Formal Verification of a Chained Multiply-Add Design: Combining Theorem Proving and Equivalence Checking

David Russinoff, Javier Bruguera, Cuong Chau,
Mayank Manjrekar, Nicholas Pfister, and Harsha Valsaraju
Austin Design Center
Arm, Inc.
Austin, Texas, U.S.A
{david.russinoff, javier.bruguera, cuong.chau, mayank.manjrekar2, nicholas.pfister, harsha.valsaraju}@arm.com

***Abstract*—We present a hybrid methodology for the formal verification of arithmetic RTL designs that combines sequential logic equivalence checking with interactive theorem proving in a two-step process. First, an intermediate model of the design is extracted by hand and coded in *Restricted Algorithmic C*, a simple C subset augmented by the C++ register class templates of Algorithmic C, which provide the bit manipulation features of Verilog. The model is designed to mirror the RTL microarchitecture closely enough to allow efficient equivalence checking, but sufficiently abstract to be amenable to formal analysis. The model is then automatically translated to the logic of the ACL2 theorem prover, which is used to establish correctness with respect to an architectural specification. As an illustration, we describe the modeling and proof of correctness of a *chained multiply-add* module, designed to test techniques for area and power reduction and intended for implementation in future Arm graphics processors.**

***Index Terms*—formal verification, theorem proving, equivalence checking, chained multiply-add**

I. Introduction

While most formal hardware verification efforts are based on automatic methods such as model checking and sequential logic equivalence checking, it is widely acknowledged that the inherent complexity limitations of these methods render then inadequate for floating-point verification. For a relatively simple arithmetic design, confidence in correctness may be attained by establishing equivalence with either a trusted high-level C++ "golden model" or a similar legacy design, but in general, comprehensive formal verification of a state-of-the-art floating-point unit is achievable only through interactive theorem proving, which is not subject to the complexity limitations of automatic tools.

Various theorem provers have been applied to the formalization of floating-point arithmetic and the verification of arithmetic algorithms [1]–[3]. However, a proof of correctness of a high-level algorithm cannot guarantee that it has been faithfully implemented in hardware. A prerequisite for applying interactive theorem proving to arithmetic circuit verification at the register-transfer level is a reliable means of converting a low-level design to a semantically equivalent representation in a formal logic. Early efforts in floating-point design verification [4], [5] were based on formal models derived by hand from microcode or register-transfer logic (RTL) designs, ignoring implementation details that could affect the result of a computation. More recently, in at least two industrial settings [6], [7], more trustworthy formalizations have been achieved by mechanical translation from Verilog directly to the logic of the ACL2 prover [8]. While these projects are based on very different translation schemes ("deep" vs. "shallow embedding"), both burden the user with an unwieldy body of ACL2 code, at least comparable in size to the Verilog source.

Fig. 1. Verification workflow

At Arm, Inc., we have addressed this problem with a hybrid solution that combines equivalence checking with theorem proving in a two-step process, as diagrammed in Figure 1. First, an intermediate model of a floating-point design is derived from the RTL by hand, coded in *Restricted Algorithmic C (RAC)*, a simple subset of C augmented by the register class templates of Algorithmic C [9], which essentially provide the bit manipulation features of Verilog. The objective is a high-level model that is more manageable than the RTL but mirrors its microarchitecture to the extent required to allow efficient equivalence checking between the design and the model. For this purpose, we have found the Mentor Graphics tool SLEC [10] to be effective. The model is then processed by a RAC-ACL2 translator, which generates a logical representation of the design. Finally, the ACL2 prover is used to check a proof of correctness with respect to a formal architectural specification, which is encoded in the same logic.

This methodology has been successfully applied in the verification of a wide range of arithmetic components of commercial CPUs and GPUs, including high-precision multipliers and adders, 64-bit integer multipliers and dividers, and a variety of floating-point division and square root modules.

