# Formal Verification of Booth Radix-8 and Radix-16 Multipliers

Mertcan Temel
*Advanced Architecture Development Group*
*Intel Corporation*
Austin, TX, USA
mert.temel@intel.com

***Abstract*—In processors where low power consumption is essential, higher Booth encodings such as radix-8 and radix-16 may be preferred over the more common radix-4 encoding. With higher radix multipliers included in commercial hardware, formal verification of such designs poses a real challenge. Verifying multipliers is difficult in general; state-of-the-art verification methods like S-C-Rewriting and computer algebra have primarily addressed the multiplier verification problem for lower Booth radices such as radix-4. However, these methods do not scale well for higher radices. This paper explores the cause of this limitation and proposes a list of solutions for automatic, sound, and fast verification of such designs. These solutions include three improvements for the S-C-Rewriting method, some of which may be applicable to computer algebra methods as well. Experiments have shown that higher Booth radix multipliers can now be verified soundly, fully automatically, and in a matter of seconds for common operand sizes such as 64 and 128 bits.**

***Index Terms*—Formal Verification, Multipliers, Hardware Verification, Booth radix-8, Booth radix-16**

## I. Introduction

Integer multipliers are pillar components in processing units. Various algorithms such as Wallace tree and Booth encoding are used to optimize multiplier designs for area, power, and propagation delay. Booth radix-4 encoding is a common choice; however, radix-8 and radix-16 have been used for systems where minimizing the power consumption is essential [1], including commercial product designs at Intel Corporation [2]. As hardware bugs can be very expensive, formal verification becomes necessary to deliver correctness guarantees about these designs.

Computer algebra and S-C-Rewriting are the state-of-the-art methods for automatically and quickly verifying multiplier designs. RevSCA [3] and AMulet [4] are some of the prominent computer algebra-based tools with notable proof-time performance for complex multipliers. AMulet stands out for being open-source and providing a mechanism to check the soundness of its work by producing proof certificates. As an alternative method to computer algebra, the S-C-Rewriting method was introduced by Temel et al. [5]. This method is implemented as an open-source and a fully verified tool (VeSCMul: Verified implementation of S-C-Rewriting for multiplier verification) [6], similarly delivering soundness guarantees about its results. S-C-Rewriting has been shown to scale better than the state-of-the-art alternatives. For example, 1024x1024-bit Booth radix-4 multipliers can be verified within minutes where others take hours or time-out. Performance of these methods for higher Booth radix multipliers was not previously reported; however, our experiments show that they do not scale.

This paper discusses the verification of Booth radix-8 and radix-16 multipliers by examining and improving the S-C-Rewriting method. Sec. II revisits state-of-the-art multiplier verification methods, which commonly use algebraic rewriting for partial products. Sec. III describes the limitations of the algebraic rewriting approach on this particular verification problem. The subsequent sections (Secs. IV, V, VI) propose three improvements to S-C-Rewriting for scalable verification. Sec. VII delivers experimental results. These experiments show the benefit of each improvement, enabling sound, fast, and automatic verification of higher radix multipliers. Sec. VIII concludes the paper.

## II. State-of-the-art Verification Methods

There are two main similarities between the computer algebra methods and S-C-Rewriting. First, both rely on their ability to identify and specially handle the adder components (e.g., full-adders) that make up multiplier designs. AMulet and RevSCA implement their own mechanisms to handle adders in flattened designs. S-C-Rewriting may use design hierarchy as a user-hint, or get bundled with a program that detects adders for full automation. The second similarity between these methods pertains to how partial product logic is processed, that is, both methods use algebraic rewriting/modeling of logical gates when reasoning about Booth encoding. Algebraic rewriting can be summarized using the following lemmas, each of which is in the form of $lhs \to rhs$ where expressions matching $lhs$ are to be replaced by $rhs$.

**Lemma 1.** $\forall x, y \in \{0,1\}\ x \land y \to xy$

**Lemma 2.** $\forall x \in \{0,1\}\ \neg x \to 1 - x$

**Lemma 3.** $\forall x, y \in \{0,1\}\ x \lor y \to x + y - xy$

These methods differ on how they define the specification and how they carry out the proofs. Computer algebra methods define a word-level (in terms of output) specification (see

Def. 1) and use more algebraic modeling with polynomial division, Gröbner basis techniques.

**Definition 1.** Word-level specification for unsigned and untruncated nxn-bit multiplication for computer algebra methods.

$out = (\sum_{i=0}^{n-1} 2^i x_i) \times (\sum_{i=0}^{n-1} 2^i y_i)$

where $x_i$ and $y_i$ are the $i$th bits of operands $x$ and $y$.

S-C-Rewriting defines a bit-level specification (see Def. 2) and use a custom rewriting strategy to simplify designs. This bit-level specification definition allows S-C-Rewriting to easily adapt and verify truncated, right-shifted, and/or saturated multipliers [7].

