A Mathematical Approach to RTL Verification

David M. Russinoff

July 5, 2007

Introduction

Ten years ago, in the wake of the Intel FDIV affair, I was hired by Advanced Micro Devices to verify the design of the floating-point unit of a chip that was later to be known as the AMD Athlon Processor. At that time, I knew very little about computer arithmetic or any other aspect of hardware design or verification. Moreover, as a theoretician with little experience in the industrial sector, I was entering a new world with strange customs, language, and culture. I could only hope that I had something useful to offer as the valorous mathematician who would create order out of chaos. There would be some interesting times ahead.

All that I brought with me to this new venture was some experience in the application of mechanical theorem proving to problems in arithmetic as well as software verification, and a general background in mathematics. So I had two questions to ponder as I investigated the nature of the problem of hardware verification. First, how similar is it to software verification? That is, how relevant are the established methodologies and conventional wisdom of program verification to the hardware problem? And second, to what extent is this a mathematical activity? How relevant are the principles and the culture of traditional mathematics? In short, what tools did I have for creating order out of the chaos of microprocessor design?

In this paper, I will submit some observations derived from my experience at AMD as I attempted to answer these questions, mainly in the limited context of arithmetic circuitry, and to describe a verification methodology that was developed in the process. Finally, I will briefly discuss prospects for extending this methodology to the broader domain of microprocessor design in general.

First, a disclaimer: when I refer to "hardware verification", I am speaking only of mechanical theorem proving. I have nothing to say about model checking, static analysis, symbolic trajectory evaluation, etc., simply because of my profound ignorance in these areas.

The Relevance of Program Verification

With regard to my first question, I was initially relieved to find that the designs to be verified took the form of software models, coded in a hardware description language that bore some resemblance to the programming languages with which I was familiar. So I had some hope that what little I knew about program verification would be of some use here.

Origins and Conventional Wisdom

My introduction to this field came in 1982 when I met Bob Boyer and J Moore at the University of Texas. Their work, as they explained to me, was derived from that of John McCarthy in the early '60s, which centered on the notion of *operational semantics*: the definition of a programming language by way of an abstract interpreter. McCarthy, of course, introduced the functional language LISP [21] as a vehicle for verification and a method that he called *recursion induction* [23] for proving properties of LISP functions. The Boyer-Moore prover, NQTHM, may be viewed as an implementation of this approach. The same is true of its successor, ACL2 [2], which is maintained by Moore and Matt Kaufmann and is the tool that I use in my work. ACL2 is both a functional programming language, essentially an applicative subset of Common LISP [31], and a first-order logic supported by a heuristic theorem prover based on mathematical induction.

There are, of course, a variety of competing approaches to the verification problem, but here I am less interested in their differences than in the precepts that are shared among them. Here are several factors that are commonly considered to be important for the success of a formal program verification effort, with regard to the problem, the solution, and the underlying formalism:

- A problem of limited size and complexity;
- A concise and unambiguous specification of correctness;
- Cooperative development of a program and its proof of correctness;
- A simple and elegant programming solution;
- A programming language with clear and simple semantics.

So, can these requirements reasonably be applied to the problem of hardware verification? For some of them, the question was easy to answer. A problem of limited size and complexity? When I received my first assignment, a floating-point multiplier consisting of half a megabyte of opaque RTL code, I knew that my experience verifying eight-line programs would be of little use to me here. On the other hand, however complex the implementation of an arithmetic operation may be, its external behavior may be described quite concisely in abstract arithmetic terms, as expressed by the IEEE Standard [17]. This, I would say, is a critical distinguishing feature of arithmetic circuitry that makes it especially suitable for formal verification.

Regarding the cooperative derivation of program and proof, it is generally held that in order to ensure that a program is susceptible to formal verification, it should be designed with that goal in mind. As David Gries puts it:

> A program and its proof should be developed hand-in-hand, with the proof usually leading the way. [12, p. 164]

Some doubt was cast on this ideal during my first week on the job, when I was asked this question by a floating-point designer:

> Do you think that we need some academic to tell us how to design a multiplier? [32]

It was true that I had some history in academia, but I felt that I had paid my debt to society and deserved a fresh start. But so much for the vision of verifier and designer strolling hand in hand. Fortunately, this turned out to be an extreme position—the engineers I've worked with been have generally been very cooperative and have taught me quite a bit. But even after ten years, design and verification remain very distinct activities. I am still utterly unqualified to write RTL code, just as those who do have little understanding of my work, and it would be absurd for me to suggest that they alter their practice in any way to suit me.

Consideration of the remaining two items on the list pointed to some interesting differences between software and hardware.

