Technical Report

UCAM-CL-TR-74
ISSN 1476-2986

Number 74

UNIVERSITY OF CAMBRIDGE
Computer Laboratory

Hardware verification by formal proof

Mike Gordon

August 1985

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

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

ISSN 1476-2986

August 19, 1985

Hardware Verification by Formal Proof

Mike Gordon
Computer Laboratory
Corn Exchange Street
Cambridge CB2 3QG

Abstract

The use of mathematical proof to verify hardware designs is explained and motivated. The hierarchical verification of a simple n-bit CMOS counter is used as an example. Some speculations are made about when and how formal proof will become used in industry.

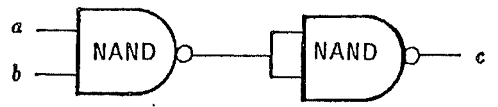
Hardware Verification by Formal Proof

The last twenty years have seen considerable progress in getting computers to generate mathematical proofs. Some of the resulting techniques are now being successfully applied to the problem of verifying that hardware designs meet their specifications.

What is verification by formal proof?

The idea of hardware verification by formal proof is not new. A traditional example is the use of Boolean algebra (see Figure 1).

A formal proof consists of a sequence of lines each of which is either a hypothesis or derived from previous lines. The final line of the proof is the theorem proved. As an example, consider the following circuit:



Below is a formal proof that the specification $c = a \wedge b$ is correctly implemented by this circuit. (The justification of each line is given in square brackets.)

1. $c = \mathsf{NAND}(\mathsf{NAND}(a,b), \mathsf{NAND}(a,b))$ [From the above circuit]
2. $\mathsf{NAND}(a,b) = \neg(a \wedge b)$ [Definition of NAND]
3. $c = \neg((\neg(a \wedge b)) \wedge (\neg(a \wedge b)))$ [Substituting 2 into 1]
4. $c = \neg(\neg(a \wedge b))$ [By 3 and the law $x \wedge x = x$]
5. $c = a \wedge b$ [By 4 and the law $\neg\neg x = x$]

Figure 1: **A formal proof using Boolean algebra**

The important difference between conventional hardware description languages and mathematical formalisms like Boolean algebra is that the latter support formal reasoning.

A design methodology employing formal verification entails:

1. Writing a high-level specification (sometimes called a "requirements specification").
2. Designing an implementation (*e.g.* a circuit diagram).
3. Proving mathematically that the design meets its specification.

In the example in Figure 1, the high-level specification is $c = a \wedge b$, the implementation uses two NAND-gates connected as in the diagram, and the proof consists of the five steps shown. The proof in this example could be done on the back of an envelope, but proofs of real devices can be thousands of lines long and the only feasible way of generating them is by computer.

Although the language of Boolean functions is adequate for specifying simple combinational circuits, it is not really suitable for more complicated systems which use a variety of data types (words, numbers, *etc.*) and have time dependent behaviour.

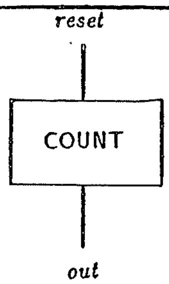
To make verification by proof feasible for real systems it is thus necessary to provide two things:

1. A high-level mathematical formalism for writing specifications.
2. Tools for mechanizing the production of correctness proofs.

A computer system that provides these is VERIFY developed by Harry Barrow of Schlumberger Palo Alto Research. VERIFY has been used to prove correct quite complex devices including an arithmetic unit containing over 18,000 transistors.

Using predicate logic for specification and verification

Predicate logic is one of several formal systems being investigated as a basis for specification and verification. Its use is illustrated in Figures 2 to 7. Figure 2 shows a behavioural specification of an n-bit counter in a version of predicate logic called higher-order logic. (See Figure 3 for the meaning of the notation used.)



$$\mathsf{COUNT}(n)(reset, out) \equiv$$
$$\forall t.\ \mathsf{VAL}(n)(out(t+1)) = (reset(t) \rightarrow 0 \mid (\mathsf{VAL}(n)(out(t))+1)\mathsf{MOD}2^{n+1})$$

Figure 2: Specification of an n-bit counter

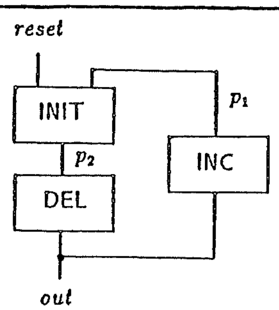
Functions like COUNT in Figure 2 take a sequence of arguments. COUNT itself denotes a function which when applied to a number n yields a predicate COUNT(n) representing the behaviour of an n-bit counter. The predicate COUNT(n) can then be applied to a pair $(reset, out)$ to yield a truth value (*i.e.* T or F). In this example *reset* and *out* are functions from times (represented by numbers) to words. Such functions represent possible 'histories of values' occurring at the input and output of the counter. For example, $out(5)$ represents the word output by the counter at time 5, and $out(5)(3)$ is bit 3 of this word (time and bit positions are counted from 0, so bit 3 is the fourth bit, *etc.*).

