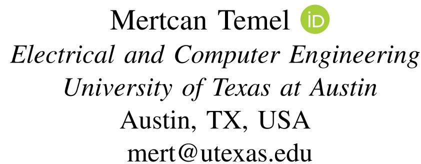
Formal Methods in Computer-Aided Design 2021

Sound and Automated Verification of Real-World RTL Multipliers


Mertcan Temel
Electrical and Computer Engineering
University of Texas at Austin
Austin, TX, USA
mert@utexas.edu

Warren A. Hunt, Jr.
Computer Science
University of Texas at Austin
Austin, TX, USA
hunt@cs.utexas.edu

***Abstract*—We have developed an algorithm, S-C-Rewriting, that can automatically and very efficiently verify arithmetic modules with embedded multipliers. These include ALUs, dot-product, multiply-accumulate designs that may use Booth encoding, Wallace-trees, and various vector adders. Outputs of the target multiplier designs might be truncated, right-shifted, or a combination of both. We evaluate the performance of other state-of-the-art tools on verification problems beyond isolated multipliers and we show that our method applies to a broader range of design techniques encountered in real-world modules. Our verification software is verified using the ACL2 theorem prover, and we can soundly verify 1024x1024-bit isolated multipliers and similarly large dot-product designs in minutes. We can also generate counterexamples in case of a design bug. Our tool and benchmarks are available online.**

***Index Terms*—Formal Verification, Integer Multipliers, Hardware Verification, Arithmetic Circuits, ACL2, Term-rewriting**

I. Introduction

Integer multipliers are fundamental building blocks for general-purpose (e.g., CPUs and GPUs), image, communications, and cryptographic processors. Multipliers are used to implement dot-product, division, square-root, and floating-point operations; in turn, these operations find their way into graphics, cryptography, and signal processing systems. In some cases, such as cryptographic processors, integer multipliers might be used to multiply numbers as large as 1024 bits.

Given the ubiquity of multipliers, it is crucial to have a sound verification method for designs that include multipliers. However, the formal verification process of multipliers is still a challenge, especially for the most common design approaches such as Wallace tree and Booth encoding. Decision-procedure-based tools such as BDDs, SAT solvers do not scale [1], [2]. In recent years, multiplier verification efforts have shifted towards using computer algebra methods [2]–[6] and they have yielded more promising results. However, these studies focused heavily on isolated multiplier designs, and they do not perform well (if at all) for multipliers with truncated output (e.g., a 32x32-bit multiplier with a 32-bit output). Studies that explore the verification problem of embedded multipliers (e.g., multiply-accumulate, dot-product) have been limited, and they do not support designs with Wallace tree and Booth encoding [1]. Additionally, only one computer-algebra-based tool [3] provides a system to check the correctness of the proof itself, leaving open the possibility that these tools might claim a design to be correct when the design is actually flawed.

In our previous work [7], we proposed a method to verify integer multipliers efficiently and automatically. Using the ACL2 theorem proving system, we developed a provably correct verification mechanism based on term-rewriting. This method has been shown to quickly verify a wide range of integer multiplier designs (e.g., 1024x1024-bit multipliers with simple partial products have been verified in less than 10 minutes). However, our focus concerned only untruncated isolated multiplier designs. Moreover, we did not discuss how the algorithm performs with buggy designs.

We have expanded our method and we have been able to:

- improve proof-time performance by a factor of 2 or more;
- verify designs beyond untruncated isolated multipliers;
- and quickly generate counterexamples.

Additionally, we retain the same level of proof automation and keep our tool provably correct.

In this paper, we aim to explore the verification problem of multipliers on more complex designs than explored in previous verification studies and deliver our solutions. We provide examples of complex multiplier architectures with optimizations that can be encountered in real-world designs. We discuss how existing state-of-the-art verification tools perform on such modules. Finally, we present our improved method and show that we can verify these complex designs very efficiently. For example, we can verify 64x64-bit isolated multipliers or similar designs within seconds and 1024x1024-bit isolated multipliers or similar dot-product designs in 5 minutes, no matter which design algorithm is used.

This paper is structured as follows. Sec. II summarizes the most common design algorithms for isolated and embedded multipliers. We show why it is important to develop a verification method for embedded and truncated multipliers and why it is not enough to have a verification tool only for isolated multipliers. In Sec. III, we summarize the related work from the most recent and/or prominent studies. Sec. IV recapitulates our term rewriting algorithm from our previous work and introduces some of its recently discovered limitations. Sec. V discusses our new improvements so that we can verify more designs with better efficiency and generate counterexamples

https://doi.org/10.34727/2021/isbn.978-3-85448-046-4_13

This article is licensed under a Creative Commons Attribution 4.0 International License

for buggy modules. Sec. VI describes how our lemmas are implemented and applied. Finally, we show our experiment results in Sec. VII and compare our performance with other state-of-the-art multiplier verification tools.

II. Multiplier Architectures

There are various algorithms to design RTL multipliers and integrate them in other arithmetic modules such as a multiply-accumulate (MAC). The difficulty of verifying these modules depends on the design algorithm. Some algorithms bring out clean and regularly structured modules, and some and most commonly used algorithms produce complex structures. This section elaborates on the verification problem by summarizing common algorithms to design multipliers and how they are implemented in other arithmetic circuits.

A. Isolated Multipliers

An isolated multiplier is a circuit with two bit-vector inputs and one bit-vector output. The output vector represents an integer equivalent to the multiplication of the input vectors, which can be signed or unsigned integers. Isolated multipliers are often implemented in two stages: partial product generation and partial product summation.

Partial products can be generated by multiplying (i.e., logical AND) each input bit with each other as in primary school multiplication. For signed numbers, the input numbers need to be sign-extended, in which case the Baugh-Wooley [8] sign extension technique can be used to lower the implementation area. Booth encoding [9] (particularly radix-4) is a more common and efficient way to generate partial products. Booth encoding incorporates more than two input bits at a time when generating partial products. This can provide more parallelism and fewer partial products. However, Booth encoding makes a circuit's structure and logic more complex, making it more difficult to reason about the circuit.

There are numerous methods to sum partial products in hardware. Unlike primary school multiplication, hardware algorithms do not sum partial products one column at a time, from right to left. Summations are performed more locally with unit adders such as half and full adders. An array multiplier is a simple example that is built with such unit adders following a shift-and-add methodology. Array multipliers have a regular structure, which makes it straightforward to verify them. However, they can have a large gate delay (i.e., propagation delay). On the other hand, Wallace-tree-like multipliers [10], such as Dadda tree [11], provide more parallelism. These summation tree algorithms sum partial products with less propagation delay and only slight changes in the implementation area. Designers can also utilize low gate-delay vector adders, such as Brent-Kung [12], Ladner-Fischer [13], and conditional sum, as a final stage adder to get the multiplication result. This can make Wallace-tree-like algorithms with complex final stage adders more preferable for hardware applications, but their irregular structures make the verification problem difficult, especially when paired with Booth encoding.