On Simple and Elegant Solutions

As Dijkstra observed, the susceptibility of a program to formal verification "is not purely a function of [its] external specification and behavior, but depends critically on its internal structure." [7, p. 5] I am particularly fond of Tony Hoare's version of this observation:

> There are two ways of constructing a software design. One way is to make it so simple that there are *obviously* no deficiencies, and the other way is to make it so complicated that there are no *obvious* deficiencies. [16, p. 155]

Clearly, elegance is a good thing. But how does it relate to more practical considerations? Robert Tarjan, an expert in the design and analysis of algorithms, says:

> Elegant algorithms are easy to program correctly, as well as being efficient. [33]

Daniel Kohansky, in his book *The Philosophical Programmer*, agrees:

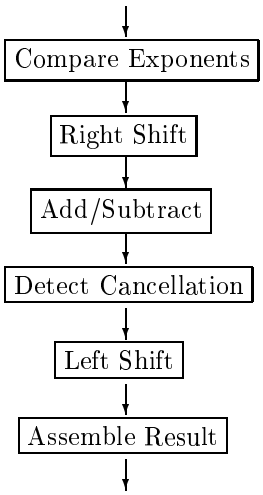
> Even so prosaic an activity as digging a ditch is improved by attention to aesthetics; a ditch dug in a straight line is both more appealing and more useful than one that zigzags at random ... [19, pp. 10–11]

This is certainly an appealing notion, and a view that I had always shared, but in hardware, it seems that the ditches to be verified are usually intended for irrigation as well as drainage, and that their designs are further complicated by issues of erosion and the like.

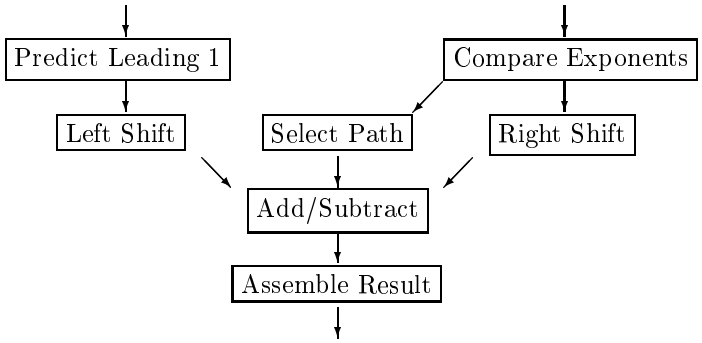
Dijkstra, who stressed the importance of elegance in programming as much as anyone, understood that it is a luxury that is afforded by increasingly powerful hardware, which, as he put it, "has mitigated the urgency of efficiency requirements." [7, p. 5] So there is a sharp distinction here between software and hardware: the concerns of usability and maintenance dictate that software be intellectually manageable, i.e., simple and elegant; the burden of efficiency must then be assumed by the underlying hardware, for which these issues are less critical. That is, while maintenance may be a consideration in hardware design—a component of a processor design may be modified for reuse in a later model—this is always outweighed by the importance of efficiency. Moreover, at least in my experience, it is rarely the case that the simplest circuit design is the most efficient one. This seems to be largely a consequence of the inherent parallelism of computer hardware.

As an illustration of this phenomenon, consider the design of a floating-point adder. The natural approach to this problem—the linear ditch—is a simple algorithm that is

readily understood and implemented and may be executed, under the constraints of contemporary technology, in six clock cycles, corresponding to the following six steps:



But since the operation occurs so frequently, in order to reduce its latency, a real adder (e.g., [28]) is a much more complex circuit involving two parallel data paths:



On one path, during the first two cycles, the exponents are compared and inputs are aligned accordingly in preparation for the operation; on the other, under the assumption that cancellation will occur through subtraction, the index of the leading one of the difference is predicted and the normalizing shift is performed in advance. Meanwhile, the exponent comparison determines which path is to be fed into the adder. Of course, this design requires considerably more hardware, is highly prone to error, and is difficult to analyze, but it runs in four cycles. This is typical of arithmetic circuitry, and the explanation is clear, although I still have trouble grasping this simple fact: gates are cheap and cycles are expensive.

On Simple Program Semantics

Dijkstra asked:

> Are you quite sure that all those bells and whistles, all those wonderful facilities of your so-called 'powerful' programming languages belong to the solution set rather than to the problem set? [9, p. xiv]

And I am convinced that this has been the primary obstacle to the goal of progressing from toy verification problems to real software: the languages in which real programs are coded are too messy to support clear semantic models. Programmers love those bells and whistles and language designers are eager to supply them. And hardware description languages are unexceptional in this regard.

Then why has hardware verification, and theorem proving in particular, enjoyed any success at all? One reason is motivation: hardware errors are difficult to correct after the fact. Another is the modularity that is imposed by timing considerations. As John Harrison has put it, timing constraints preclude "spaghetti hardware". [14, p. 1] But the real story—and for me, this was the biggest surprise of the entire experience—is that coding guidelines are effectively enforced to limit RTL design to a very small and manageable subset of Verilog. The main reason for this, I believe, constitutes a fundamental distinction between hardware and software development. A software developer does not have a complete explicit understanding of the language in which he is programming; he relies on the experimental use of a compiler to expose his errors. But the behavior of a compiled Verilog program is an unreliable model of a circuit. The real "compiler" of an RTL design is the process of implementation in silicon, which is of course unavailable for testing during the design process. Consequently, in practice, strict coding guidelines are required to ensure predictable behavior. The result is a language with a cycle-based semantic model that is simple enough to be amenable to formal analysis.

A program in this language consists of a set of signal definitions. For our purpose, a signal is either a *wire* or a *register*, as distinguished syntactically by the "`=`" and "`<=`" symbols, respectively:

```
rc_co = esub ? {1'b0, rc[70:1]} :
               {rc[69:0], rc[0]};

sum[70:0] <= rc[70:0] ^ a[70:0] ^ b[70:0];
```

Each signal assumes a value on each cycle of an execution. The value of a wire on a given cycle is computed according to its defining equation from the values of other signals on the same cycle; the value of a register is determined by values on the preceding cycle.

