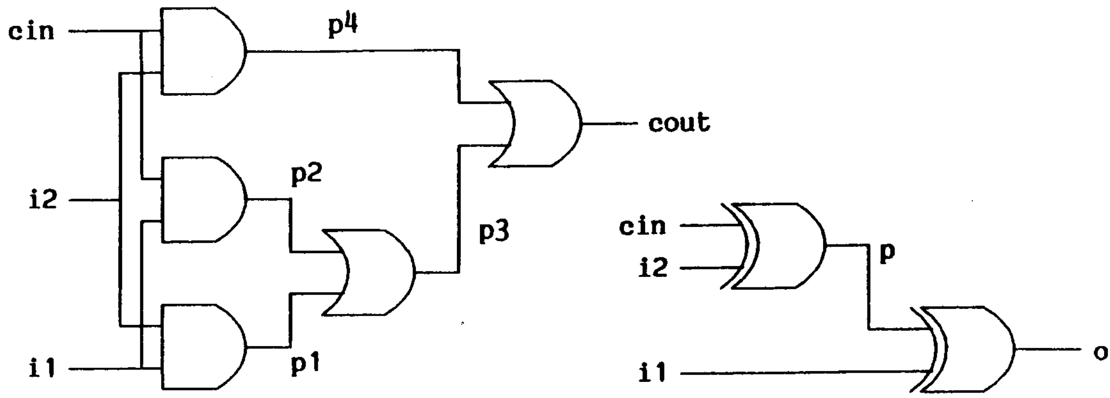


Figure 4: Implementations of Carry (left) and Sum (right).

We take the components in the implementation of Sum and Carry to be primitives, defined by:

$$\mathsf{OR}(i_1, i_2, o) \equiv (o = i_1 \vee i_2)$$

$$\mathsf{AND}(i_1, i_2, o) \equiv (o = i_1 \wedge i_2)$$

$$\mathsf{XOR}(i_1, i_2, o) \equiv (o = (i_1 \wedge \neg i_2) \vee (\neg i_1 \wedge i_2))$$

The definitions of the predicates Sum_Imp and Carry_Imp below are based on the structures shown in Figure 4.

$$\mathsf{Sum_Imp}(i_1, i_2, cin, o) \equiv \exists p. \mathsf{XOR}(i_1, p, o) \wedge \mathsf{XOR}(i_2, cin, p)$$

$$\begin{array}{l} \mathsf{Carry_Imp}(i_1, i_2, cin, cout) \equiv \\ \quad \exists\, p1\ p2\ p3\ p4\ . \\ \quad\quad \mathsf{AND}(i_1, i_2, p1) \wedge \mathsf{AND}(i_1, cin, p2) \wedge \\ \quad\quad \mathsf{AND}(i_2, cin, p4) \wedge \mathsf{OR}(p1, p2, p3) \wedge \\ \quad\quad \mathsf{OR}(p3, p4, cout) \end{array}$$

From these definitions it is routine to prove that:

$$\mathsf{Sum}(i_1, i_2, cin, o) \equiv \mathsf{Sum_Imp}(i_1, i_2, cin, o)$$

$$\mathsf{Carry}(i_1, i_2, cin, o) \equiv \mathsf{Carry_Imp}(i_1, i_2, cin, o)$$

The following is an outline of the proof of correctness for the implementation of Sum. The proof for the implementation of Carry is almost identical.

1. The theorem we wish to prove is:

$$\mathsf{Sum}(i_1, i_2, cin, o) \equiv \mathsf{Sum_Imp}(i_1, i_2, cin, o)$$

2. Expanding using the definition of Sum_Imp, gives:

$$\mathsf{Sum}(i_1, i_2, cin, o) \equiv \exists p. \mathsf{XOR}(i_1, p, o) \wedge \mathsf{XOR}(i_2, cin, p)$$

3. Expanding using the definition of XOR yields:

$$\begin{aligned}
&\mathsf{Sum}(i_1, i_2, cin, o) \equiv \\
&\exists p.(o = (i_1 \wedge \neg p) \;\vee\; (\neg i_1 \wedge p)) \wedge \\
&\quad (p = (i_2 \wedge \neg cin) \;\vee\; (\neg i_2 \wedge cin))
\end{aligned}$$

4. Eliminating the existential quantifier gives:

$$\begin{aligned}
&\mathsf{Sum}(i_1, i_2, cin, o) \equiv \\
&(o = (i_1 \wedge \neg((i_2 \wedge \neg cin) \;\vee\; (\neg i_2 \wedge cin))) \vee \\
&\qquad (\neg i_1 \wedge ((i_2 \wedge \neg cin) \;\vee\; (\neg i_2 \wedge cin))))
\end{aligned}$$

5. Expanding using the definition of Sum and breaking up the equivalence into two implications we obtain:

$$\begin{aligned}
&(o = (\neg i_1 \wedge \neg i_2 \wedge cin) \;\vee\; (\neg i_1 \wedge i_2 \wedge \neg cin) \vee \\
&\qquad (i_1 \wedge \neg i_2 \wedge \neg cin) \;\vee\; (i_1 \wedge i_2 \wedge cin)) \\
&\supset \\
&(o = (i_1 \wedge \neg((i_2 \wedge \neg cin) \;\vee\; (\neg i_2 \wedge cin))) \vee \\
&\qquad (\neg i_1 \wedge ((i_2 \wedge \neg cin) \;\vee\; (\neg i_2 \wedge cin))))
\end{aligned}$$

and

$$\begin{aligned}
&(o = (i_1 \wedge \neg((i_2 \wedge \neg cin) \;\vee\; (\neg i_2 \wedge cin))) \vee \\
&\qquad (\neg i_1 \wedge ((i_2 \wedge \neg cin) \;\vee\; (\neg i_2 \wedge cin)))) \\
&\supset \\
&(o = (\neg i_1 \wedge \neg i_2 \wedge cin) \;\vee\; (\neg i_1 \wedge i_2 \wedge \neg cin) \vee \\
&\qquad (i_1 \wedge \neg i_2 \wedge \neg cin) \;\vee\; (i_1 \wedge i_2 \wedge cin))
\end{aligned}$$

