ACL2 Theorems about Commercial Microprocessors

Bishop Brock, Matt Kaufmann[*] and J Strother Moore

Computational Logic, Inc., 1717 West Sixth Street, Austin, TX 78703-4776, USA[**]

Abstract. ACL2 is a mechanized mathematical logic intended for use in specifying and proving properties of computing machines. In two independent projects, industrial engineers have collaborated with researchers at Computational Logic, Inc. (CLI), to use ACL2 to model and prove properties of state-of-the-art commercial microprocessors prior to fabrication. In the first project, Motorola, Inc., and CLI collaborated to specify Motorola's complex arithmetic processor (CAP), a single-chip, digital signal processor (DSP) optimized for communications signal processing. Using the specification, we proved the correctness of several CAP microcode programs. The second industrial collaboration involving ACL2 was between Advanced Micro Devices, Inc. (AMD) and CLI. In this work we proved the correctness of the kernel of the floating-point division operation on AMD's first Pentium-class microprocessor, the AMD5$_K$86. In this paper, we discuss ACL2 and these industrial applications, with particular attention to the microcode verification work.

1 ACL2

ACL2 stands for "A Computational Logic for Applicative Common Lisp." ACL2 is both a mathematical logic and system of mechanical tools which can be used to construct proofs in the logic. The logic, which formalizes a subset of Common Lisp, is a high level programming language which can be executed efficiently on many host platforms. Thus, programmers can define models of computational systems and these models can be executed ("simulated") to test them on concrete data. But because the language is also a formal mathematical logic it is possible to reason about the models symbolically. Indeed, it is possible to prove theorems establishing properties of the models and to check these proofs with mechanical

[*] Matt Kaufmann's address is now Motorola @ Lakewood, P.O. Box 6000, MD F52, Austin, TX 78762

[**] The theorem prover used in this work was supported in part at Computational Logic, Inc., by the Defense Advanced Research Projects Agency, ARPA Order 7406, and the Office of Naval Research, Contract N00014-94-C-0193. The views and conclusions contained in this document are those of the author(s) and should not be interpreted as representing the official policies, either expressed or implied, of Advanced Micro Devices, Inc., Motorola, Inc., Computational Logic, Inc., the Defense Advanced Research Projects Agency, the Office of Naval Research, or the U.S. Government.

tools that are part of the ACL2 system. The ACL2 system is essentially a reimplemented extension, for applicative Common Lisp, of the so-called "Boyer-Moore theorem prover" Nqthm [2, 3].

1.1 The Logic

The ACL2 logic is a first-order, essentially quantifier-free logic of total recursive functions providing mathematical induction and two extension principles: one for recursive definition and one for "encapsulation."

The syntax of ACL2 is a subset of that of Common Lisp. Formally, an ACL2 term is either a variable symbol, a quoted constant, or the application of an n-ary function symbol or lambda expression, f, to n terms, written `(`$f\ t_1 \ldots t_n$`)`. This formal syntax is extended by Common Lisp's facility for defining constant symbols and macros.

The rules of inference are those of Nqthm, namely propositional calculus with equality together with instantiation and mathematical induction on the ordinals up to $\epsilon_0 = \omega^{\omega^{\omega^{\cdots}}}$.

The axioms of ACL2 describe five primitive data types: the complex rationals, characters, strings, symbols, and ordered pairs or lists. The complex rationals are complex numbers with rational components and hence include the rationals, the integers and the naturals. Symbols are logical constants denoting words, such as `DIV` and `STEP`. Symbols are in "packages" which provide a convenient way to have disjoint name spaces. `SMITH::DIV` is a different symbol than `JONES::DIV`; but if the user has selected `"SMITH"` as the "current package" then the former symbol can be written more succinctly as `DIV`.

Essentially all of the Common Lisp functions on the above data types are axiomatized or defined as functions or macros in ACL2. By "Common Lisp functions" here we mean the programs specified in [36] or [37] that are (i) applicative, (ii) not dependent on state, implicit parameters, or data types other than those in ACL2, and (iii) completely specified, unambiguously, in a host-independent manner. Approximately 170 such functions are axiomatized or defined. In addition, we add definitions of new function symbols to provide fast multiply-valued functions, an explicit notion of state with appropriate applicative input/output primitives, fast applicative arrays, and fast applicative property lists.

Common Lisp functions are partial; they are not defined for all possible inputs. In ACL2 we complete the domains of the Common Lisp functions, making every function total by adding "natural" values on arguments outside the "intended domain" of the Common Lisp function. For example, if ACL2 is used to evaluate `(car 7)` no exception is raised and the result is `nil`; if a Common Lisp were used to evaluate that expression the behavior is unpredictable, depends on which Common Lisp implementation is used, and could include memory violations and system crashes.

ACL2 and Common Lisp agree on the result of a computation provided the functions involved are exercised only on their intended domain. We formalize the notion of the intended domain of each function in our notion of *guard*. A guard is a predicate that recognizes arguments in the intended domain. The guards of the Common Lisp primitives are provided by the system; the user may provide guards for defined functions. The system then provides a mechanism, called *guard verification*, that insures that a function is Common Lisp compliant. Guard verification simply generates and attempts to prove a set of conjectures sufficient to imply that the evaluation of the function on its intended domain exercises its subroutines on their intended domains.

Some comments about guards and guard verification are in order. First, guards and guard verification are optional: ACL2 is a syntactically untyped logic of total functions. Theorems may be proved without considering guards; any ground term can be evaluated to a constant (provided no undefined functions are involved) using the axioms. But such theorems and computations may not be consistent with Common Lisp. If you want assurance that an ACL2 function will run in Common Lisp as predicted by an ACL2 theorem, you must verify the guards on the function and check that the arguments satisfy the guard. Second, guard verification is akin to type checking in that users may provide guards that express arbitrarily strong restrictions. Guard verification then insures that functions are "well-typed." Since guards are arbitrary predicates, guard checking is technically undecidable; but for commonly used primitive guards, guard checking is fast and silent. Finally, ACL2's implementation takes advantage of the fact that ACL2's axioms and Common Lisp agree when guards have been verified. If called upon to evaluate a function on constants, ACL2 can "short-circuit" its interpreter and execute compiled Common Lisp provided the function's guards have been verified and the arguments are in the intended domain. The user wishing to improve the execution speed of an ACL2 model is therefore encouraged to do guard verification. We discuss guards in more detail in [24].