The simplicity of this structure allows us to translate RTL designs mechanically into the ACL2 logic in a fairly straightforward way. The primitive RTL operations correspond naturally to ACL2 functions, either built-in or simply defined, and each signal generates one of a set of mutually recursive ACL2 functions, each taking a single argument n, representing the number of cycles that have elapsed during the course of an execution:

```
(defun rc_co (n)
  (if (not (= (esub n) 0))
      (bits (rc n) 70 1)
```

```
    (cat (bits (rc n) 69 0) (bitn (rc n) 0) 1)))

(defun sum (n)
  (if (zp n)
      (reset 'sum 71)
    (logxor (logxor (bits (rc (1- n)) 70 0)
                    (bits (a (1- n)) 70 0))
            (bits (b (1- n)) 70 0))))
```

Along with this formal model, we also encode a statement of correctness in the same logic, essentially a formalization of IEEE-compliance, relating these signal functions to high-level arithmetic concepts. Thus, we have a complete formal representation of the problem, on which the power of the ACL2 prover may be brought to bear.

The Relevance of Traditional Mathematics

I'd like to turn now to the process of proof and the question of the relevance of traditional mathematics.

Early Resistance to Computer-Assisted Proof

I first became aware of the use of computing in support of mathematical proof as a graduate student in the '70s when I heard that the four color conjecture had been proved with the aid of a computer [1]. This was a novel development at the time and was met with some uncertainty in the mathematical community. Here is an excerpt from a paper by Daniel Cohen, a mathematician who had himself worked on the four-color problem, delivered at a conference on *The Mathematical Revolution Inspired by Computers*: [5, p. 327]

> In 1976, Appel and Haken announced that they had solved the Four Colour Problem by a computer examination of nearly two thousand cases . . . Furthermore, the procedure employed by the machine to analyze each case of necessity involved billions of logical inferences; this means that even though a human can duplicate by hand any small subset of the machine's deliberations there is not even a remote chance that, in an entire lifetime, a human could trace the program's run on even one case . . . [5, p. 327]

Cohen's remarks were something other than an expression of reverence for the power of the modern electronic computer. He continues:

> . . . Convictions derived in this manner might be valid but they are not mathematics. Such a result is still unproven, and should be so considered. . . . The real thrill of mathematics is to show as a feat of pure reasoning, it can be understood that four colours suffice. Admitting the shenanigans of Appel and Haken to the ranks of mathematics would only leave us intellectually unfulfilled. [5, p. 328]

This view was not uncommon at the time. In fact, this paper was written as recently as 1991. I find it amusing to observe that if my work is at all interesting, it is only

because of the use that I've made of computing in support of my results, all of which are relatively trivial and of little interest in themselves. Here, on the other hand, is a proof of a very deep result that was condemned for precisely the same reason. Are the goals of industrial hardware verification so very different from those of traditional mathematics? Or have attitudes changed so radically in sixteen years? Well, our goals are somewhat different, and attitudes have indeed shifted, but of course what has really changed is the technology of mechanical theorem proving.

Modern Theorem Proving: ACL2

Moving ahead a few years to 2005, we find a report of a new proof in *Mathematical Association of America Online*, under the headline, "Last Doubts Removed About the Proof of the Four Color Theorem" [8]. But in fact, this was yet another computer-assisted proof. It was developed by Georges Gonthiers of Microsoft Research, who used the Coq proof assistant [6] to formalize a variant of the Appel-Haken argument, including both its manual and mechanical components, as well as all of the topology and graph theory needed for a comprehensive proof from first principles. As noted in the MAA report:

> What makes the new result particularly significant from a reliability point of view is that the proof assistant Gonthiers employed, called Coq, is a widely-used general purpose utility, which can be verified experimentally, unlike the special-purpose programs used in the earlier proofs of the Four Color Theorem.

The point is that a modern theorem proving tool such as Coq, ACL2, HOL [15], or PVS [26] is trustworthy because it has been widely tested in a variety of domains by a community of users over a period of perhaps several decades. Such a tool is also more transparent and easily understood than the programmed proofs of earlier days. But is reliability the only issue here? What did Daniel Cohen mean by the remark, "Convictions derived in this manner might be valid but they are not mathematics."? I'd like to return to that question after taking a look at the theorem prover of my choice, ACL2.

It is difficult to say very much that is meaningful about the relative merits of different provers. Most comparisons are quite subjective, very much like religious preferences. (A notable exception is Freek Wiedijk's study [35].) With regard to ACL2, some of us appreciate the simplicity of its syntax (e.g., [10, 22]), while others are troubled by all of those parentheses (e.g., [11, Section 4] and [34]). But there seems to be a consensus on a number of points:

- Unlike Coq, ACL2 is intended primarily for computer system verification rather than mathematics, although I'm not sure of the significance of this statement. I asked Bob Boyer to comment on this; his observation was that the atom bomb was not intended primarily for digging ditches. (This will be my final ditch metaphor.)
- ACL2 is efficiently executable, since it may be compiled and executed as Common LISP.
- It provides a relatively high degree of automation, mainly through a system of powerful induction heuristics, conditional rewriting, and integrated decision pro-

cedures. I would note, however, that the term *automated theorem prover* is misleading: any nontrivial proof involves considerable interaction with the user, who usually begins with a fairly complete proof in mind, which he uses to guide the prover interactively through a long sequence of lemmas and hints.

