Adding Dual Variables to Algebraic Reasoning for Gate-Level Multiplier Verification

Daniela Kaufmann*, Paul Beame†, Armin Biere*,‡, Jakob Nordström§
*Johannes Kepler University Linz, Austria †University of Washington, Seattle, WA, USA
‡Albert-Ludwigs-University Freiburg, Germany §University of Copenhagen, Denmark & Lund University, Sweden

***Abstract*—Algebraic reasoning has proven to be one of the most effective approaches for verifying gate-level integer multipliers, but it struggles with certain components, necessitating the complementary use of SAT solvers. For this reason validation certificates require proofs in two different formats. Approaches to unify the certificates are not scalable, meaning that the validation results can only be trusted up to the correctness of compositional reasoning. We show in this paper that using dual variables in the algebraic encoding, together with a novel tail substitution and carry rewriting method, removes the need for SAT solvers in the verification flow and yields a single, uniform proof certificate.**

I. Introduction

Formal verification of arithmetic circuits helps prevent issues like the infamous Pentium FDIV bug. Although several techniques for circuit verification have been developed, the problem of fully automatic verification of arithmetic circuits is still considered to be hard. In particular, integer gate-level multipliers are challenging to verify due to their structural composition. Just to point out some issues with previous approaches used for multiplier verification, decision diagrams rely on manual decomposition [4], satisfiability checking (SAT) does not scale [3], and automated reasoning using theorem provers relies on hierarchical information [20].

Currently, the most successful technique for flattened gate-level multipliers is based on algebraic reasoning [5], [10], [16], [17], [19], where the circuit is modeled as a set of polynomials, and a polynomial reduction algorithm is used to check whether the circuit specification is implied by these polynomials.

However, certain multipliers pose a challenge for algebraic reasoning. More precisely, when the final-stage (FS) adder is a generate-and-propagate (GP) adder [18], the size of the intermediate reduction results increases drastically, which leads to a non-termination of the reduction. The reason is that these specific adders are composed of sequences of OR-gates with common inputs, which are not shared internally. The encoding of OR-gates by polynomials grows exponentially in the number of inputs [10]. The specification polynomial must to be reduced by these exponential-size polynomials before any cancellation of common inputs happens, causing the blow-up.

To overcome this issue, several preprocessing techniques have been proposed. The approach in [16] aims to identify and extract atomic blocks of the multiplier. A follow-up method [17] applies a dynamic substitution order to limit the size of the intermediate reduction results.

In [12] the problem of the OR-gates is explicitly addressed and GP adders are replaced by simple ripple carry (RC) adders that do not contain sequences of OR-gates. The equivalence of the adders is verified using SAT solving and the rewritten multiplier is verified using computer algebra. This method [12] is, to our knowledge, the only approach that additionally provides formal proof certificates to validate the verification result. Combining computer algebra and SAT solving produces certificates in two different proof formats: PAC [14] for algebraic reasoning and DRUP [8] for SAT solving. Hence a gap is left in the validation arguments. It is possible to simulate DRUP proofs in PAC [13], but this method does not scale and generates a tremendous overhead in the proof certificate.

In this paper we extend the approach of [12] by adding dual variables for positive and negative literals in the polynomial encoding, which results in a compact representation. The theoretical utility of dual variables in keeping polynomial reasoning compact (in particular capable of efficiently representing resolution-based reasoning) has long been known [1] and its precise utility has recently been investigated [7]. On the applied side, dual variables were considered in [19], however the authors allow only one sign of a literal within the same polynomial, whereas we do not restrict the signedness. Furthermore, dual variables allow algebraic simulation of resolution in practice [13], but, as discussed above, that approach does not scale and can barely handle 32-bit multipliers. One reason for the difficulty in making practical use of dual variables is that the key method for polynomial inference in practice, the Gröbner basis algorithm, relies on a reduction method based on a fixed variable order that will immediately eliminate one of each pair of dual variables by re-expressing it using its partner.

Our novel carry rewriting approach also introduces the concept of tail substitution, which enables sharing. Combining these contributions allows us to eliminate the SAT solver invocation in multiplier verification. This not only speeds up verification by up to an order of magnitude, and accordingly scales to larger bit-widths, but also directly produces a compact, uniform and easy-to-check proof certificate in PAC.

II. Preliminaries

We briefly review algebraic circuit verification as per the approach in [12] and describe the algebraic proof system PAC [14]. The mathematical notation follows [6].

D. Kaufmann and A. Biere were supported by the LIT AI Lab funded by the state of Upper Austria. J. Nordström was supported by the Swedish Research Council grant 2016-00782 and the Independent Research Fund Denmark grant 9040-00389B. P. Beame was supported by NSF-SHF grant CCF-1714593.