Finally, ACL2 has two extension principles: definition and encapsulation. Both preserve the consistency of the extended logic. See [22]. The definitional principle insures consistency by requiring a proof that each defined function terminates. This is done, as in Nqthm, by the identification of some ordinal measure of the formals that decreases in recursion.

The *encapsulation* principle allows the introduction of new function symbols axiomatized to have certain properties. It preserves consistency by requiring the exhibition of witness functions that can be proved to have the alleged properties. To encapsulate a sequence of logical events — definitions and theorems — one embeds the sequence in an `encapsulate` form and marks certain of the events as `local`. Such an encapsulation is *admissible* to the logic if all the events are admissible; hence the definitions all satisfy the definitional principle and the theorems can be proved in the resulting extension of the logic. But the axiomatic *effect* of an encapsulation is to add to the logic the axioms corresponding to the *non-local* events. Thus, for example, to introduce an undefined function f constrained to be rational, it suffices to define f to be the rational 23, say, prove the theorem that f returns a `rationalp`, and then encapsulate those two events marking the definition as `local`. Effectively the definition serves as a *witness* for the satisfiability of the axiom added.

1.2 The System

Like Nqthm, ACL2's theorem prover orchestrates a variety of proof techniques. As suggested by Figure 1, the user puts the formula to be proved into a pool. The simplifier, the most important proof technique, draws a formula from the pool and either "simplifies" it — replacing it in the pool by the several new cases sufficient to prove it — or passes it to the next proof technique. The simplifier employs many different proof techniques, including conditional (back chaining) rewrite rules, congruence-based rewriting, efficient ground term evaluation, forward chaining, type-inference, the OBDD propositional decision procedure, a rational linear arithmetic decision procedure, and user-defined, machine-verified meta-theoretic simplifiers.

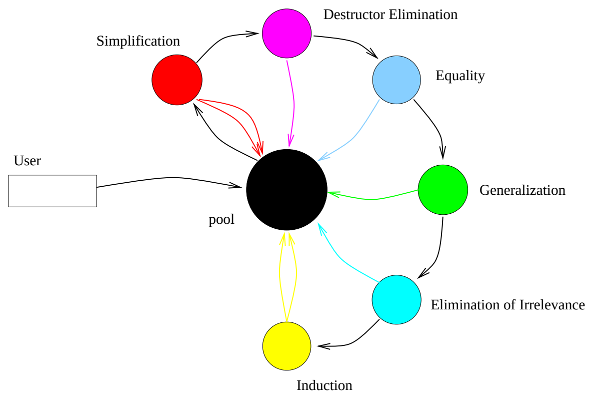

Fig. 1. The Orchestration of Proof Techniques

Roughly speaking, as the formula moves clockwise around the ring in Figure 1 it becomes more general. Eventually, if all else fails, the induction mechanism is applied.

The proof techniques are extensions of those used by Nqthm; see [2]. Most of the techniques are rule-driven. The rules are derived from previously proved theorems. For example, if the user has instructed ACL2 to prove that `append` is associative

```
(equal (append (append x y) z)
       (append x (append y z)))
```

and to use that fact as a `:rewrite` rule (from left to right), then — after the associative law is proved — the simplifier will right-associate all `append`-nests.

Because of the conservative nature of the extensions created by encapsulation, that logical mechanism is also very useful as a proof management tool. A complicated theorem can be derived in an encapsulation in which the uninteresting but necessary details are developed locally. See [22] for a detailed logical justification of encapsulation.

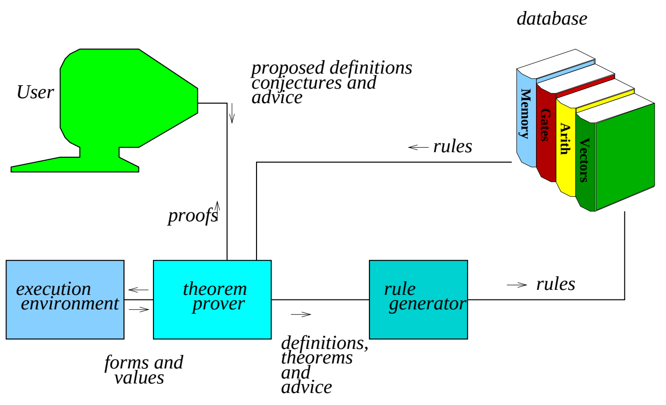

Fig. 2. System Architecture

By the proper choice of theorems to prove, the user can inform the system that a certain type-inference is possible, that a given relation is an equivalence relation to be used in congruence-based rewriting, that a certain function is a correct meta-theoretic simplifier, etc. Via this mechanism the informed user of ACL2 can essentially program it so that it constructs proofs following a certain strategy. By only using rules derived from previously proved theorems we insure that the user's advice cannot lead to logical unsoundness. See Figure 2.

With each proposed definition and theorem the user can supply hints to guide the theorem prover during the admission of the definition or proof of the theorem. A very common hint specifies which rules in the database are to be considered available during a proof or proof step. Other hints allow the user to suggest the use of a particular instance of a given theorem or to skip some step in the ring illustrated in Figure 1. Hints can be computed by ACL2 expressions so the user can codify strategies such as "if the goal contains these function symbols then the following rules should be used."

