August 16, 1991

Why higher-order logic is a good formalism for specifying and verifying hardware[1]

Mike Gordon
Computer Laboratory
Corn Exchange Street
Cambridge CB2 3QG

Abstract

Higher-order logic was originally developed as a foundation for mathematics. In this paper we show how it can be used as:

1. a hardware description language, and
2. a formalism for proving that designs meet their specifications.

Examples are given which illustrate various specification and verification techniques. These include a CMOS inverter, a CMOS full adder, an n-bit ripple-carry adder, a sequential multiplier and an edge-triggered Dtype register.

[1]An earlier version of this paper was published in *Formal Aspects of VLSI Design*, edited by G. Milne and P. A. Subrahmanyam, North Holland, 1986.

Contents

1 Introduction

The purpose of this paper is to show, via examples, that:

1. Many kinds of digital systems can be formally specified using the notation of formal logic; specialized hardware description languages are not needed.

2. The inference rules of logic provide a practical means of proving systems correct; specialized deductive systems are not needed.

The idea of using 'pure logic' as a hardware description and verification language is not new. The general approach advocated here has been promoted by, among others, Keith Hanna [Hanna & Daeche] and Ben Moszkowski [Halpern *et al.*]. It was as a result of working with Moszkowski that I realized that everything that I had previously been doing using an *ad hoc* formalism called LSM (Logic of Sequential Machines [Gordon83]) could be done *much better* in pure logic.

The particular logical system used here is called higher-order logic, and is very briefly explained in the next section. It is hoped that this section will enable readers who are not familiar with predicate calculus to understand what follows. Thorough introductions to higher-order logic can be found in textbooks on mathematical logic [Hatcher], in Church's original paper [Church], or in the report on the HOL logic [Gordon85(a)].

2 Introduction to higher-order logic

Higher-order logic uses standard predicate logic notation:

- "$P(x)$" means "x has property P",
- "$\neg t$" means "not t",
- "$t_1 \vee t_2$" means "t_1 or t_2",
- "$t_1 \wedge t_2$" means "t_1 and t_2",
- "$t_1 \supset t_2$" means "t_1 implies t_2",
- "$t_1 \equiv t_2$" means "t_1 if and only if t_2",
- "$\forall x.\ t[x]$" means "for all x it is the case that $t[x]$",
- "$\exists x.\ t[x]$" means "for some x it is the case that $t[x]$",

- "$(t \rightarrow t_1 \mid t_2)$" means "if t is true then t_1 else t_2".

There are three important ways that higher-order logic extends first-order logic.

1. Variables can range over functions and predicates. Such variables are called *higher-order* and can be quantified. For example, the principle of mathematical induction can be expressed using a variable P that ranges over predicates:

$$\forall P.\ P(0) \wedge (\forall n.\ P(n)\ \supset\ P(n{+}1))\ \supset\ \forall n.\ P(n)$$

   The existence of a function satisfying a simple recursive definition can be stated using variables f and s that range over functions.

$$\forall n_0.\ \forall f.\ \exists s.\ (s(0) = n_0) \wedge (\forall n.\ s(n{+}1) = f(s(n)))$$

   This asserts that for each number n_0 and function f there exists a function s such that $s(0){=}n_0$ and $s(n{+}1){=}f(s(n))$ for all n. The two examples just described make essential use of higher-order variables, and thus they can't be expressed in first-order logic.

2. Functions and predicates can be the arguments and results of other functions and predicates. For example, a function Iterate can be defined such that:

$$\mathsf{Iterate}(m,n)(f)\ \equiv\ f(n)\ \wedge\ f(n{-}1)\ \wedge\ \cdots\ \wedge\ f(m)$$

   Iterate maps a pair of numbers (m,n) to a higher-order predicate whose argument is a function. This predicate is true of a function f if and only if $f(n) \wedge f(n{-}1) \wedge \cdots \wedge f(m)$ holds.

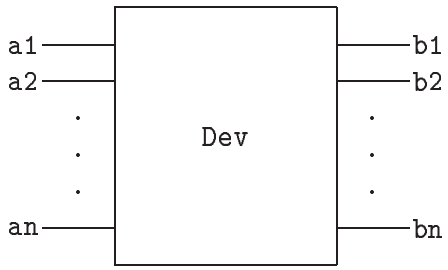
3. Higher-order logic has special function-denoting terms called λ-expressions. These have the form $\lambda x.\ t$ where x is a variable and t is an expression. Such a λ-term denotes the function $a \mapsto t[a/x]$ where $t[a/x]$ is the result of substituting a for x in t. For example, $\lambda x.\ x{+}3$ denotes the function $a \mapsto a{+}3$ which adds 3 to its argument. A common use of λ-expressions is as arguments to higher-order functions. It will be explained later how the term

$$\mathsf{Iterate}\ (0,n)\ (\lambda i.\ \mathsf{Add1}(a(i), b(i), c(i), sum(i), c(i{+}1)))$$

   can be used to represent an array of $n{+}1$ 1-bit adders.

3 Representing behaviour with predicates

A device is a 'black box' with a specified behaviour; for example:



This device is called `Dev` and has external lines `a1`, `a2`, ... , `am`, `b1`, `b2`, ... , `bn`. These lines correspond to the 'pins' of an integrated circuit. When the device is in operation each line has a value drawn from some set of possible values. Different kinds of device are modelled with different sets of values. The behaviour of device `Dev` is 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`.

The following font conventions will be used:

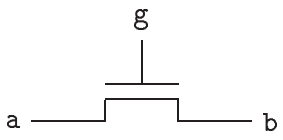
- Physical objects like devices and lines will be written in `typewriter` font.
- Mathematical variables will be written in *italic* font.
- Mathematical constants, (*e.g.* predicate and function constants) will be written in sans serif font.

The same letter will be used for a physical object and its mathematical representation. Thus, for example, l will range over the values allowed at line `l`, and Dev denotes the predicate describing the behaviour of device `Dev`.

We now describe two examples that illustrate the use of predicates to specify behaviour. In the first of these examples the values on lines are modelled with truth-values. In the second example the values on lines are modelled with functions, and consequently the predicate used to specify the behaviour of the device is higher-order.

3.1 A delayless switch

Zero-delay combinational devices can be modelled by taking the boolean values T and F as the allowed values on their lines. An example is a switch:



The intended behaviour of this is that a is connected to b if g has the value T and a and b are not connected if g has the value F. This behaviour 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.