We should also note that an isolated multiplier implementation may not always return the full multiplication result. Instead, the result might be truncated, right-shifted, or a combination of both. For example, when two 32-bit numbers are multiplied, a lossless multiplier would output a 64-bit number. On the other hand, if the design only calculates the lower, say, 32-bits of the result, we say that the result is truncated. Similarly, when, say, only the upper 32-bits of the result are returned from the multiplier, we say that the result is right shifted. If only the middle portion of the result is returned, which may happen in fixed-point arithmetic, we say that the result is right shifted and truncated. Some designs implement rounding or saturation when a certain portion of the result is discarded when truncating and/or shifting.

B. Simple Arithmetic Modules with Embedded Multipliers

Integer multipliers can be implemented in various arithmetic modules such as MAC, dot-product, and floating-point arithmetic units. This section summarizes how a MAC module can be implemented in hardware.

A simple MAC computes $a*b+c$, where a, b and c are bit-vectors. When designing a MAC module, one may implement an isolated multiplier that computes $a*b$ and a vector adder that adds c to the multiplier's output. To verify such a MAC module, one can decompose the design, use different tools to verify the isolated multiplier and the final adder separately, and compose the proofs to show that the overall MAC module is correct. However, this design methodology uses two vector adders consecutively (one vector adder as part of the isolated multiplier and one for adding c). Vector adders can make up a large portion of the gate delay (and/or area) in such circuits, and this design technique can increase the gate delay considerably, making this approach a poor design choice.

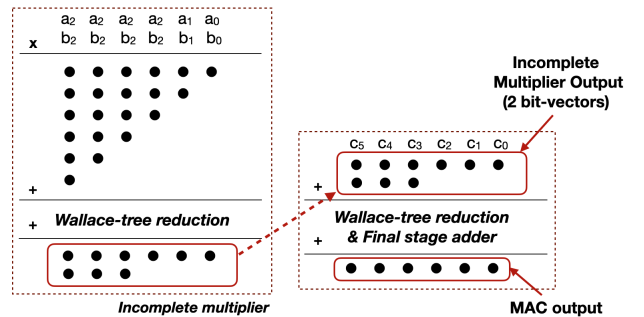


Fig. 1. An efficient way to compute MAC result

Fig. 1 shows an alternative approach that uses only one vector adder. This MAC module does *not* implement a complete isolated multiplier. Instead, it uses an *incomplete* multiplier. We define incomplete multipliers as modules that multiply two bit-vectors but do not use a final stage adder to return the complete multiplication result; instead, they return the two bit-vectors generated after the Wallace-tree reduction (summing these two vectors would give the multiplication result). This output form is also referred to as *redundant*

form. After the incomplete multiplication, the two bit-vector outputs are summed together with the addend (c) using another Wallace tree and a vector adder. This can be a preferable design approach as it provides better gate-delay performance. However, it removes the boundaries between multiplication and summation, which complicates the job of a verification engineer. Further complicating verification, an alternative design technique may sum c with the initial partial products with a single Wallace-tree and vector adder, which can remove the boundaries even further. In such cases, we cannot simply decompose the design and use a multiplier verification tool that works only with isolated multipliers.

We can see similar design methodologies in other modules. For example, a dot product design may use multiple *incomplete* multiplier modules and sum all the output vector pairs together in another summation tree using a Wallace-tree and a final stage adder. This method would prevent the increase in area and gate delay by using only one final stage adder in the overall design. Similarly, a floating-point module implementing FMA (fused multiply-add) may use an incomplete integer multiplier.

C. Multi-purpose Multipliers

Some processing units may implement multipliers for various arithmetic operations with different operand sizes. For example, x86 chips have many integer multiplication instructions such as PMADDWD (multi-lane multiply and add together, in other words, dot-product), PMULHW (multi-lane multiply and store upper half of the result), and PMULLW (multi-lane multiply and store lower half). Multiplier circuits can occupy a large implementation area, and it is common for such instructions to share resources and reuse multiplier modules.

We have created an example arithmetic circuit that shows how multiplier modules can be reused for different operations. We call this arithmetic unit *integrated multipliers* whose schematic diagram is shown in Fig. 2. This design multiplexes various multipliers and adders to perform 4-point 32-bit dot-product, 1-lane 64-bit multiply-accumulate, or 4-lane 32-bit multiply-accumulate with options to return lower or upper significant halves of the result. This module also includes an accumulator register that can be used, for example, to perform an 8-point 32-bit dot-product in two clock cycles, or 12-point 32-bit dot-product in three clock cycles, and so on. The mode of operation is determined by the control signal `mode`.

This module implements four identical 32x32-bit *incomplete* multipliers whose inputs are two 32-bit numbers with an additional sign bit and whose outputs are two bit-vectors. Depending on the mode of operation, the outputs of these multipliers are summed with another summation tree, and the final result is calculated with vector adders. The datapaths for 32-bit MAC and dot-product operations are as described in the previous section (Sec II-B). This module also supports 64-bit operands, in which case the outputs of the 32x32-bit incomplete multipliers are appropriately shifted, sign-extended, and summed to calculate the 64x64-bit multiplication result. We call such operations *merged multiplication*, where multiple

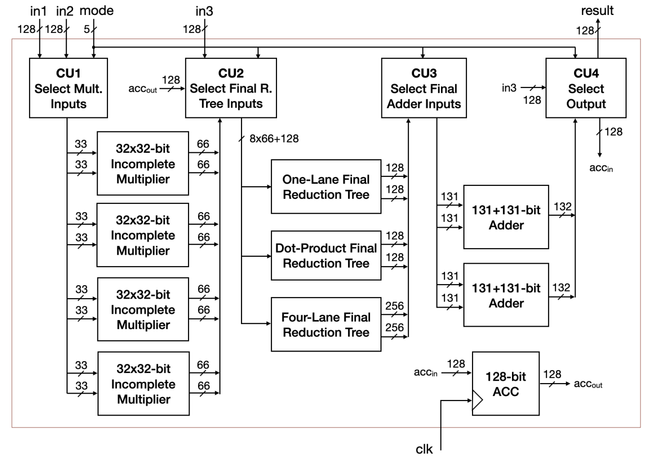


Fig. 2. The circuit diagram of integrated multipliers, our example arithmetic unit.

smaller multipliers are used to implement a larger multiplier. The module can also add a number to the 64x64-bit multiplication result and make this a 64-bit MAC operation.

We can verify this design for each possible mode of operation. For example, we can set the `mode` signal to perform dot product and check if the result matches the mode's specification. Industrial designs are often much more intricate than this module; however, it is often possible to reason about one arithmetic operation at a time. Then, the verification problem becomes as complex as verifying a single arithmetic operation.

III. Related Work

The verification problem of multipliers continues to have a great deal of research interest, and researchers offer new techniques every year. This section covers the most recent and prominent studies that attempt to solve this problem, particularly for RTL designs with Booth encoding and Wallace-tree-like structures.

A. BDDs, BMDs, SAT and SMT Solvers

Automated and well-studied generic tools and methods such as BDDs, SAT, and SMT Solvers can theoretically be used to verify multiplier designs. However, it has been shown that these methods do not scale for designs larger than 12x12-bit multipliers [1], [2]. SAT solvers may scale better when generating counterexamples for buggy designs. Some success has been achieved with BMDs but only for regularly structured multipliers [14]. On the other hand, these automated tools may be used to verify some multiplier design components, such as the final stage adder [3].