It is possible to collect together a body of definitions, theorems and advice into a file, here called a *book*. If ACL2 *certifies* the book then it can be loaded into any compatible ACL2 session to extend the logic and rule database appropriately. Books are incremental; multiple books can be loaded provided they compatibly define shared names. Packages allow the authors of books to have disjoint name spaces. Encapsulation can be used to hide unwanted aspects of a book or to extract desirable theorems. Events used in development of books may be marked `local` in order to prevent them from being exported.

The ACL2 system is written in ACL2, except for a relatively small amount of "boot-strapping" code, and is in essence a collection of ACL2 books. Coding ACL2 in its own logic forced many features into the language and insured that the resulting language is efficient and powerful enough for large projects demanding great computational resources. At the moment only one of the ACL2 source books, representing about 15% of the system, has been certified by ACL2; however, all the source books have been checked syntactically by ACL2.

2 Motorola CAP

The CAP is a single-chip, DSP co-processor optimized for communications signal processing, currently under development by Motorola Government and Systems Technology Group, Scottsdale, Arizona [16]. During the first seven months of the project one of us (Brock) relocated from Austin, Texas, to Scottsdale to work in close collaboration with the CAP design team. For the succeeding 18 months a member of the design team (Calvin Harrison) relocated to Austin to work at CLI. The CAP design was evolving throughout this period.

2.1 CAP Architecture

The CAP design follows the 'Harvard architecture', i.e., there are separate program and data memories. The design includes 252 programmer-visible data and control registers. There are six independently addressable data and parameter memories. The data memories are logically partitioned into 'source' and 'destination' memories; the sense of the memories may be switched under program control. The arithmetic unit includes four multiplier-accumulators and a 6-adder array. The CAP executes a 64-bit instruction word, which in the arithmetic units is further decoded into a 317-bit, low-level control word. The instruction set includes no-overhead looping constructs and automatic data scaling. As many as 10 different registers are involved in the determination of the next program counter. A single instruction can simultaneously modify well over 100 registers. In practice, instructions found in typical applications simultaneously modify several dozen registers. Finally, the CAP has a three-stage instruction pipeline which contains many programmer-visible pipeline hazards.

The motivation behind this complexity and unusual design is to allow the Motorola engineers to code DSP applications programs on the CAP and have those programs execute with stunning efficiency. The CAP was designed to execute a 1024-point complex FFT in 131 microseconds. There is no compiler targeting the CAP, perhaps because only a relatively few applications programs are required. For the time being at least, all CAP programs are coded by hand in CAP assembly language (CASM) and then assembled into binary and loaded into ROM. But programming the CAP efficiently and effectively requires a tremendous amount of knowledge and skill. One motivation for our involvement was to demonstrate that it was possible to create and then use a formal model of a state-of-the-art processor to verify applications programs.

2.2 The Model

We began by creating a formal, executable, ACL2 specification of the CAP [5]. This specification closely followed the style of earlier Nqthm work on modeling microprocessors, e.g., [19, 20, 28, 4]. Readers unfamiliar with that style need merely imagine defining, as a Lisp function, an interpreter for the intended machine language. We owe a special debt of gratitude to Yuan Yu, whose techniques in [39] we followed closely.

Our behavioral-level specification describes every well-defined behavior of the CAP including all legal instructions, I/O [21], traps, and interrupts. Only a few hardware and software initiated reset sequences are not modeled by our specification; these sequences were unnecessary to our intended verification work. In an effort to validate the specification we compared its execution with the results from executing Motorola's SPW engineering model of the processor [1].[3] For example, we compared the results of executing an end-to-end application (a QPSK modem) on both the SPW model and the ACL2 model; we found the final states bit-exact for all programmer visible registers. Other tests exposed discrepancies between the two models, most — but not all — of which were specification errors; some bugs were found in the CAP design. The ACL2 model runs several times faster than the compiled SPW model and hence is a potentially valuable debugging tool in its own right. A small part of the hardware implementation of the processor, the XY memory address generation unit, has been formally verified to agree with with the corresponding part of the ACL2 model [18]; this involved the hand-translation into ACL2 (in a very mechanical fashion) of the SPW description of the hardware.

We believe that CAP is the most complex processor for which a complete formal specification has been produced. The MC68020 modeled in [39], the first commercial processor for which a substantially complete formal model was produced, has only sixteen general purpose registers and a simple instruction set, albeit one with 18 addressing modes. Until the CAP work, the most complicated commercial processor subjected to formal modeling at the functional level was probably the Rockwell-Collins AAMP5, a special purpose avionics processor. The functionality of the AAMP5 was partially specified with PVS [15], as described in [26]: 108 of 209 instructions were specified and the two-stage pipelined microcode for eleven was verified. The state of the AAMP5 is much simpler than that of the CAP, involving only two memories, two flags, and six registers; nevertheless, the AAMP5 is far from simple: its 209 variable length CISC-like instructions closely resemble the intermediate output of compilers, and include some real-time executive capabilities such as interrupt handling, task state saving and context switching.

Before turning to the verification of CAP applications programs we undertook the logical elimination of the CAP pipeline. The CAP implementation includes a three-stage instruction pipeline with many visible pipeline hazards. Using ACL2 we demonstrated that under "normal" conditions, which we rigorously defined,

[3] As of this writing no CAP chips have been fabricated.

the behavior of CAP programs can be understood without reference to the instruction pipeline. We did this by proving an appropriate correspondence between the pipelined CAP model and a simpler pipeline-free model of the CAP [6]. Aside from making subsequent code proofs easier, this work had the important benefit of identifying (with both precision and assurance) an *equivalence condition* sufficient to avoid hazards yet weak enough to admit CAP application programs. Most of the requirements, in some suitably informal sense, were already part of the evolving "folklore" in the small community of CAP programmers, who generally take great pains to avoid pipeline hazards in their DSP application code. But the equivalence condition is sufficiently complicated that, short of a rigorous proof such as the one we constructed, confidence in its sufficiency is difficult to obtain. Our condition is weak enough to accept every ROM-resident DSP application on the CAP.