3.2 An inverter with delay

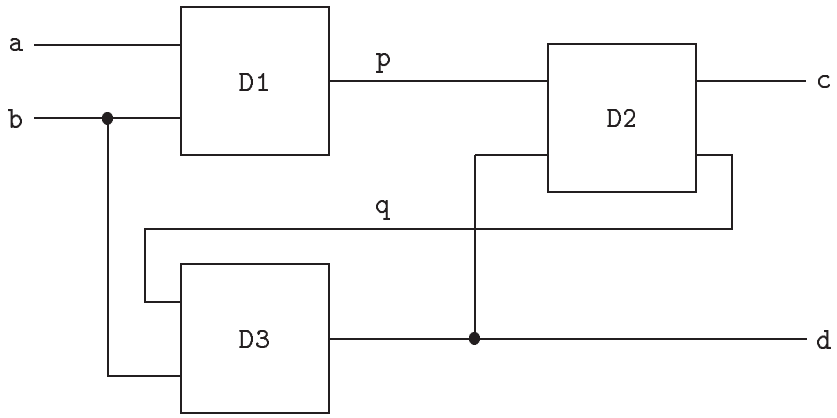
The values occurring on the lines of devices may vary over time. When this happens, their behaviour must be represented by predicates whose arguments are 'time varying values'. Such values correspond to 'waveforms' and can be modelled by functions of time. For example, the behaviour of an inverter with a delay of δ units of time can be specified with a predicate Invert defined by:

$$\mathsf{Invert}(i, o) \;\equiv\; \forall t.\; o(t{+}\delta) = \neg i(t)$$

Here the values on lines i and o are functions i and o which map times (represented by numbers) to values (represented by booleans). These functions are in the Invert relation if and only if for all times t, the value of o at time $t{+}\delta$ equals the value of i at time t.

4 Representing circuit structure with predicates

Consider the following structure (called D):



This device is built by connecting together three component devices D1, D2 and D3. The external lines of D are a, b, c and d. The lines p and q are internal and are not connected to the 'outside world'. (External lines might correspond to the pins of an integrated circuit, and internal lines to tracks.)

Suppose the behaviours of D1, D2 and D3 are specified by predicates D_1, D_2 and D_3 respectively. How can we derive the behaviour of the system D shown above? Each device constrains the values on its lines. If a, b and p denote the values on the lines a, b and p, then D1 constrains these values so that $\mathsf{D}_1(a, b, p)$ holds. To get the constraint imposed by the whole device D we just conjoin (*i.e.* $\wedge$-together) the constraints imposed by D1, D2 and D3; the combined constraint is thus:

$$\mathsf{D}_1(a, b, p) \ \wedge \ \mathsf{D}_2(p, d, c) \ \wedge \ \mathsf{D}_3(q, b, d)$$

This expression constrains the values on both the external lines a, b, c and d and 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. We can therefore define a predicate D representing the behaviour of D by:

$$\mathsf{D}(a, b, c, d) \ \equiv \ \exists p \ q. \ \mathsf{D}_1(a, b, p) \ \wedge \ \mathsf{D}_2(p, d, c) \ \wedge \ \mathsf{D}_3(q, b, d)$$

Thus we see that the behaviour corresponding to a circuit is got by:

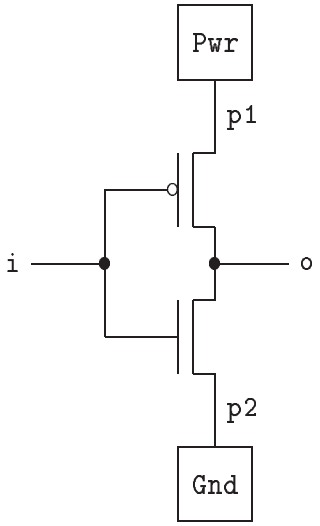
- Conjoining the constraints corresponding to the components, and

- existentially quantifying the variables corresponding to the internal lines.

This technique of representing circuit diagrams in logic is fairly well known [Hoare]. In the terminology of CCS [Milner] we are using conjunction for parallel composition and existential quantification for hiding. Other ways of representing structure in logic are also possible [Clocksin].

5 A CMOS inverter

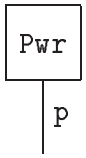
The standard CMOS implementation of an inverter is:



5.1 Specification of the components

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.

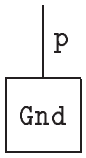
5.1.1 Power



This is a power source (sometimes called $\mathsf{V_{dd}}$) and can be modelled by a predicate Pwr that constrains the value on the line p to be T.

$$\mathsf{Pwr}(p) \;\equiv\; (p = \mathsf{T})$$

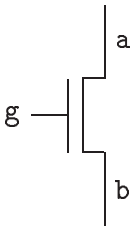
5.1.2 Ground



This represents 'ground' and can be modelled by a predicate Gnd that constrains the value on the line p to be F.

$$\mathsf{Gnd}(p) \;\equiv\; (p = \mathsf{F})$$

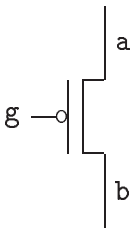
5.1.3 n-transistor



This represents an n-transistor. It can be modelled as a switch.

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

5.1.4 p-transistor



This represents a p-transistor. It can be modelled as a switch which conducts when its gate (*i.e.* line g) is low.

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

5.2 Logic representation of the inverter circuit

Conjoining together the constraints from the four components 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 contrained to be in the relation determined by the inverter circuit above.

5.3 Verification by proof

It follows by standard logical reasoning that if Inv is defined as above, then

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

This shows that the constraint on i and o imposed by the inverter circuit is exactly what we want: o is the inverse of i.

An outline of the formal proof of this is as follows:

1. By definition of Inv:

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

2. Substituting in the definitions of Pwr and Gnd yields:

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

3. Substituting with the equations p_1=T and p_2=F yields:

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

4. In general, if t_1 and t_2 are any terms such that t_2 doesn't contain x then:

$$(\exists x.\ t_1\ \wedge\ t_2)\ =((\exists x.\ t_1)\ \wedge\ t_2)$$

and

$$(\exists x.\ t_2\ \wedge\ t_1)\ =(t_2\ \wedge\ (\exists x.\ t_1))$$

are both true. Using these properties we can move the existential quantifiers inwards to derive:

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

5. Both $(\exists p_1.\ p_1{=}\mathsf{T})$ and $(\exists p_2.\ p_2{=}\mathsf{F})$ are logical truths and so can be deleted from conjunctions; hence:

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

6. Next we substitute in the definitions of Ptran and Ntran to get:

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

7. From this we can derive

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

and

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

from which it follows by case analysis that:

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

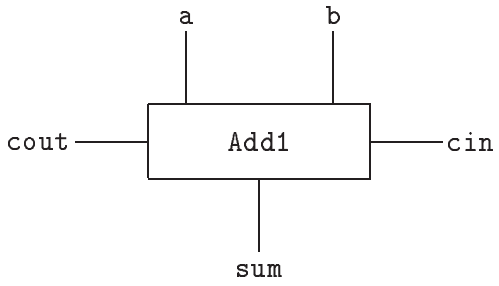
Proofs such as these can be generated by the HOL system [Gordon85(b)].