B. Computer Algebra Methods

In computer algebra-based methods, multiplier circuits are modeled with a set of polynomials. Basic logical gates of a circuit are represented in terms of algebraic expressions (e.g., $\forall x, y \in \{0, 1\}\ x \vee y = x + y - xy$) as well as the multiplication result (see Example 1 for a 2x2-bit unsigned multiplier specification). The algebraic representation on its

own does not scale when verifying multipliers. Researchers implement various heuristics and optimizations that are specific to multiplier designs to achieve efficient and practical results. A notable optimization is identifying the logic from adder modules implemented in target multiplier designs [3], [4], [15]–[17].

Example 1. $4a_1b_1 + 2a_1b_0 + 2a_0b_1 + a_0a_0$

Computer algebra methods have made a lot of progress towards the multiplier verification problem. However, these studies have focused mainly on isolated multipliers with untruncated outputs and the same operand sizes (nxn-bit multipliers with 2n-bit outputs). This makes it more difficult to utilize them for real-world designs where truncation, shifting, and integration with other arithmetic operations are common (See Sec. II).

Ciesielski et al. [1] showed that their method could be used for other multiplier-centric arithmetic operations, such as MAC; however, they showed that they only verified multiplier modules with regular structures. The benchmarks and their verification tool are not provided. We do not know of any publicly available tool that can scale and automatically verify designs such as MAC and dot-product. The underlying theory used by the computer-algebra methods may support verification of such arithmetic circuits. However, some optimizations that make these tools efficient may or may not be directly applicable to modules beyond isolated multipliers.

Verifying multipliers whose output is truncated or shifted is difficult for the computer algebra approach. Su et al. [18] discussed why computer algebra techniques are inefficient when verifying truncated arithmetic circuits. They stated that intermediate expressions, which are manageable in untruncated modules, can grow exponentially in truncated designs. They suggested a method to reconstruct a truncated multiplier into a complete multiplier by adding missing elements before verification. They did not discuss the soundness of their approach, their experiments were only on simple multipliers, and the benchmarks and the tool are not provided. Kaufmann et al. [3] suggested using modular arithmetic and defined a specification in the ring $\mathbb{Z}_{2^n}[X]$ where n is the multiplier output size. They showed that this approach works on a simple multiplier model, but our experiments with RTL designs resulted in time-out. We are not aware of any computer algebra studies that can verify truncated and/or shifted RTL multipliers.

C. Industrial Methods

Verification efforts of commercial multipliers often involve a great deal of manual work. A common method is to create a simple reference design that is structurally close (isomorphic) to the original and then repeatedly equivalence-check a litany of ever-increasingly complex designs [19]. Some engineers verify reference designs using mechanized proof systems [20]. Another common analysis method is to decompose a design into smaller parts, reason about these parts separately, and then compose these proofs into a top-level theorem [21]–[23]. Finding a workable decomposition and combining individual proofs of multiplier fragments can be a cumbersome task. Such methods help formal verification engineers verify various multiplication operations such as multiply-accumulate and dot-product; however, this usually entails extensive manual effort. Moreover, these proofs are often design-specific, and even a slight change in the design might cause a previous proof procedure to fail.

IV. S-C-Rewriting Algorithm

In our previous work [7], we introduced a verified term-rewriting algorithm that can verify a wide range of isolated multiplier designs more quickly than the other state-of-the-art tools. In this section, we summarize this term-rewriting algorithm and discuss its recently discovered limitations.

We use the ACL2 theorem prover to verify and run our multiplier verification tool. ACL2 is an interactive and automated theorem proving system, and a programming language that is used by both industry and academia [24]. For a target multiplier design, we try to prove conjectures of the form given in Listing 1. `defthm` is a commonly used utility by ACL2 users, and it asks the ACL2 system to check conjectures. On the left hand side, we specify symbolic simulation of a multiplier design representation. We use the SVL semantics [25] to simulate designs, which are automatically translated from Verilog (our verification tool can be used with other simulators as well). The right hand side has the multiplier specification; in this example, the target multiplier module returns a 128-bit number equivalent to the multiplication of two 64-bit signed numbers.

Listing 1. A correctness conjecture for a signed 64x64-bit isolated multiplier

```
(defthm multiplier_is_correct
  (implies (and (integerp a)
                (integerp b))
    (equal (simulate :inputs (a b)
                     :design <signed_64x64_mult>)
           (truncate 128
                     (* (signext 64 a)
                        (signext 64 b))))))
```

We prove such conjectures by rewriting both sides of the equality to fixed final forms. We define two functions s (short for *sum*) and c (short for *carry*) as given in Def. 1. The target representations for the first few output bits of some modules (half, full, vector adders, and multipliers) are given in Table I. Our goal is to rewrite all such modules/operations to this form. We call this s-c representation or s-c form.

Definition 1. Functions s and c are defined as follows.

$$\forall x \in \mathbb{Z}\ s(x) = mod_2(x)$$
$$\forall x \in \mathbb{Z}\ c(x) = \left\lfloor \frac{x}{2} \right\rfloor$$

While verifying multiplier designs, we wish not to work with the logical definition of adder modules but instead work with their s-c representations. The SVL semantics allow hierarchical reasoning such that if we previously prove that symbolic simulation of an adder module can be replaced with this s-c form, then the SVL system can use this form (as

TABLE I
TARGETED FINAL FORMS FOR SOME MODULES/FUNCTIONS

Function	out_2	out_1 / c_{out}	out_0 / s_{out}
Half-adder	-	$c(a+b)$	$s(a+b)$
Full-adder	-	$c(a+b+c_{in})$	$s(a+b+c_{in})$
Bit-vector addition $a+b$	$s(a_2+b_2$ $+c(a_1+b_1$ $+c(a_0+b_0)))$	$s(a_1+b_1$ $+c(a_0+b_0))$	$s(a_0+b_0)$
Bit-vector multiplication $a*b$	$s(a_0b_2+a_1b_1$ $+a_2b_0$ $+c(a_1b_0+a_0b_1$ $+c(a_0b_0)))$	$s(a_1b_0+a_0b_1$ $+c(a_0b_0))$	$s(a_0b_0)$

opposed to the adder's logical definition) while expanding the definition of multiplier designs. Therefore, we first prove that each distinct adder module can be represented with the s-c form. We use a term-rewriting algorithm to carry out the proofs for adder modules [7]. Since verifying adders is straightforward [3], we omit this rewrite algorithm here for brevity. After the adder proofs, we start verifying the target multiplier design. As we expand the definition of the multiplier, our program replaces each instance of its adder modules automatically with their s-c representation.

Using the s-c form for adders instead of their logical definitions can bring about simpler expressions representing the output bits of a multiplier. An example of such an expression is given in Example 2 for a Wallace-tree multiplier with simple partial products.