Our equivalence proof follows the approach suggested by Burch and Dill [11]. We found their method for stating equivalence, involving the idea of flushing the pipeline, intuitive for writing specifications. Applying it to the full CAP specification was somewhat challenging. The proof requires the symbolic expansion of the both the pipelined and non-pipelined models for three clock-cycles and then the comparison of the resulting symbolic states. Because of the control complexity of the CAP specification and the (necessary) weakness in the equivalence condition, a naive expansion took ACL2 about 10 minutes and produced a term whose printed representation was about 300,000 lines long (about 20 megabytes). After a couple of days' work by the user developing the proper theories, ACL2 can do a three step expansion on arbitrary (hazard-free) code in a few seconds and produce a term of about 3000 lines. Once the libraries needed to show the equivalence of these two models are in place (non-trivial), the proof takes about 4 minutes. ACL2's ability to handle large terms is one of its primary advantages over our earlier theorem prover, Nqthm, when dealing with industrial-sized problems.

The CAP instruction pipeline is certainly not the most complex instruction pipeline that has been formally specified and analyzed. However, although others have considered a few instructions on a pipelined commercial processor design (such as the AAMP5 work, [26]), or all instructions on a simple academic example [34, 35], our results cover *every* instruction sequence in a complete, bit-accurate model of a commercially designed processor. This work is also the only case that we are aware of where the problem of the equivalence of pipelined and non-pipelined models is so dependent on the program being executed, rather than predominately an intrinsic property of the hardware implementation.

Finally, we used the pipeline-free model as a basis for the mechanized verification of CAP application code. We briefly describe two CAP assembly language programs for which we constructed mechanically checked correctness proofs.

The first is a finite impulse response (FIR) filter. The FIR filter is an archetypical DSP application, a discrete-time convolution of an input signal with a set of filter coefficients. This FIR code is one that we developed specifically as an example, although it is based on a FIR algorithm for the CAP originally coded by a Motorola engineer. We prove that the code computes an appropriate fixed-precision result, but do not address any of the numerical analysis or DSP content of the algorithm. The second program is a high-speed searching application. This application uses specialized data paths in the CAP adder array to locate the 5 maxima of an input data vector in just over one clock cycle per input vector element. The 5-peak search is a fascinating example of an application with a straightforward specification, implemented by a clever combination of hardware and software, whose correctness is far from obvious. In this case the code we verified was obtained directly from the CAP program ROM, exactly as written by a Motorola engineer.

2.3 FIR

The finite impulse response (FIR) filter is a commonly used DSP algorithm; for details on its development and utility see for example [30]. Boiled down to its essentials, the FIR filter maps a discrete input signal $x[i]$ to a discrete output signal $y[i]$ by a discrete-time convolution of $x[i]$ with a set of filter coefficients h_k. The equation defining a non-adaptive FIR filter is often shown as something similar to

$$y[n] = \sum_{k=0}^{n-1} h_k \cdot x[n-k].$$

Given that the above equation defines a FIR filter, our goal is simply to prove that certain CAP microcode computes something suitably equivalent. The implementation uses the fixed-precision arithmetic of the CAP; the sense in which it implements the above equation involves the assumption that no overflows, for example, occur. There is no guarantee that every set of coefficients will produce a meaningful result, since the choice of coefficients together with input scaling parameters may lead to catastrophic overflows in the CAP accumulators that compute the sums of products.

The FIR code we verified consists of 30 64-bit microcode instructions. The program simultaneously filters X and Y source memories with a single set of coefficients, writing the results to X and Y destination memories. This can either be interpreted as two real FIR filters on two real signals, or a single real FIR filter on a complex signal. The small size of the FIR program belies its complexity because the CAP instruction set is so complicated. The effects of the single instruction in the inner loop depends upon 9 memories, 3 stacks and 31 registers; 17 registers are simultaneously modified by the instruction. The reader is reminded that we are not considering some idealized algorithm but an actual program written in a new and arcane programming language designed by industrial DSP experts for DSP applications.

The code has been mechanically proved to implement the specification equation. Because the CAP assembly code has no formal semantics, we actually addressed ourselves to the binary object code produced by Motorola's CAP assembler. Our theorem establishes that when (our formal model of) the CAP is run a certain number of steps on a suitably configured initial state the resulting destination memories are configured as required by the equation.

2.4 5PEAK

It is often necessary to perform statistical filtering and peak location in digital spectra for communications signal processing. The so-called "5PEAK" program uses the CAP's array of adder/subtracters with their dedicated registers as a systolic comparator array as shown in Figure 3. The program streams data through the comparator array and finds the five largest data points and the five corresponding memory addresses. In this discussion we largely ignore the fact that

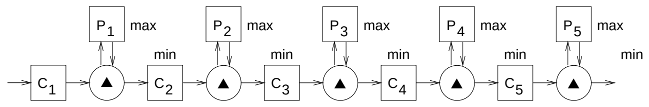

Fig. 3. Abstract View of Comparator Array

the comparator array also maintains the memory address of each data point.

Data points enter the array by way of register C_1 and move through the array, towards the right in the diagram. Maximum values remain in the array in the P_n registers, and the minima are eventually discarded when they pass out of the last comparator. On each cycle the comparator array updates the registers as follows:

$$\begin{aligned} C_1 &= \text{next data point}, \\ C_n &= \min(C_{n-1}, P_{n-1}), \quad n > 1, \\ P_n &= \max(C_n, P_n). \end{aligned}$$

Informally, the peak registers, P_i, maintain the maximum value that has passed by that point in the comparator array.

Suppose we initialize the array with the smallest possible data values (called "negative infinity" but actually just the most negative CAP integer). Then, as we stream an arbitrary amount of data through the array, the peaks accumulate in the peak registers. Immediately after the last data point has entered the array, we know that P_1 is the maximal data point. But P_2 may not be the second highest peak because it will not have been compared to C_2. To cause the necessary comparisons to occur, we must stream more data in. Note that by feeding one "positive infinity" (the most positive CAP integer) into the array we can cause the comparison of C_2 with P_2, along with other comparisons. This also displaces the actual maximal value from P_1 into C_2 and flushes one minimal value from the right end of the array. Thus by feeding five positive infinities in we can accumulate at the right end of the array the five maximal values.