One important result is a proof of correctness of a fused multiply-add (FMA) module that serves as a central component of Arm's GPUs. The defining characteristic of a graphics processing architecture is the coordination of many identical multi-purpose arithmetic units, which may be executed in parallel to perform the same operations on large sets of data. Such hardware is useful in a variety of scientific applications that are naturally accelerated by massive data parallelism, including efficient image rendering. Since the FMA module is replicated many times per core, it accounts for a significant portion of the core's area as well as its power consumption. Most of the choices underlying its design are driven by these considerations.

As an illustration of our methodology, we shall describe the modeling and formal verification of a variant of this design, a *chained multiply-add (CMA)* module, designed for the purpose of investigating some proposed implementation techniques intended to improve performance and reduce area and power. Although our concerns regarding its correctness may not be of the same urgency that attends the imminent release of a product, it is nonetheless an important objective, as it provides assurance that the data derived from the experiment are reliable. Since this CMA prototype is the first of its kind, RTL-RTL equivalence is not a viable alternative to theorem proving. Moreover, since the design is nontrivial but less complicated than, for example, a high-precision high-radix divider, it provides a suitable illustration of our verification methodology, which is described in Section II. In Section III, we present an overview of the CMA design, its RAC model, and the proof of correctness. Section IV discusses the results of the experiment.

II. Verification Methodology

The RAC programming language, which is fully documented in [11, chap. 15], is intended for modeling RTL designs that employ complex and sophisticated arithmetic algorithms and optimization techniques. For our purpose, a RAC model must be sufficiently faithful to the RTL to allow efficient processing by SLEC, thereby guaranteeing functional equivalence. On the other hand, the language and model should be as simple and abstract as possible in order to facilitate formal mathematical analysis and, in particular, mechanical translation to ACL2.

The RAC data types include the basic integer types of C along with the register class templates of Algorithmic C, which provide signed and unsigned integer and fixed-point registers of arbitrary width and precision. In addition to the standard arithmetic, logical, and shift operations, the register classes include bit slice extraction and assignment methods with the semantics of the corresponding Verilog operators, thereby easing the burden of equivalence checking. All language features are supported by SLEC.

ACL2 is both an executable programming language, essentially an applicative subset of Common Lisp, and a first order logic supported by a heuristic interactive theorem prover based on recursion and mathematical induction. A RAC program

```
ui32 add8(ui32 a, ui32 b) {
  ui32 res; ui8 sum;
  for (uint i=0; i<4; i++) {
    si8 aS = a.slc<8>(8 * i);
    si8 bS = b.slc<8>(8 * i);
    si9 sumS = aS + bS;
    if (sumS < -128) {sum = -128;}
    else if (sum >= 128) {sum = 127;}
    else {sum = sumS;}
    res.set_slc(8 * i, sum);}
  return res;}
```

Fig. 2. A vector signed integer adder

is subject to a number of restrictions intended to enable its translation to ACL2. A functional programming paradigm is enforced by disallowing C pointers and reference parameters. Thus, a RAC function is free of side-effects and simply returns a value. In order to compensate for these restrictions, two standard C++ library class templates are supported:

- Since the elimination of pointers precludes the usual C mechanism for passing arrays as function parameters, RAC supports the `array` template, which allows arrays to be passed by value;
- The `tuple` template may be instantiated as the return type of a function, providing the effect of multiple-valued functions.

Various constraints are imposed on the control structure of a function in order to facilitate its transformation into an ACL2 function. Conditional branching (`if ... then ... else`) is supported along with restricted versions of `switch` and `for`. A `return` statement can occur only as the final statement of a function branch.

A simple example of a RAC function illustrating the use of integer registers is listed in Figure 2. Note that unsigned and signed integers of width n are denoted by the type declarations `ui`n and `si`n, respectively. Thus, the function takes 2 32-bit unsigned arguments, from which it extracts corresponding 8-bit slices, adds them as signed integers, saturates the sums to lie in the range $[-128, 128)$, and stores them in a 32-bit result vector. The bit slice extraction method `slc` has a template parameter indicating the width of the slice and an argument representing the base index. The `set_slc` method, which writes a value to a slice, takes two arguments, which determine the base index of the slice and the value to be written, the type of which determines the width of the slice.