- The underlying logic is relatively "weak", i.e., lacking in expressiveness. For example, it provides little support for existential quantification, and none for quantification over relations, sets, or functions.

These last two points constitute a trade-off: limiting the logic facilitates automatic analysis. Personally, I've never found the lack of expressiveness of ACL2 to be a serious drawback. Occasionally, some thought is required to find a way to say what I want to say, but that's a price I'm willing to pay in order to be relieved of some of the details of a proof. Other opinions may differ; I seem to have a natural tendency to think recursively and inductively. And, I might add parenthetically, parenthetically.

Illustration: A Test for Primality

Here is a small example of an ACL2 program, a characterization of prime numbers. (See [29] for an ACL2 proof script that includes all of the results listed in this section, culminating in a formalization of Gauss's Law of Quadratic Reciprocity.) The predicate `primep` tests for primality using a function `least-divisor`, which recursively searches for a divisor of n by dividing n by successively larger integers, starting at a designated value k, until it finds an integer quotient:

```
(defun least-divisor (k n)
  (if (and (integerp n)
           (integerp k)
           (< 1 k)
           (<= k n))
      (if (divides k n)
          k
        (least-divisor (1+ k) n))
    nil))

(defun primep (n)
  (and (integerp n)
       (= (least-divisor 2 n) n)))
```

This is a case where one would naturally like to use existential quantification, but is forced by the ACL2 logic to use recursion instead, and the result is a specification that can be compiled and executed.

For example, combining this predicate with the primitive ACL2 exponentiation function, we have a simple procedure for classifying Mersenne primes, i.e., identifying those primes p for which $2^p - 1$ is also a prime.

The Mersenne number $2^{23} - 1$, which happens to be divisible by 47 (as first observed by Fermat in 1640), is disposed of in a fraction of a second on my workstation:

```
(defthm mersenne-23
    (not (primep (- (expt 2 23) 1))))
```

```
[Time: .02 seconds]
```

The case $p = 31$ (which was settled by Euler in 1772) takes about an hour:

```
(defthm mersenne-31
    (primep (- (expt 2 31) 1)))

[Time: 65 minutes]
```

Here is a Mersenne number, generated by a six-digit prime, that takes a couple of hours to factor:

```
(defthm mersenne-999671
    (not (primep (- (expt 2 999671) 1))))

[Time: 165 minutes]
```

Obviously, this method requires no special expertise on the part of the user. I am confident that I could train a team of the meanest of engineers to administer it flawlessly. In a sense, it is completely general, but it suffers from practical limitations. Given that it took an hour to prove the primality of $2^{31} - 1$, we can estimate that the next smallest Mersenne prime, which happens to correspond to $p = 61$, would take about a billion hours. And the Mersenne number generated by an 8-digit exponent is already too large even to be represented in the memory of my machine:

```
(defthm mersenne-19876271
    (not (primep (- (expt 2 19876271) 1))))

[Error: Attempt to create an integer that is too large to represent.]
```

The most obvious optimization is based on the simple observation that if n has a proper divisor, then it has one that does not exceed $\sqrt{n}$. Thus, we define an alternative to the function `least-divisor` that stops at $\sqrt{n}$, and establish a rewrite rule:

```
(defun least-divisor-fast (k n)
  (if (and (integerp n)
           (integerp k)
           (< 1 k)
           (<= k n))
      (if (> (* k k) n)
          n
        (if (divides k n)
            k
          (least-divisor-fast (1+ k) n)))
    nil))

(defthm least-divisor-rewrite
    (equal (least-divisor 2 n)
           (least-divisor-fast 2 n)))
```

Once we arrange for this theorem (which was proved quite easily by means of ACL2's induction heuristics) to be applied in the computation of `primep`, the case $p = 31$ takes a fraction of a second, and $p = 61$ completes in under an hour:

```
(defthm mersenne-31-revisited
  (primep (- (expt 2 31) 1)))

[Time: .05 seconds]


(defthm mersenne-61
  (primep (- (expt 2 61) 1))

[Time: 54  minutes]
```

However, this optimization can't get us any further than this, and it is of no help in handling the composite case. At some point, in order to continue to make progress, we eventually must abandon algorithmic methods and resort to real theorem proving. For example, several of the cases that we've considered can be handled effectively by a nice theorem of Euler involving quadratic residues (see Theorem 103 of [13]). This exercise requires a little number theory, but I hope it will help illustrate the ACL2 experience.

If p is an odd prime, then an integer a is said to be a quadratic residue modulo p if there exists an integer x such that x^2 is congruent to a mod p. It may be shown that this property is equivalent to the condition

$$a^{(p-1)/2} \equiv 1 \pmod{p}.$$

(This congruence is known as *Euler's Criterion*.) In particular, it turns out that 2 is a quadratic residue mod p iff $p \equiv \pm 1 \pmod 8$. (This result is called the *Second Supplement* to the Law of Quadratic Reciprocity.)

Now we can easily prove the following:

Theorem If $p = 4k + 3$ and $q = 2p + 1$ are both prime, then $q | 2^p - 1$.