We verified the 5PEAK object code obtained from the CAP program ROM. We proved that when the abstract CAP machine (as defined in ACL2) executes that binary code, on an appropriate initial state and for the appropriate number of cycles, the five highest peaks and their addresses are deposited into certain locations. We defined the “highest peaks and their addresses” by defining, for specification purposes only, a sort function in ACL2 which sorts such address/data pairs into descending order; in our 5PEAK specification we refer to the first five pairs in the ordering.

The argument that 5PEAK is correct is quite subtle, in part because an arbitrary amount of data is streamed through and in part because the positive and negative infinities involved in the algorithm can be legitimate data values but are accompanied by bogus addresses; correctness depends on a certain “anti-stability” property of the comparator array. A wonderfully subtle generalization of a key lemma was necessary in order to produce a theorem that could be proved by mathematical induction, [7]. In addition, as in the FIR example, correctness also depends on the invariance of many low-level properties of the initial CAP state.

2.5 Manpower Breakdown

Our involvement with the CAP project lasted 31 months. Only one formal methods expert (Brock) worked on the project continuously during that time. He was responsible for modeling the CAP in ACL2, producing the non-pipelined abstraction of the CAP, inventing the equivalence condition, mechanically proving the conditional equivalence of the two models, and mechanically proving the two microcode programs reported here. He also developed a library of ACL2 books for use in this work. Except where noted below, he worked on these tasks alone, although in the early months of the project he talked frequently with the CAP design team. Several other researchers and engineers contributed to the formal methods part of the project (e.g., the validation testing of the ACL2 against the SPW, a mechanically checked correctness proof for the address generation unit, an ACL2 macro to help formulate lemmas for expanding function definitions, a CAP assembler, a pin-level specification of the CAP IO interface) but their work has not been the focus of this paper and is not further discussed here. Brock's efforts can be broken down as follows:

– The CAP Specification: 15 months. Brock produced the first executable version of the specification in about 6 months, while resident in Scottsdale and interacting with the design team. During that time he was also learning ACL2[4]. The first model was simple and incomplete. As the project progressed, a 4-valued logic was introduced, and the pipeline, full ALU and IO modules were included. In addition, the CAP design evolved more or less continuously during this interval and the formal model tracked the design. Finally, ACL2 evolved also and Brock had to convert his work from Version 1.7 to Version 1.8, which dramatically changed the treatment of guards, making proofs much easier.

[4] He was already an accomplished Nqthm user.

– Reusable Books: 6 months. To do proofs about the CAP specification Brock had to develop many ACL2 books about modular arithmetic, logical operations on integers, hardware arithmetic and bit vectors, arrays, record structures, and list processing. These books are not CAP specific and are hence reusable.
– Equivalence to Non-Pipelined Model: 5 months. Not surprisingly, the non-pipelined model of the CAP shares perhaps 90% of its definition with the (pipelined) CAP specification. It was relatively easy to produce; however it required the reorganization of the CAP specification so that such sharing was possible. The first conditional equivalence proof was carried out on a simplified model of the CAP and then repeated when the IO module was added. A serendipitous visit by David Dill occurred just as Brock was beginning the first proof; Dill explained the methods in [11], which proved very useful. The details of the equivalence condition were derived from failed proofs. The condition was later weakened to so that it would accept the ROM-resident DSP application codes and the (now largely automatic) proof was repeated a third time to confirm the new condition. Brock then incorporated the predicate into an (unverified) automated tool that analyzes CAP programs for pipeline visibility conditions. The tool requires no formal methods expertise to use or to interpret its output.
– FIR: 1 month. Some of the work on reusable books was done in response to difficulties encountered during this task. In addition, CAP-specific books were developed for controlling the unwinding of the CAP model during such proofs. Both the FIR and 5PEAK work benefit from the fact that the non-pipelined model is simpler to reason about.
– 5PEAK: 1 month. About half of the time here was devoted to the development of the specification of the 5PEAK code. After several false starts, it eventually involved the idea of sorting an arbitrary amount of data with a generalization of the 5PEAK algorithm and then collecting the first 5 values. Moore spent one week proving the fundamental properties of the generalized sort algorithm.
– Reporting: 3 months. This includes time to write regular progress reports, travel to meetings, and write the final reports documenting the effort.

The times above are essentially the time it took the user to create the material, discover the proofs and lead the theorem prover to them the first time. The theorem prover can actually carry out the proofs relatively quickly. All of the CAP proofs can be reproduced in 2:30 (2 hours and 30 minutes) on a Sun Microsystems Sparcstation-20/712 with dual 75 MHz SuperSparc-II CPU's (each with 1 megabyte of cache), 256 megabytes of memory and 2 gigabytes of local disk. Many books are shared between the three main proofs (equivalence, FIR, and 5PEAK). A book need be certified only once and thereafter can simply be referenced without proof. The equivalence proof takes 1:45 (1:29 of which is spent certifying books that are re-used in the two applications proofs). FIR can then be done in 33 minutes (17 minutes of which is spent certifying books used also in 5PEAK). 5PEAK then takes 13 minutes.

3 AMD5K86

In the Spring of 1995, the Intel Pentium floating-point division bug was in the headlines. At that time, Advanced Micro Devices, Inc., was working on their first Pentium-class microprocessor, the AMD5_K86. Because the AMD5_K86 was not yet fabricated, there was time to reassess its division algorithm for bugs and perhaps fix any that might be found. But the algorithm was proprietary so it could not be reviewed by the numerical analysis community; nor was there time for the "social process" of mathematics to review any proffered proof of correctness.