6 A 1-bit CMOS full adder

The full adder described in this section was shown to me by Inder Dhingra; it illustrates the use of bidirectional transistors in CMOS. The transistor models Ptran and Ntran can be used to prove the circuit correct. Such a proof would be difficult with the usual representation of combinational circuits as boolean functions. Relations rather than functions are needed to model bidirectionality.

6.1 Specification

Here is a diagram of a full adder:



The lines `a`, `b`, `cin`, `sum` and `cout` carry the boolean values T or F. To relate these logical values to the numbers 1 and 0 we define a function Bit_Val by:

$$\mathsf{Bit\_Val}(\mathsf{T}) = 1 \quad \text{and} \quad \mathsf{Bit\_Val}(\mathsf{F}) = 0$$

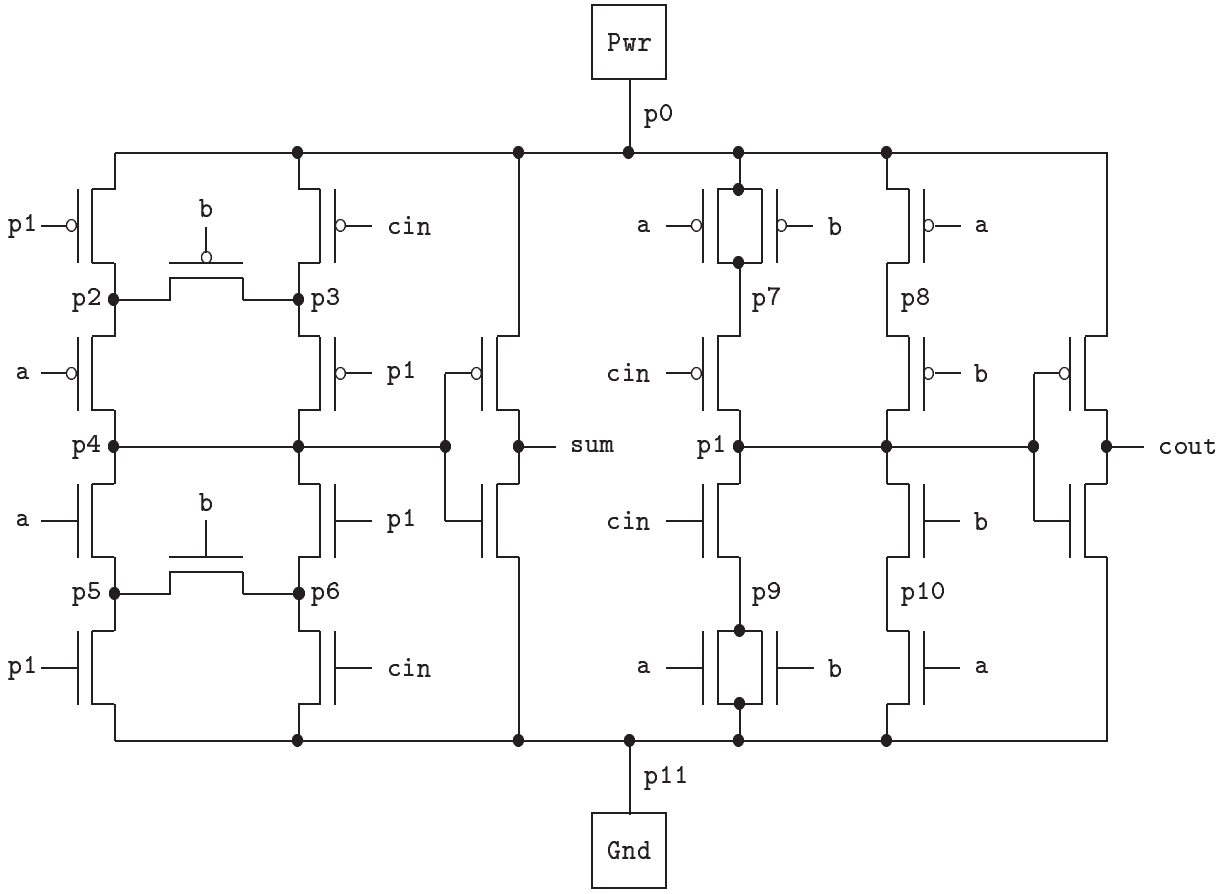
The specification of the adder can then be defined by:

$$\begin{aligned}&\mathsf{Add1}(a, b, cin, sum, cout) \;\equiv\\ &(2 \times \mathsf{Bit\_Val}(cout) + \mathsf{Bit\_Val}(sum) =\\ &\;\mathsf{Bit\_Val}(a) + \mathsf{Bit\_Val}(b) + \mathsf{Bit\_Val}(cin))\end{aligned}$$

A correct implementation of this specification is a circuit with lines `a`, `b`, `cin`, `sum` and `cout` such that the constraints imposed on the values a, b, cin, sum and, $cout$ that can exist on these lines imply that $\mathsf{Add1}(a, b, cin, sum, cout)$ always holds.

6.2 Implementation

A CMOS implementation of the adder is given below. Lines with the same name are connected. The lines `p0`, ... , `p11` are internal. The two transistors drawn horizontally function bidirectionally.



This circuit can be represented in logic by defining:

$$\begin{aligned}
&\mathsf{Add1_Imp}(a, b, cin, sum, cout) \equiv \\
&\exists p_0\ p_1\ p_2\ p_3\ p_4\ p_5\ p_6\ p_7\ p_8\ p_9\ p_{10}\ p_{11}. \\
&\quad \mathsf{Ptran}(p_1, p_0, p_2) \wedge \mathsf{Ptran}(cin, p_0, p_3) \wedge \mathsf{Ptran}(b, p_2, p_3) \wedge \mathsf{Ptran}(a, p_2, p_4) \wedge \\
&\quad \mathsf{Ptran}(p_1, p_3, p_4) \wedge \mathsf{Ntran}(a, p_4, p_5) \wedge \mathsf{Ntran}(p_1, p_4, p_6) \wedge \mathsf{Ntran}(b, p_5, p_6) \wedge \\
&\quad \mathsf{Ntran}(p_1, p_5, p_{11}) \wedge \mathsf{Ntran}(cin, p_6, p_{11}) \wedge \mathsf{Ptran}(a, p_0, p_7) \wedge \mathsf{Ptran}(b, p_0, p_7) \wedge \\
&\quad \mathsf{Ptran}(a, p_0, p_8) \wedge \mathsf{Ptran}(cin, p_7, p_1) \wedge \mathsf{Ptran}(b, p_8, p_1) \wedge \mathsf{Ntran}(cin, p_1, p_9) \wedge \\
&\quad \mathsf{Ntran}(b, p_1, p_{10}) \wedge \mathsf{Ntran}(a, p_9, p_{11}) \wedge \mathsf{Ntran}(b, p_9, p_{11}) \wedge \mathsf{Ntran}(a, p_{10}, p_{11}) \wedge \\
&\quad \mathsf{Pwr}(p_0) \wedge \mathsf{Ptran}(p_4, p_0, sum) \wedge \mathsf{Ntran}(p_4, sum, p_{11}) \wedge \\
&\quad \mathsf{Gnd}(p_{11}) \wedge \mathsf{Ptran}(p_1, p_0, cout) \wedge \mathsf{Ntran}(p_1, cout, p_{11})
\end{aligned}$$

6.3 Verification

To verify that the implementation Add1_Imp correctly implements the specification Add1, it must be proved that if a, b, cin, sum and $cout$ satisfy the constraints imposed by the implementation, then they also satisfy the specification. Formally:

$$\mathsf{Add1\_Imp}(a, b, cin, sum, cout) \supset \mathsf{Add1}(a, b, cin, sum, cout)$$

The most straightforward way to prove this implication is to consider separately the eight possible input combinations.