6. Assuming the antecedent of each of the implications, and rewriting in the consequents, we derive:

$$\begin{aligned}
&(\neg i_1 \wedge \neg i_2 \wedge cin) \;\vee\; (\neg i_1 \wedge i_2 \wedge \neg cin) \vee \\
&(i_1 \wedge \neg i_2 \wedge \neg cin) \;\vee\; (i_1 \wedge i_2 \wedge cin) \\
&= \\
&(i_1 \wedge \neg((i_2 \wedge \neg cin) \;\vee\; (\neg i_2 \wedge cin))) \vee \\
&(\neg i_1 \wedge ((i_2 \wedge \neg cin) \;\vee\; (\neg i_2 \wedge cin)))
\end{aligned}$$

and

$$\begin{aligned}
&(i_1 \wedge \neg((i_2 \wedge \neg cin) \;\vee\; (\neg i_2 \wedge cin))) \vee \\
&(\neg i_1 \wedge ((i_2 \wedge \neg cin) \;\vee\; (\neg i_2 \wedge cin))) \\
&= \\
&(\neg i_1 \wedge \neg i_2 \wedge cin) \;\vee\; (\neg i_1 \wedge i_2 \wedge \neg cin) \vee \\
&(i_1 \wedge \neg i_2 \wedge \neg cin) \;\vee\; (i_1 \wedge i_2 \wedge cin)
\end{aligned}$$

which can be proven by simple boolean algebra, hence proving the original term.

8 An n-bit adder

We now outline the verification of the n-bit adder shown in Figure 5. We represent n-bit inputs and outputs with functions from integers (representing bit-positions) to booleans. For example, $a(0)$, $a(1)$ and $a(2)$ represent the first three bits of input a. In order to relate bit-strings to integers we define a function Val by primitive recursion, as follows:

$$\begin{aligned}&\mathsf{Val}(0,f) = \mathsf{Bitval}(f(0)) \wedge \\ &\mathsf{Val}(n{+}1,f) = (2^{n+1} \times \mathsf{Bitval}(f(n{+}1))) + \mathsf{Val}(n,f)\end{aligned}$$

Bitval is a function which maps the boolean truth-values, T and F, to the integer values, 1 and 0 respectively.

$$\mathsf{Bitval}(x) = (x \rightarrow 1|0)$$

Next we need to capture the logic in Figure 5 by first defining a 1-bit adder slice, using the Sum and Carry predicates defined in the previous section.

$$\mathsf{Add1}(i_1, i_2, cin, out, cout) \equiv \mathsf{Sum}(i_1, i_2, cin, out) \wedge \mathsf{Carry}(i_1, i_2, cin, cout)$$

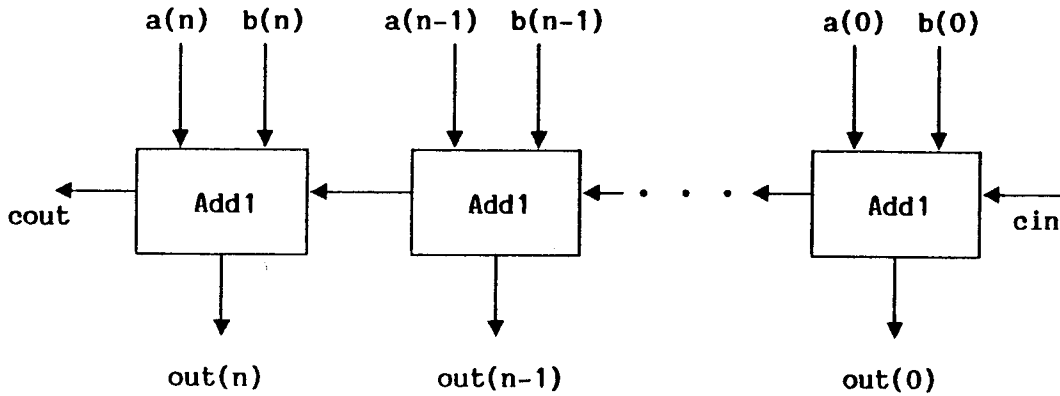


Figure 5: Implementation of a Binary Adder.

Figure 5 above, showing iterated 1-bit adder slices, can be represented in logic in various ways [6]. The most straightforward is the simple recursive definition given below:

$$\begin{aligned}&\mathsf{Adder}(0, a, b, cin, out, cout) \equiv (cout = cin) \wedge \\ &\mathsf{Adder}(n{+}1, a, b, cin, out, cout) \equiv \\ &\quad \exists\, cn.\ \mathsf{Adder}(n, a, b, cin, out, cn) \wedge \mathsf{Add1}(a(n), b(n), cn, out(n), cout)\end{aligned}$$

Hence, as one can see from the definition of Adder, if the value of n is 0 then we are performing absolutely no addition at all but merely wiring cin through to $cout$.

One can now show that an n-bit adder correctly performs binary addition by proving the following theorem.

$$\begin{aligned}&\forall n.\ \forall f\ g\ cin\ out\ cout. \\ &\quad \mathsf{Adder}(n{+}1, f, g, cin, out, cout) \supset \\ &\qquad (2^{n+1} \times \mathsf{Bitval}(cout)) + \mathsf{Val}(n, out) = \\ &\qquad \mathsf{Val}(n,f) + \mathsf{Val}(n,g) + \mathsf{Bitval}(cin)\end{aligned}$$