Proof: Since $q = 2(4k+3)+1 = 8k+7 \equiv -1 \pmod 8$, we know that 2 is a quadratic residue mod q, and therefore, by Euler's Criterion,

$$2^p = 2^{(q-1)/2} \equiv 1 \pmod{q},$$

or equivalently, $2^p - 1$ is divisible by q. □

Getting back to the Mersenne prime problem, what this result tells us is that under the stated hypothesis, $2^p - 1$ is not a prime. Here is an ACL2 formulation of this statement:

```
(defthm euler-corollary
  (implies (and (primep p)
                (= (mod p 4) 3)
                (> p 3)
                (primep (1+ (* 2 p))))
           (not (primep (- (expt 2 p) 1)))))
```

In order to generate its proof from scratch, over 100 lemmas were fed to the prover, along with generous hints, but that's the nature of "automated" theorem proving. It's also worth noting that through an oversight, my original formulation did not include the hypothesis that $p > 3$. It was only by examining the output of a failed proof attempt that I realized that when $p = 3$, while $2^p - 1$ (i.e., 7) is indeed divisible by $2p + 1$, it is in fact equal to $2p + 1$ and is thus nonetheless a prime.

We now have new proofs of two of our earlier results, requiring practically no computation, as the cases 23 and 999,671 both conform to the hypotheses of our theorem:

```
(defthm mersenne-23-revisited
  (not (primep (- (expt 2 23) 1))))

[Time: .01 seconds]

(defthm mersenne-999671-revisited
  (not (primep (- (expt 2  999671) 1))))

[Time: .01 seconds]
```

So what? We haven't proved anything new. But I claim that some proofs are better than others, and these last two are the only proofs we've seen that I'm really happy with, because not only have they been checked by ACL2, but I can understand them and check them by hand as well. Now, not only am I confident that there are no errors hidden in my proof (and Euler may share in this reassurance), but I actually know *why* $2^{23} - 1$ is divisible by 47. In other words, I've used formal methods to *support* mathematical rigor, rather than to replace it. Moreover, I have discovered a method that I can use in cases that I was previously unable to handle, such as this one:

```
(defthm mersenne-19876271
  (not (primep (- (expt 2 19876271) 1))))

[Time: 47 seconds]
```

The Value of Mathematical Proof

Now returning to Cohen's objection to computer-assisted proof, I think that his main point was that a proof should serve purposes other than merely to establish the correctness of a result. What then are the goals of mathematical proof?

There is, of course, the cynical view. Another item from the folklore of topology is the story of the knot theorist who presented a new result before a learned society and was asked about the real significance of his proof: "Your work is very beautiful, but what good is it?" "Well," he replied, "I write papers about knot theory; they get published, and I get promoted." [3, p. 164]

No doubt, there is some truth in this story. But I believe there are better answers to the question. Aside from the obvious one, that we prove theorems in order to know they are true, at least two others are suggested by our exercise in number theory:

- Explication of underlying principles: we rely on a proof to tell us *why* a result is true, to provide clues as to how it might be generalized, and to increase our understanding in order to make things easier in the future. Gauss published eight

distinct proofs of the law of quadratic reciprocity between 1796 and 1818, not because he remained unconvinced of the truth of the proposition, but rather, I suspect, because he was dissatisfied with the depth of understanding that was provided by the existing proofs. The way he put it was that he was looking for a proof that could be generalized to higher-order reciprocity laws. [18, p. 815]

- Refinement of hypotheses: often it is not until we explore the proof of a statement that we see that a required hypothesis has been omitted or that a superfluous one has been included. And mechanical provers, I find, are especially useful in exposing errors of that sort.

I would argue that all of these observations are just as valid in the context of industrial hardware verification as they are in pure mathematics. The primary conclusion of my investigation is that while common sense suggests that there are lessons to be derived from several decades of research in program verification, there is even more reason not to ignore the wisdom of several millennia of mathematics. So I would like to look more closely at the goals listed above and discuss how they can be addressed by formal hardware verification.

On Establishing Confidence in Correctness

With regard to confidence in correctness, I have tried to produce proofs that are both human-readable and machine-checked, as I do not believe that mechanical theorem proving negates the value of the social review process. The case of the Athlon floating-point adder was typical, and it is one for which I happen to have some relevant statistics handy, which may be of interest. This is the course that I followed:

- First, I learned what I could about the algorithms, studied the RTL, developed a statement of correctness, and wrote out a rigorous detailed proof, which filled 33 pages and consumed 4 weeks of my time.
- The RTL module, consisting of 86 KB of source code, was mechanically translated, generating 219 KB of ACL2 code.
- For 8 weeks, I sat with ACL2 and my hand-written proof and transformed it into ACL2 lemmas, line by line, until the formal proof was complete. The result was a proof script consisting of about 2200 lemmas.
- The process of mechanization exposed one fatal bug in the RTL, which was easily fixed, along with several minor errors in my hand-written proof.

Eighteen months later, that 33-page proof was published in its entirety [28]. (By then, my management was willing to concede that the competition had learned how to build their own adder.) I mention this because I often read reports of correctness proofs that sound interesting but are nowhere to be found. This is especially frustrating when the (neo-Pythagorean?) author favorably compares his own secret proof to my published one. So as long as I am presuming to tell you what I like or don't like about mathematical proofs, let me add that if we can't see a proof, then we are deprived of much of its potential value.