The translation from RAC to ACL2 is performed in two phases: a special-purpose C++ parser based on Flex and Bison [12] transforms a RAC program into a set of S-expressions (linked lists, the data of Lisp and ACL2), which are converted to ACL2 functions by a code generator written in ACL2 itself. As a by-product, the parser produces a more readable pseudocode version of a model, replacing the cryptic syntax of C++ methods with the familiar notation of Verilog. Thus, the first two assignments of the function `add8` are printed as

```
si8 aS = a[8*i+7:8*i];
si8 bS = b[8*i+7:8*i];
```

```
(DEFUN ADD8-LOOP (I A B SUM RES)
  (IF (AND (INTEGERP I) (< I 4))
      (LET* ((AS (BITS A (+ (* 8 I) 7) (* 8 I)))
             (BS (BITS B (+ (* 8 I) 7) (* 8 I)))
             (SUMS (BITS (+ (SI AS 8) (SI BS 8)) 8 0))
             (SUM (IF1 (LOG< (SI SUMS 9) -128)
                       (BITS -128 7 0)
                       (IF1 (LOG>= SUM 128)
                            (BITS 127 7 0)
                            (BITS (SI SUMS 9) 7 0))))
             (RES (SETBITS RES 32
                           (+ (* 8 I) 7) (* 8 I)
                           SUM)))
        (ADD8-LOOP-0 (+ I 1) A B SUM RES))
      (MV SUM RES)))

(DEFUN ADD8 (A B)
  (MV-LET (SUM RES) (ADD8-LOOP 0 A B 0 0)
    RES))
```

Fig. 3. Translation of the signed integer adder

and the final slice assignment as

```
res[8*i+7:8*i] = sum;
```

The translation of `add8` is displayed in Figure 3. Note that the the problem of mapping an imperative paradigm to a functional language is addressed by two tactics:

- Every RAC `for` loop is converted to an auxiliary recursive ACL2 function. Constraints imposed on the format of a loop guarantee that execution of this function terminates. In this case, the arguments of the recursive function `ADD8-LOOP` include the loop variable `I`, which is incremented in the recursive call. The variables that are updated by the loop, `SUM` and `RES`, are returned as a multiple value.
- A sequence of RAC assignment statements is converted to a nest of bindings, using two ACL2 macros: `LET*`, which performs sequential bindings, and `MV-LET`, which handles the result of a multiple-valued function and binds variables in parallel.

The translation employs a library of ACL2 functions that are defined to correspond to RAC primitives and emulate the semantics of C:

- `LOG<`, `LOG>=`, etc., are boolean comparators that return 1 or 0;
- `IF1` is similar to the Lisp `IF` but tests its first argument against 0 instead of `NIL`;
- `BITS` and `SETBITS`, respectively, extract and assign slices of a bit vector;
- `SI` computes the signed integer value of a register of a given width.

Note that variable types do not explicitly appear in the translation. The problem of translating a typed language to an untyped language is addressed mainly by converting implicit register evaluations and type conversions to explicit computations. Thus, in the expression that is derived from the declaration

```
si9 sumS = aS + bS;
```

the registers *AS* and *BS* are evaluated according to their type, i.e., by computing their 8-bit signed integer values, and when the sum is assigned to the 9-bit register *SUMS*, the low order 9 bits are extracted.

In comparison to direct translation from Verilog to ACL2, the primary advantage of an intermediate RAC model is that it provides an abstract and readable representation of a design that is amenable to mathematical analysis, and consequently a compact ACL2 model that is more susceptible to formal proof. The construction of a RAC model is generally a compromise between two opposing objectives: while a higher-level model allows a simpler correctness proof, successful equivalence checking of a complex design generally depends on structural similarities between the model and the design. SLEC provides a facility for exploiting such similarities by mapping intermediate internal RTL signals to corresponding local variables of the model, which effectively decomposes the equivalence computation.