The proof proceeds as follows by performing mathematical induction on n. The basis and step cases obtained are:

$$
\begin{array}{l}
\forall\, f\ g\ cin\ out\ cout\ . \\
\quad \mathsf{Adder}(0{+}1, f, g, cin, out, cout) \supset \\
\qquad (2^{0+1} \times \mathsf{Bitval}(cout)) + \mathsf{Val}(0, out) = \\
\qquad \mathsf{Val}(0, f) + \mathsf{Val}(0, g) + \mathsf{Bitval}(cin)
\end{array}
$$

and

$$
\begin{array}{l}
\forall\, f\ g\ cin\ out\ cout. \\
\quad \mathsf{Adder}(n{+}1, f, g, cin, out, cout) \supset \\
\qquad (2^{n+1} \times \mathsf{Bitval}(cout)) + \mathsf{Val}(n, out) = \\
\qquad \mathsf{Val}(n, f) + \mathsf{Val}(n, g) + \mathsf{Bitval}(cin) \\
\supset \\
\forall f\ g\ cin\ out\ cout. \\
\quad \mathsf{Adder}(n{+}1{+}1, f, g, cin, out, cout) \supset \\
\qquad (2^{n+1+1} \times \mathsf{Bitval}(cout)) + \mathsf{Val}(n{+}1, out) = \\
\qquad \mathsf{Val}(n{+}1, f) + \mathsf{Val}(n{+}1, g) + \mathsf{Bitval}(cin)
\end{array}
$$

Proving the basis case is straightforward but the step case is very tedious though not too difficult. Only a very brief outline of this proof will therefore be given.

1. The proof of the basis case is done by repeatedly expanding using the definition of Adder to obtain:

$$
\begin{array}{l}
(\exists\, cn.\, (cn = cin) \wedge \mathsf{Add1}(f(0), g(0), cn, out(0), cout)) \supset \\
(2^{0+1} \times \mathsf{Bitval}(cout)) + \mathsf{Val}(0, out) = \mathsf{Val}(0, f) + \mathsf{Val}(0, g) + \mathsf{Bitval}(cin)
\end{array}
$$

   which is proved by eliminating the existential quantifiers, expanding using the definitions of Add1, Sum, Carry, Val and Bitval, and using the excluded middle axiom to perform boolean case analysis on $f(0)$, $g(0)$ and cin.

2. The induction step is proved by expanding using the definitions of Adder, Add1, Sum, Carry and Val, assuming the induction hypothesis, and performing resolution to obtain:

$$
\begin{array}{l}
\forall\, n\ f\ g\ cn\ out\ cout\ . \\
\quad (out(n{+}1) = (\neg f(n{+}1) \wedge \neg g(n{+}1) \wedge cn) \vee \\
\qquad\qquad (\neg f(n{+}1) \wedge g(n{+}1) \wedge \neg cn) \vee \\
\qquad\qquad (f(n{+}1) \wedge \neg g(n{+}1) \wedge \neg cn) \vee \\
\qquad\qquad (f(n{+}1) \wedge g(n{+}1) \wedge cn)) \wedge \\
\quad (cout = (f(n{+}1) \wedge g(n{+}1)) \vee (f(n{+}1) \wedge cn) \vee (g(n{+}1) \wedge cn)) \\
\quad \supset \\
\quad (2^{n+1} \times \mathsf{Bitval}(out(n{+}1))) + (2^{n+1} \times \mathsf{Bitval}(cout)) = \\
\quad (2^{n+1} \times \mathsf{Bitval}(g(n{+}1))) + (2^{n+1} \times \mathsf{Bitval}(f(n{+}1))) + \\
\quad (2^{n+1} \times \mathsf{Bitval}(cn))
\end{array}
$$

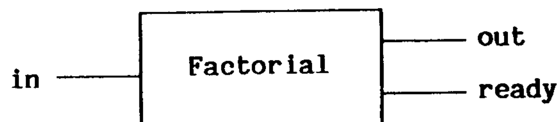
   which can be proved by rewriting and repeated use of the excluded middle axiom.

9 A sequential device

We now consider the formal verification of a sequential device for computing the factorial function. The design of this device was first considered in [5], where it was proven correct using a formalism based on denotational semantics.

9.1 The specification

The factorial device has an input line, in and two output lines, out and ready:



The boolean output line ready indicates when the device is ready to read an integer on the input line, in. If ready is high at some time t, then the numerical value on the input line at that time, say n, is loaded into the device. The ready line will then remain low for n units of time, while the device computes the factorial of n. During this computation time the output line, out, will have the value 0 and the value on in will be ignored. When the computation is finished, the result $n!$ will appear on out for one unit of time (at time $t+n+1$). The ready line will then go high again (at time $t+n+2$) and the whole cycle will be repeated for a new input.

The behaviour of Factorial is specified formally by the higher-order predicate Factorial, defined as follows:

$$
\begin{array}{l}
\mathsf{Factorial}(ready, in, out) \equiv \\
\quad \forall n\, t.\, ready(t) \wedge n{=}in(t) \supset \\
\qquad \mathsf{Next}(t, t{+}n{+}2, ready) \wedge \\
\qquad \forall t'.\, (t{\le}t' \wedge t'{<}t{+}n{+}1) \supset (out(t'){=}0) \wedge \\
\qquad out(t{+}n{+}1) = \mathsf{Fact}(n)
\end{array}
$$

Here, the time-varying value on the line ready is modelled by the function *ready* from time (integers) to booleans. The numeric values on the lines in and out also vary over time; they are therefore represented by the functions *in* and *out* which map time to integers.

Notice that this model is an abstraction; any physical realisation of Factorial will necessarily have maximum possible input and output values. While there are ways of formally relating descriptions which are at different levels of abstraction, we will consider here only the higher level of abstraction specified by Factorial.

The definition of Factorial states that if the ready line is high at some time t and n is the value on in at that time — *i.e.* '$ready(t) \wedge n{=}in(t)$' — then it will be the case that:

1. The next time after time t that ready will be high is time $t+n+2$:

$$\mathsf{Next}(t, t{+}n{+}2, ready)$$

2. The output out will be zero from time t up to time $t+n+1$:

$$\forall t'.\,(t \leq t' \wedge t' < t+n+1) \supset (out(t')=0)$$

3. The output out at time $t+n+1$ will be the factorial of n:

$$out(t+n+1) = \mathsf{Fact}(n)$$

The variable n is used in the specification for readability only, allowing us to write, for example, '$t'<t+n+1$' rather than '$t'<t+in(t)+1$'.