But the best way for me to inspire confidence on the part of my customers is to show them that the proof process exposes bugs that would otherwise have gone undetected—bugs in algorithms, in implementations, and in interfaces. To give an idea of the sort

of bug that might survive traditional testing, here is one that I found in a square root algorithm [27], which proceeded as follows:

- A 64-bit approximation q of $\sqrt{x}$ is derived, accurate to 38 bits.
- A 64-bit correction term c is added to q: $q+c$ is an underestimate of $\sqrt{x}$, accurate to 74 bits, and $0 < c < 2^{-38}q$.
- $q + c$ is rounded to 64 bits in both directions to produce r_1 and r_2.

In most cases, according to the rounding mode to be applied, either r_1 or r_2 is returned as the final result. In the case of rounding toward $+\infty$, if $(q + c) - r_1$ is not too big, then r_2 is returned.

All of this sounds reasonable, and it was not until I attemped to check my proof with ACL2 (which I sometimes think of as an unimaginative but unerring colleague peering over my shoulder) that I noticed the underlying assumption that r_1 and r_2 are distinct, which is not the case if $q + c$ happens to be a 64-bit number itself. This seems unlikely, and it is, because we are adding two 64-bit numbers that are misaligned by 38 bits, so in order for the sum to be 64-exact, the lower 38 bits of the smaller number would have to be 0. But I could not think of any reason to preclude this possibility, and neither could the designer. In fact, if we make the naive assumption that in this context, any given 38-bit sequence is as likely to occur as any other, then we may expect this situation to arise in one test out of every 2^{38}, which is about a quarter of a trillion. This number is in a range that makes it unlikely that the bug would be found in testing, but rather likely that it would occur during the life of a commercial processor.

By the way, this bug was corrected before the part was taped out, but not without some vigorous discussion. When one is accustomed to finding bugs only through testing, one might reasonably expect any bug report to be accompanied by a test failure. Unfortunately, that is not always easy to achieve. This was not the only occasion on which I was asked the question, "If you know what a bad result looks like, why can't you just work backwards through the algorithm to compute inputs that produce such a result?" My answer is to observe that the algorithm consists of a sequence of perhaps a dozen multiplications interspersed with various other operations, and that there is a widely used algorithm—namely, RSA public key encryption [4]— that is based on the practical impossibility of "working backwards" through even a single multiplication. I have enjoyed varying degrees of success with this argument.

On the Explication of Underlying Principles: Science vs. Art

With respect to the explanatory value of a proof and the guidance that it provides in applying underlying principles, I believe that there is an important contribution to be made by formal verification to the art of circuit design. I say "art" because I believe that this field in its current state is not so much *science*, which depends on the explicit knowledge and conscious application of principles, as *art*, which depends on traditional rules and skill acquired by practice.

I have seen complex logic ripped out of one design and inserted into another, with an incomplete understanding of why it worked in the first place, and then tested to ensure that it still does. There is a good deal of knowledge that is shared implicitly by design engineers but not written anywhere. There are textbooks on the subject [20, 25], but these are more concerned with the application of particular techniques than with their

theoretical underpinnings. These techniques are usually justified by means of examples rather than proofs, just as their implementations are validated by testing.

When I first observed these practices, I was reminded of something that I had encountered in my reading:

> It lacks so completely all plan and system that it is peculiar that so many men could have studied it. The worst is, it has never been treated stringently. There are very few theorems ... which have been demonstrated in a logically tenable manner. Everywhere one finds this miserable way of concluding from the specific to the general ... [18, p. 947]

This is from a letter written in 1826 by Niels Abel on the state of the calculus at that time, which had much in common with the present situation. This was a relatively new area of mathematical endeavor, lacking a solid foundation. Rigorous analysis had been replaced by appeal to geometric intuition and diagrammed examples, resulting in uncertainty and error. But it also produced results of tremendous practical significance, so that it was tempting to overlook these deficiencies.

In the end, of course, nineteenth century analysis was redeemed by a rigorous formulation derived solely from basic arithmetic principles, and I believe that a similar remedy is called for here: a unified arithmetic theory of register-transfer logic and floating-point arithmetic. This has been one of my objectives almost from the beginning. Over the course of developing various correctness proofs, I have tried to identify those results that pertain to the general theory and collected them in a library that is now a part of the ACL2 standard release [2]. It currently includes about 600 lemmas pertaining to bit vectors and logical operations, floating-point formats and rounding, and special-purpose techniques for efficient implementation of elementary operations. I have also written a hypertext document that is both an exposition of the theory and a user's manual for the library [30]. Naturally, the library makes my job much easier, since it allows me to reuse results from one project to the next. It has also found some use by ACL2 users outside of AMD. But there remains the real challenge of convincing engineers of the value of such a rigorous approach and integrating it into the design process.

On the Refinement of Hypotheses: Interface Specifications

This brings me to my last point on the value of proof: the refinement of hypotheses. In the context of RTL verification, this usually means precise specification of interface constraints, and is particularly important because this is probably the most common source of errors in RTL designs. When I examine a new RTL module, the piece of the puzzle that is invariably the most elusive is the interface. There is rarely any useful documentation to be found in the code or elsewhere. In fact, cooperation between modules commonly depends on informal oral agreements between RTL writers, which are highly prone to misunderstanding.