**Definition 2.** Bit-level multiplication ($out = x * y$) specification for S-C-Rewriting.

$$w_j = \begin{cases} (\sum_{i=0}^{j} x_i y_{j-i}) + c(w_{j-1}), & \text{if j} \geq 0 \\ 0 & \text{otherwise} \end{cases}$$

$$out_j = s(w_j)$$

where $c(x) = \left\lfloor \frac{x}{2} \right\rfloor$, $s(x) = mod_2(x)$, and $x_i y_{j-i}$ is a monomial with the $i$th and $(j-i)$th bits of operands $x$ and $y$.

S-C-Rewriting's simplification strategy is described with a set of rewrite rules (see Lemmas 4-11). Along with Lemmas 1-3, these rewrite rules aim to convert designs' complex multiplier expressions to the same designated form as the bit-level specification.

**Lemma 4.** $\forall x, y, z \in \{0, 1\}\ fulladder(x, y, z) \rightarrow \{carry : c(x + y + z),\ sum : s(x + y + z)\}$

**Lemma 5.** $\forall x, y \in \{0, 1\}\ halfadder(x, y) \rightarrow \{carry : c(x+y),\ sum : s(x + y)\}$

**Lemma 6.** $\forall x, y \in \mathbb{Z}\ s(s(x) + y) \rightarrow s(x + y)$

**Lemma 7.** $\forall x, y \in \mathbb{Z}\ c(s(x) + y) \rightarrow c(x + y) - c(x)$

**Lemma 8.** $\forall x, y \in \mathbb{Z}\ s(x + x + y) \rightarrow s(y)$

**Lemma 9.** $\forall x, y \in \mathbb{Z}\ c(x + x + y) \rightarrow x + c(y)$

**Lemma 10.** $\forall x, y \in \mathbb{Z}\ s((-x) + y) \rightarrow s(x + y)$

**Lemma 11.** $\forall x, y \in \mathbb{Z}\ c((-x) + y) \rightarrow (-x) + c(x + y)$

Computer algebra methods are mathematically complete in theory, and S-C-Rewriting in isolation is not. However, the S-C-Rewriting can be extended with SAT solving by automatically passing its returned expressions to an external SAT solver, which can keep trying to complete the proofs (only useful for fringe cases) or generate counterexamples. This makes for a sound and complete verification flow.

In the next sections, this paper focuses on S-C-Rewriting to examine the verification problem for higher Booth radices and propose improvements. As both S-C-Rewriting and computer algebra methods use algebraic rewriting/modeling of gates for partial product logic, the next sections may help improve the computer algebra methods as well.

## III. Complexity of Algebraic Rewriting

Algebraic rewriting is useful for reasoning about partial products, but it is an expensive operation. Algorithm 1 shows a procedure, as implemented by the S-C-Rewriting method, for performing algebraic rewriting on gates. This section demonstrates why this procedure can be problematic particularly for higher Booth radices.

**Algorithm 1** Flatten partial product logic through algebraic rewriting with Lemmas 1-2

```
1: procedure PP_FLATTEN($exp$)
2:     if $exp = x \wedge y$ then
3:         return $p_*$(PP_FLATTEN($x$), PP_FLATTEN($y$))
4:     else if $exp = \overline{x}$ then
5:         return $p_+(1,\ p_-($PP_FLATTEN($x$)))
6:     else return $exp$
```

The goal of `pp_flatten` given in Algorithm 1 is to convert a Boolean expression to a polynomial. For example, `pp_flatten`('$\overline{x} \wedge (y \wedge \overline{z})$') returns '$-xy + xyz + y - yz$'. In this procedure, $p_*$, $p_+$, $p_-$ perform polynomial multiplication, summation, and negation, respectively, for two-valued (0 or 1) variables. These functions keep all the variables and terms in generated polynomials in lexical order, they cancel or merge terms (e.g., $(x - x + y) \rightarrow y$, $(xy + xy) \rightarrow 2xy$), and roll-down variable exponents to 1 (e.g., $x^2 y \rightarrow xy$) when necessary. `pp_flatten` may be extended to include other common logical operators such as logical OR ($\vee$) using Lemma 3.

Correctness (i.e., soundness of rewriting) of `pp_flatten`, $p_*$, $p_+$, and $p_-$ can be shown through induction. The actual code implementing and running `pp_flatten` is verified using the ACL2 interactive theorem prover. The source code and the verification events are available online[1].

Runtime for `pp_flatten` grows exponentially with input expression's distinct variable count. To show this, let's start by stating and proving Prop. 12.