The predicate Next is defined by:

$$\mathsf{Next}(t_1, t_2, f) \equiv t_1 < t_2 \wedge f(t_2) \wedge \forall t.\,(t_1 < t \wedge t < t_2) \supset \neg f(t)$$

and the function Fact is defined by primitive recursion as follows:

$$\mathsf{Fact}(0) = 1 \wedge$$
$$\mathsf{Fact}(n+1) = (n+1) \times \mathsf{Fact}(n)$$

9.2 The implementation

The implementation of the factorial device, shown in figure 6, uses three components, Down, Mult and Test.

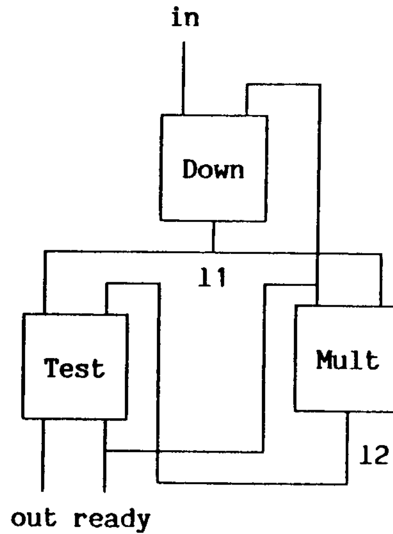


Figure 6: The Factorial Device Implementation

The Down device latches the input on line in when ready is high and counts down, on line l1, from the input value to zero. The successive values of the count generated by Down are input to Mult which builds up the factorial result on line l2 by repeated multiplication. The line l2 is given the initial value 1 by Mult when ready is high. The device Test controls the other two components via the ready line and outputs the result once l1 becomes zero.

The formal specifications of these three components are:

$$\mathsf{Down}(in, ready, l1) \equiv \forall t.\, l1(t+1) = (ready(t) \rightarrow in(t) \mid l1(t) - 1)$$

$$\mathsf{Mult}(ready, l1, l2) \equiv \forall t.\, l2(t{+}1) = (ready(t) \rightarrow 1 \mid l2(t) \times l1(t))$$

$$\begin{aligned}\mathsf{Test}(l1, l2, out, ready) \equiv\ & \forall t.\, out(t) = (l1(t){=}0 \wedge \neg ready(t) \rightarrow l2(t) \mid 0) \wedge \\ & \forall t.\, ready(t{+}1) = (l1(t){=}0 \wedge \neg ready(t))\end{aligned}$$

and the connection of components, Fact_Imp, is defined by:

$$\begin{aligned}\mathsf{Fact_Imp}(ready, in, out) \equiv \exists l1\, l2.\ & \mathsf{Down}(in, ready, l1) \wedge \\ & \mathsf{Mult}(ready, l1, l2) \wedge \\ & \mathsf{Test}(l1, l2, out, ready)\end{aligned}$$

9.3 The correctness proof

We wish to show that Fact_Imp is a correct implementation of the factorial device as specified by Factorial. Formally, the logical theorem we want to prove is:

$$\mathsf{Fact_Imp}(ready, in, out) \supset \mathsf{Factorial}(ready, in, out)$$

That is, whenever the signals *ready*, *in* and *out* satisfy the Fact_Imp relationship then they will also satisfy the Factorial relationship.

Expanding the correctness statement using the definition of Fact_Imp gives:

$$\begin{aligned}& \exists l1\, l2.\, \mathsf{Down}(in, ready, l1) \wedge \mathsf{Mult}(ready, l1, l2) \wedge \mathsf{Test}(l1, l2, out, ready) \supset \\ & \mathsf{Factorial}(ready, in, out)\end{aligned}$$

Because the right hand side of the equation for $l1$ in the definition of Down involves the variable $l1$ itself, we will not be able to use the simple technique for eliminating existential quantifiers outlined previously. Notice, however, that $l1$ and $l2$ do not occur in the consequent of the above implication. Since, for all terms t_1 and t_2, '$(\exists x.t_1) \supset t_2$' is equivalent to '$\forall x.(t_1 \supset t_2)$' if x does not occur free in t_2, the above implication can be reduced to:

$$\begin{aligned}& \mathsf{Down}(in, ready, l1) \wedge \mathsf{Mult}(ready, l1, l2) \wedge \mathsf{Test}(l1, l2, out, ready) \supset \\ & \mathsf{Factorial}(ready, in, out)\end{aligned}$$

Expanding with the definition of Factorial gives:

$$\begin{aligned}& \mathsf{Down}(in, ready, l1) \wedge \mathsf{Mult}(ready, l1, l2) \wedge \mathsf{Test}(l1, l2, out, ready) \supset \\ & \forall n\, t.\, ready(t) \wedge n{=}in(t) \supset \\ & \qquad \mathsf{Next}(t, t{+}n{+}2, ready) \wedge \\ & \qquad \forall t'.\, (t{\leq}t' \wedge t'{<}t{+}n{+}1) \supset (out(t'){=}0) \wedge \\ & \qquad out(t{+}n{+}1) = \mathsf{Fact}(n)\end{aligned}$$

The three conjuncts in the consequent are just the three correctness conditions of the specification. The proof of this correctness statement proceeds by considering each of these correctness conditions in turn.