Example 2. The 4th LSB of a Wallace-tree multiplier output when its adders are represented in the s-c form:

$$\begin{aligned}&s(\ s(\ s(a_3b_0+a_2b_1+a_1b_2)\\&\qquad +a_0b_3\\&\qquad +c(a_2b_0+a_1b_1+a_0b_2))\\&\quad +c(s(a_2b_0+a_1b_1+a_0b_2)+c(a_1b_0+a_0b_1)))\end{aligned}$$

We rewrite such terms to make them syntactically equivalent to our target final form. To do that, we define a set of lemmas of the form $lhs = rhs$ such that terms that match lhs are replaced with rhs with appropriate term bindings. All lemmas are proved using ACL2 and we omit the proofs here.

We investigated such terms from multiplier designs and realized that we could rewrite and simplify nested calls of s with Lemma 1. Rewriting with this lemma when applicable can simplify the term from Example 2 to the form given in Example 3.

Lemma 1. $\forall x, y \in \mathbb{Z}\ s(s(x)+y) = s(x+y)$

Example 3. Example 2 simplified with Lemma 1:

$$\begin{aligned}&s(a_3b_0+a_2b_1+a_1b_2+a_0b_3\\&\quad +c(a_2b_0+a_1b_1+a_0b_2)\\&\quad +c(s(a_2b_0+a_1b_1+a_0b_2)+c(a_1b_0+a_0b_1)))\end{aligned}$$

Now, we observe more than one instance of c on the same summation level. We rewrite and simplify them by a set of lemmas. Lemmas 2-5 are applied to the term as rewrite rules, where the function d is defined as $\forall x \in \mathbb{Z}\ d(x) = \frac{x}{2}$. Then, we get the term in Example 4. This is syntactically equivalent to our target form for the 4th output bit, and we can conclude that the multiplier is correct for this output bit.

Lemma 2. $\forall x, y \in \mathbb{Z}\ c(x)+c(y) = d(x+y-s(x)-s(y))$

Lemma 3. $\forall x, y \in \mathbb{Z}\ c(x)+d(y) = d(x+y-s(x))$

Lemma 4. $\forall x, y \in \mathbb{Z}\ d(x)+d(y) = d(x+y)$

Lemma 5. $\forall x \in \mathbb{Z}\ d(-s(x)+x) = c(x)$

Example 4. Example 3 rewritten with Lemma 2-5:

$$\begin{aligned}&s(a_3b_0+a_2b_1+a_1b_2+a_0b_3\\&\quad +c(a_2b_0+a_1b_1+a_0b_2\\&\qquad +c(a_1b_0+a_0b_1)))\end{aligned}$$

As Booth encoding can incorporate multiple input bits when generating partial products, we can see operators for logical gates (e.g., logical OR, XOR) when verifying Booth encoded multipliers. We use a few more simple lemmas to simplify terms from Booth encoding and we derive the same final form. These lemmas, along with examples, are provided in our previous work [7], and we omit them here for brevity. These extra lemmas are triggered automatically when Booth encoding is present, and they do not affect other proofs when simple partial products are used.

Once we are done rewriting the left-hand side in Listing 1, we rewrite the right hand side (specification) to the same form through proved rewrite rules from our library. When we see that the two sides are syntactically equivalent, we conclude that the multiplier is correct.

Note that our target representation has a separate term for each output bit whereas the computer algebra methods specify all output bits with a single expression (see Example 1). This makes it easier for our method to verify designs whose output may be manipulated on bit level such as by truncating, shifting, and bit-masking.

Example 5. The first instance of a_2b_0 in Example 2 is replaced by a_2b_1 to simulate a bug. Then, the rewriting algorithm returned:

$$\begin{aligned}&s(\ a_3b_0+a_2b_1+a_1b_2+a_0b_3\\&\quad +d(-s(a_2b_1+a_1b_1+a_0b_2)\\&\qquad -s(a_2b_0+a_1b_1+a_0b_2+c(a_1b_0+a_0b_1))\\&\qquad +s(a_2b_0+a_1b_1+a_0b_2)\\&\qquad +a_2b_1+a_1b_1+a_0b_2\\&\qquad +c(a_1b_0+a_0b_1)))\end{aligned}$$

In our previous work, we did not investigate what happens when the design has a bug and whether or not the algorithm can work beyond isolated multipliers. If our program cannot verify a multiplier for some reason, it returns a term rewritten with our lemmas. For example, when we introduce a simple bug to the term in Example 2, the described rewriting algorithm will return the term given in Example 5. The resulting term is larger than the initial term, and the gap can grow even larger for big designs. When a proof attempt fails, either due

to a bug in the design or some problem with our verification method, resulting terms are often very large and users do not receive a useful feedback from the program.

A proof attempt might fail even when the target design is correct. We have found such an instance and we could not verify some Booth encoded *merged* multipliers (See Sec. II) larger than 16x16-bit multiplication. Since the resulting terms are so large, we could not understand if there was a missing lemma that could help finish the proofs. We encountered similar issues with some dot-product and MAC designs, and we were likewise unable to verify them.

V. Improvements to S-C-Rewriting

We have developed and experimented with various alternatives to the existing S-C-Rewriting algorithm. Our goal is to verify designs beyond isolated multipliers and return small terms if a proof attempt fails due to a design bug or a problem in the verification system. We have found a rewriting scheme that meets these goals. Instead of rewriting c terms with Lemmas 2-5, we use only the new Lemma 6. Similar to Lemma 1, this lemma extracts the arguments of inner s calls but it also creates a byproduct $-c(x)$.

Lemma 6. $\forall x, y \in \mathbb{Z}\ c(s(x) + y) = c(x + y) - c(x)$

When the given designs are correct, this lemma helps simplify multiplier designs without needing Lemmas 2-5. We have also seen that when this lemma is used, proofs are actually much faster for Booth encoded designs as well as array multipliers by an order of magnitude (see Sec. VII).

For cases where a proof-attempt fails, we apply another lemma (Lemma 7) to cancel out common terms shared between the specification and the design. After all our lemmas are applied and the design is simplified, the rewriter compares if the simplified design is syntactically equivalent to the specification for each output bit. If they are not, then we rewrite the term that represents the equivalence of these two sides with Lemma 7.

Lemma 7. $\forall x, y \in \{0, 1\}\ (x = y) \iff (s(x + y) = 0)$

Lemma 6 and Lemma 7 help the program return a much smaller term if a proof attempt fails. Assume that we are rewriting a term that checks the equivalence of the term from Example 2 to its specification (Example 4). When we introduce the same bug from Example 5 to this term, our new rewrite method will return the term in Example 6.

Example 6. When the same bug from Example 5 is rewritten with the improved rewriting algorithm:

$$\begin{aligned} s(\ & c(a_0b_2 + a_1b_1 + a_2b_0) \\ & +c(a_0b_2 + a_1b_1 + a_2b_1)) \\ =\ & 0 \end{aligned}$$

As seen in this example, the returned term is considerably smaller than what we would get from the older algorithm (Example 5). We have observed the same behavior with larger multipliers so much so that the returned term can sometimes give a hint as to where the bug exists within the design. Moreover, since these terms are often small, we use the FGL [26] or the GL [27], [28] utilities in ACL2 to send such returned terms to an external SAT Solver. We have seen through our experiments (Sec. VII) that SAT Solver can return a counterexample very quickly from simplified terms.