**Proposition 12.** `pp_flatten` *can return a polynomial with up to* $2^n$ *terms, where n is the number of distinct variables.*

The proof for Prop. 12 is as follows. The generated polynomials have these properties: 1) coefficients may be integers, 2) variables can have an exponent of either 0 or 1 only, and 3) a term may not be repeated with the same or a different coefficient. Such a polynomial will have the following form:

$(c_1 x_1^0 x_2^0 ...\ x_n^0) + (c_2 x_1^1 x_2^0 ...\ x_n^0) + ...\ (c_{2^n} x_1^1 x_2^1 ...\ x_n^1)$

which demonstrates that there can be up to $2^n$ terms in the described polynomial. `pp_flatten` can be shown to reach this upper-bound through Example 1, which concludes the proof for Prop. 12.

**Example 1.** `pp_flatten` returns a polynomial with $2^n$ terms for: $\overline{x_1} \wedge \overline{x_2} \wedge\ ...\ \overline{x_n}$ (can be confirmed through induction). E.g., for $n = 3$, the result will have 8 terms: $1 - x_1 + x_1 x_2 - x_1 x_2 x_3 + x_1 x_3 - x_2 + x_2 x_3 - x_3$

[1]https://github.com/acl2/acl2/tree/master/books/projects/vescmul/

Let $T(e)$, $T_*(e_1, e_2)$, $T_+(e_1, e_2)$, and $T_-(e)$ denote the time to compute pp_flatten($e$), $p_*(e_1, e_2)$, $p_+(e_1, e_2)$, and $p_-(e)$, respectively, and let $N_e$ denote the number of terms in some expression $e$. As variables and monomials are kept in lexical order (with merge-sort), the upper-bound of $T_*$ is in the order of $(N_{e_1} N_{e_2} log_2(N_{e_1} N_{e_2}))$. Similarly, the upper-bounds for $T_+$ and $T_-$ are in the order of $(N_{e_1} + N_{e_2})$ and $(N_e)$, respectively. This makes the '$exp = x \wedge y$' case in pp_flatten the slowest path, so we examine this case for worst-case analysis. Let $t$ denote some time unit, $x'$ and $y'$ be the outputs of pp_flatten for respective arguments, $z$ denote the expression for '$x \wedge y$', and $n_e$ denote the number of distinct variables in some expression $e$.

$T(z) = T(x) + T(y) + T_*(x', y')$

$\rightarrow T(z) \leq T(x) + T(y) + N_{x'} N_{y'} log_2(N_{x'} N_{y'})t$

From Prop. 12, $N_{x'} \leq 2^{n_x}$ and $N_{y'} \leq 2^{n_y}$:

$\rightarrow T(z) \leq T(x) + T(y) + 2^{n_x} 2^{n_y} (n_x + n_y)t$

Since $n_x \leq n_z$ and $n_y \leq n_z$:

$\rightarrow T(z) - T(x) - T(y) \leq 2^{(2n_z)}(2n_z)t$

A few remarks: 1) we need an upper-bound for $T(z)$ and this relation gives an upper-bound for a single recursive call $T(z) - T(x) - T(y)$, 2) this upper-bound is a function of number of distinct variables, and 3) as pp_flatten makes more recursive calls, the number of distinct variables may or may not go down; however, the total number of recursive calls is limited to the number of Boolean operators in the expression. Let that number be denoted as $m_z$. These give:

$T(z) \leq m_z 2^{(2n_z)}(2n_z)t$

Even though expression size matters, the upper-bound for time grows exponentially with distinct variable count.

This high-complexity with distinct variable count does not pose a big problem for lower Booth radices. For example, Booth radix-4 uses 3-bits from multiplier to calculate multiples of multiplicand from the coefficient set {-2, -1, 0, 1, 2}. These multiples can be calculated with simple shifts and negations, and an additional carry-in bit would be added to the summation/compression tree for the two's-complement cases. This gives at most 5 distinct variables (3 selector bits from multiplier, and 2 bits from multiplicand) for each partial product bit.

On the other hand, higher Booth radices incorporate more multiplier selector bits and uses more coefficients such as {-4, -3, ... 3, 4} for radix-8 and {-8, -7, ... 7, 8} for radix-16. Calculating the non-power-of-2 multiples of multiplicand requires vector additions, which can substantially increase the number of distinct variables per partial product bit. For example, as part of a radix-8 encoding, we may see $\overline{b_5} b_4 \overline{b_3} b_2 (3a)_{60}$ inside an expression for partial products. Here, $(3a)_{60}$ represents the Boolean expression for the 60th output bit of the circuitry calculating $3a$, which is a function of the first 60 bits of number $a$. Given the sheer number of distinct variables, rewriting such logical expressions with pp_flatten does not scale. The next three sections propose solutions to address this problem for scalable verification of higher Booth radix multipliers by extending the S-C-Rewriting method.