9.4 Verification of condition 1

The first correctness condition specifies the behaviour of the ready signal. Formally, we must show that:

$$\begin{array}{l}\mathsf{Down}(in, ready, l1)\ \wedge \mathsf{Mult}(ready, l1, l2)\ \wedge \mathsf{Test}(l1, l2, out, ready) \supset \\ ready(t) \wedge n{=}in(t) \supset \mathsf{Next}\ (t, t{+}n{+}2, ready)\end{array}$$

The proof of this lemma proceeds by first observing that if the ready line is high at time t then it will be low at time $t{+}1$; this follows from the equation for *ready* in the definition of Test (see section 9.2). Also, if ready is high at time t and the input at time t is n then the line 11 will have the value n at time $t{+}1$, by the definition of Down. Formally, we have:

$$\begin{array}{l}\mathsf{Down}(in, ready, l1)\ \wedge \mathsf{Mult}(ready, l1, l2)\ \wedge \mathsf{Test}(l1, l2, out, ready) \supset \\ ready(t) \wedge n{=}in(t) \supset (\neg ready(t{+}1) \wedge l1(t{+}1){=}n)\end{array} \tag{1}$$

We then show that if the ready is low at some time t and the line 11 has some value, say *num*, at time t then ready will remain low until the moment of time after line 11 reaches zero. That is, the next time ready will be high will be time $t{+}num{+}1$. Formally:

$$\begin{array}{l}\mathsf{Down}(in, ready, l1)\ \wedge \mathsf{Mult}(ready, l1, l2)\ \wedge \mathsf{Test}(l1, l2, out, ready) \supset \\ \forall num\ t.\ \neg ready(t) \wedge l1(t){=}num \supset \mathsf{Next}(t, t{+}num{+}1, ready)\end{array}$$

The proof of this lemma is done by induction on *num* and use of the definitions of *ready* and $l1$. The variable *num* is not introduced here merely for readability but is included to put the lemma in a form suitable for doing induction. Specialising the variables *num* and t to n and $t{+}1$ respectively, gives:

$$\begin{array}{l}\mathsf{Down}(in, ready, l1)\ \wedge \mathsf{Mult}(ready, l1, l2)\ \wedge \mathsf{Test}(l1, l2, out, ready) \supset \\ (\neg ready(t{+}1) \wedge l1(t{+}1){=}n) \supset \mathsf{Next}(t{+}1, t{+}1{+}n{+}1, ready)\end{array}$$

This theorem, together with (1) and the simple fact that:

$$\neg f(t_1{+}1) \wedge \mathsf{Next}(t_1{+}1, t_2, f) \supset \mathsf{Next}(t_1, t_2, f)$$

give the result:

$$\begin{array}{l}\mathsf{Down}(in, ready, l1)\ \wedge \mathsf{Mult}(ready, l1, l2)\ \wedge \mathsf{Test}(l1, l2, out, ready) \supset \\ ready(t) \wedge n{=}in(t) \supset \mathsf{Next}\ (t, t{+}n{+}2, ready)\end{array}$$

which completes the proof of the first correctness condition.

9.5 Verification of condition 2

The second condition of the specification is that the value on output line, out, during the loading and computation phase is 0. The lemma we wish to prove is:

$$\begin{array}{l}\mathsf{Down}(in, ready, l1)\ \wedge \mathsf{Mult}(ready, l1, l2)\ \wedge \mathsf{Test}(l1, l2, out, ready) \supset \\ ready(t) \wedge n{=}in(t) \supset (\forall t'.\ (t{\le}t' \wedge t'{<}t{+}n{+}1) \supset out(t'){=}0)\end{array}$$

To prove this lemma, we begin by showing that if **ready** is low at time t and line **l1** has the value num at time t, then the value on **out** between time t and $t{+}num$ will be 0:

$$\begin{array}{l}\mathsf{Down}(in,ready,l1)\ \wedge \mathsf{Mult}(ready,l1,l2)\ \wedge \mathsf{Test}(l1,l2,out,ready) \supset \\ \forall num\, t.\,(\neg ready(t) \wedge l1(t){=}num) \supset (\forall t'.\,(t{<}t' \wedge t'{<}t{+}num) \supset out(t'){=}0)\end{array} \quad (2)$$

This lemma is proved by induction on num and use of the definitions of $l1$, $ready$ and out. Again the variable num is introduced to make induction possible.

By case analysis on the condition '$num = 0$', it is possible to show that:

$$\begin{array}{l}\mathsf{Down}(in,ready,l1)\ \wedge \mathsf{Mult}(ready,l1,l2)\ \wedge \mathsf{Test}(l1,l2,out,ready) \supset \\ \forall num\, t.\,(\neg ready(t) \wedge l1(t){=}num) \supset (\forall t'.\,(t{=}t' \wedge t'{<}t{+}num) \supset out(t'){=}0)\end{array} \quad (3)$$

since, if $num{\neq}0$ then $out(t){=}0$ by the specification of out and, if $num{=}0$, then the term

$$(t{=}t' \wedge t'{<}t{+}num)$$

is false for all t' and so the implication

$$\forall t'.\,(t{=}t' \wedge t'{<}t{+}num) \supset out(t'){=}0$$

will be true.

Combining (2) and (3) yields:

$$\begin{array}{l}\mathsf{Down}(in,ready,l1)\ \wedge \mathsf{Mult}(ready,l1,l2)\ \wedge \mathsf{Test}(l1,l2,out,ready) \supset \\ \forall num\, t.\,(\neg ready(t) \wedge l1(t){=}num) \supset (\forall t'.\,(t{\leq}t' \wedge t'{<}t{+}num) \supset out(t'){=}0)\end{array}$$

Specialising the variables num and t to n and $t{+}1$ respectively, we have:

$$\begin{array}{l}\mathsf{Down}(in,ready,l1)\ \wedge \mathsf{Mult}(ready,l1,l2)\ \wedge \mathsf{Test}(l1,l2,out,ready) \supset \\ (\neg ready(t{+}1) \wedge l1(t{+}1){=}n) \supset (\forall t'.\,(t{+}1{\leq}t' \wedge t'{<}t{+}1{+}n) \supset out(t'){=}0)\end{array}$$

Rewriting $t{+}1{\leq}t'$ to $t{<}t'$ in and again using (1) we have:

$$\begin{array}{l}\mathsf{Down}(in,ready,l1)\ \wedge \mathsf{Mult}(ready,l1,l2)\ \wedge \mathsf{Test}(l1,l2,out,ready) \supset \\ ready(t) \wedge n{=}in(t) \supset (\forall t'.\,(t{<}t' \wedge t'{<}t{+}1{+}n) \supset out(t'){=}0)\end{array}$$

which, along with the fact that $ready(t) \supset out(t){=}0$, immediately yields:

$$\begin{array}{l}\mathsf{Down}(in,ready,l1)\ \wedge \mathsf{Mult}(ready,l1,l2)\ \wedge \mathsf{Test}(l1,l2,out,ready) \supset \\ ready(t) \wedge n{=}in(t) \supset (\forall t'.\,(t{\leq}t' \wedge t'{<}t{+}n{+}1) \supset out(t'){=}0)\end{array}$$

which completes the proof of the second lemma needed for the correctness statement.