As noted in Sec. IV, proof attempts may fail even when the design is correct. This was the case with our initial term rewriting strategy for some Booth encoded merged multipliers and some MAC and dot-product modules. Since the returned terms are smaller with the modified term-rewriting, we could find the source of the problem and determine the missing lemmas needed to verify these designs. We found out that we simply need to rewrite some c and s instances in terms of logical operators (see Lemmas 8-11) when certain syntactic conditions on their arguments are met. Those conditions are: the arguments x, y and z (if available) need to be instances of the logical AND ($\wedge$) function only, and the operands in y and z (if available) need to be a subset of the operands of x. For example, we can apply Lemmas 8-9 if $x = a \wedge b \wedge c \wedge d$, $y = a \wedge c$, and $z = b \wedge c$ but we cannot apply it if $z = b \wedge e$. The resulting terms from these rewrites are simplified the same way as Booth encoding logic. We have these strict syntactical conditions so that the rewriting system is more deterministic and there is minimal effect on the verification procedures for other designs. We leave these lemmas enabled in our program, and they help automatically verify the previously failed designs, such as merged multipliers.

Lemma 8. $\forall x, y, z \in \{0, 1\}\ c(x+y+z) = x \wedge y \vee x \wedge z \vee y \wedge z$

Lemma 9. $\forall x, y, z \in \{0, 1\}\ s(x + y + z) = x \oplus y \oplus z$

Lemma 10. $\forall x, y \in \{0, 1\}\ c(x + y) = x \wedge y$

Lemma 11. $\forall x, y \in \{0, 1\}\ s(x + y) = x \oplus y$

Additionally, we tested this method with another simulation tool, SVTV [24], to show that our method does not have to be used with the SVL system. The SVTV system sources designs from Verilog and flattens them before (symbolic) simulation. We found a way to mark the adder modules before flattening to easily rewrite them in the s-c form. We omit the details here for brevity, and the readers may refer to our online tutorials for details (http://mtemel.com/fmcad21).

VI. Implementation

All of our rewriting system consists of lemmas of the form $lhs = rhs$. When patterns found in conjectures match lhs, they should be replaced by rhs. Since conjectures for multiplier designs may yield very large terms, we implement a scalable mechanism to find such patterns and apply our lemmas.

We use a verified rewriter [29] that follows an inside-out rewriting strategy [30], [31]. Example 7 shows how a rewrite rule can modify a term from inside out. We can prove the associativity of summation (see the upper-left corner) using the existing libraries and the built-in axioms in ACL2. The

`defthm` event saves the proved lemma as a rewrite rule. When this rewrite rule is in the system, we can apply it to terms whenever the left hand side pattern finds a match. Assume that this is the only enabled rule, and we would like to prove another conjecture which contains the term shown on the upper-right corner. Since the rewriter performs inside-out rewriting, it will start with the innermost term to search for matching patterns. The first match occurs for the following bindings: `a` to `x3`, `b` to `x4`, and `c` to `x5`. With these term bindings, the term is replaced using the right hand side of the rewrite rule, and we obtain the term in the lower-left corner. The rule can find another match on this new term. After similarly rewriting this term, we obtain the term in the lower-right corner.

Example 7. A target term is rewritten with a rewrite rule.

Rewrite Rule	**Target Term**

`(defthm sum-assoc`
`  (equal (+ (+ a b) c)`
`         (+ a (+ b c))))`
	`(+ (+ x1 x2)`
	`   (+ (+ x3 x4) x5))`

After the First Rewrite	**After the Second Rewrite**

`(+ (+ x1 x2)`
`   (+ x3 (+ x4 x5)))`
	`(+ x1`
	`   (+ x2`
	`      (+ x3`
	`         (+ x4 x5))))`

Even though the rewriter dives into every subterm, it keeps track of already processed terms and it does not attempt to rewrite them again. For example, assume that `x4` in the target term from Example 7 is not a variable but it is a very large term that is already rewritten. After the first rewrite, `x4` will have moved within the term. Since the applied rule has a fixed pattern on the left and right hand sides, the rewriter knows to not process `x4` again. On the other hand, if there was an applicable rule, the new subterm `(+ x4 x5)` could be rewritten.

Our overall rewriting system follows this basic rewriting strategy with many more lemmas that work together harmoniously. Fig. 3 shows a flow diagram when the rewriter processes a conjecture for multiplier designs. Assume that we are using the SVL system for simulation, and the user has already created rewrite rules for adder modules to represent them in the s-c form. When the user states a conjecture for the target multiplier design (see Listing 1) and submits it to ACL2, the rewriter dives into the innermost terms to search for applicable rules. The first subterm that it rewrites is the symbolic simulation instance for the target multiplier design.

The SVL system simulates designs by executing all the functional blocks (e.g., Verilog assignments and submodules) and one by one calculating the values for all internal wires and registers. As the rewriter is symbolically simulating an SVL design, derived expressions for internal wires and registers are tested against rewrite rules. If the rewriter encounters an instantiation of an adder module, then it is replaced by the s and c functions using the rules created by the user. If the rewriter encounters some other module or an assignment, then regular ACL2 expressions representing their functionality are created from their logical definitions.

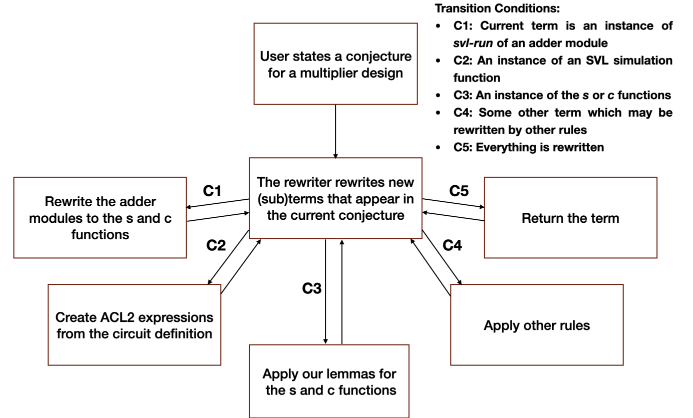


Fig. 3. Steps taken by the rewriter when rewriting a conjecture for a multiplier design

When new instances of the s and c are created after the adder modules are rewritten, our lemmas for these functions are triggered and our simplification algorithm is applied. For example, when the new term is an instance of c and one of its arguments is an instance of s, then Lemma 6 will be applied. If the arguments of the new s and c instances contain some Boolean expressions, then our lemmas for Booth encoding [7] are applied.

As the symbolic simulation of the circuit finishes, we get a term that is completely rewritten with our algorithm. After that, the system rewrites the right hand side (specification) to the s-c form with other rewrite rules in our library, compares the two sides syntactically, and exits. If the final term is `t`, then we can conclude that the multiplier is correct. Otherwise, we can investigate this term and/or send it to a SAT solver so as to generate counterexamples or attempt to finish the proofs.