## IV. Improvement A: Do Not Flatten Addition Logic in Partial Products

The S-C-Rewriting method was described to rewrite all of partial product logic with pp_flatten. The method then rewrites the adder components in summation trees to the $s$ and $c$ functions (see Lemmas 4, 5). After rewriting the adders in terms of $s$ and $c$, a custom rewriting scheme with Lemmas 6-11 is applied to resulting expressions to simplify them to the same form as the specification (see Def. 2).

For higher Booth radices, the extra addition logic in partial product bits causes the distinct variable count for pp_flatten to drastically increase, so rewriting the overall partial product logic as it is done for lower radices does not scale. As a workaround, we ask the question if the addition logic inside partial products can be processed the same way as the other adder components in multiplier designs.

We guide VeSCMul (the tool implementing S-C-Rewriting) to rewrite the adders inside partial products the same way as the other adder components in multipliers, that is, rewrite them in terms of $s$ and $c$. This causes $s$ and $c$ expressions, which are outputs of the adder logic to calculate non-power-of-2 multiples of multiplicand, to appear inside partial product logic. We program the pp_flatten function to treat these output bits ($s$ and $c$ instances) as variables inside the given Boolean expression. For example, $(3a)_{60}$ in $\overline{b_5} b_4 \overline{b_3} b_2 (3a)_{60}$ would now be an $s$ or $c$ instance to be processed as if it is a distinct variable. This substantially reduces the number of observed variables for algebraic rewriting. For radix-8, now the program works with up to 8 variables (4 from multiplier, 4 from multiplicand); and for radix-16, it works with up to 13 variables (5 from multiplier, 8 from multiplicand).

Treating the $s$ and $c$ instances as variables yields terms containing multiplication of actual input variables with an $s$ or $c$ instance in the returned polynomial. For example:

$b_4 b_3 b_2 * s(a_4 + a_2 + c(a_2 a_0 + a_3 + a_1))$

where $b$ is the multiplier operand selecting a multiple of multiplicand $a$. Such expressions/patterns are not recognized by the previously described S-C-Rewriting algorithm, and proof-attempts would fail. Two new rewrite rules are proposed to extend the method to process such expressions:

**Lemma 13.** $\forall x \in \{0, 1\}\ y \in \mathbb{Z}\ x * s(y) \rightarrow s(x * y)$

**Lemma 14.** $\forall x \in \{0, 1\}\ y \in \mathbb{Z}\ x * c(y) \rightarrow c(x * y)$

Lemmas 13-14 distribute the multiplier operand bits into the multiplicand's summation expressions. Rewriting the above example with these lemmas will yield:

$s(b_4 b_3 b_2 a_4 + b_4 b_3 b_2 a_2 + c(b_4 b_3 b_2 a_2 a_0 + b_4 b_3 b_2 a_3 + b_4 b_3 b_2 a_1))$

In addition to these lemmas, the program follows its already established rewriting strategy to now successfully verify some small Booth radix-8 and Booth radix-16 multipliers (see experiments in Sec. VII). However, the scaling factor was still subpar as compared to lower radix encodings, which necessitated the search for further improvements.

## V. Improvement B: A Shortcut Rewrite Rule for S-C-Rewriting

After Improvement A, the program implementing S-C-Rewriting started encountering a new pattern:

$...c(-s(...) + ...)$

Such negated instances of $s$ comes from flattened partial product logic of radix-8 and radix-16 multipliers. For example, a term returned from `pp_flatten` may be $-b_3b_2{*}s(a_4{+}...)$, which will later be rewritten by Lemma 13 to $-s(a_4b_3b_2{+}...)$. Even though these are new patterns, the previously described rewriting method could already handle such cases for a successful design verification; however, the rewriting path it took turned out to be very slow and caused scalability issues. As an example, assume that the system aims to rewrite the term below.

$s(k + c(x + c(-s(x) + y)))$

Lemma 11 is applied to the underlined expression:

$s(k + c(x + \underline{c(-s(x) + y)}))$

and yield:

$s(k + c(x + -s(x) + \underline{c(s(x) + y)}))$

Lemma 7 rewrites the underlined expression above to:

$s(k + \underline{c(-s(x) - c(x)} + x + c(x + y)))$

Apply Lemmas 11 (2 times) and 7 again to get:

$\underline{s(-s(x) - c(x) - c(x)} + k + \underline{c(x + x} + c(x) + c(x + y)))$

Finally, Lemmas 6, 8 (2 times), 9, 10 can rewrite this to:

$s(k + c(c(x) + c(x + y)))$

This final form may not have much meaning out of context, but it can be the intended form to carry out a correctness proof.

In the above example, there are initially 2 nested calls of $c$, and 10 rewriting steps are taken. For each additional nested call of $c$, the number of steps would go up. In order to converge more quickly, Lemma 15 is proposed as a shortcut rule to extend the S-C-Rewriting method. This new rule is given a higher application priority over the other rules.