Manipulations like those given for the inverter yield the following eight facts:

$$\begin{aligned}
\mathsf{Add1_Imp}(\mathsf{T},\mathsf{T},\mathsf{T},sum,cout) &\equiv (sum{=}\mathsf{T}) \wedge (cout{=}\mathsf{T}) \\
\mathsf{Add1_Imp}(\mathsf{T},\mathsf{T},\mathsf{F},sum,cout) &\equiv (sum{=}\mathsf{F}) \wedge (cout{=}\mathsf{T}) \\
\mathsf{Add1_Imp}(\mathsf{T},\mathsf{F},\mathsf{T},sum,cout) &\equiv (sum{=}\mathsf{F}) \wedge (cout{=}\mathsf{T}) \\
\mathsf{Add1_Imp}(\mathsf{T},\mathsf{F},\mathsf{F},sum,cout) &\equiv (sum{=}\mathsf{T}) \wedge (cout{=}\mathsf{F}) \\
\mathsf{Add1_Imp}(\mathsf{F},\mathsf{T},\mathsf{T},sum,cout) &\equiv (sum{=}\mathsf{F}) \wedge (cout{=}\mathsf{T}) \\
\mathsf{Add1_Imp}(\mathsf{F},\mathsf{T},\mathsf{F},sum,cout) &\equiv (sum{=}\mathsf{T}) \wedge (cout{=}\mathsf{F}) \\
\mathsf{Add1_Imp}(\mathsf{F},\mathsf{F},\mathsf{T},sum,cout) &\equiv (sum{=}\mathsf{T}) \wedge (cout{=}\mathsf{F}) \\
\mathsf{Add1_Imp}(\mathsf{F},\mathsf{F},\mathsf{F},sum,cout) &\equiv (sum{=}\mathsf{F}) \wedge (cout{=}\mathsf{F})
\end{aligned}$$

Deriving these equations is equivalent to exhaustive simulation for all input values and is best done by computer. It follows from these eight equations that:

$$\mathsf{Add1\_Imp}(a,b,cin,sum,cout) \equiv \mathsf{Add1}(a,b,cin,sum,cout)$$

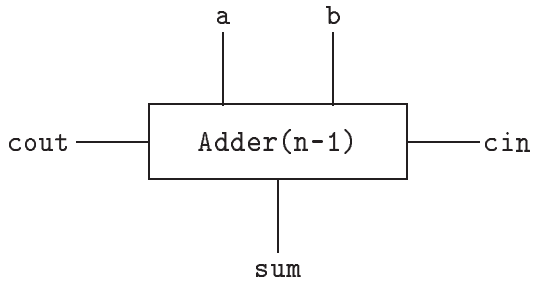
This entails the implication we want, since if $t_1 \equiv t_2$ then $t_1 \supset t_2$ holds *a fortiori*.

7 An n-bit adder

The example in this section is intended to illustrate the use of higher-order logic to represent parameterized systems. An n-bit adder computes an n-bit sum and 1-bit carry-out from two n-bit inputs and a 1-bit carry-in. Each value of the parameter n determines an adder operating on words of size $n{+}1$ ("$n{+}1$" because n counts from 0; *e.g.* if n is 0 we get a 1-bit adder).

7.1 Specification

Here is a diagram of an n-bit adder:



The lines `cin` and `cout` carry 1-bit words and the lines `a`, `b` and `sum` carry n-bit words. We will model 1-bit words as booleans and n-bit words as functions from natural numbers to booleans. Thus the 4-bit word 1101 is represented by a function f such that $f(0)$=T, $f(1)$=F, $f(2)$=T and $f(3)$=T. To relate words to numbers we use a function Val such that $\mathsf{Val}(n, f)$ is the number denoted by the $(n{+}1)$-bit word $f(n)f(n{-}1)\cdots f(0)$. Val can be defined by *primitive recursion*.

The primitive recursive definition of Val consists of two parts, a *basis* which specifies $\mathsf{Val}(0, f)$ and a *recursion* which specifies $\mathsf{Val}(n{+}1, f)$ in terms of $\mathsf{Val}(n, f)$. The basis is:

$$\mathsf{Val}(0, f) = \mathsf{Bit\_Val}(f(0))$$

and the recursion is:

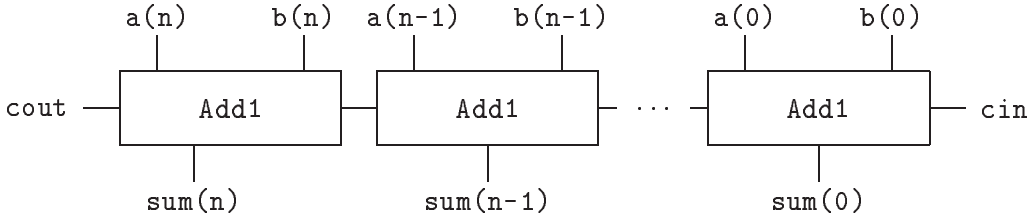
$$\mathsf{Val}(n{+}1, f) = 2^{n+1} \times \mathsf{Bit\_Val}(f(n{+}1)) + \mathsf{Val}(n, f)$$

To specify an n-bit adder we will define a higher-order function Adder which when applied to the number $n{-}1$ yields a predicate specifying the adder. Thus, for example, Adder(3) is a predicate specifying a 4-bit adder. The definition of Adder is:

$$\begin{aligned}&\mathsf{Adder}(n)(a, b, cin, sum, cout) \;\equiv\\ &(2^{n+1} \times \mathsf{Bit\_Val}(cout) + \mathsf{Val}(n, sum) =\\ &\;\mathsf{Val}(n, a) + \mathsf{Val}(n, b) + \mathsf{Bit\_Val}(cin))\end{aligned}$$

7.2 Implementation

An n-bit adder can be built by connecting together n full adders. The diagram below shows an $(n{+}1)$-bit adder. The inputs are a single bit carry-in `cin` and two $(n{+}1)$-bit words `a(n)a(n-1)...a(0)` and `b(n)b(n-1)...b(0)`. The outputs are an $(n{+}1)$-bit sum `sum(n)sum(n-1)...sum(0)` and a 1-bit carry-out `cout`.



