Formal Verification of Floating-Point RTL with ACL2

David M. Russinoff

June 30, 2019

1 Introduction

The intent of the original IEEE floating-point specification, Standard 754-1985[1], was to establish an industry standard for the most common arithmetic operations in order to ensure consistent results across all computing platforms. Inevitably, the major floating-point architectures that have emerged since the publication of that document have largely ignored its recommendations, especially with respect to the handling of exceptional conditions. Consequently, no single specification can be expected to capture the behavior of even the most prevalent architectures.

A prerequisite for the verification of correctness of a commercial floating-point unit, for which backward compatibility is a strict requirement, is a comprehensive behavioral specification of the underlying architecture. For this purpose, we have defined such specifications for the most common elementary arithmetic instructions of the x86 and Arm architectures. These were developed over a period of more than two decades, during which they were validated against a variety of implementations produced by AMD, Intel, and Arm through extensive co-simulation and formal verification, while subjected to continual expert review.

Given the dual objectives of efficient execution and formal analysis, a natural choice of formalism for these specifications is the ACL2 logic [2], the versatility of which allows a proposed implementation to be encoded in the same language. A mathematical proof of correctness of the design with respect to its specification may then be mechanically checked with the ACL2 prover.

This plan presupposes a faithful representation in the ACL2 logic of a floating-point design, which is typically coded in Verilog at the register-transfer level. One solution to this problem, which was pursued in our earlier work, is to build a semantics-preserving Verilog-ACL2 translator that generates an ACL2 program consisting of a function corresponding to each RTL signal [6]. A problem faced by this approach is that the resulting ACL2 model is generally unwieldy, of code size at least comparable to that of the original RTL, leasing to concomitantly complex correctness proofs.

A more common industrial practice is the use of an automated sequential logic equivalence checker [4, 7] to compare a proposed RTL module either to an older trusted design or to a high-level C++ model. One deficiency of this approach is that the so-called "golden model", whether coded in Verilog or C++, has typically never been formally verified itself and thus cannot be guaranteed to be free of errors. Another is the inherent complexity limitations of such tools, which have been found to render them inadequate for the comprehensive verification of complex high-precision floating-point modules.

Our experience has led to a hydrid solution that combines theorem proving with equivalence checking. We have identified a simple subset of C++, called *Rescricted Algorithmic C* (RAC), and implemented a special-purpose parser for this language and a translator to ACL2. A simplified hand-coded RAC model of a Verilog module generates a more compact and managable ACL2 program than could be produced by direct translation. Functional equivalence between the Verilog code and the model may be established automatically by a commercial checker, and the proof of correctness of the ACL2 representation, although still a substantial effort, may proceed relatively easily.

Our collection of formal specifications is a component of an evolving ACL2 library of general results pertaining to register-transfer logic and floating-point arithmetic, which resides in the directory `books/rtl` of the ACL2 repository [2]. The directory `books/projects/rac` contains the

RAC parser and translator, and `books/projects/arm` includes ACL2 scripts for correctness proofs of the elementary arithmetic operations as implemented in an Arm floating-point unit. The library, language, and applications are thoroughly documented in [5] and briefly summarized below.

2 The ACL2 RTL Library

The instruction specifications as well as our formulation and analysis of implementations are based on a unified mathematical theory, derived from the first principles of arithmetic and encompassing two distinct domains of interest: register-transfer logic and floating-point arithmetic. The first of these comprises the primitive data types and operations on which microprocessor designs are built: bit vectors and logical operations. A bit vector in our theory is simply an integer, and the operations are all defined in terms of the standard *floor* and *modulus* functions. For example, the slice of a bit vector x bounded by indices i and j is defined by

$$x[i:j] = \lfloor (x \bmod 2^{i+1})/2^j \rfloor$$

and the extraction of a single bit is given by

$$x[i] = x[i:i].$$

(In this note, we shall use standard mathematical notation instead of ACL2 syntax.)