**Lemma 15.** $\forall x, y \in \mathbb{Z}\ c(-s(x) + y) \to c(x + y) + c(x) - x$

Let's apply this lemma on the example term above:

$s(k + c(x + \underline{c(-s(x) + y)}))$

and quickly arrive at the same expression:

$s(k + c(c(x) + c(x + y)))$

Lemma 15 helped the expression converge to the form in only 1 step. It would not make a difference if there were more nested calls of $c$ in this case. Such instances seem to be common enough for this shortcut rule to make a significant difference in improving the proof-time performance and reduce memory allocation for large multipliers (see Sec. VII).

To prove Lemma 15, let's define an additional lemma: $\forall x \in \mathbb{Z}\ s(x) \to x - 2c(x)$. This lemma is correct because when x is even, $s(x) = 0$ and $c(x) = x/2$; and when x is odd, $s(x) = 1$ and $c(x) = (x - 1)/2$. Now, in Lemma 15, we can replace the $s(x)$ instance with $x - 2c(x)$, and use Lemmas 9 and 11 to show that Lemma 15 is correct.

## VI. Improvement C: Dynamically Learn Pattern Reductions

On the way to the next extension, we make three observations:

*1)* The `pp_flatten` function was implemented with optimal (and verification friendly) data-structures for the ACL2 system, and then the overall program's functions were profiled to evaluate `pp_flatten`'s performance. This profiling showed that with the other proposed improvements in place, 76% of overall verification time was spent in `pp_flatten` for 64x64-bit radix-16 multipliers. This ratio went down to 58% for radix-8, and 20% for radix-4 encodings (see Table II).

*2)* `pp_flatten` is invoked for each partial product bit in a multiplier design. This can amount to hundreds or thousands of outermost calls of `pp_flatten` for a 64x64-bit multiplier. The calculations for partial products may be expressed very differently from design to design; however, within the same design, the logical gate combinations among partial product bits are expected to be shared. This means that it is likely for `pp_flatten` to be called repeatedly for the same gate patterns but with different variables.

*3)* Algebraic rewriting may take a large number of steps but eventually produce a small polynomial due to vanishing monomials, which are terms that get canceled out in summations. After implementing Improvement A, `pp_flatten` was profiled for an instance of a radix-16 partial product bit, which showed that for an input expression with 13 distinct variables (expected maximum), the number of terms in the returned polynomial was only 126 as opposed to the worst case scenario of $2^{13} = 8192$. During this rewriting process, $p_*$ multiplied monomials $1.80 \times 10^4$ times; a total of $1.17 \times 10^3$ distinct monomials were created; and monomials were canceled out $4.95 \times 10^3$ times. Given that `pp_flatten` may be called thousands of times, these computations can exert their toll on a system's resources.

These observations suggest that there might be performance benefits from implementing a mechanism that dynamically learns `pp_flatten`'s results based on input gate patterns. This can help avoid repeated algebraic rewriting for expressions of the same shape but with different variables. The pseudo-code for such a mechanism is given in Algorithm 2.

**Algorithm 2** Reuse `pp_flatten` results for the same gate patterns with different variables

```
1: procedure PP_FLATTEN_WITH_BINDS($exp$)
2:     $\langle exp, binds \rangle \leftarrow$ REPLACE_VARS($exp$)
3:     $poly \leftarrow$ PP_FLATTEN($exp$)            $\triangleright$ a memoized call
4:     $poly \leftarrow$ REBIND_VARS($poly$, $binds$)
5:     return $poly$
```

`pp_flatten_with_binds` from Algorithm 2 operates as follows. The `replace_vars` function parses a given expression and replaces every distinct variable (or *s*/*c* instances that come from Improvement A) with alternative variables. The alternative variables will always be named the same (e.g.,

$var_0$, $var_1$, and so on) in the same order as they replace the original variables. Then, replace_vars will return an expression with the same gate pattern but new variables in place, along with a variable binding list to designate what each variable replaced in the original expression. This is followed by invoking pp_flatten, whose outermost calls are memoized. Memoization prevents repeated computation of a function for the same arguments; the system will instead look up the returned value from a dynamically expanded hash table. The overall procedure is finalized by rebinding the variables in the returned polynomial to their original values with the rebind_vars function. Changing variable names might break lexical order so rebind_vars also reorders terms and variables.

For example, for $exp = \overline{a_3} \vee (a_2 \wedge a_3)$, replace_vars will return $\overline{var_0} \vee (var_1 \wedge var_0)$ along with binding list $\{var_0 \leftarrow a_3, var_1 \leftarrow a_2\}$. Then, pp_flatten will compute the corresponding polynomial: $1 - var_0 + var_0 var_1$. As pp_flatten is memoized, this input-output pair will be remembered. Finally, rebind_vars will replace variables using the binding list and return: $1 + a_2 a_3 - a_3$.