- "$P \wedge Q$" means "P and Q".
- "$P \supset Q$" means "P implies Q".
- "$P \equiv Q$" means "P if and only if Q" (*i.e.* $(P \supset Q) \wedge (Q \supset P)$).
- "$\forall t.\ P[t]$" means "$P[t]$ is true for all values of t".
- "$\exists t.\ P[t]$" means "$P[t]$ is true for some value of t".
- "$(P \rightarrow t_1 \mid t_2)$" equals t_1 if P=T and equals t_2 if P=F.
- "$\mathsf{VAL}(n)f$" denotes the number represented by the $(n+1)$-bit word whose i^{th} bit is $f(i)$ (where i runs from 0 to n).
- "$\mathsf{WORD}(n)m$" is defined by the equation $\mathsf{VAL}(n)(\mathsf{WORD}(n)m) = m$.
- "m MOD n" denotes the remainder after dividing n into m.

Figure 3: Notation used in the examples

The idea of the specification in Figure 2 is that $\mathsf{COUNT}(n)(reset, out)$ should equal T if and only if *reset* and *out* correspond to possible histories of values occurring at the input and output of the counter — *i.e.* if the number denoted by the word occurring at *out* at time $t+1$ is one plus the value at *out* at time t (except that if *reset* is asserted at time t then 0 is output at *out* at $t+1$).

Figure 4 shows a diagram of an implementation of the counter, together with behavioural specifications of its components.



$$\mathsf{INIT}(n)(reset, p_1, p_2) \equiv \forall t.\ p_2(t) = (reset(t) \rightarrow \mathsf{WORD}(n)0 \mid p_1(t))$$

$$\mathsf{DEL}(p_2, out) \equiv \forall t.\ out(t+1) = p_2(t)$$

$$\mathsf{INC}(n)(out, p_1) \equiv \forall t.\ \mathsf{VAL}(n)(p_1(t)) = (\mathsf{VAL}(n)(out(t)) + 1)\mathsf{MOD}2^{n+1}$$

Figure 4: An implementation of the counter

This implementation is specified at the register-transfer level, a level at which clock lines are implicit. At a lower level the delay element DEL might be implemented by a clocked D-type register with an explicit clock line; setup and hold times would then have to be represented. This lower level is not elaborated here, but it can also be modelled in predicate logic.

In Figure 5 there is an outline proof that the implementation of the counter is correct.

1. Translating the diagram in Figure 4 into logic gives:

$$\exists p_1\ p_2.\ \mathsf{INIT}(n)(reset, p_1, p_2)\ \wedge\ \mathsf{DEL}(p_2, out)\ \wedge\ \mathsf{INC}(n)(out, p_1)$$

2. Substituting in the definitions of INIT, DEL and INC results in:

$$\begin{aligned}\exists p_1\ p_2.\ & (\forall t.\ p_2(t) = (reset(t) \rightarrow \mathsf{WORD}(n)0 \mid p_1(t))) \wedge \\ & (\forall t.\ out(t+1) = p_2(t)) \wedge \\ & (\forall t.\ \mathsf{VAL}(n)(p_1(t)) = (\mathsf{VAL}(n)(out(t)) + 1)\mathsf{MOD}2^{n+1})\end{aligned}$$

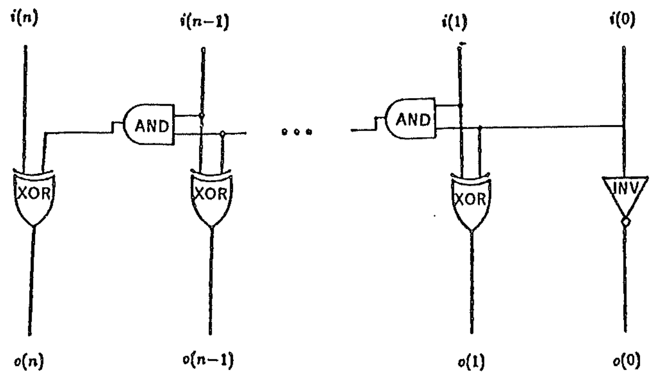
3. Routine formal manipulations yield:

$$\forall t.\ \mathsf{VAL}(n)(out(t+1)) = (reset(t) \rightarrow 0 \mid (\mathsf{VAL}(n)(out(t)) + 1)\mathsf{MOD}2^{n+1})$$

4. This is equal to $\mathsf{COUNT}(n)(reset, out)$, the behavioural specification of the counter in Figure 2.

Figure 5: Outline correctness proof of the counter implementation

Figure 6 shows an implementation of the incrementer INC. Mathematical induction on n can be used to formally prove that this implementation meets the behavioural specification of INC in Figure 4. The proof is too long to be shown here, but it is not difficult and is easily generated by computer.