9.6 Verification of condition 3

The third and final condition which the implementation has to satisfy is that it correctly computes the factorial:

$$\begin{array}{l}\mathsf{Down}(in,ready,l1)\ \wedge \mathsf{Mult}(ready,l1,l2)\ \wedge \mathsf{Test}(l1,l2,out,ready) \supset \\ ready(t) \wedge n{=}in(t) \supset out(t{+}n{+}1) = \mathsf{Fact}(n)\end{array}$$

It is possible to proceed to prove this lemma directly. We will not do this but will instead illustrate the use a more general theorem, which can be applied to the proofs of many similar devices. The theorem is:

$$\begin{array}{l} \forall\, fun\ f\ g\ h\ load\ i\ s. \\ \quad \forall t.\, load(t) = f(s(t)) \wedge \\ \quad \forall t.\, s(t+1) = (load(t) \rightarrow g(i(t)) \mid h(s(t))) \wedge \\ \quad \forall b.\, fun(b) = (f(h(b)) \rightarrow b \mid fun(h(b))) \supset \\ \qquad \forall t.\, load(t) \wedge \mathsf{Next}(t, t+d+2, load) \supset s(t+d+1) = fun(g(i(t))) \end{array} \tag{4}$$

This theorem is intended to be applied to a device which operates as follows. The device has a load indicator output, `load`, whose value is some function of the device's internal state. When `load` is high at some time t then the device's internal state at the next moment of time (*i.e.* time $t+1$) will be determined by some function of the input at time t. When `load` is false at time t then the device's internal state at time $t+1$ will be some function of only its state at time t — input is ignored when the load indicator is false. Formally, this behaviour is expressed in (4) by the equations:

$$\begin{array}{l} \forall t.\, load(t) = f(s(t)) \\ \forall t.\, s(t+1) = (load(t) \rightarrow g(i(t)) \mid h(s(t))) \end{array}$$

Here, the functions i and s model the input and state of the device as they change over time; $i(t)$ models the input at time t and $s(t)$ the internal state at time t. The function f defines the function $load$ which models the value of the load indicator, `load`. Notice that the value of $load$ at time t depends only on the state at time t. The function h is used to compute the next state of the device when the load indicator is false and the function g is used to compute the initialised state of the device when the load indicator is true.

The variable fun in (4) denotes a recursively defined function which when given a state, b, of the device yields the first state after b where the load indicator is just about to become true — *i.e.* the first state where one more application of the next state function, h, will give a state in which the load indicator, as defined by f, is true. Thus the function fun 'iterates' the application of the next state function h until the just before load indicator becomes true. Formally:

$$\forall b.\, fun(b) = (f(h(b)) \rightarrow b \mid fun(h(b)))$$

The conclusion of theorem (4) is:

$$\forall t.\, load(t) \wedge \mathsf{Next}(t, t+d+2, load) \supset s(t+d+1) = fun(g(i(t)))$$

This states that if `load` is true at some time t and the next time after t that `load` is true will be time $t+d+2$ then the internal state at time $t+d+1$ will be given by the function fun applied to the state initialised by the input at time t.

The proof of theorem (4) is straightforward but tedious and so will not be given here. The main step consists of an induction on the variable d.

We begin the application of theorem (4) to the proof the final correctness condition by representing the state of the factorial device with a single function. The factorial device

has 'internal state' at time t modelled by the three signals $l1$, $l2$ and $ready$. We will represent this internal state by the *triple*: $(l2(t),\ l1(t),\ ready(t))$ and define the internal state function, S, as follows:

$$\mathsf{S}(t) = (l2(t),\ l1(t),\ ready(t))$$

The *ready* signal will be used as the load indicator for the factorial device. Notice that $ready(t)$ is a function of the internal state at time t, as required by theorem (4):

$$\begin{array}{l}\mathsf{Down}(in, ready, l1)\ \wedge \mathsf{Mult}(ready, l1, l2)\ \wedge \mathsf{Test}(l1, l2, out, ready) \supset \\ \forall t.\, ready(t) = \mathsf{Third}(\mathsf{S}(t))\end{array} \qquad (5)$$

where $\mathsf{Third}(v_1, v_2, v_3) = v_3$.

When $ready(t)$ is true the next state of the device is the triple $(1, in(t), \mathsf{F})$ and, when $ready(t)$ is false, the next state is the triple $(l2(t)\times l1(t), l1(t)-1,\ l1(t)=0)$ so the next state and initialisation state functions, H and G, are defined:

$$\mathsf{H}(m, n, t) = (m \times n, n-1, n=0)$$

$$\mathsf{G}(num) = (1, num, \mathsf{F})$$

Notice that the state equation for S is:

$$\begin{array}{l}\mathsf{Down}(in, ready, l1)\ \wedge \mathsf{Mult}(ready, l1, l2)\ \wedge \mathsf{Test}(l1, l2, out, ready) \supset \\ \forall t.\, \mathsf{S}(t+1) = (ready(t) \rightarrow \mathsf{G}(in(t))\ |\ \mathsf{H}(\mathsf{S}(t)))\end{array} \qquad (6)$$

as required by theorem (4). We are now ready to define the function which iterates the state changes of the factorial device in the way required of fun in theorem (4). The state iteration function (which operates on a triple of values) is defined by primitive recursion as follows:

$$\begin{array}{l}\mathsf{Fun}(m, 0, t) = (m, 0, t)\ \wedge \\ \mathsf{Fun}(m, n+1, t) = \mathsf{Fun}(m\times(n+1), n, \mathsf{F})\end{array}$$