Establishing the appropriate level of detail of the model is guided by experience and often involves some experimentation. As a rule of thumb, the model should be as abstract as possible while performing the same essential computations as the design. For example, in the case of an iterative divider, this means precisely replicating the partial remainder and quotient at each iteration. Successful processing of a high-precision multiplier typically requires replication of the Booth encoder as well as the intermediate computations at each level of the compression tree.

The translation of Arm floating-point units through this method has been found to result in an average reduction in code size by approximately 85%. The RAC model serves other purposes as well, including documentation, design guidance, and simulation in a C++ environment. Extracting the model from the design may require significant effort, but much of this effort also contributes to the proof of correctness.

Such proofs are based on a unified arithmetic theory of register-transfer logic and computer arithmetic, which is supported by an ACL2 RTL library of formal results developed over the course of twenty-five years of analysis and verification of commercial hardware designs produced by AMD, Intel, and Arm. These results cover the following areas:

- Register-transfer logic: properties of bit vectors, integer and fixed-point representations, bitwise logical operations;
- Floating-point arithmetic: floating-point formats and the relevant properties of rational numbers, IEEE rounding and other modes of rounding that are commonly used in the implementation of floating-point units;
- Implementation of elementary arithmetic operations: integer addition, parallel prefix adders, leading zero anticipation; variations of Booth multiplication; SRT division and square root extraction; multiplication-based division;
- Instruction set architectures: comparison of the floating-point behaviors of IEEE-compliant x87, SSE, and Arm CPUs with emphasis on variations in exception handling.

The theory and the RTL library are thoroughly documented in [11, chaps. 1–14].

It cannot be denied that the application of this methodology to a design of any complexity involves significant effort, a major portion of which lies in the development of the RAC model, which requires a detailed understanding of the design. Any bugs in the RTL are most likely to be detected during this phase. Once the model is in place, coding the SLEC script is straightforward and execution of the tool proceeds automatically. Initial mismatches usually indicate minor bugs in the model. Once these are corrected, a successful equivalence check may run in several minutes or up to several hours.

ACL2, on the other hand, requires considerable guidance by the user. Development of a formal proof of correctness is an exacting interactive process, resulting in a script consisting of a long sequence of lemmas and hints that are sufficient to guide the prover to the final result. Occasionally, this process reveals a bug in the model, which at this point indicates a bug in the RTL. A successful proof, once achieved, provides high confidence in the correctness of a design.

Subsequent RTL modifications necessitate rerunning the SLEC script and may also require corresponding changes in the model as well as the ACL2 script. The most common modifications, however, are driven by timing requirements and are not reflected at the level of abstraction of the model.

III. Chained Multiply-Add

Fused multiply-add is a single floating-point instruction of three arguments that combines addition and multiplication to compute $A * B + C$, rounding only after the full computation has completed. Chained multiply-add is a single instruction that is functionally equivalent to a sequence of two distinct instructions, rounding after the multiplication and again after the addition. The main advantage of FMA is the improved accuracy that results from the elimination of the intermediate rounding of the product, especially in the case of an effective subtraction when the product and the addend are comparable in magnitude. A disadvantage is the increased widths of the shifters and adders that are required to accommodate the double-width unrounded product. Since the two instructions have identical issue bandwidth, there is no appreciable difference in performance.

Arm GPUs currently use a single-precision FMA for most graphics applications. Experiments have shown, however, that minor variations in rendered images are invisible to the human eye, and therefore, the benefit of higher accuracy relative to a CMA is outweighted by considerations of area and power consumption, which are high priorities in graphics processing. Thus, we have undertaken a research effort to determine the viability of replacing this FMA with a CMA in future Arm graphics processors, focusing on the potential benefits with respect to area, power, and performance that may be achieved through reduction in hardware.