In May, 1995 AMD hired CLI to construct and mechanically check a proof of the correctness of the kernel of the AMD5_K86's floating-point division algorithm. The principals of this collaboration were Tom Lynch of AMD, the designer of the division algorithm, and authors Kaufmann and Moore. All three were located in Austin, Texas, which facilitated collaboration. Work commenced in June, 1995 and the proof was completed by mid-August, 1995. Moore was the only person working full-time on the project. No bugs were found; the AMD algorithm was correct. We sketch the algorithm and theorem proved below. See [29] for details.

The AMD5_K86 algorithm is supposed to divide p by d, where both are floating-point numbers and $d \neq 0$, and round the result according to a specified rounding mode, m.

There has been much interest in the mechanical analysis of division algorithms since the Pentium bug. Almost all of the work focuses on SRT division [8, 32, 12]. The easiest way to contrast that work with ours is to point out that the results established in the above-mentioned papers do not formalize or discuss the notions of "floating-point number" or "rounding modes." Those concepts have been formalized elsewhere, for example [27], but no significant mechanically checked theorems about them are reported. Our work, on the other hand, focuses almost entirely on the concepts of floating-point number and directed rounding and the properties of the elementary floating-point operations. It is the first substantial body of mechanically checked formal mathematics about floating-point arithmetic.

By an $n,,m$ *floating-point number* we mean a rational that can be represented in the form $\sigma \times s \times 2^e$ where $\sigma \in \{+1, -1\}$, s is a rational, either $s = 0$ or $1 \leq s < 2$ and the binary representation of s fits in n bits, and e is an integer whose (biased) binary representation fits in m bits. A *rounding mode* specifies a rounding procedure and a precision n by which one is to round a rational to an $n,,17$ floating-point number. Common procedures include truncation of excessive bits (rounding toward 0), rounding away from 0, and "sticky" rounding in which the least significant bit of the rounded significand is 1 if precision is lost.

The AMD5_K86 algorithm uses a lookup table to obtain an 8,,17 floating-point approximation of $1/d$, twice applies a floating-point implementation of a variant of Newton-Raphson iteration to refine this into a 28,,17 approximation of $1/d$, uses the approximation to compute four 24,,17 quotient digits in an algorithm similar to long-division, and then sums the digits using sticky rounding for the first two floating-point additions and the user-specified mode m for the last (and most significant).

Using ACL2 we proved that if p and d are $64,,15$ (possibly denormal) floating-point numbers, $d \neq 0$, and m is a rounding mode specifying a positive precision $n \leq 64$, then the answer described above is the $n,,17$ floating-point number obtained by rounding the infinitely precise p/d according to m. In addition, and of particular interest to the design team, we proved that all intermediate results computed by the algorithm fit in the floating-point registers allocated to them.

Over 1600 definitions and lemmas were involved in this proof. While ACL2 (unlike Nqthm) has built-in support for the rationals, this was the first time that floating-point numbers and directed rounding had been formalized in ACL2. A substantial body of numerical analysis had to be formalized and mechanically checked in order to follow a fairly subtle mathematical argument constructed by the three collaborators. The proof complexity management tools, in particular books, encapsulation and macros, were crucial to the timely completion of this project. Note that only 9 weeks elapsed from the time CLI first saw the microcode to the time the final "Q.E.D." was printed. The final proof script can be replayed by ACL2 in two hours on the Sparcstation-20/712 described above.

4 Conclusion

We have described several theorems proved about commercial microprocessors. Make no mistake, these were "heavy duty proofs" requiring many skills, including great familiarity and insight into the applications areas, engineering issues, mathematics, formal logic, and the workings of the ACL2 proof tool. Furthermore, a fair amount of dedication and persistence were also required. See for example [25] for a case study describing practical problems in the use of a general purpose theorem prover. Simply put, it is hard work producing proofs of conjectures like these, a fact which stands in stark contrast to the impressive results obtained by "lightweight" analysis tools requiring so much less of the user [10, 14, 17, 38].

So why should people use general purpose theorem provers? If correctness is important and the problem cannot be solved by special-purpose "lightweight" tools, then a "heavyweight" tool is not only appropriate but is the only alternative. But is microcode verification within reach of today's model checking tools? If one assumes one has a bit-accurate model of a microcode engine, that the relevant microcode is in ROM (and so does not change during execution) and that all loops are controlled by counting down registers (and so are bounded by the word size), then microcode verification is a finite problem that can, in principle, be done by model checking. The question is simply one of practicality.

It is, of course, difficult to say with certainty that an untried problem is beyond the capabilities of a given model checker. The reason is partly that such tools can often handle designs with surprisingly large state spaces (e.g., several hundred boolean variables). Many techniques exist for reducing the "naive" state-space (e.g., “scaling”, the removal of all bits of state that do not support any property being checked); model checkers at higher levels of abstraction are also being developed [13, 9]. But the reason an unequivocal rejection of model checking is so difficult here is that with creative insight (i.e., “heavyweight thinking?”) the user of a “lightweight” tool can often abstract the problem into one that is manageable.

An easier question to answer, then, is whether today’s model checkers have been used to do microcode verification for machines of industrial interest. The answer seems to be no. While today’s model checkers can often succeed at this kind of verification for small values of the input parameters, they rarely do for the full range of values. We believe that the CAP and AMD programs discussed here are beyond the range of today’s model checkers. We return to this point shortly.

Assuming that these problems “cannot” be done by “lightweight” tools, theorem proving is the only alternative.[5] We have demonstrated that such problems, in their full complexity, are not beyond the range of ACL2. The manpower requirements of these ACL2 projects are not extreme, considering the amount of manpower industry currently throws at testing.