Assume that pp_flatten_with_binds is called again for $exp = \overline{a_7} \vee (a_6 \wedge a_7)$. replace_vars will return the same expression as before $\overline{var_0} \vee (var_1 \wedge var_0)$ but with a different bindings list $\{var_0 \leftarrow a_7, var_1 \leftarrow a_6\}$. Since pp_flatten is memoized and it already computed the polynomial for this intermediary expression, the system will not call pp_flatten but instead it will look up the memoize tables to return the same result: $1 - var_0 + var_0 var_1$. Then, rebind_vars will return: $1 + a_6 a_7 - a_7$.

As per the second observation, making a hit in the memoize tables for pp_flatten in this scheme is expected to be frequent, which is supported by the experiments with over 99% hit rate. Given how pp_flatten was taking a large portion of the computation time (the first observation), this system made a notable difference in improving memory and proof-time performance (see Sec. VII).

Establishing a functional correctness proof for this procedure is more tedious than pp_flatten itself. This is due to the fact that this procedure introduces new variables in intermediary expressions. Proofs include ensuring that rebinding these intermediary variables on the polynomial returned by pp_flatten would yield a polynomial that symbolically evaluates to the same value as the original expression. This proof is formalized in the ACL2 theorem prover via meta-reasoning. The proof events along with the functions span almost 3000 lines of ACL2 code, which is available online[2].

## VII. Experiments

Proposed improvements have been implemented in VeSCMul (the program implementing S-C-Rewriting). The entirety of the verification functions and the overall procedure is end-to-end verified using the ACL2 interactive theorem prover. Therefore, when designs are claimed to be correct, users can trust the results with high confidence. Over 1000 Booth encoded multipliers have been gathered for experiments[3]. These include signed/unsigned Wallace (WT), Dadda (DT), and 4-to-2 compressor trees (4:2) with various fast vector adders, such as Han-Carlson (HC), Brent-Kung (BK).

Table I delivers the experiment results showing the benefit of the proposed improvements. The impacts of Improvements A and B are substantial. Even though relatively less significant, Improvement C also delivers around 4 times better performance for radix-16 multipliers. Addition of Improvement C provides some improvements for radix-4 multipliers as well.

TABLE I
Experiments showing the impact of each improvement on proof-time and memory allocation

| Size | PP | Prev Work | | Impr A | | Impr A, B | | Impr A, B, C | |
|---|---|---|---|---|---|---|---|---|---|
| 8 | r4 | 0.1s | 15MB | 0.1s | 15MB | 0.1s | 15MB | 0.1s | 12MB |
| | r8 | 0.2s | 24MB | 0.1s | 13MB | 0.1s | 15MB | 0.1s | 13MB |
| | r16 | 2.3s | .4GB | 0.3s | 53MB | 0.3s | 55MB | 0.2s | 43MB |
| 16 | r4 | 0.2s | 37MB | 0.2s | 37MB | 0.2s | 37MB | 0.2s | 32MB |
| | r8 | 7.8s | 1.3GB | 0.3s | 56MB | 0.3s | 58MB | 0.2s | 37MB |
| | r16 | Stack ovrflw | | 2.5s | .73GB | 1.6s | .28GB | 0.5s | .11GB |
| 32 | r4 | 0.6s | .13GB | 0.6s | .13GB | 0.6s | .13GB | 0.5s | .11GB |
| | r8 | Stack ovrflw | | 3.6s | .88GB | 1.3s | .25GB | 0.6s | .13GB |
| | r16 | Stack ovrflw | | 104s | 27GB | 7.2s | 1.3GB | 1.7s | .39GB |
| 64 | r4 | 2.2s | .47GB | 2.2s | .47GB | 2.2s | .47GB | 1.8s | .39GB |
| | r8 | Stack ovrflw | | 145s | 37GB | 5s | 1GB | 2.1s | .50GB |
| | r16 | Stack ovrflw | | 74m | 1.3TB | 31s | 6.1GB | 7.5s | 1.7GB |
| 128 | r4 | 7.8s | 1.9GB | 7.8s | 1.9GB | 7.8s | 1.9GB | 6.2s | 1.6GB |
| | r8 | Stack ovrflw | | 81m | 1.3TB | 20.4s | 4.4GB | 8.6s | 2.2GB |
| | r16 | Stack ovrflw | | Time-out | | 128s | 26GB | 33s | 8.3GB |

Multiplier sizes range from 8x8 to 128x128. Each row consists of 72 different benchmarks grouped with respect to size and Booth encoding (radix-4, 8, or 16), amounting to a total of 1080 different designs. Results are averaged per benchmark. System memory is limited to 16GB. Allocated memory is shown; memory-in-use might be lower. Time-out is set to 2 hours.