Our starting point is a legacy RTL module that shares hardware among a variety of operations including single- and half-precision floating-point addition, multiplication, and FMA with full support for subnormals and IEEE rounding, as well an integer dot product that combines 4 8×8-bit multiplications. The proposed design has been derived as a simplification of this module that replaces FMA with CMA. Note that our GPUs, unlike CPUs, are not subject to the architectural constraints imposed by IEEE-754. Thus, our CMA flushes subnormal parameters to zero and does not report exceptions (although an invalid multiplication or addition is encoded in a NaN payload). Both roundings are limited to the default IEEE rounding mode, "round-to-nearest-even".

We have extracted a RAC model of this simplified design that splits its functionality into three separate modules. This produces an expanded model with some redundant code, but it clarifies the design and simplifies the analysis, with no appreciable effect on the complexity of the equivalence check. We shall focus on the single-precision CMA component of the model as represented by the RAC function *CMA32*, the pseudocode version of which is displayed in Figure 4.[1] The first argument of this function selects one of three operations corresponding to the values *FADD*, *FMUL*, and *CMA* of the enumeration type `OP`. The remaining three arguments are 32-bit single-precision operands, *a*, *b*, and *c*, the third of which is used only in the *CMA* case. The returned value is a single-precision encoding of the result.

Correctness of *CMA32* is defined by formal high-level behavioral specifications of the *FMUL* and *FADD* operations, *fmul32spec* and *fadd32spec*, coded directly in ACL2. Each of these operates on two arguments and returns a single value, all of which are 32-bit SP encodings. The specification of the ternary *CMA* operation is simply the composition

$$cma32spec(a, b, c) = fadd32spec(fmul32spec(a, b), c).$$

These functions must handle the special cases of NaN, infinity, and subnormal operands, as well as overflow and underflow. They are much simpler, however, than the standard specifications of CPU operations presented in [11, Chap. 14], since all subnormal inputs and outputs are flushed to zero, only one rounding mode is supported, and the GPU does not generate exceptions.

In our ACL2 theory of floating-point arithmetic, the bit vector representations of rational numbers with respect to a given format are defined by decoding and encoding functions. Thus, if x is a single-precision encoding of a rational r, then $r = decode(x, SP)$ and $x = encode(r, SP)$. A rounding mode is formulated as a function that computes an approximation of a given number with a given number of bits of precision. In particular, the result of rounding r to n bits according to the round-to-nearest-even mode is computed as $RNE(r, n)$. Our instruction specifications are defined in terms of these functions. For example, if a and b are normal SP encodings and the sum of their values is also within the normal range, then the value of $fadd32spec(a, b)$ is

$$encode(RNE(decode(a, SP) + decode(b, SP), 24), SP).$$

[1]The corresponding FMA module is documented in [11, Chap. 22], and its RAC model may be found at `https://go.sn.pub/fma32`.

```
ui32 CMA32(OP op, ui32 a, ui32 b, ui32 c) {
  // Flush subnormals to 0:
  if (a[30:23] == 0) a[22:0] = 0;
  if (b[30:23] == 0) b[22:0] = 0;
  if (c[30:23] == 0) c[22:0] = 0;
  // Special cases:
  if (isNaN32(a) || isNaN32(b) || op == CMA && isNaN32(c) ||
      isZero32(a) || isZero32(b) ||
      op == FADD && a[31] != b[31] && a[30:0] == b[30:0]) {
    return specialCase(op, a, b, c);
  }
  else {
    // Multiplication:
    bool pSign;
    ui8 pExp;
    ui25 pMant;
    ui2 pInc = 0;
    if (op == FADD) {
      // FADD bypasses the multiplier and converts a
      // to the format of the multiplier outputs:
      pSign = a[31];
      pExp = a[30:23];
      pMant = a[22:0];
      pMant[23] = 1;
      pInc[0] = 0;
    }
    else {
      <pSign, pExp, pMant, pInc> = mulSP(a, b);
    }
    // Addition and rounding:
    if (op == CMA && isZero32(c) || op == FMUL) {
      // In this case, the adder is bypassed and the
      // result is derived from the multiplier outputs:
      ui32 sideRes = sidePath(pSign, pExp, pMant, pInc);
      if (op == CMA && isZero32(sideRes)) {
        sideRes[31] = sideRes[31] && c[31];
      }
      return sideRes;
    }
    else {
      ui32 addend = op == FADD ? b : c;
      return addSP(addend, pSign, pExp, pMant, pInc);
    }
  }
}
```