Of course, a formal proof of correctness of a module requires a formal specification of its external behavior, comprising all input and output constraints. At first, it seemed natural to me to write this specification directly in ACL2, based on information that I gathered from the designer. Then I would go over it with him, trying to explain what I had written, until he told me that he thought I had it right. This process did expose some bugs, but it was unsatisfying. I needed a formal language that was accessible to engineers. After some experimenting, it was clear that I had to find a way to write these

specifications in their native language. I had resisted this conclusion because Verilog is not an ideal specification language, but I found that with a few minor extensions, it served the purpose. The most important of these are:

- A rational data type, to allow high-level specifications of arithmetic operations;
- A facility for assertions representing both safety and liveness constraints on inputs and outputs.

Both of these extensions were readily implemented in both the Verilog compiler and the ACL2 translator. The result is a sort of pidgin Verilog that allows me to communicate more effectively with engineers. As a bonus, a specification written in this language may be integrated into the simulation environment and executed with the RTL for the purpose of testing. But once again, the real value of this approach will be measured by its ultimate impact on the design process.

Future Directions

Where are we today and where do we go from here? There is no doubt that the methodology that I've described has gained general acceptance by floating-point designers within AMD. New designs are now routinely and thoroughly verified. Recently, I attended a meeting to discuss plans for a new FPU that was considered to be innovative and therefore somewhat risky. I was shocked when the manager of the project, when asked how he would proceed if there were no resources available for theorem-proving verification, said that if this were the case, then he would revert to a more conservative design.

But why has our formal verification effort been limited to theorem proving to the exclusion of model checking, etc? When I ask the question, I'm told that it is a matter of resources: to assign anyone to a new area would be to detract from an existing project that is already considered to be essential to the verification process. But this is changing as well, as an investigation into commercial model checking tools and other automatic methods is underway.

In order to make all of this possible, we have increased our staffing. Until recently, I have been responsible for nearly all theorem proving at AMD, although I have had help, especially with development of the RTL library, from Eric Smith as a summer intern and occasionally from Matt Kaufmann. But in the past year, we have hired three people with expertise in this area. As a result, I am now enjoying more freedom to investigate new applications.

Specification and Verification of Control Logic

As an experiment, I wrote a specification for a bus interface unit and verified some invariance properties. I was not surprised to find that, in contrast to arithmetic designs, the main challenge of this project was in the level of detail of the interface and the internal structure of the module rather than the complexity of operations.

The specification had to account for elaborate interfaces with internal data and instruction caches and an external memory controller. Read and write requests are received from the caches, along with probes from the system, in response to which various transactions are initiated by the bus unit. Formulating the constraints on these transactions requires a complete abstract model of internal state: various buffers, an

L2 cache, outstanding requests, etc. As a benefit, we discovered that executing this model in RTL simulation exposed bugs that were less likely to be found by traditional checkers, which are more closely tied to the RTL. I also found that this process requires more cooperation between specification and design than does floating-point analysis. Once that is achieved, I believe that this is an area in which formal methods will have a major impact.

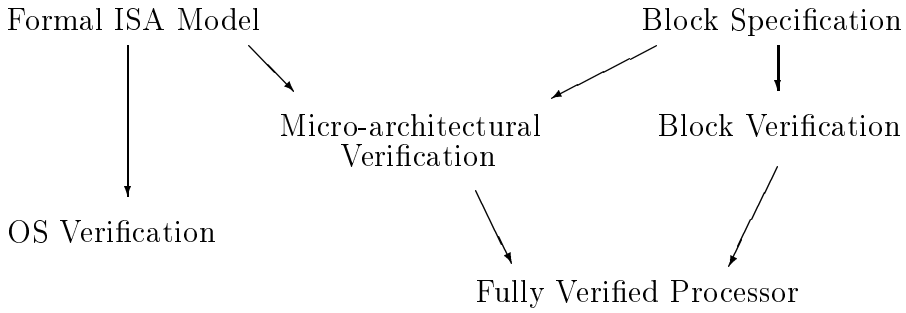
A Formal X86 Model

My current focus is a formal model of the x86 instruction set architecture, which I am developing in collaboration with my colleagues Bill Bevier and Larry Smith. We have two initial applications in mind, neither of which involves theorem proving: simulation and documentation.

The model is coded in a simple formal language called *XFL* that we have designed for this purpose. XFL is semantically embedded in C++, the language of AMD's simulation software. This allows our instruction set model to serve as the core of our simulation environment. The same model also forms the foundation of a formally based on-line programming manual. The goals are software that is robust and maintainable and documentation that is accurate and unambiguous. The idea is to achieve these goals through a unified model, the reliability of which is ensured by *reciprocal validation*. That is, since a bug in the simulator is also a bug in the documentation, every bug has exposure on several fronts, which increases the likelihood of detection.

Future Directions in Formal Verification

Each of these last two projects plays a role in our longer-term plans for formal verification. Here is my diagram for the future:



The formal instruction set model has potential applications in both software and hardware verification. On the one hand, software vendors have an interest in verifying properties of operating systems, especially security, for which such a model is a prerequisite. On the hardware side, imagine that we have an ACL2 specification for each block of an RTL processor design. We also have a translator from XFL to ACL2, which allows us to apply the ACL2 prover to the problem of micro-architectural verification, i.e., proving that the blocks fit together effectively to implement the instruction set. Once we also prove that each block satisfies its specification, we have achieved the ultimate goal of a fully verified processor. All of this, of course, rests on my ambition to live an inordinately long and mentally competent life.