The bitwise logical operations are built-in ACL2 functions (inherited from Common Lisp), represented here in a Verilog-like notation. These include the unary complement,

$$\tilde{}x = -x - 1,$$

and the three standard binary operations, for which we derive equivalent recursive formulations that provide a basis for inductive proofs of their properties. Thus, the "inclusive or" is

$$x \mid y = \begin{cases} y & \text{if } x = 0 \text{ or } x = y \\ x & \text{if } y = 0 \\ 2 \cdot (\lfloor x/2 \rfloor \mid \lfloor y/2 \rfloor) + (x \bmod 2) \mid (y \bmod 2) & \text{otherwise.} \end{cases}$$

The usual bitwise characterizations,

$$\tilde{}x[i] = 1 - x[i],$$

$$(x \mid y)[i] = x[i] \mid y[i],$$

etc., are logical consequences these definitions.

The higher-level domain of floating-point arithmetic is based on the decomposition of a non-zero rational number into *sign*, *significand*, and *exponent*,

$$x = sgn(x) \cdot sig(x) \cdot 2^{expo(x)},$$

where $sgn(x) \in \{1, -1\}$, $1 \le sig(x) < 2$, and $expo(x) \in \mathbb{Z}$, and the fundamental notion of exactness:

$$x \text{ is } n\text{-}exact \Leftrightarrow 2^{n-1} sig(x) = 2^{n-1-expo(x)}|x| \in \mathbb{Z}.$$

A floating-point *format* is a mapping of the rationals of a specified degree of exactness and exponent range to the bit vectors of a specified width. A *rounding mode* is function that computes, for given rational x and integer n, an n-exact approximation of x. The library defines and extensively characterizes the formats and rounding modes employed by the major floating-point instruction sets, as well as those that are commonly used internally by commercial floating-point units. For example, the simplest rounding mode, "round toward zero", is defined by

$$RTZ(x,n) = sgn(x)\lfloor 2^{n-1} sig(x) \rfloor 2^{expo(x)-n+1}.$$

One that does not appear in the IEEE standard but is critical to many implementations is "round to odd":

$$RTO(x,n) = \begin{cases} x & \text{if } x \text{ is } (n-1)\text{-exact} \\ RTZ(x,n-1) + sgn(x)2^{expo(x)+1-n} & \text{otherwise.} \end{cases}$$

The library includes a section devoted to behavioral specifications of three major floating-point instruction set architectures: Intel's original x87 instructions, the newer x86 SSE instructions, and the Arm architecture. Central to all of these is the IEEE *principle of correct rounding*, according to which each of the elementary arithmetic operations of addition, multiplication, division, square root extraction, and fused multiplication-addition (FMA)

> ... shall be performed as if it first produced an intermediate result correct to infinite precision and then rounded that result according to one of the [supported] modes ...

The formalization of this principle in our theory in terms of floating-point formats, rounding modes, and rational arithmetic is straightforward. However, the architectures of interest differ significantly in the handling of exceptional conditions, especially the detection and handling of underflow, the response to a denormal operand, the order of precedence of the pre-computation conditions, the precedence of operands when more than one is a NaN, and the interaction of exceptions reported by the component operations of a SIMD (single instruction, multiple data) instruction. Consequently, a separate set of instruction specifications is required for each architecture. Each of these takes the form of an executable ACL2 function that computes a bit vector result corresponding to a set of bit vector operands in the context of a register environment and updates the environment. For example, the Arm FMA instruction is specified by a function of five arguments: a format (double-, single-, or half-precision), three bit vectors of width determined by the format, and the initial contents of the 32-bit Floating-Point Status and Control Register (FPSCR), which records exceptions and controls exception handling and rounding. Two values are returned: a bit vector data result and the updated contents of the FPSCR.

Finally, a section of the library collects and analyzes algorithms and optimization techniques that are commonly employed in the RTL implementation of the elementary operations of addition (various integer adders, leading and trailing zero anticipation), multiplication (several versions of Booth encoding), and division and square root extraction (SRT algorithms and FMA-based division). These results have been invaluable in the verification of a variety of designs.