Fig. 4. CMA32

Execution of *CMA32* begins with the flushing of subnormal operands. Several trivial cases are handled specially and will be ignored here: (1) a NaN operand, (2) *a* or *b* zero, and (3) *FADD* with opposite infinities or a zero sum. The main computation is performed by the multiplier and the adder, *mulSP* and *addSP*. In the *FADD* case, *mulSP* is bypassed and the summands are passed directly to *addSP*. In the case of *FMUL*, or *CMA* with a zero addend, the adder is bypassed and the result is derived from the multiplier output by the function *sidePath* of Figure 5.

The multiplier computes the 24×24-bit product of the significands of *a* and *b*. In the original FMA, the implementation of this computation is deferred to synthesis. Since hardware is shared by operations on several data types, a consequence of that choice is that smaller data (11-bit half-precision significands and 8-bit integers) are padded or sign-extended to 24 bits in order to share one large multiplier. In the CMA version, an explicitly coded multiplier sub-array is designed to assemble the 48-bit single-precision product by combining the outputs of a set of narrow multipliers applied to segments of the operands. The benefit of this technique is that the same multipliers may be efficiently applied directly to the smaller data types, providing improvements in timing and area.

```
ui32 sidePath(bool pSign, ui8 pExp, ui25 pMant, ui2 pInc) {
  ui32 res = 0;
  res[31] = pSign;
  if (pExp == 0 && !pMant[24]) {
    if (pMant[22:0] == 0x7FFFFF) {
      res[23] = 1;
    }
  }
  else if (pExp == 0xFF ||
           pExp == 0xFE && pMant[24] ||
           pExp == 0xFE && pMant[22:0] == 0x7FFFFF
                        && pInc[0]) {
    res[30:0] = 0x7F800000;
  }
  else if (pMant[24]) {
    res[30:23] = pExp + 1;
    res[22:0] = pMant[23:1];
    res += pInc[1];
  }
  else {
    res[30:23] = pExp;
    res[22:0] = pMant[22:0];
    res += pInc[0];
  }
  return res;
}
```

Fig. 5. Computation of the rounded product

The crux of the CMA design is the interface between *mulSP* and *addSP*. The rounding of the product is not performed by *mulSP*, which instead produces an implicit representation of the rounded product, consisting of four components: *pSign*, *pExp*[7:0], *pMant*[24:0], and *pInc*[1:0]. The first two of these values are the sign and biased exponent of the unrounded product. Underflow and overflow are indicated by the two extreme values of *pExp*. In the normal case, *pMant* consists of the leading 25 bits of the 48-bit product of the significands, and therefore, either *pMant*[24] = 1 or *pMant*[23] = 1. If *pMant*[24] = 0, then *pInc*[0] is the rounding increment, to be added to *pMant*; if *pMant*[24] = 1, then *pInc*[1] is the rounding increment, to be added to *pMant*[24:1], and *pMant*[0] is ignored. In the multiplier bypass case, *pSign*, *pExp*, and *pMant* are simply extracted from a, and since no rounding is required and *pMant*[24] = 0, the rounding increment *pInc*[0] is set to 0.