It is easy to show that Fun has the recursive form required by theorem (4). *I.e.* that:

$$\begin{array}{l}\mathsf{Down}(in, ready, l1)\ \wedge \mathsf{Mult}(ready, l1, l2)\ \wedge \mathsf{Test}(l1, l2, out, ready) \supset \\ \forall b.\, \mathsf{Fun}(b) = (\mathsf{Third}(\mathsf{H}(b)) \rightarrow b\ |\ \mathsf{Fun}(\mathsf{H}(b)))\end{array} \qquad (7)$$

Specialising the variables fun, f, g, h, $load$, i and s in theorem (4) to Fun, Third, G, H, $ready$, in and S respectively, yields:

$$\begin{array}{l}\forall t.\, ready(t) = \mathsf{Third}(\mathsf{S}(t)) \wedge \\ \forall t.\, \mathsf{S}(t+1) = (ready(t) \rightarrow \mathsf{G}(in(t))\ |\ \mathsf{H}(\mathsf{S}(t))) \wedge \\ \forall b. \mathsf{Fun}(b) = (\mathsf{Third}(\mathsf{H}(b)) \rightarrow b\ |\ \mathsf{Fun}(\mathsf{H}(b))) \supset \\ \quad \forall t.\, ready(t) \wedge \mathsf{Next}(t, t+d+2, ready) \supset \mathsf{S}(t+d+1) = \mathsf{Fun}(\mathsf{G}(in(t)))\end{array}$$

Which, using theorems (5), (6) and (7) simplifies to:

$$\begin{array}{l}\mathsf{Down}(in, ready, l1)\ \wedge \mathsf{Mult}(ready, l1, l2)\ \wedge \mathsf{Test}(l1, l2, out, ready) \supset \\ \forall t.\, ready(t) \wedge \mathsf{Next}(t, t+d+2, ready) \supset \mathsf{S}(t+d+1) = \mathsf{Fun}(\mathsf{G}(in(t)))\end{array}$$

Which can be used along with the first correctness condition, to derive:

$$\begin{array}{l}\mathsf{Down}(in, ready, l1) \wedge \mathsf{Mult}(ready, l1, l2) \wedge \mathsf{Test}(l1, l2, out, ready) \supset \\ ready(t) \wedge n{=}in(t) \supset \mathsf{S}(t{+}d{+}1) = \mathsf{Fun}(\mathsf{G}(n))\end{array} \quad (8)$$

Using induction on n, the definition of Fact and the definition of G, it is easy to show that:

$$\mathsf{Fun}(\mathsf{G}(n)) = (\mathsf{Fact}(n), 0, \mathsf{F})$$

Which, along with the definition of S, and theorem (8) gives:

$$\begin{array}{l}\mathsf{Down}(in, ready, l1) \wedge \mathsf{Mult}(ready, l1, l2) \wedge \mathsf{Test}(l1, l2, out, ready) \supset \\ ready(t) \wedge n{=}in(t) \supset \\ (l2(t{+}n{+}1), l1(t{+}n{+}1), ready(t{+}n{+}1)) = (\mathsf{Fact}(n), 0, \mathsf{F})\end{array}$$

The conclusion of this theorem states that *ready* is false at time $t{+}n{+}1$ and $l1$ has the value 0 at time $t{+}n{+}1$ and that $l2(t{+}n{+}1) = \mathsf{Fact}(n)$ so, by the equation for *out*, we have that:

$$\begin{array}{l}\mathsf{Down}(in, ready, l1) \wedge \mathsf{Mult}(ready, l1, l2) \wedge \mathsf{Test}(l1, l2, out, ready) \supset \\ ready(t) \wedge n{=}in(t) \supset out(t{+}n{+}1) = \mathsf{Fact}\end{array}$$

which is the last correctness lemma to be proven. This completes the correctness proof of the factorial device design.

10 The 'false implies everything' problem

The correctness of the CMOS examples and the 1-bit full adder were formulated as logical equivalences of the form:

$$\forall i_1 \cdots i_m o_1 \cdots o_n.\ \mathsf{Imp}(i_1, \ldots, i_m, o_1, \ldots, o_n) \equiv \mathsf{Spec}(i_1, \ldots, i_m, o_1, \ldots, o_n)$$

where Imp is a predicate representing the implementation, Spec is a predicate representing the specification, i_1, ... , i_m are the inputs and o_1, ... , o_n are the outputs. For simple examples, this is often the appropriate way to state correctness and some systems can only handle such equivalences [2,5]. For complex devices, it is usually wrong to require that the implementation be logically equivalent to the specification. The specification will typically be some sort of abstract description of behaviour, operating on different data-types and at a different timescale to the implementing circuit. In such cases correctness is more appropriately formulated as an implication of the form:

$$\forall i_1 \cdots i_m o_1 \cdots o_n.\ \mathsf{Imp}(i_1, \ldots, i_m, o_1, \ldots, o_n) \supset \mathsf{Spec}(\mathsf{Abs}(i_1, \ldots, i_m, o_1, \ldots, o_n))$$

where Abs is a data abstraction function. The correctness of both the adder and the factorial device were formulated in this way (the data abstraction for the factorial device was trivial).

It is a property of implication that $\mathsf{F} \supset x$ is true for any x, *i.e.* 'false implies everything'. Thus if the implementation formula $\mathsf{Imp}(i_1, \ldots, i_m, o_1, \ldots, o_m)$ were equivalent to F then it

would correctly implement every specification. An example of such a paradoxical implementation is easily obtained by considering a circuit in which both power (modelled by Pwr) and ground (modelled by Gnd) are connected to a single line p. Since $((p = \mathsf{T}) \wedge (p = \mathsf{F})) = \mathsf{F}$ it follows that the the formula representing this circuit will be equivalent to F. It is easy to see what has gone wrong. The definitions $\mathsf{Pwr}(p) \equiv (p = \mathsf{T})$ and $\mathsf{Gnd}(p) \equiv (p = \mathsf{F})$ assert that the outputs of Pwr and Gnd are *always* equal to T and F respectively, but this cannot be the case when these outputs are connected.