To express this diagram in logic we define $\mathsf{Adder\_Imp}(n)(a, b, cin, out, cout)$ where Adder_Imp is a higher-order function which when applied to a number n yields the predicate specifying the implementation of an $n{+}1$-bit adder.

Two logically equivalent definitions of Adder_Imp are given below. The first one is a simple primitive recursive definition. The second one is intended to mimic how the adder might be described in a commercial hardware description language like MODEL [Lattice]. This second definition illustrates the claim that pure logic has all the expressive power found in special purpose languages.

7.2.1 Recursive description of the adder circuit

A primitive-recursive definition of Adder_Imp corresponding to the above diagram has the following basis:

$$\mathsf{Adder\_Imp}(0)(a, b, cin, sum, cout) \equiv \mathsf{Add1}(a(0), b(0), cin, sum(0), cout)$$

The recursive part of the definition says that an $(n{+}2)$-bit adder is built by first building an $n{+}1$-bit adder and then connecting its carry-out to the carry-in of a 1-bit adder.

$$\begin{aligned}&\mathsf{Adder\_Imp}(n{+}1)(a, b, cin, sum, cout) \equiv\\ &\quad \exists c.\ \mathsf{Adder\_Imp}(n)(a, b, cin, sum, c) \wedge\\ &\qquad \mathsf{Add1}(a(n{+}1), b(n{+}1), c, sum(n{+}1), cout)\end{aligned}$$

7.2.2 Iterative description of the adder circuit

To indicate the expressiveness of higher-order logic we will show how to mimic the following 'part specification' written in MODEL.

```
PART Adder_Imp (n) [a(0:n),b(0:n),cin] -> sum(0:n),cout
 SIGNAL c(0:n+1)
 INTEGER i
 cin -> c(0)
 FOR i = 0:n CYCLE
  Add1[a(i),b(i),c(i)] -> sum(i),c(i+1)
 REPEAT
 c(n+1) -> cout
END
```

Here is a line by line explanation of this code.

1. `PART Adder_Imp (n) [a(0:n),b(0:n),cin] -> sum(0:n),cout`

   A part called `Adder_Imp` is being specified.

   (a) `(n)`

   indicates that the part is parameterized on a number `n`.

   (b) `[a(0:n),b(0:n),cin]`

   indicates that `Adder_Imp` has two `(n+1)`-bit inputs called `a` and `b`, and one single bit input called `cin`.

   (c) `-> sum(0:n),cout`

   indicates that `Adder_Imp` has an `(n+1)`-bit output called `out` and a `1`-bit output called `cout`.

2. `SIGNAL c(0:n+1)`

   declares a 'local' `(n+1)`-bit line called `c`.

3. `INTEGER i`

   declares a local integer-valued variable called `i`.

4. `cin -> c(0)`

   specifies that `cin` be connected to the 0^{th} bit of `c`.

5. `FOR i = 0:n CYCLE`

   starts an iteration in which the body of the iteration (see below) is executed with `i` successively taking values `0`, `1`, ... , `n`.

6. `Add1[a(i),b(i),c(i)] -> sum(i),c(i+1)`

   is the body of the iteration; it specifies an instance of `Add1` having as inputs the $\texttt{i}^{\text{th}}$ bits of `a`, `b` and `c` and as outputs the $\texttt{i}^{\text{th}}$ bit of `sum` and the $\texttt{i+1}^{\text{th}}$ bit of `c`.

7. `REPEAT`

   ends the iteration.

8. `c(n+1) -> cout`

   specifies that the $\texttt{n+1}^{\text{th}}$ bit of `c` be connected to `cout`.

9. `END`

   ends the specification of `Adder_Imp`.

A transcription of the MODEL part specification into higher-order logic is:

$$\begin{array}{l}\mathsf{Adder\_Imp}(n)(a,b,cin,sum,cout) \;\equiv\\ \exists c.\\ \quad cin = c(0) \;\wedge\\ \quad \mathsf{Iterate}\ (0,n)\ (\lambda i.\ \mathsf{Add1}(a(i),b(i),c(i),sum(i),c(i{+}1))) \;\wedge\\ \quad c(n{+}1) = cout\end{array}$$

Note that in logic:

- a 'part specification' is just the definition of a function,
- a 'signal declaration' is an existential quantification,
- a wiring specification (*e.g.* `cin -> c(0)`) is an equation, and
- an iteration is just an invocation of the higher-order function Iterate.

No *ad hoc* hardware description constructs are needed, pure logic is enough.

7.3 Verification

To verify the adder one proves by induction on n that:

$$\mathsf{Adder\_Imp}(n)(a,b,cin,sum,cout) \;\supset\; \mathsf{Adder}(n)(a,b,cin,sum,cout)$$

The basis of the induction is:

$$\mathsf{Adder\_Imp}(0)(a,b,cin,sum,cout) \;\supset\; \mathsf{Adder}(0)(a,b,cin,sum,cout)$$

This is easily proved by substituting the definitions of Adder_Imp and Adder into the above implication and then reducing the resulting expression to a tautology. The induction step is:

$$\begin{array}{l}(\mathsf{Adder\_Imp}(n)(a,b,cin,sum,cout) \;\supset\; \mathsf{Adder}(n)(a,b,cin,sum,cout))\\ \supset\\ (\mathsf{Adder\_Imp}(n{+}1)(a,b,cin,sum,cout) \;\supset\; \mathsf{Adder}(n{+}1)(a,b,cin,sum,cout))\end{array}$$

This can be proved by simple arithmetic. A correctness proof of the recursively specified adder has been generated by Albert Camilleri using the HOL system.

8 Sequential Devices

All of the examples so far have been combinational; *i.e.* the values on the outputs have only depended on the current input values, not on input values at past times. Sequential devices can be modelled by taking the values on lines to be functions of time. For example, a unit-delay element `Del`, with input line `i` and output line `o`, is modelled by specifying that the value output at time $t+1$ equals the value input at time t. This is expressed in higher-order logic by:

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

Combinational devices can be modelled as sequential devices having no delay. To illustrate this, recall the specification of the adder:

$$\begin{array}{l}\mathsf{Adder}(n)(a, b, cin, sum, cout) \;\equiv \\ (2^{n+1} \times \mathsf{Bit\_Val}(cout) + \mathsf{Val}(n, sum) = \\ \quad \mathsf{Val}(n, a) + \mathsf{Val}(n, b) + \mathsf{Bit\_Val}(cin))\end{array}$$

The variables a, b and sum range over words (modelled as functions) and the variables cin and $cout$ range over truth-values. To model the adder as a zero-delay sequential device we must represent its behaviour with a predicate whose arguments are functions of time.