Since the function *sidePath* derives the encoded product from the multiplier outputs, its definition is a precise formulation of the product representation scheme and is therefore the basis of the statement of correctness of the multiplier. The following is an informal version of the formal statement that has been proved by ACL2:

Lemma 1: If $op \neq FADD$, then

$$sidePath(pSign, pExp, pMant, pInc) = fmul32spec(a, b).$$

The adder performs the shift required to align the product with the addend and computes the sum of the product, addend, and rounding increment. The result is the same as if the rounding were performed before the addition, but with the aid of a half-adder applied to the three summands, only one carry-propagate addition is required. Moreover, in contrast to the 52-bit adder used in the original FMA design, the width of the

adder is reduced to 30 bits. In addition to the savings in area and power, the latency of the operation is thereby decreased from 4 cycles to 3.

Rounding of the sum is implemented through the use of an auxiliary mode known as "round-to-odd" [4], [11], [13]: $RTO(r, n)$ is computed by truncating r to n bits and setting the least significant bit in the event that it is 0 and the result is inexact. As documented in [11, Sec. 6.4], the ACL2 RTL library contains a number of properties of this mode that account for its utility in the implementation of floating-point addition, including the following:

If $n > m + 1$, then $RNE(RTO(r, n), m) = RNE(r, m)$.

Appealing to this result, *addSP* applies *RTO* in an intermediate rounding of the sum (by appending a sticky bit) before the final rounding to 24 bits is computed by *RNE*.

The behavior of *addSP* in the *CMA* case is characterized by the following:

Lemma 2: Assume $op = CMA$ and c is nonzero. Let

$$p = sidePath(pSign, pExp, pMant, pInc).$$

Then

$$addSP(c, pSign, pExp, pMant, pInc) = fadd32spec(c, p).$$

In the *FADD* case, the same result holds with p and c replaced with a and b. Overall correctness of the module follows easily from the lemmas stated above.

Establishing the appropriate level of detail of the RAC model involved some experimentation. The equivalence check did not converge until the RTL multiplier sub-array was accurately modeled at the bit level. We also found it necessary for the function *addSP* to adhere to the structure of the RTL adder, including the rounding process. Convergence also required decomposition of the check by means of explicit intermediate maps between the inputs of *addSP* and the corresponding RTL signals. On the other hand, several critical optimization techniques employed in the RTL are not reflected in the model. These include an explicitly code Han-Carlson adder [11, Sec. 8.2] and a sophisticated leading zero anticipator [11, Sec. 8.3]. In the verification of similar double-precision adders, we have found it necessary to model such features in detail, but in this single-precision case, they were eliminated from the model without sacrificing convergence. This resulted in a minor increase in the SLEC run-time, but simplified the model and significantly eased the burden on ACL2. The final model consists of 36 kilobytes of code, 1/6 of the 216 kilobytes of Verilog source.

IV. Conclusion

As for all projects of this sort in our experience, verification of the chained multiply-add module involved two significant efforts: design of the RAC model and development of the correctness proof, each of which occupied one of the authors for several weeks. The modeling effort required a detailed study of the RTL in consultation with the designers. Once the design was understood, coding the model was straightforward, although some debugging with SLEC was necessary. The proof effort, as usual, began with an informal but mathematically rigorous hand-written proof based on previously established properties of the underlying algorithms and techniques. This was interactively transformed into a sequence of ACL2 lemmas, which, in combination with the ACL2 RTL library, led to a general correctness theorem. The final ACL2 proof script consists of 64 definitions and 506 lemmas.

Owing to the close correspondence between the model and the RTL, the development of the SLEC script that guides the equivalence check was a much simpler task, utilizing only the most basic features of the tool. After an initial run failed, we inserted the intermediate maps at the multiplier-adder interface, which were suggested by previous experience with similar designs. The check succeeded with no further guidance.

The SLEC and ACL2 scripts are executed independently. We run SLEC on Arm clusters via LSF, and ACL2 on an Apple laptop. The equivalence check runs in 19 minutes, while the ACL2 proof takes 2 minutes. We note that a more conventional attempt, using a commercial model checker, to establish the correctness of the multiplier sub-array, a small component of the design, ran for 24 hours without success.