3 Modeling RTL Designs: Restricted Algorithmic C

The intermediate modeling language, RAC, has been designed with several objectives in mind. One purpose of the model is documentation: it is intended to be a simplified and readable representation of the design, reflecting the underlying algorithms and essential computations while eliminating incidental implementation details. C++ is a natural candidate in view of its versatility and widespread use in system modeling, but we require a subset that allows a clear and easily understood semantic definition, and therefore include only the most basic integer data types, arithmetic operations, and control constructs.

On the other hand, the RAC model must be sufficiently faithful to the RTL to allow efficient sequential logic equivalence checking. This motivates the incorporartion of the Algorithmic C register class templates [3], which model integer and fixed-point registers of arbitrary width and provide the basic bit manipulation features of Verilog. These data types, along with all other RAC features, are supported by the commercial equivalence checkers Hector [7] and SLEC [4].

The objectives of translation to ACL2 and formal mathematical analysis dictate a functional programming paradigm, which we promote by replacing the pointers and reference parameters of C++ with other suitable extensions. The stipulation that function parameters are passed only by value dictates that native C arrays may be used only as global constants and in instances where an array is used only locally. The effect of passing arrays by value is achieved by including the standard

C++ `array` class template. We further compensate for the absence of side-effects by including the `tuple` class template, which provides the effect of multiple-valued functions and a convenient means of simultaneously assigning the components of a returned value to local variables of the caller.

The RAC-ACL2 translation is simplified by a number of control restrictions. For example, an important feature of the translator is the replacement of iteration with recursion, which is facilitated by imposing strict requirements on the structure of a `for` loop to allow the straightforward generation of an equivalent recursive function that may be automatically admitted to the ACL2 logic by the prover.

4 Example: Verification of an Arm Floating-Point Unit

The sample RAC application included in the ACL2 repositiory is the formal verification of the FPU of an Arm Cortex-A class high-end processor, addressing double-precision multiplication, addition, FMA, division, and square root extraction. The principal algorithms used in this design are taken from the ACL2 RTL library, including leading zero anticipation, radix-4 Booth encoding, and radix-4 SRT division and square root.

Experimentation with these modules has shown that the multiplier and adder are susceptible to equivalence checking against a general-purpose high-level C++ FPU model, although requiring considerable computing resources. This approach was found to be useless, however, for the iterative operations of division and square root, and for this implementation of FMA, which involves passing a full 106-bit product from the multiplier to the adder.

As usual, the RAC models were designed to be as compact and abstract as possible while based on the same algorithms as the RTL and computing precisely the same essential intermediate values in order to minimize the complexity of equivalence checking, which was performed in this case by SLEC. This required no user guidance other than supplying correspondences between inputs and outputs and the latencies of the operations, as the tool has the capability of independently discovering correspondences between intermediate values, thereby effectively decomposing the equivalence check.

The RAC abstraction of the RTL amounted to a reduction in code of approximately 85%, resulting in some 86 KB of RAC code and a slightly larger ACL2 model. The correctness proofs were found to be manageable but more cumbersome than one would like, consisting of a total of nearly 4000 ACL2 lemmas. Future work will focus on exploiting opportunities for simplifying and automating the proof process.

References

[1] Institute of Electrical and Electronic Engineers: IEEE standard for floating point arithmetic, Std. 754-1985 (1985)

[2] Kaufmann, M., Moore, J S.: ACL2 web site. http://www.cs.utexas.edu/users/moore/acl2/

[3] Mentor Graphics Corp.: Algorithmic C datatypes. Available at https://www.mentor.com/hls-lp/downloads/ac-datatypes

[4] Mentor Graphics Corp.: Sequential logic equivalence checker. https://www.mentor.com/-products/fv/questa-slec

[5] Russinoff, D.: Formal Verification of Floating-Point Hardware Design: A Mathematical Approach. Springer (2018)

[6] Russinoff, D., Kaufmann, M., Smith, E., Sumners, R.: Formal verification of floating-point rtl at amd using the acl2 prover. In: IMACS World Congress: Scientific Computation, Applied Mathematics and Simulation (2005)

[7] Synopsys, Inc.: Hector. http://www.synopsys.com/Tools/Verification/FunctionalVerification/-Pages/hector.aspx