Table II shows detailed profiling results for Improvement C. Even though some additional time is spent in the auxiliary functions replace_vars and rebind_vars of Algorithm 2, the run-time for pp_flatten is cut down considerably with a lot of memoize hits, which results in reduction in the overall run-time.

Table III shows the performance of various tools and compares them to VeSCMul. Kissat is a modern SAT solver. RevSCA2 [3] supports signed/unsigned multipliers but only for nxn-bit configurations with 2n-bit output. There is another tool called DyPoSub from the same research group, but it has been reported to have unsound and incomplete results [8]. AMulet [4] similarly only supports nxn-bit multipliers. AMulet can produce certificates to check its work. Its newest version (AMulet2) timed out in the majority of cases; the owner of the tool is notified, and only the results of AMulet1 are included. Around 1/3 of the RTL designs could not be converted to a suitable input format for AMulet and RevSCA2 following the recommended way with yosys and abc utilities; and those designs are not considered when evaluating AMulet

[2]https://github.com/acl2/acl2/tree/master/books/projects/vescmul

[3]https://github.com/temelmertcan/multgen

TABLE II

EXPERIMENTS SHOWING THE IMPACT OF IMPROVEMENT C

| Size | PP | Prep | Without Impr C | | | With Impr C | | |
|---|---|---|---|---|---|---|---|---|
| | | | M. hits | pp_fltn | SCR | M. hits | ppf_wb | SCR |
| 32 | r4 | 0.4s | 1.0% | 0.1s | 0.13s | 99.8% | 0.01s | 0.04s |
| | r8 | 0.5s | 0.8% | 0.66s | 0.73s | 99.7% | 0.02s | 0.09s |
| | r16 | 1.1s | 0.4% | 5.38s | 6s | 99.6% | 0.08s | 0.6s |
| 64 | r4 | 1.6s | 0% | 0.43s | 0.59s | 99.7% | 0.04s | 0.2s |
| | r8 | 1.6s | 0% | 2.89s | 3.35s | 99.5% | 0.1s | 0.5s |
| | r16 | 2.6s | 0% | 23.5s | 27.7s | 99.4% | 0.54s | 4.8s |
| 128 | r4 | 5.1s | 0% | 1.8s | 2.73s | 99.8% | 0.2s | 1.1s |
| | r8 | 5.3s | 0% | 12.6s | 15.37s | 99.6% | 0.5s | 3.3s |
| | r16 | 8.2s | 0% | 99.1s | 120.3s | 99.5% | 1.5s | 24.5s |

Multiplier sizes range from 32x32 to 128x128. Each row consists of 72 different benchmarks amounting to a total of 648 different designs. Results are averaged. *Prep* column includes the time spent in the automatic adder detection program, file reads, and other misc operations. Prep time is the same whether or not Improvement C is implemented. *SCR* column gives the averaged time spent by the rewrite rules of S-C-Rewriting. Averaged total verification time equals SCR+Prep times. *pp_fltn* column represents time spent in the `pp_flatten` function. *M. hits* is ratio of memoize hits for outermost `pp_flatten` calls. *ppf_wb* column represents time spent in `pp_flatten_with_binds`. Time spent in `pp_flatten` is omitted for Improvement C as it is close to 0 for all cases.

TABLE III

COMPARING THE PERFORMANCE OF STATE-OF-THE-ART-TOOLS

| Size | PP | Kissat | RevSCA2 | AMulet1 | VeSCMul Before | ***VeSCMul Now*** |
|---|---|---|---|---|---|---|
| 8 | r4 | 6.5s | 0.05s (87%) | 0.1s (91%) | 0.1s | *0.1s* |
| | r8 | 6.2s | 1.7s (45%) | 4.5s (38%) | 0.2s | *0.1s* |
| | r16 | 7.2s | TO | TO | 2.3s | *0.2s* |
| 16 | r4 | TO | 0.2s (87%) | 0.3s (72%) | 0.2s | *0.2s* |
| | r8 | TO | 27s (45%) | 941s (25%) | 7.8s | *0.2s* |
| | r16 | TO | TO | TO | St.O | *0.5s* |
| 32 | r4 | TO | 1.4s (87%) | 1.3s (91%) | 0.6s | *0.5s* |
| | r8 | TO | 241s (44%) | TO | St.O | *0.6s* |
| | r16 | TO | TO | TO | St.O | *1.7s* |
| 64 | r4 | TO | 19s (75%) | 4.9s (88%) | 2.2s | *1.8s* |
| | r8 | TO | 1630s (19%) | TO | St.O | *2.1s* |
| | r16 | TO | TO | TO | St.O | *7.5s* |
| 128 | r4 | TO | 642s (50%) | 274s (91%) | 7.8s | *6.2s* |
| | r8 | TO | TO | TO | St.O | *8.6s* |
| | r16 | TO | TO | TO | St.O | *33s* |