TABLE I
Area and power reductions when replacing FMA with CMA

	CMA	CMA with multiplier sub-array
Area reduction	18%	24%
Dynamic power reduction	21%	15%
Leakage power reduction	18%	23%
Total power reduction	20%	18%

The baseline FMA module consists of 4 SP FMA pipelines, 8 HP FMA pipelines, and 16 8-bit integer multiplication pipelines. It is replicated 128 times per core and accounts for 13% of the core's area and 7% of its power consumption. The test CMA module is similarly structured, with the 4-cycle SP pipeline reduced to 3 cycles. The results of our experiment are summarized in Table I. An initial version of the CMA, prior to the implementation of the multiplier sub-array, exhibited savings of 18% in area in 20% in power. With the sub-array, area was further reduced, but a small increase in power was observed. The reason for this is that total power consumption comprises dynamic power, which is a function of switching activity, and leakage power, which is determined by the number and structure of the gates. Since leakage power is proportional to area, we see a further improvement in that component with the introduction of the sub-array. On the other hand, since the sub-array combines SP and HP logic to save area, a benchmark that runs HP applications can cause switching activity in SP logic, resulting in higher dynamic power and a slight increase in total power. Given the importance of area in this context, the overall effect of the sub-array is clearly beneficial. Moreover, the applicability of the experiment to future implementations is ensured by our

verification results.

REFERENCES

[1] S. Boldo and G. Melquiond, "Flocq: A unified library for proving floating-point algorithms in Coq," in *20th IEEE Symposium on Computer Arithmetic*, Tubingen, Germany, July 2011, pp. 243–252.

[2] M. Daumas, L. Rideau, and L. Thery, "A generic library for floating-point numbers and its application to exact computing," in *Theorem Proving in Higher Order Logics, Lecture Notes in Computer Science*, Heidelberg, Germany, 2001, pp. 169–184.

[3] J. Harrison, "A machine-checked theory of floating point arithmetic," in *12th International Conference on Theorem Proving in Higher Order Logics*, Y. Bertot, G. Dowek, A. Hirschowitz, C. Paulin, , and L. Thery, Eds., Nice, France, 1999, pp. 113–130.

[4] 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," *IEEE Transactions on Computers*, vol. 47, no. 9, September 1998.

[5] D. M. Russinoff, "A mechanically checked proof of IEEE compliance of the AMD-K7 floating point multiplication, division, and square root instructions," *London Mathematical Society Journal of Computation and Mathematics*, vol. 1, pp. 148–200, December 1998, available at http://www.russinoff.com/papers/k7-div-sqrt.html.

[6] D. Russinoff, M. Kaufmann, E. Smith, and R. Sumners, "Formal verification of floating-point RTL at AMD using the ACL2 theorem prover," in *17th IMACS World Conference: Scientific Computation, Applied Mathematics and Simulation*, Paris, France, 2005.

[7] W. Hunt, S. Swords, J. Davis, and A. Slobodova, "Use of formal verification at Centaur Technology," in *Design and Verification of Microprocessor Systems for High-Assurance Applications*, D. Hardin, Ed. Springer, 2010, pp. 65–88.

[8] ACL2 Web site, https://www.cs.utexas.edu/users/moore/acl2/.

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

[10] ——. Sequential Logic Equivalence Checker. [Online]. Available: https://www.mentor.com/products/fv/questa-slec

[11] D. M. Russinoff, *Formal Verification of Floating-Point Hardware design: A Mathematical Approach*, 2nd ed. Springer, 2022.

[12] J. Levine, *Flex and Bison*. Oreilly Media, 2009.

[13] S. Boldo and G. Melquiond, "Emulation of a FMA and correctly rounded sums: Proved algorithms using rounding to odd," *IEEE Transactions on Computers*, May 2008.