Note that our lemmas described in Sec. IV, Sec. V, and our previous work [7] do not trigger an expensive rewriting chain upon application. They each have an almost constant time complexity. The slowest component of the rewriting algorithm is lexicographical sorting of the terms in column summations, which are expected to be very small sets as compared to the overall size of the given design. Since our lemmas are applied as the circuit's definition is expanded and we never perform a global search, we observe an almost linear time complexity with respect to the design size as shown in the next section.

VII. Experiments

We verified various multiplier designs using our tool and applicable tools from related work. We ran our experiments on an Intel(R) Core(TM) i7-4790K CPU @ 4.00GHz computer with 32GB system memory. We used three RTL multiplier

TABLE II
PROOF-TIME RESULTS IN SECONDS (ROUNDED) FOR VARIOUS UNTRUNCATED, SIGNED ISOLATED MULTIPLIER DESIGNS

Size	Architecture	RS [4]	AMu [3]	Prev [7]	This work
64x64	sp-cwt-ks	39	42	1	.5
	sp-ar-rc	3	2	1	.5
	sp-dt-bk	5	2	1	.5
	b4-wt-hc	154	28	1	1
	b2-wt-hc	123	77	4	1
	b4-dt-ks	17	28	1	1
	b4-dt-csel	19	5	4	1
	b4-os-bk	15	5	6	1
	b4-wt-csu	21	5	5	2
	b4-bdt-hc	131	6	5	2
	b4-rbat-ks	19	7	5	2
	b4-ar-vcska	17	5	12	2
	b4-4:2-lf	30	5	8	3
	b4-7:3-bcla	44	TO	12	6
	b4-wt-cla	22	14	21	12
128x128	sp-cwt-ks	1001	TO	3	2
	sp-ar-rc	96	10	20	2
	b4-wt-hc	TO	803	13	4
	b4-dt-ks	773	785	8	4
256x256	sp-cwt-ks	TO	TO	16	7
	sp-ar-rc	2416	176	556	11
	b4-wt-hc	TO	TO	62	15
	b4-dt-ks	TO	TO	47	15
512x512	sp-wt-lf	TO	1577	76	44
	sp-dt-bk	TO	1562	64	40
	b4-wt-hc	TO	TO	418	65
	b4-dt-ks	TO	TO	282	71
1024x1024	sp-wt-lf	TO	14005	345	240
	sp-dt-bk	TO	13247	397	220
	b4-wt-hc	TO	TO	MO	288
	b4-dt-ks	TO	TO	MO	300

MO: Out of memory (32GB) **TO**: Time-out (5400 secs./90 mins. for 64x64 and 128x128 multipliers, 16200 secs./270 mins. for the rest)

TABLE III
PROOF-TIME RESULTS IN SECONDS FOR SOME MULTIPLIER DESIGNS IN VARIOUS CONFIGURATIONS

Function & I/O Size	Architecture	AMu [3]	Prev [7]	This work
16x16 = 16	usp-dt-hc	TO	.1	.04
16x16 = 16	ssp-dt-hc	NS	.1	.04
16x16 = 16	ub4-dt-hc	TO	.1	.06
16x16 = 16	sb4-dt-hc	NS	.1	.05
20x40 = 60	ub2-wt-rp	NS	.3	.1
20x40 = 60	sb2-wt-rp	NS	.3	.1
33x17 = 40	ub4-wt-hc	NS	.2	.1
33x17 = 40	sb4-wt-hc	NS	.2	.1
64x64 = 64	ub4-dt-hc	TO	1	.5
64x64 = 64	sb4-dt-hc	NS	1	.4
64x64 = 64 (r. shifted)	ub4-dt-hc	NS	2	1
64x64 = 64 (r. shifted)	sb4-dt-hc	NS	2	1
64x64 = 128	ub4-mdt-ks	45	F	1
64x64 = 128	sb4-mdt-ks	44	F	1
64x64 = 128	ub2-mdt-lf	61	F	1
64x64 = 128	sb2-mdt-lf	59	4	1
2(32x32)+32 = 66	sb4-dt-hc	NS	F	1
2(32x32)+32 = 66	sb4-os-bcla	NS	F	1
2(32x32)+32 = 66	sb4-bdt-csu	NS	F	1
2(32x32)+32 = 66	sb4-ar-csel	NS	F	1
2(32x32)+32 = 66	sb4-4:2-rp	NS	F	2
2(32x32)+32 = 66	sb4-7:3-bk	NS	F	3
64x64+128 = 128	ub4-dt-ks	NS	2	1
64x64+128 = 128	sb4-dt-ks	NS	2	1
64x64+128 = 129	sb4-dt-hc	NS	F	2

TO: Time-out (5400 secs) **NS**: Configuration is not supported by the tool. **F**: Failed proof-attempt. The tool returns a large rewritten term.

generators [32]–[34] to generate isolated multipliers, MAC, and dot-product designs. The benchmarks and our tool are available online (http://mtemel.com/fmcad21).

We verified various architectures with different configurations. For partial product generation algorithms, the designs use either simple partial products (*sp*), Booth encoding radix-4 (*b4*) or radix-2 (*b2*). Summation tree reduction algorithms include counter-based Wallace (*cwt*), array (*ar*), Dadda (*dt*), traditional Wallace (*wt*), overturned-stairs (*os*), balanced delay (*bdt*), redundant binary addition (*rbat*), 4-to-2 compressor (*4:2*), 7-to-3 compressor (*7:3*) trees, and merged multipliers with Dadda tree (*mdt*). For final stage addition, these multipliers implement Kogge-Stone (*ks*), ripple-carry (*rc*), Brent-Kung (*bk*), Han-Carlson (*hc*), Ladner-Fischer (*lf*), carry-select (*csel*), conditional sum (*csu*), variable-length carry-skip (*vcska*), block carry-lookahead (*bcla*) and regular carry-lookahead (*cla*) adders.

As far as we are aware, there are only two other publicly available tools from two different research groups that can verify these complex architectures for isolated multipliers. These are computer-algebra-based tools RevSCA2 [4] (shortened as RS) and AMulet 2.0 [3], [35] (shortened as AMu). The tools from other studies are not publicly available and/or they do not provide competitive results for the designs in question. RevSCA2 does not produce certificates and it is not verified. AMulet provides certificates to check the validity proofs by external tools; we include the certification time in our results (they can be around 3 times faster without certification). The verification tools from our previous and current work are verified using ACL2; thus, no additional check is required.

Table II delivers the proof-time results in seconds for signed and untruncated isolated multipliers. Our previous work scales substantially better than (RS [4]) and (AMu [3]) but the performance is not as strong for Booth encoded designs. Our improved rewriting algorithm is much faster than our previous work and others, and it can verify even very large Booth encoded multipliers in at most 5 minutes.

Table III delivers proof-time results for various architectures and configurations. This includes truncated or right shifted outputs, merged multipliers, multipliers with different operand sizes, two-point dot-product designs with accumulate, and truncated or untruncated MAC modules. The designs in this table are produced with two different generators [32], [33]. AMulet has a hard-coded specification and does not support many of these configurations. Users can determine the design specifications for our previous work, but our older tool cannot prove some merged multipliers, dot-product, and MAC designs. On the other hand, our new method could verify all of them very quickly.