$$\begin{array}{l}\mathsf{Combinational\_Adder}(n)(a', b', cin', sum', cout') \;\equiv \\ \quad \forall t.\; \mathsf{Adder}(n)(a'(t), b'(t), cin'(t), sum'(t), cout'(t))\end{array}$$

The variables a', b' and sum' range over functions from time to words, and the variables cin' and $cout'$ range over functions from time to truth-values. Thus, for example, $out'(7)(5)$ is bit 5 of the word output at time 7. If we wanted to specify the adder as having a unit-delay then we could define:

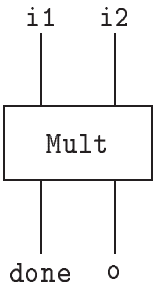
$$\begin{array}{l}\mathsf{Unitdelay\_Adder}(n)(a', b', cin', sum', cout') \;\equiv \\ \quad \forall t.\; \mathsf{Adder}(n)(a'(t), b'(t), cin'(t), sum'(t+1), cout'(t+1))\end{array}$$

9 A sequential multiplier

As an illustration of the specification and verification of a sequential device, we describe below a multiplier. To simplify details, it is (unrealistically) assumed that lines carry numbers rather than words. This enables us to use ordinary arithmetic operators in the specification.

9.1 Specification

The multiplier is specified to have two inputs and two outputs.



An informal specification of the required behaviour of `Mult` is:

If

- `done` has value T at time t_1, and
- t_2 is the first time after t_1 that `done` again has value T, and
- the values at `i1` and `i2` are stable from t_1 to t_2,

then

- the value at `o` at time t_2 is the product of the values at `i1` and `i2` at time t_1.

In order to formalize this in logic various temporal notions like "the first time after" and "stable" must be represented.

9.1.1 Some temporal predicates

The predicate Stable is defined so that $\mathsf{Stable}(t_1, t_2)(f)$ is true if and only if the value of f is constant from t_1 until just before time t_2. Formally:

$$\mathsf{Stable}(t_1, t_2)(f) \;\equiv\; \forall t.\; t_1 \leq t \;\wedge\; t < t_2 \;\supset\; (f(t){=}f(t_1))$$

The predicate Next is defined so that $\mathsf{Next}(t_1, t_2)(f)$ is true if and only if t_2 is the first time after t_1 that $f(t_2)$=T. Formally:

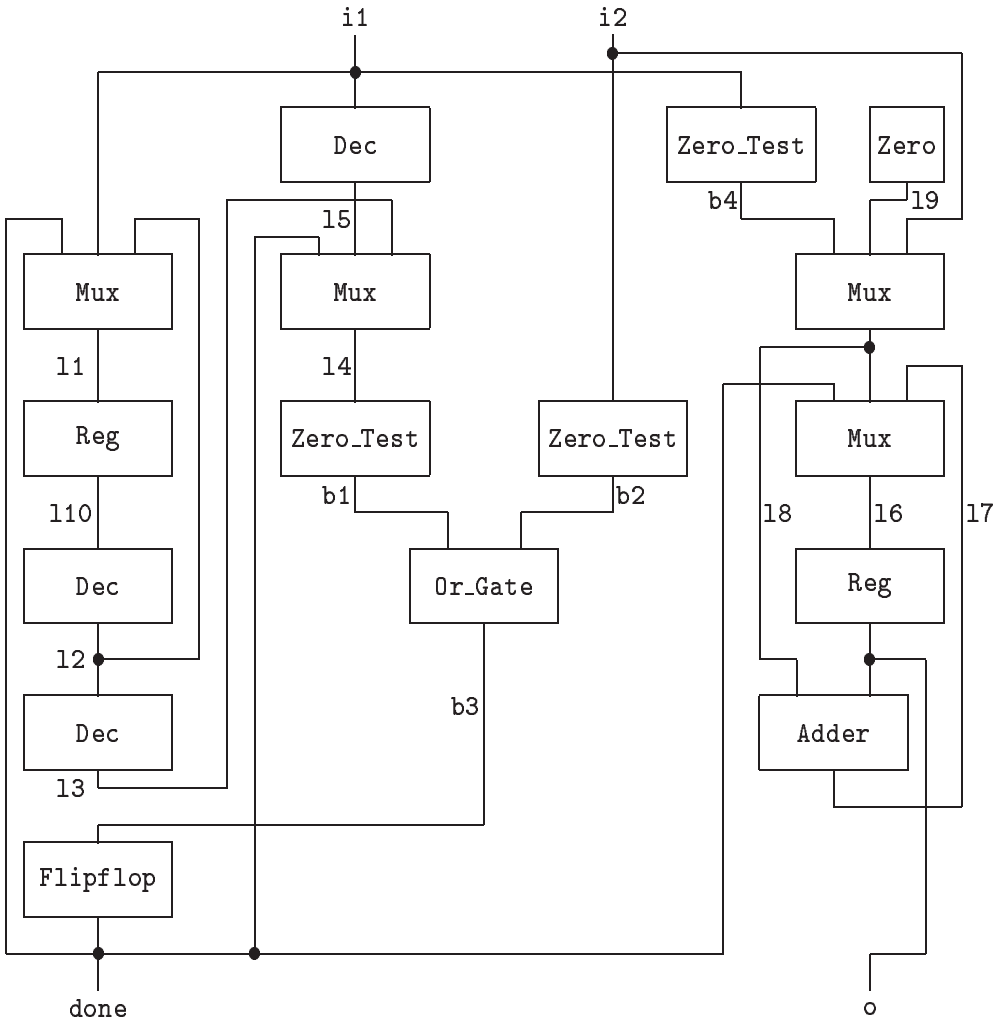
$$\mathsf{Next}(t_1, t_2)(f) \equiv t_1 < t_2 \wedge f(t_2) \wedge (\forall t.\ t_1 < t \wedge t < t_2 \supset \neg f(t))$$

Using Stable and Next, the specification of `Mult` can be represented with the predicate Mult defined by:

$$\begin{array}{l}\mathsf{Mult}(i_1, i_2, o, done) \equiv \\ done(t_1) \wedge \\ \mathsf{Next}(t_1, t_2)(done) \wedge \\ \mathsf{Stable}(t_1, t_2)(i_1) \wedge \\ \mathsf{Stable}(t_1, t_2)(i_2) \wedge \\ \supset \\ (o(t_2) = i_1(t_1) \times i_2(t_1))\end{array}$$

9.2 Implementation

Here is a circuit which meets the specification $\mathsf{Mult}(i_1, i_2, o, done)$:



The components of this device are specified by:

$$\mathsf{Mux}(ctl, i_1, i_2, o) \equiv \forall t.\ o(t) = (ctl(t) \rightarrow i_1(t) \mid i_2(t))$$
$$\mathsf{Reg}(i, o) \equiv \forall t.\ o(t{+}1) = i(t)$$
$$\mathsf{Flipflop}(i, o) \equiv \forall t.\ o(t{+}1) = i(t)$$
$$\mathsf{Dec}(i, o) \equiv \forall t.\ o(t) = i(t){-}1$$
$$\mathsf{Add}(i_1, i_2, o) \equiv \forall t.\ o(t) = i_1(t) + i_2(t)$$
$$\mathsf{Zero\_Test}(i, o) \equiv \forall t.\ o(t) = (i(t){=}0)$$
$$\mathsf{Or\_Gate}(i_1, i_2, o) \equiv \forall t.\ o(t) = i_1(t) \vee i_2(t)$$
$$\mathsf{Zero}(o) \equiv \forall t.\ o(t) = 0$$

The behaviour corresponding to the diagram above is captured in logic as follows:

$$\mathsf{Mult\_Imp}(i_1, i_2, o, done) \equiv$$
$$\exists b_1\ b_2\ b_3\ b_4\ l_1\ l_2\ l_3\ l_4\ l_5\ l_6\ l_7\ l_8\ l_9\ l_{10}.$$

$$\begin{aligned}
&\mathsf{Mux}(done, l_8, l_7, l_6) \ \wedge\ \mathsf{Reg}(l_6, o) \ \wedge\ \mathsf{Add}(l_8, o, l_7) \ \wedge\ \mathsf{Dec}(i_1, l_5) \ \wedge \\
&\mathsf{Mux}(done, l_5, l_3, l_4) \ \wedge\ \mathsf{Mux}(done, i_1, l_2, l_1) \ \wedge\ \mathsf{Reg}(l_1, l_{10}) \ \wedge \\
&\mathsf{Dec}(l_{10}, l_2) \ \wedge\ \mathsf{Dec}(l_2, l_3) \ \wedge\ \mathsf{Zero}(l_9) \ \wedge\ \mathsf{Mux}(b_4, l_9, i_2, l_8) \ \wedge \\
&\mathsf{Zero_Test}(i_1, b_4) \ \wedge\ \mathsf{Zero_Test}(l_4, b_1) \ \wedge\ \mathsf{Zero_Test}(i_2, b_2) \ \wedge \\
&\mathsf{Or_Gate}(b_1, b_2, b_3) \ \wedge\ \mathsf{Flipflop}(b_3, done)
\end{aligned}$$

9.3 Verification

The correctness of the multiplier implementation is established by proving that for all values of i_1, i_2, o and $done$:

$$\mathsf{Mult\_Imp}(i_1, i_2, o, done) \ \supset\ \mathsf{Mult}(i_1, i_2, o, done)$$

Expanding the definition of Mult and then slightly rearranging the result yields:

$$\begin{aligned}
\forall t_1\ t_2.\ &\mathsf{Mult_Imp}(i_1, i_2, o, done) \ \wedge \\
&done(t_1) \ \wedge \\
&\mathsf{Next}(t_1, t_2)(done) \ \wedge \\
&\mathsf{Stable}(t_1, t_2)(i_1) \ \wedge \\
&\mathsf{Stable}(t_1, t_2)(i_2) \ \wedge \\
&\supset \\
&(o(t_2) = i_1(t_1) \times i_2(t_1))
\end{aligned}$$

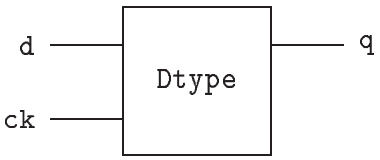
This can be proved by mathematical induction on $t_2 - t_1$. The proof is mostly routine, but there are a few slightly tricky bits. Some elementary results concerning $+$ and $\times$ are required, together with the following lemmas about time:

$$\begin{aligned}
&f(t{+}1) \ \supset\ \mathsf{Next}(t, t{+}1)(f) \\
&\mathsf{Next}(t_1, t_2)(f) \ \wedge\ \neg f(t_1{+}1) \ \supset\ \mathsf{Next}(t_1{+}1, t_2)(f) \\
&\mathsf{Next}(t_1, t_2)(f) \ \wedge\ \mathsf{Next}(t_1, t_3)(f) \ \supset\ (t_2{=}t_3) \\
&\mathsf{Next}(t, (t{+}1){+}d)(f) \ \wedge\ \neg f(t{+}1) \ \supset\ \neg(d{=}0) \\
&\mathsf{Stable}(t_1, t_2)(f) \ \supset\ \mathsf{Stable}(t_1{+}1, t_2)(f) \\
&\mathsf{Stable}(t, (t{+}1) + d)(f) \ \wedge\ \neg(d{=}0) \ \supset\ (f(t) = f(t{+}1))
\end{aligned}$$

We do not give details of the proof here. It was not difficult to generate using the HOL system. Tom Melham has done a similar example: a device for computing the factorial function. In the process he discovered some general principles for reasoning about self-timed devices. Details will appear in a forthcoming paper.

10 An edge-triggered Dtype

The implementation of the multiplier described in the preceding section was described at the register-transfer level. This is an abstract level in which devices are viewed as sequential machines. At this level registers are modelled as unit-delay elements without explicit clock lines. To implement such a register using actual hardware, something like a Dtype device must be used:



10.1 Specification

An informal behavioural specification of `Dtype` is:

If

- the clock `ck` has a rising edge at time t_1, and
- the next rising edge of `ck` is at t_2, and
- the value at `d` is stable for c_1 units of time before t_1 (c_1 is the setup time), and
- there are at least c_2 units of time between t_1 and t_2 (c_2 is the minimum clock period),

then

- the value at `q` will be stable from c_3 units of time after t_1 (c_3 is the start time) until c_4 units of time after t_2 (c_4 is the finish time), and
- the value at `q` between the start and finish times will equal the value held stable at `d` during the setup time.

To formalize this we need to define what a "rising edge" is. We will continue to use a discrete model of time, but the grain of time will be finer than before. A function from time to truth-values is defined to rise at time t if it is F at time $t-1$ and T at t. Formally:

$$\mathsf{Rise}(f)(t) \;\equiv\; (f(t-1) = \mathsf{F}) \;\wedge\; (f(t) = \mathsf{T})$$

If the function Rise is applied to a single argument f, then the resulting expression $\mathsf{Rise}(f)$ denotes a predicate that is true of t if and only if f rises at t. The specification of the Dtype below illustrates the use of this kind of 'partial application'.