This circuit can be represented in logic by defining:

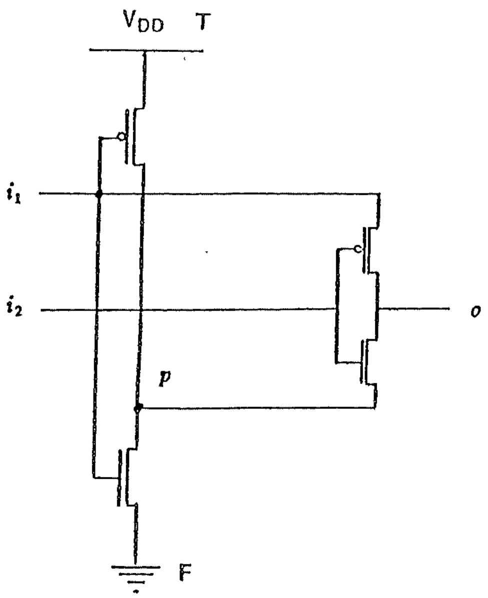
$$\begin{aligned}&\mathsf{INC\_IMP}(n)(i, o)\ \equiv \\ &((n{=}0) \rightarrow \mathsf{INV}(i(0), o(0)) \mid \\ &\qquad \exists c.\ \mathsf{XOR}(i(n), c, o(n)) \wedge \mathsf{INC\_SLICE}(n-1)(i, o, c))\end{aligned}$$

Where INC_SLICE is defined by:

$$\begin{aligned}&\mathsf{INC\_SLICE}(n)(i, o, cout)\ \equiv \\ &((n{=}0) \rightarrow \mathsf{INV}(i(0), o(0)) \wedge (cout = i(0)) \mid \\ &\qquad \exists c.\ \mathsf{XOR}(i(n), c, o(n) \wedge \mathsf{AND}(i(n), c, cout) \wedge \mathsf{INC\_SLICE}(n-1)(i, o, c))\end{aligned}$$

Figure 6: An implementation of INC

Finally, in Figure 7, a CMOS circuit implementing the exclusive-or gate used in Figure 6 is verified.



1. From the circuit diagram:

$$\exists p.\ \mathsf{PTRAN}(i_1, \mathsf{T}, p) \wedge \mathsf{NTRAN}(i_1, p, \mathsf{F}) \wedge \mathsf{PTRAN}(i_2, i_1, o) \wedge \mathsf{NTRAN}(i_2, o, p)$$

2. By the definition of NTRAN and PTRAN:

$$\exists p.\ (\neg i_1 \supset (p = \mathsf{T})) \wedge (i_1 \supset (p = \mathsf{F})) \wedge (\neg i_2 \supset (o = i_1)) \wedge (i_2 \supset (o = p))$$

3. Boolean algebra and laws for equality give:

$$\exists p.\ (p = \neg i_1) \wedge (o = \neg(i_1 = i_2))$$

4. Moving $\exists$ inwards yields:

$$(\exists p.\ p = \neg i_1) \wedge (o = \neg(i_1 = i_2))$$

5. Hence $o = \neg(i_1 = i_2)$.

Figure 7: **Formal verification of a CMOS exclusive-or gate**

Although the example outlined in Figures 2 to 7 is simple, it does illustrate an important point, namely that formal specification and verification can be done hierarchically.

When should formal verification be used?

Formal verification is expensive and currently it may only be only worthwhile for systems whose failure would result in disaster (*e.g.* loss of life, destruction of costly equipment, or recall of a mass produced product). Examples include aircraft control systems, nuclear reactor monitors, satellite systems, medical devices and chips in automobiles.

It has been suggested that aircraft that fly-by-wire should only get air worthiness certificates if critical parts of their control systems have been formally proved correct. Until recently this was not considered practical, but a group at RSRE have proved correct a complete processor. This establishes that real systems can be formally verified and so contractors will no longer be able to claim that doing it is impossible.

As designs get larger and more complicated, the cost of conventional verification methods appears to grow faster than the cost of formal methods. It might thus actually be cheaper to verify VLSI designs by formal proof than by standard CAD techniques like simulation. This is especially plausible for complex single chip systems containing a lot of hard wired logic (*e.g.* RISC machines).

Verification engineering

The first commercially available formal verification systems will probably handle simple designs fully automatically, but may require manual guidance for more complicated ones. For example, the proofs shown in Figures 1, 5 and 7 could be generated automatically, but the inductive proof of the circuit in Figure 6 might require manual guidance. A new kind of expert (a 'verification engineer') will be needed for the production of complex correctness proofs. Such people are likely to be in short supply and may well work on a consulting basis. System designers will verify in-house those parts of their designs which can be done automatically. The few complicated parts of the proof that require a specialist might then be contracted out to a 'verification shop'.

Current research

Several university groups in the UK are actively researching hardware specification and verification. Keith Hanna at the University of Kent is developing a system called VERITAS which can be used to mechanize the sort of proofs described in this article. At the University of Edinburgh, George Milne is working on a circuit calculus called CIRCAL. He has implemented some tools for animating formal specifications. CIRCAL is particularly appropriate for analysing low-level timing behaviour. At Cambridge, the hardware verification group is studying several formalisms including higher-order logic and temporal logic.

References

1. H. Barrow. *Proving the Correctness of Digital Hardware Designs.* VLSI Design, Volume 5, Number 7, July 1984.

2. 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.

3. G. J. Milne. CIRCAL*: A calculus for circuit description.* Integration, Volume 1, Numbers 2 & 3, October 1983.