The manpower requirements in projects focused on code proofs for a given machine (the FM9001 assembly language [28], the MC68020 machine code proofs [4], the CAP work here, and the AAMP5 work [26]) all reflect a common theme: we are repeatedly measuring the startup costs. Building the formal model of the “new” machine or language and getting the appropriate library of rules in place dominates the costs. Yu’s work, [4], demonstrated that if one simply focuses on proving one program after another, it begins to get routine. Until automated theorem provers surpass humans in their creative ability to find proofs — today’s theorem provers must be regarded primarily as “proof checkers” that fill in the thousands of missing steps in what is commonly called a “proof” by students of mathematics — the mechanical verification of a program will remain essentially a two step process: the user discovers the “intuitive proof” and then gives the machine enough advice to check it. Once the startup costs have been paid for a given machine, the step of intuiting the informal proof becomes much more dominant.

Our CAP and AMD5_K86 projects, as well as the AAMP5 work [26], also support the conclusion that theorem-prover based formal methods can “keep up” with an evolving hardware design and make important contributions. Why then is industry apparently so reluctant to adopt these techniques? Part of the answer is that the skills required to use our tools are different than those ordinarily found in a hardware design team. Oddly enough, people with these skills, namely people trained in discrete mathematics, are readily available but their skills are not appreciated by industry. However, another major reason our tools are not more

[5] Testing the CAP code, while useful, is probably less likely to find all errors than testing would on a more conventional processor simply because of the sensitivity of the instruction sequencing to large portions of the state. Testing is also unconvincing in the case of sophisticated floating-point algorithms.

widely used is that they are not integrated into the *design process.* The ACL2 specification of the CAP, while efficiently executable and uniquely enabling of mechanically checked proofs, cannot at present be processed by conventional CAD tools. We had intended to build a translator from SPW to ACL2 but, for non-technical reasons beyond our control, could not pursue the task.

Such discouragement not withstanding, the fact remains that general purpose tools, such as ACL2, are truly general purpose. A powerful specification language is provided. As a user one rarely feels *unable* to express the key ideas. Arbitrary complexity can be modeled. The user carries the responsibility of managing this complexity but the system provides a wealth of ways to help, ranging from such traditional mathematical devices as functional composition and lemmas, to system features such as macros, encapsulation, books, symbol packages, and computed hints. The development of these features is the biggest improvement of ACL2 over Nqthm and is the reason that the CAP project could be carried out with ACL2 much more expeditiously, we believe, than with Nqthm. As with a good programming language or other universal tool, one rarely feels that a problem is too big or too complicated to address. Rather, the question is simply how to proceed. In that sense, the system feels open-ended and empowering, especially given that the system carries the burden of logical correctness and leaves the user entirely focused on the larger issues.

Finally, and most importantly, we believe there is value in demonstrating that certain things are merely *possible.* At least one respected numerical analyst told us several years ago that floating-point error analysis was too complicated to imagine checking mechanically. We now know otherwise. After our checking of the division microcode, David Russinoff, a former CLI employee now working at AMD, used ACL2 to analyze the AMD5_K86 floating-point square root code; his proof attempt exposed a bug in the original code which had escaped testing.[33] In collaboration with Tom Lynch, the code's designer, the code was changed and proved correct with ACL2. This work would not have been done had we persisted in believing that it was impossible. Similarly, perhaps microcode for such complex processors as the CAP will someday be verified automatically by "lightweight" tools; after all, we know that with general methods it is not only possible but practical and we are now merely talking about the cost.

In summary, general purpose theorem proving tools such as ACL2 are up to the demands of state-of-the-art industrial microprocessor design. In the short run, the contribution of such "heavy duty" tools is merely that they provide answers that can be obtained no other way. Mechanically checked proofs are essentially the only technology allowing the reliable exploration of arbitrarily deep problems. On the frontier of what is thought possible, there will always be a need for such tools and they will likely be in hands of people skilled in both the application area and mathematics. But in the long run, perhaps the main contribution of such "heavy duty" tools is that they allow us to enlarge the realm of what is thought possible.

References

1. K. Albin. *Validating the ACL2 CAP Model.* CAP Technical Report 9, Computational Logic, Inc., 1717 W. 6th, Austin, TX 78703 March, 1995.
2. R. S. Boyer and J S. Moore. *A Computational Logic.* Academic Press: New York, 1979.
3. R. S. Boyer and J S. Moore. *A Computational Logic Handbook*, Academic Press: New York, 1988.
4. R. S. Boyer and Y. Yu. Automated Proofs of Object Code for a Widely Used Microprocessor, *JACM*, **43**(1) January, 1996, pp. 166–192.
5. B. Brock. *The CAP 94 Specification*, CAP Technical Report 8, Computational Logic, Inc., 1717 W. 6th, Austin, TX 78703, July, 1995.
6. B. Brock. *Formal Analysis of the CAP Instruction Pipeline*, CAP Technical Report 10, Computational Logic, Inc., 1717 W. 6th, Austin, TX 78703, June, 1996.
7. B. Brock. *Formal Verification of CAP Applications*, CAP Technical Report 15, Computational Logic, Inc., 1717 W. 6th, Austin, TX 78703, June, 1996.
8. R. E. Bryant. Bit-Level Analysis of an SRT Divider Circuit, CMU-CS-95-140, School of Computer Science, Carnegie Mellon University, Pittsburg, PA 15213.
9. R. E. Bryant and Y. A. Chen. Verification of arithmetic functions with binary moment diagrams. In *Proceedings of the 32nd ACM/IEEE Design Automation Conference* IEEE Computer Society Press, June 1995.
10. J. R. Burch, E. M. Clarke, D. E. Long, K. L. McMillan and D. L. Dill. Symbolic Model Checking for Sequential Circuit Verification, *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems* **13**(4) April, 1994, pp. 401–424.
11. J. R. Burch and D. L. Dill. Automatic verification of pipelined microprocessor control. in David Dill, editor, *Computer-Aided Verification, CAV '94,* Stanford, CA, Springer-Verlag *Lecture Notes in Computer Science* Volume 818, June, 1994, pp. 68–80.
12. E. M. Clarke, S. M. German and X. Zhao. Verifying the SRT Division Algorithm using Theorem Proving Techniques, Proceedings of Conference on Computer-Aided Verification, CAV '96, July, 1996.
13. E. M. Clarke, M. Fujita, and X. Zhao. Hybrid Decision Diagrams, ICCAD95, 1995, pp. 159-163.
14. E. M. Clarke, O. Grumberg, H. Hiraishi, S. Jha, D. E. Long, K. L. McMillan and L. A. Ness. Verification of the Futurebus+ Cache Coherence Protocol, *Proc. CHDL*, 1993.
15. J. Crow, S. Owre, J. Rushby, N. Shankar, and M. Srivas. A Tutorial Introduction to PVS, presented at *Workshop on Industrial-Strength Formal Specification Techniques*, Boca Raton, FL, April 1995 (see http://www.csl.sri.com/pvs.html).
16. S. Gilfeather, J. Gehman, and C. Harrison. Architecture of a Complex Arithmetic Processor for Communication Signal Processsing in *SPIE Proceedings, International Symposium on Optics, Imaging, and Instrumentation*, **2296** *Advanced Signal Processing: Algorithms, Architectures, and Implementations V*, July, 1994, pp. 624–625.
17. Z. Har'El and R. P. Kurshan. Software for Analytical Development of Communications Protocols, *AT&T Bell Laboratories Technical Journal*, **69**(1) Jan-Feb, 1990, pp. 45–59.