Table IV shows how the proof-time performance of our tool

TABLE IV
OUR TOOL'S PROOF-TIME RESULTS IN SECONDS FOR SIGNED MAC AND DOT-PRODUCT DESIGNS

Size	Dot-product length				
	N=1	N=2	N=4	N=8	N=16
N(32x32)	0.2	0.5	1.0	2.0	4.5
N(32x32)+64	0.2	0.5	0.9	1.9	4.2
N(64x64)	0.9	1.9	3.8	8.2	19
N(64x64)+128	0.9	1.8	3.7	7.7	17
N(128x128)	3.5	7.8	18	35	81
N(128x128)+256	3.5	7.6	15	33	76
N(256x256)	15	32	67	151	356
N(256x256)+512	14	30	64	144	340

All designs use Booth radix-4 encoding, Dadda tree and Ladner-Fischer adder.

TABLE V
OUR TOOL'S PROOF-TIME RESULTS IN SECONDS FOR OUR EXAMPLE MODULE, INTEGRATED MULTIPLIERS, DESCRIBED IN SEC. II-C

Mode	SVL		SVTV	
	Signed	Unsigned	Signed	Unsigned
1-lane MAC	1.0	0.9	2.8	2.9
4-lane MAC (lower half)	1.0	0.9	2.8	2.8
4-lane MAC (upper half)	1.0	1.0	3.0	2.9
4-point dot-product	1.8	1.2	4.4	3.4
8-point dot-product (seq.)	4.9	2.9	14.5	10.1

scales on dot-product designs with different sizes. Even though it is not shown here, allocated system memory scales similarly. Finally, Table V shows the proof-time results for our example module integrated multipliers (see Sec. II-C) for both the SVL and SVTV simulation systems.

In addition to the designs reported here, we have also verified some private industrial designs at Centaur Technology with a similar performance. These designs include multiply-accumulate, dot-product, multiplication of signed and unsigned numbers, truncation, right-shifting, rounding, and saturation. Our program is not designed to handle branches implemented for saturation. Therefore, after our program simplified the saturated designs, we sent the resulting terms to a SAT Solver (*glucose* [36]) with the FGL utility [26], [37], and we have seen that proofs finished successfully in a few seconds.

We have also tried our tool on buggy designs and used a SAT solver (*glucose* [36]) to create counterexamples from simplified terms. We randomly inserted (one or more) bugs into various 64x64-bit, 128x128-bit, and 256x256-bit designs and experimented with 20 different scenarios. Our tool rewrote each multiplier design and returned simplified terms within the same amount of time as given in Table II. It took the SAT solver between 0.1 to 10 seconds to return a counterexample from rewritten terms. Our previous tool could not be used in this workflow because it returns massive terms when proof-attempts fail (see Sec. IV). Using the SAT solver with the original conjecture (in other words, without rewriting with our tool) could give a counterexample in some cases after a few minutes, but it timed out (60 minutes) in the majority of cases. Additionally, our tool can tell exactly which output bits are mismatching the specification. With our new method, we see that our term-rewriting strategy can be very practical and efficient for debugging flawed designs.

VIII. CONCLUSION

We have presented a term-rewriting method that can be used to verify digital circuit designs with embedded integer multipliers. Our tool is efficient, automated, and provably correct. We have shown that we can verify isolated multipliers as large as 1024x1024-bit in less than 5 minutes. Our system allows the user to modify the specification per the target design. Therefore, we can verify multipliers with unusual operand sizes, whose output may be truncated, right-shifted, rounded or saturated. In addition, we can verify other multiplier-centric arithmetic operations such as dot-product and multiply-accumulate. Our library and tutorials are distributed with the ACL2 system, and this content is available online for public use (http://mtemel.com/fmcad21).

This work has been a continuation of our earlier study [7]. With the improvements detailed in this paper, we can verify Booth encoded designs with a much better proof-time efficiency, along with MAC, dot-product, and merged multiplier designs. In addition, we can now generate counterexamples for buggy designs. Moreover, we provide a more comprehensive summary of various multiplier design techniques and discuss why they might be challenging for verification tools.

We use the ACL2 programming language and interactive theorem prover to run and verify our multiplier verification tool, and we use the SVL semantics as our preferred method to simulate Verilog designs. However, our term rewriting algorithm does not require any specific feature from a particular a theorem prover or anything unique to the SVL system. Using a term rewriter and a simulator with hierarchical reasoning can be enough to implement our algorithm on any platform.

We have exploited design hierarchy when implementing our algorithm, whereas the other state-of-the-art tools [3], [4] work on flattened designs. We should note that these tools more or less depend on the original design having clear boundaries for adder modules for their good proof-time performance in the majority of cases. Our choice to use a symbolic simulation system that allows hierarchical reasoning reduces engineering costs and simplifies our program. This way, we do not need to implement any detection algorithm for adder logic. If necessary, using our term-rewriting algorithm for flattened designs might be possible by implementing some preprocessing techniques to reconstruct the design hierarchy. On the other hand, incorporating hierarchical reasoning into computer algebra methods may help improve their performance.

We continue to exercise and improve our method with ever more complex designs such as floating-point multiplication. We have laid a groundwork to permit verification procedures with improved automation and efficiency. The convenience that comes with our fast and automatic verification process can contribute to building reliable hardware systems that include embedded integer multipliers of varying sizes, including but not limited to general-purpose processing units, image processors, digital signal processors, and secure cryptoprocessors.

REFERENCES

[1] M. Ciesielski, C. Yu, W. Brown, D. Liu, and A. Rossi, "Verification of gate-level arithmetic circuits by function extraction," in *2015 52nd ACM/EDAC/IEEE Design Automation Conference (DAC)*, 2015, pp. 1–6. [Online]. Available: https://doi.org/10.1145/2744769.2744925

[2] A. Sayed-Ahmed, D. Große, U. Kühne, M. Soeken, and R. Drechsler, "Formal Verification of Integer Multipliers by Combining Gröbner Basis with Logic Reduction," in *Proceedings of the 2016 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. Research Publishing Services, 2016, pp. 1048–1053.

[3] D. Kaufmann, A. Biere, and M. Kauers, "Verifying Large Multipliers by Combining SAT and Computer Algebra," in *2019 Formal Methods in Computer Aided Design (FMCAD)*, Oct 2019, pp. 28–36.

[4] A. Mahzoon, D. Große, and R. Drechsler, "RevSCA: Using Reverse Engineering to Bring Light into Backward Rewriting for Big and Dirty Multipliers," in *Proceedings of the 56th Annual Design Automation Conference 2019*, ser. DAC '19. New York, NY, USA: ACM, 2019, pp. 185:1–185:6.

[5] M. Ciesielski, T. Su, A. Yasin, and C. Yu, "Understanding Algebraic Rewriting for Arithmetic Circuit Verification: a Bit-Flow Model," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2019.