References

[1] Appel, K., and W. Haken, "Every planar map is four colorable", *Illinois J. Math.* 21 (1977), 429-567.

[2] ACL2 Web site, `http://www.cs.utexas.edu/users/moore/acl2/`.

[3] Barnette, David, *Map coloring, polyhedra, and the four-color problem*, Mathematical Association of America, 1983.

[4] Boyer, Robert S., and J Strother Moore, "Proof Checking the RSA Public Key Encryption Algorithm", *American Mathematical Monthly*, vol. 91, no. 3, 1984.

[5] Cohen, Daniel, "The Superfluous Paradigm", in *The Mathematical Revolution Inspired by Computing*. Oxford University Press, 1991.

[6] Coq Web page, `http://coq.inria.fr/`.

[7] Dahl, O.-J, E.W. Dijkstra, and C.A.R. Hoare, *Structured Programming*, Academic Press, 1972.

[8] Devlin, Keith, "Last doubts removed about the proof of the Four Color Theorem", *MAA Online*, January, 2005, `http://www.maa.org/devlin/devlin_01_05.html`.

[9] Dijkstra, Edsger W., *A Discipline of Programming*, Prentice-Hall, 1976.

[10] Dupilka, Chris E., "Formal Methods in the Movies", `http://jaguar.it.miami.edu/~chris/formal_methods_in_the_movies/~ApocalypseNow.html`.

[11] Gray, David E., "Quadratic Reciprocity in Isabelle", `http://pressurecooker.phil.cmu.edu/Academic/Papers/quadRes.htm`.

[12] Gries, David, *The Science of Programming*, Springer-Verlag, New York, 1981.

[13] Hardy, G.H., and E.M. Wright, *An Introduction to the Theory of Numbers*, Oxford University Press, London, 1938.

[14] Harrison, John, "Floating-Point Verification", *Proceddings of FM2005: International Symposium of Formal Methods Europe*, Springer-Verlag, 2006.

[15] HOL Web page, `http://hol.sourceforge.net/#papers`.

[16] Hoare, C.A.R., "The Emperor's Old Clothes", in *ACM Turing Award Lectures: The First Twenty Years*, ACM Press, New York, 1987.

[17] Institute of Electrical and Electronic Engineers, "IEEE Standard for Binary Floating Point Arithmetic", Std. 754-1985, New York, NY, 1985.

[18] Kline, Morris, *Mathematical Thought from Ancient to Modern Times*, Oxford University Press, New York, 1972.

[19] Kohansky, Daniel, *The Philosophical Programmer: Reflections on the Moth in the Machine*, St. Martins Press, 1998.

[20] Koren, Israel, *Computer Arithmetic Algorithms*, 2nd Edition, A.K. Peters, Natick, MA, 2002.

[21] Levin, Michael I., *LISP 1.5 Programmer's Manual*, MIT Press, 1962.

[22] Liberman, Mark, "Emotional Code", *Language Log*, August 5, 2007, `http://itre.cis.upenn.edu/~myl/languagelog/archives/004789.html`.

[23] McCarthy, John M., "A Basis for a Mathematical Theory of Computation", *Computer Programming and Formal Systems*, edited by P. Braffort and D. Hirschberg, N. Holland Press, 1963.

[24] NQTHM Web site, `http://www.cs.utexas.edu/users/boyer/ftp/nqthm/`.

[25] Omondi, A.R., *Computer Arithmetic Systems: Algorithms, Architecture and Implementation*, Prentice Hall, 1994.

[26] PVS Web page, `http://pvs.csl.sri.com/`.

[27] Russinoff, David M., "A Mechanically Checked Proof of IEEE Compliance of the AMD-K5 Floating Point Square Root Microcode", *Formal Methods in System Design*, 1998.
`http://www.russinoff.com/papers/fsqrt.html`.

[28] Russinoff, David M., "A Case Study in Formal Verification of Register-Trasfer Logic: The Floating-Point Adder of the AMD Athlon Processor", Invited paper, *Formal Methods in Computer-Aided Design*, Springer LNCS 1954, November, 2000.
`http://www.russinoff.com/papers/fsqrt.html`.

[29] Russinoff, David M., "An ACL2 Proof of Quadratic Reciprocity",
`http://www.russinoff.com/qr/`.

[30] Russinoff, David M., "A Formal Theory of Register-Transfer Logic and Computer Arithmetic",
`http://www.russinoff.com/libman/`.

[31] Steele, G.L., Jr., *Common Lisp The Language*, 2nd edition, Digital Press, 1990.

[32] Stiles, David, Private communication, April, 1997.

[33] Tarjan, Robert, *Hewlett-Packard News* interview, October, 2004.
`http://www.hpl.hp.com/news/2004/oct_dec/tarjan.html`.

[34] Wheeler, David A., "Readable s-expressions and sweet-expressions: Getting the infix fix and fewer parentheses in Lisp-like languages",
`http://www.dwheeler.com/blog/2006/06/17/`.

[35] Wiedijk, Freek (ed.), *The Seventeen Provers of the World*, Springer LNAI 3600, Berlin, 2006.