18. C. Harrison. *Hardware Verification of the Complex Arithmetic Processor XY Address Generator.* CAP Technical Report 16, Computational Logic, Inc., 1717 W. 6th, Austin, TX 78703, August, 1995.
19. W. A. Hunt, Jr. Microprocessor Design Verification. *Journal of Automated Reasoning*, **5**(4), pp. 429–460, 1989.
20. W. A. Hunt, Jr. and B. Brock. A Formal HDL and its use in the FM9001 Verification. *Proceedings of the Royal Society*, 1992.
21. W. A. Hunt, Jr. *CAP Pin-level Specifications*, CAP Technical Report 12, Computational Logic, Inc., 1717 W. 6th, Austin, TX 78703, April, 1996.
22. M. Kaufmann and J S. Moore. High-Level Correctness of ACL2: A Story, URL ftp://ftp.cli.com/pub/acl2/v1-8/acl2-sources/reports/story.txt, September, 1995.
23. M. Kaufmann and J S. Moore. *ACL2 Version 1.8*, URL ftp://ftp.cli.-com/pub/acl2/v1-8/acl2-sources/doc/HTML/acl2-doc.html, 1995.
24. M. Kaufmann and J S. Moore. ACL2: An Industrial Strength Version of Nqthm. In *Proceedings of the Eleventh Annual Conference on Computer Assurance* (COMPASS-96), IEEE Computer Society Press, June, 1996, pp. 23–34.
25. M. Kaufmann and P. Pecchiari. Interaction with the Boyer-Moore and Theorem Prover: A Tutorial Study Using the Arithmetic-Geometric Mean Theorem. *Journal of Automated Reasoning* 16(1–2) March, 1996, pp. 181–222.
26. S. P. Miller and M. Srivas. Formal Verification of the AAMP5 Microprocessor: A Case Study in the Industrial Use of Formal Methods, in *Proceedings of WIFT '95: Workshop on Industrial-Strength Formal Specification Techniques*, IEEECS, April, 1995, pp. 2–16.
27. P. M. Miner. Defining the IEEE-854 Floating-Point Standard in PVS, NASA Technical Memorandum 110167, NASA Langely Research Center, Hampton, VA 23681, 1995.
28. J S. Moore. *Piton: A Mechanically Verified Assembly-Level Language*, Automated Reasoning Series, Kluwer Academic Publishers, 1996.
29. J S. Moore, T. Lynch, and M. Kaufmann. A Mechanically Checked Proof of the Correctness of the Kernel of the AMD5_K86 Floating-Point Division Algorithm, March, 1996, URL http://devil.ece.utexas.edu:80/~lynch/divide/-divide.html.
30. A. V. Oppenheim and R. W. Scahfer. *Discrete-Time Signal Processing.* Prentice Hall, Englewood Cliffs, New Jersey, 1989.
31. K. M. Pitman *et al. draft proposed American National Standard for Information Systems — Programming Language — Common Lisp; X3J13/93-102.* Global Engineering Documents, Inc., 1994.
32. H. Rueß, M. K. Srivas, and N. Shankar. Modular Verification of SRT Division, Computer Science Laboratory, SRI International, Menlo Park, CA 49025, 1996.
33. D. Russinoff, "A Mechanically Checked Proof of Correctness of the AMD-5_K86 Floating-Point Square Root Microcode," http://www.onr.com/user/-russ/david/fsqrt.html, February, 1997.
34. M. Srivas and M. Bickford. Formal Verification of a Pipelined Microprocessor, *IEEE Software*, September, 1990, pp. 52–64.
35. V. Stavridou. Gordon's Computer: A Hardware Verification Case Study in OBJ3, *Formal Methods in System Design*, **4**(3), 1994, pp. 265–310.

36. G. L. Steele, Jr. *Common LISP: The Language*, Digital Press: Bedford, MA, 1984.
37. G. L. Steele, Jr. *Common Lisp The Language, Second Edition.* Digital Press, 30 North Avenue, Burlington, MA 01803, 1990.
38. U. Stern and D. L. Dill. Automatic Verification of the SCI Cache Coherence Protocol, in *Proceedings of IFIP WG 10.5 Advanced Research Working Conference on Correct Hardware Design and Verification Methods*, 1995, pp. 21–34.
39. Y. Yu. *Automated Proofs of Object Code for a Widely used Microprocessor*, Technical Report 92, Computational Logic, Inc., 1717 W. 6th, Austin, TX 78703, May, 1993. URL http://www.cli.com/reports/files/92.ps.