The informal behavioural specification of a Dtype can now be formalized in logic by:

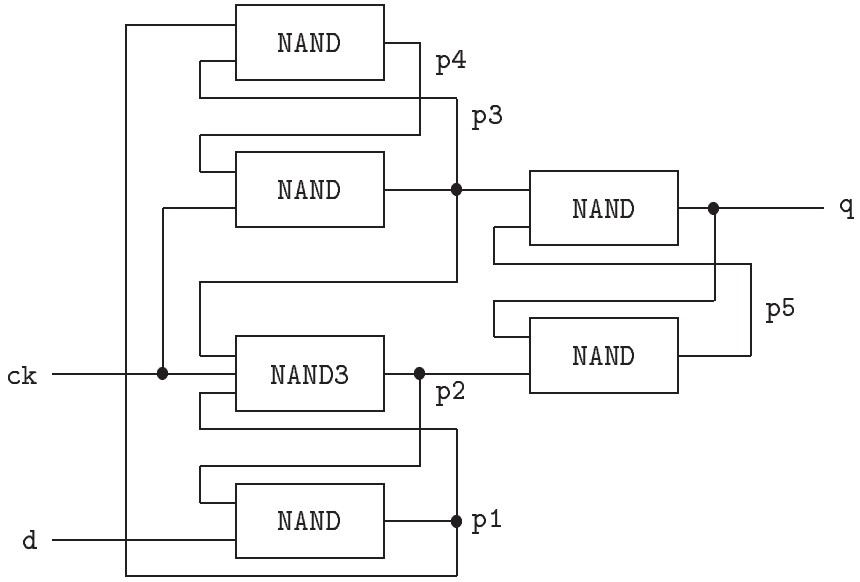
$$
\begin{array}{l}
\mathsf{Dtype}(\mathsf{c}_1, \mathsf{c}_2, \mathsf{c}_3, \mathsf{c}_4)(d, ck, q) \;\equiv \\
\quad \forall t_1\ t_2.\ \mathsf{Rise}(ck)(t_1) \wedge \\
\qquad\qquad \mathsf{Next}(t_1, t_2)(\mathsf{Rise}(ck)) \wedge \\
\qquad\qquad (t_2 - t_1 > \mathsf{c}_2) \wedge \\
\qquad\qquad \mathsf{Stable}(t_1 - \mathsf{c}_1, t_1)(d) \\
\qquad\qquad \supset \\
\qquad\qquad (\mathsf{Stable}(t_1 + \mathsf{c}_3, t_2 + \mathsf{c}_4)(q) \;\wedge\; (q(t_2) = d(t_1)))
\end{array}
$$

The parameters c_1, c_2, c_3 and c_4 are the timing constants of the Dtype; their value depends on how the device is fabricated. Note that $\mathsf{Next}(t_1, t_2)(\mathsf{Rise}(ck))$ is an expression formed by applying $\mathsf{Next}(t_1, t_2)$ to the predicate $\mathsf{Rise}(ck)$.

A Dtype becomes a unit-delay if we abstract signals to the sequence of values occurring at rising edges of the clock. The formal analysis of such abstractions is currently being studied by Tom Melham as part of his Ph.D. research.

10.2 Implementation

A common implementation of a Dtype uses NAND-gates:



10.3 Verification

To show that this implementation works we must use a model in which the NAND-gates have delay, since it is the delay in feedback loops that provides memory. The simplest such model is one in which each gate has unit-delay:

$$\mathsf{NAND}(i_1, i_2, o) \;\equiv\; \forall t.\; o(t{+}1) = \neg(i_1(t) \wedge i_2(t))$$
$$\mathsf{NAND3}(i_1, i_2, i_3, o) \;\equiv\; \forall t.\; o(t{+}1) = \neg(i_1(t) \wedge i_2(t) \wedge i_3(t))$$

The Dtype implementation can be represented in logic by defining:

$$\begin{aligned}&\mathsf{Dtype\_Imp}(d, ck, q) \;\equiv\\ &\exists p_1\; p_2\; p_3\; p_4\; p_5.\\ &\quad \mathsf{NAND}(p_2, d, p_1) \;\wedge\; \mathsf{NAND3}(p_3, ck, p_1, p_2) \;\wedge\\ &\quad \mathsf{NAND}(p_4, ck, p_3) \;\wedge\; \mathsf{NAND}(p_1, p_3, p_4) \;\wedge\\ &\quad \mathsf{NAND}(p_3, p_5, q) \;\wedge\; \mathsf{NAND}(q, p_2, p_5)\end{aligned}$$

It can then be proved that:

$$\mathsf{Dtype\_Imp}(d, ck, q) \;\supset\; \mathsf{Dtype}(2, 3, 4, 1)(d, ck, q)$$

This shows that if each NAND-gate has unit-delay then the Dtype has a setup time of 2, a minimum clock period of 3, a start time of 4 and a finish time of 1. The formal proof of this is fairly complicated. It has been done by hand by John Herbert.

11 Conclusions

The examples presented here demonstrate that higher-order logic is a formalism in which a wide variety of behaviour and structure can be specified. With the aid of suitable functions (*e.g.* Iterate), specifications can be made to look like conventional hardware descriptions whilst retaining logical purity (and hence formal tractability).

Hardware verification requires various kinds of reasoning.

- The adder example shows the need for mathematical induction (both to deal with iterated structures and for proving arithmetic lemmas).
- The multiplier example shows the need for reasoning about temporal concepts (Next, Stable *etc.*).
- The Dtype and unit-delay show the need for reasoning about abstractions between different time scales.

All these kinds of reasoning can be done using the standard inference rules of logic.

12 Acknowledgements

The use of higher-order logic for hardware specification and verification has been pioneered by Keith Hanna. Many of the techniques presented here have been adapted from Ben Moszkowski's work on applying temporal logic to hardware description. I learnt from him the representation of circuits as predicates described in Section 4. I have had numerous useful discussions with the users of the HOL system. These include Albert Camilleri, Nives Chaplin, Inder Dhingra, John Herbert, Tom Melham and Edmund Ronald, all from Cambridge, and Jeff Joyce from the University of Calgary. Don Gaubatz, Andy Hopper and Peter Robinson patiently explained to me various electrical phenomena that arise in MOS circuits.

Avra Cohn, Inder Dhingra and John Herbert pointed out errors in a first draft of this paper.

13 References

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

[**Clocksin**] W. F. Clocksin. *Logic Programming and the Specification of Circuits.* Computer Laboratory Technical Report No. 72, 1985.

[**Gordon83**] M. J. C. Gordon. *LCF_LSM.* University of Cambridge Computer Laboratory Technical Report No. 41, 1983.

[**Gordon85(a)**] M. J. C. Gordon. *HOL: A Machine Oriented Formulation of Higher-Order Logic.* University of Cambridge Computer Laboratory Technical Report No. 68, 1985.

[**Gordon85(b)**] M. J. C. Gordon. *HOL: A Proof Generating System for Higher-Order Logic.* Forthcoming technical report.

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

[**Hanna & Daeche**] F. K. Hanna and N. Daeche. *Specification and Verification using Higher-Order Logic.* Proceedings of the *7th International Conference on Computer Hardware Design Languages.* Tokyo, 1985.

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

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

[**Lattice**] *Designing with Gate Arrays.* Lattice Logic Limited, 9 Wemyss Place, Edinburgh EH3 6DH, 1982.

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