[6] A. Mahzoon, D. Große, and R. Drechsler, "PolyCleaner: Clean your Polynomials before Backward Rewriting to verify Million-gate Multipliers," *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pp. 1–8, 2018.

[7] M. Temel, A. Slobodova, and W. A. Hunt, "Automated and scalable verification of integer multipliers," in *Computer Aided Verification*. Cham: Springer International Publishing, 2020, pp. 485–507. [Online]. Available: http://doi.org/10.1007/978-3-030-53288-8%5F23

[8] C. R. Baugh and B. A. Wooley, "A Two's Complement Parallel Array Multiplication Algorithm," *IEEE Transactions on Computers*, vol. C-22, pp. 1045–1047, 1973.

[9] A. D. Booth, "A Signed Binary Multiplication Technique," vol. 4, no. 2. Oxford University Press (OUP), 1951, pp. 236–240.

[10] C. S. Wallace, "A Suggestion for a Fast Multiplier," *IEEE Trans. Electronic Computers*, vol. 13, pp. 14–17, 1964.

[11] L. Dadda, "Some Schemes for Parallel Multipliers," 1965.

[12] Brent and Kung, "A Regular Layout for Parallel Adders," *IEEE Transactions on Computers*, vol. C-31, no. 3, pp. 260–264, mar 1982.

[13] R. E. Ladner and M. J. Fischer, "Parallel Prefix Computation," *Journal of the ACM (JACM)*, vol. 27, no. 4, pp. 831–838, oct 1980.

[14] R. E. Bryant and Y.-A. Chen, ""Verification of Arithmetic Functions with Binary Moment Diagrams"," in *DAC 1994*, 1994.

[15] M. A. Basith, T. Ahmad, A. Rossi, and M. Ciesielski, "Algebraic approach to arithmetic design verification," in *Proceedings of the International Conference on Formal Methods in Computer-Aided Design*, ser. FMCAD '11. Austin, Texas: FMCAD Inc, 2011, p. 67–71.

[16] O. Wienand, M. Wedler, D. Stoffel, W. Kunz, and G.-M. Greuel, "An algebraic approach for proving data correctness in arithmetic data paths," in *Computer Aided Verification*, A. Gupta and S. Malik, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 473–486.

[17] A. Mahzoon, D. Große, C. Scholl, and R. Drechsler, "Towards formal verification of optimized and industrial multipliers," in *2020 Design, Automation Test in Europe Conference Exhibition (DATE)*, 2020, pp. 544–549.

[18] T. Su, C. Yu, A. Yasin, and M. Ciesielski, "Formal verification of truncated multipliers using algebraic approach and re-synthesis," in *2017 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, 2017, pp. 415–420.

[19] C. Jacobi, K. Weber, V. Paruthi, and J. Baumgartner, "Automatic formal verification of fused-multiply-add fpus," in *Proceedings of the Conference on Design, Automation and Test in Europe - Volume 2*, ser. DATE '05. USA: IEEE Computer Society, 2005, p. 1298–1303.

[20] D. M. Russinoff, *Formal Verification of Floating-Point Hardware Design: A Mathematical Approach*. Springer, 2019.

[21] W. A. 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*. Springer, 2010, pp. 65–88.

[22] A. Slobodova, J. Davis, S. Swords, and W. A. Hunt, "A Flexible Formal Verification Framework for Industrial Scale Validation," in *Proceedings of the 9th IEEE/ACM International Conference on Formal Methods and Models for Codesign (MEMOCODE)*. Cambridge, UK: IEEE/ACM, July 2011, pp. 89–97.

[23] R. Kaivola and N. Narasimhan, "Formal Verification of the Pentium ® 4 Floating-Point Multiplier," in *2002 Design, Automation and Test in Europe Conference and Exposition (DATE 2002), 4-8 March 2002, Paris, France*, 2002, pp. 20–27.

[24] W. A. Hunt, M. Kaufmann, Moore, J S., and A. Slobodova, "Industrial Hardware and Software Verification with ACL2," *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, vol. 375, no. 2104, p. 20150399, sep 2017.

[25] M. Temel, "ACL2 SVL Documentation," 2019. [Online]. Available: http://www.cs.utexas.edu/users/moore/acl2/manuals/current/manual/?topic=ACL2____SVL

[26] S. Swords, "New rewriter features in FGL," *Electronic Proceedings in Theoretical Computer Science*, vol. 327, p. 32–46, Sep 2020. [Online]. Available: http://dx.doi.org/10.4204/EPTCS.327.3

[27] S. Swords and J. Davis, "Bit-blasting ACL2 theorems," *Electronic Proceedings in Theoretical Computer Science*, vol. 70, p. 84–102, Oct 2011. [Online]. Available: http://dx.doi.org/10.4204/EPTCS.70.7

[28] S. Swords, "Term-level reasoning in support of bit-blasting," *Electronic Proceedings in Theoretical Computer Science*, vol. 249, p. 95–111, May 2017. [Online]. Available: http://dx.doi.org/10.4204/EPTCS.249.7

[29] M. Temel, "RP-Rewriter: An optimized rewriter for large terms in ACL2," vol. 327. Open Publishing Association, Sep 2020, p. 61–74. [Online]. Available: http://dx.doi.org/10.4204/EPTCS.327.5

[30] H. R. Chamarthi, "Rewriting in ACL2," 2021. [Online]. Available: http://www.ccs.neu.edu/home/harshrc/courses/cs2800-fall2010/f10-lec26.pdf

[31] M. Temel, "Automated, efficient, and sound verification of integer multipliers," Ph.D. dissertation, The University of Texas at Austin, 2021.

[32] ——, "Multgen: a fast multiplier generator," 2021. [Online]. Available: https://github.com/temelmertcan/multgen

[33] N. Homma, Y. Watanabe, T. Aoki, and T. Higuchi, "Arithmetic module generator (AMG)," 2006. [Online]. Available: https://www.ecsis.riec.tohoku.ac.jp/topics/amg/

[34] A. Mahzoon, D. Große, and R. Drechsler, "SCA multiplier generator GenMul," 2019. [Online]. Available: http://www.sca-verification.org

[35] D. Kaufmann and A. Biere, "AMulet 2.0 for verifying multiplier circuits," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems , TACAS 2021.*, ser. Lecture Notes in Computer Science, J. F. Groote and K. G. Larsen, Eds., vol. 12652. Springer, 2021, pp. 357–364.

[36] N. Sörensson and N. Een, "Minisat v1.13-a sat solver with conflict-clause minimization," *International Conference on Theory and Applications of Satisfiability Testing*, 01 2005.

[37] S. Goel, A. Slobodová, R. Sumners, and S. Swords, "Balancing automation and control for formal verification of microprocessors," in *Computer Aided Verification - 33rd International Conference, CAV 2021, Virtual Event, July 20-23, 2021, Proceedings, Part I*, ser. Lecture Notes in Computer Science, A. Silva and K. R. M. Leino, Eds., vol. 12759. Springer, 2021, pp. 26–45. [Online]. Available: https://doi.org/10.1007/978-3-030-81685-8_2