Several approaches to this problem have been proposed. For example, Mike Fourman has suggested that correctness be reformulated as

$$(\forall i_1 \cdots i_m o_1 \cdots o_n.\ \mathsf{Imp}(i_1, \ldots, i_m, o_1, \ldots, o_m) \supset \mathsf{Spec}(i_1, \ldots, i_m, o_1, \ldots, o_n)) \wedge$$
$$(\forall i_1 \cdots i_m.\ \exists o_1 \cdots o_n.\ \mathsf{Imp}(i_1, \ldots, i_m, o_1, \ldots, o_m))$$

The second conjunct of this asserts that for all possible input values there are some output values consistent with it. If this is true then the implementation formula is never equal to F and so the 'false implies everything' problem cannot arise. Although this maneuver appears to work it seems a bit *ad hoc* and is not easily generalised to bidirectional circuits. Another approach is to look for a better model in which the problem doesn't arise. The essential idea is to recognise that there are more values than can be modelled with just T and F. For example, if power is connected to ground then some sort of 'high current' value appears. The value appearing on a line must be a 'union' of all the values being driven onto the line. The union of high and low is then defined to be a 'high current' value. Recent work by Glynn Winskel on combining Milner's calculus of communicating systems with Bryant's discrete circuit theory offers the hope of a successful model along these lines.

11 Related work

The general idea of representing hardware directly in predicate calculus is well known. Various approaches differ in the particular details of the representation and the particular version of predicate calculus used. Work based on first-order logic includes the proof of equivalence of some asynchronous hardware devices [1] and Warren Hunt's impressive use of the Boyer-Moore theorem-prover to verify the FM8501 microprocessor [11]. The approach described in this paper is influenced by the elegant work of Keith Hanna on 'Veritas' [8] and Ben Moszkowski's work on modelling hardware using Interval Temporal Logic [7]. Considerable success in generating proofs fully automatically has been obtained by Harry Barrow of Schlumberger Palo Alto Research [2]. His 'VERIFY' system employs a specialised representation of behaviour based of state machines. It has been used to automatically verify a number of complicated circuits with several levels of hierarchy including a multiplier and a little computer. As the CMOS XOR example described above shows, we need better models of transistors. A promising and completely different approach to representing behaviour is being developed by George Milne at Edinburgh. Milne's circuit calculus called 'CIRCAL' may be a better framework in which to model transistors than pure logic. As

mentioned in the previous section, Glynn Winskel at Cambridge is working on combining Bryant's model of MOS with Milner's CCS. This also looks as though it will lead to more accurate transistor models.

12 Conclusions and current research

The examples described in this paper illustrate the general approach being taken at Cambridge but are not representative of the current state-of-the-art. Several much more complex examples have been successfully specified and verified including a microprocessor [12] and a simple local area network.

Formal verification is very expensive using current theorem-proving technologies. Experts are needed to guide proof generating tools and typical proofs take months of work. In the short term it is likely that verification by formal proof will only be worthwhile for those systems whose failure would result in disasters such as loss of life, destruction of costly equipment, or recall of a mass produced product. Examples of such systems include aircraft control systems, nuclear reactor monitors, satellite systems, medical devices and chips in automobiles.

The first commercially available formal verification systems may be able to generate correctness proofs of simple designs fully automatically, but they are likely to need manual guidance for more complicated ones. A possible scenario is that in-house system designers will verify those parts of their designs that can be done automatically but will contract-out the difficult parts to a 'verification shop' staffed by a new kind of professional ('verification engineers'). To see if this scenario is feasible one of us (Gordon) has taken on a contract to verify a microprocessor designed for safety critical applications. This contract has only just started; it is hoped that it will show that existing methods and tools are robust and powerful enough to be used on real examples.

Acknowledgements

We have had many instructive discussions with the members of the Cambridge Hardware Verification Group — especially Inder Dhingra, Miriam Leeser, John Herbert and Don Gaubatz.

Tom Melham is funded by a scholarship from the Royal Commission for the Exhibition of 1851. Albert Camilleri is supported by the Commonwealth Scholarship Commission in the U.K.

References

[1] Barros, J.C. and Johnson, B.W., "Equivalence of the Arbiter, the Synchronizer, the Latch and the Inertial Delay", IEEE Transactions on Computers, Vol. C-32, No. 7, July 1983.

[2] Barrow, H., "Proving the Correctness of Digital Hardware Designs", VLSI Design, Vol. 5, No. 7, July 1984.

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

[4] Clocksin, W.F., "Logic Programming and the Specification of Circuits", Technical Report No. 72, Computer Laboratory, University of Cambridge, August, 1985.

[5] Gordon, M.J.C., "A Model of Register Transfer Systems with Applications to Microcode and VLSI correctness", Technical Report CSR-82-81, Dept. of Computer Science, University of Edinburgh, May 1982.

[6] Gordon, M.J.C., "Why Higher-Order Logic is a Good Formalism for Specifying and Verifying Hardware", Technical Report no. 77, University of Cambridge, Computer Laboratory, September 1985.

[7] Halpern, J., Manna, J. and Moszkowski, B., "A Hardware Semantics based on Temporal Intervals", In the proceedings of the 10-th International Colloquium on Automata, Languages and Programming, Barcelona, Spain, 1983.

[8] Hanna, F.K. and Daeche, N., "The VERITAS Theorem Prover", Internal Report, University of Kent, August 1985.

[9] Hatcher, W., "The Logical Foundations of Mathematics", Pergamon Press, 1982.

[10] Hoare, C.A.R., "A Calculus of Total Correctness for Communicating Processes", Science of Computer Programming, Vol. 1, No. 1, 1981.

[11] Hunt, W. A. Jr., "FM8501: A Verified Microprocessor", Technical Report 47, Institute for Computing Science, The University of Texas at Austin, 1985.

[12] Joyce, J., Birtwistle, G. and Gordon, M., "Proving a Computer Correct in Higher Order Logic", Research Report No. 85/208/21, Department of Computer Science, The University of Calgary, August 1985.

[13] Milner, R., "A Calculus of Communicating Systems", Lecture Notes in Computer Science No. 92, Springer-Verlag, 1980.