Multiplier sizes range from 8x8 to 128x128. Each row consists of 72 different benchmarks amounting to a total of 1080 different designs. Proof-times in seconds are averaged when successful. Time-out (TO) is set to 1 hour. For cases where tools could not finish the proofs, success ratios are given inside parentheses. Stack overflow (St.O) occurred after around 10 minutes on a system with 16GB memory.

and RevSCA2. As seen from these results, every tool except VeSCMul performed poorly for radix-8 and radix-16 multipliers. Only RevSCA2 could verify some 64x64-bit radix-8 multipliers in around 27 minutes (only less than 20% of the benchmarks were verified, the rest timed-out). VeSCMul is more comprehensive and faster, verifying all the cases in a matter of seconds. VeSCMul only used S-C-Rewriting to verify these designs without needing a SAT solver.

Finally, Table IV shows that VeSCMul can be used for designs with arbitrary operand sizes and other arithmetic operations such as multiply-add, dot-product, truncation, and shifting. Being able to verify such configurations is important as they are prevalent in industrial designs such as in x86 processors. No other comparable verification tool is known to support such configurations.

TABLE IV

PROOF-TIME RESULTS IN SECONDS FOR MULTIPLIER DESIGNS IN ARBITRARY CONFIGURATIONS

| Function Summary | Architecture | Result |
|---|---|---|
| 64x64+128 (Multiply add) | 4:2-**r8**-HC | 2.2s |
| 5(32x64)+64 (5-point dot-product-accumulate) | DT-**r16**-BK | 18.2s |
| (64x64)[63:0] (Multiply, return lower half) | WT-**r8**-LF | 1s |
| (64x64)[95:32] (Multiply, return mid. portion) | WT-**r8**-LF | 2.1s |
| (64x64)[127:64] (Multiply, return upper half) | WT-**r8**-LF | 2.5s |
| 10x1024 (asymmetric multiply) | DT-**r8**-KS | 42.8s |
| 1024x10 (asymmetric multiply) | DT-**r8**-KS | 24.7s |
| 10x1024 (asymmetric multiply) | DT-**r16**-KS | 75.2s |
| 1024x10 (asymmetric multiply) | DT-**r16**-KS | 39s |

In addition, we have used VeSCMul to automatically verify propriety high-radix designs at Intel®.

## VIII. CONCLUSION

This paper has explored the verification problem of Booth radix-8 and radix-16 multipliers, and has offered three improvements to the S-C-Rewriting method. These improvements are implemented in the fully verified VeSCMul tool, which delivers soundness guarantees for its results. Experiments with up to 128x128-bit multipliers showed that target designs can now be verified fully automatically in a matter of seconds. No other automated method is known to scale as well. Future work involves testing the verification tool on more radix-8 and radix-16 benchmarks as they become publicly available.

## REFERENCES

[1] E. Antelo, P. Montuschi, and A. Nannarelli, “Improved 64-bit radix-16 booth multiplier based on partial product array height reduction,” *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 64, no. 2, pp. 409–418, 2017.

[2] R. Riedlinger, R. Arnold, L. Biro, B. Bowhill, J. Crop, K. Duda, E. S. Fetzer, O. Franza, T. Grutkowski, C. Little, C. Morganti, G. Moyer, A. Munch, M. Nagarajan, C. Parks, C. Poirier, B. Repasky, E. Roytman, T. Singh, and M. W. Stefaniw, “A 32 nm, 3.1 billion transistor, 12 wide issue itanium® processor for mission-critical servers,” *IEEE Journal of Solid-State Circuits*, vol. 47, no. 1, pp. 177–193, 2012.

[3] 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, 2019, pp. 185:1–185:6.

[4] D. Kaufmann and A. Biere, “Improving AMulet2 for verifying multiplier circuits using SAT solving and computer algebra,” *Int. J. Softw. Tools Technol. Transf.*, vol. 25, no. 2, pp. 133–144, 2023.

[5] 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

[6] M. Temel, “VeSCMul: Verified implementation of S-C-Rewriting for multiplier verification,” in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems , TACAS 2024 (to appear)*.

[7] M. Temel and W. A. Hunt, “Sound and automated verification of real-world RTL multipliers,” in *Formal Methods in Computer Aided Design, FMCAD 2021, New Haven, CT, USA, October 19-22, 2021*. IEEE, 2021, pp. 53–62. [Online]. Available: https://doi.org/10.34727/2021/isbn.978-3-85448-046-4_13

[8] D. Kaufmann and A. Biere, “Fuzzing and delta debugging and-inverter graph verification tools,” in *Tests and Proofs*, L. Kovács and K. Meinke, Eds. Cham: Springer International Publishing, 2022, pp. 69–88.