978-3-9819263-6-1/DATE22/©2022 EDAA

In this work we focus on gate-level unsigned integer multipliers described as and-inverter graphs (AIGs) with input bits $a_0, \ldots, a_{n-1}$, $b_0, \ldots, b_{n-1} \in \{0,1\}$ output bits $s_0, \ldots, s_{2n-1} \in \{0,1\}$, and the internal gates defined by $l_1, \ldots, l_k \in \{0,1\}$. Let the set of all these variables be $X = \{a_0, \ldots, a_{n-1}, b_0, \ldots, b_{n-1}, l_1, \ldots, l_k, s_0, \ldots, s_{2n-1}\}$.

In algebraic reasoning, the circuit is modeled using multivariate polynomials and correctness is established by showing that the *specification* – the desired relation between the outputs and inputs – is implied by the polynomial encoding of the circuit. A multiplier C is correct iff for all inputs a_i, $b_i \in \{0,1\}$, the specification $\mathcal{L} = 0$ holds, where

$$\mathcal{L} = -\sum_{i=0}^{2n-1} 2^i s_i + \left(\sum_{i=0}^{n-1} 2^i a_i\right)\left(\sum_{i=0}^{n-1} 2^i b_i\right).$$

A *term* $\tau = x_1^{d_1} \cdots x_r^{d_r}$ is a power product of variables for $d_1, \ldots, d_r \in \mathbb{N}$. We denote the set of terms by $[X]$. The *degree* of a term is $\deg(\tau) = \sum_{i=1}^{r} d_i$. A *monomial* $c\tau$ is a term τ multiplied by $c \in \mathbb{Z}$, and a *polynomial* p is a finite sum of monomials. We write $q|p$ if polynomial p is a multiple of polynomial q. As we will only consider polynomial equations with right-hand side zero, we write f instead of $f = 0$.

An order $\leq$ is fixed on the set of terms such that $1 \leq \tau$ and $\sigma_1 \leq \sigma_2 \Rightarrow \tau\sigma_1 \leq \tau\sigma_2$ for all terms τ, σ_1, σ_2. Such an order is a *lexicographic term order* if for all terms $\sigma_1 = x_1^{d_1} \cdots x_r^{d_r}$, $\sigma_2 = x_1^{e_1} \cdots x_r^{e_r}$ we have $\sigma_1 < \sigma_2$ iff there exists i with $d_j = e_j$ for all $j < i$, and $d_i < e_i$. The largest term (w.r.t. $\leq$) in a polynomial $p = c\tau + \cdots$ is called the *leading term* $\mathrm{lt}(p) = \tau$ which, together with the *leading coefficient* c, constitutes the *leading monomial* $c\tau$, and the *tail of* p is $\mathrm{tail}(p) = p - c\tau$.

The semantics of each circuit gate implies a polynomial relation as shown in Fig. 1 (covering all cases for AIGs).

Let $G(C) \subseteq \mathbb{Z}[X]$ be the set of *gate polynomials* that contains the corresponding polynomial for each gate. All variables $x \in X$ are in $\{0,1\}$ and we enforce this property by the set of *Boolean value constraints* $B(X) = \{x(1-x) \mid x \in X\} \subseteq \mathbb{Z}[X]$. The polynomials in $G(C) \cup B(X)$ are ordered according to a lexicographic order, the so-called *reverse topological term ordering* (RTTO), such that the output variable of any gate is always greater than its inputs [15].

The question whether $\mathcal{L}$ is implied by $G(C) \cup B(X) \subseteq \mathbb{Z}[X]$ can be answered by solving a so-called ideal membership problem [12]. A set of polynomials $I \subseteq \mathbb{Z}[X]$ is called an *ideal* if for all $p_1, p_2 \in I$ and all $q \in \mathbb{Z}[X]$ it holds that $p_1 + p_2 \in I$ and $qp_1 \in I$. A set $G = \{g_1, \ldots, g_s\} \subseteq \mathbb{Z}[X]$ is called a *basis* of I if $I = \{g_1h_1 + \cdots + g_sh_s \mid h_1, \ldots, h_s \in \mathbb{Z}[X]\}$. We say I is generated by G and write $I = \langle G \rangle$.

If we let $J(C) = \langle G(C) \cup B(X) \rangle \subseteq \mathbb{Z}[X]$ be the ideal generated by $G(C) \cup B(X)$, we see that the circuit C fulfils its specification if and only if $\mathcal{L} \in J(C)$ [12].

We can determine membership in $J(C)$ by finding a *D-Gröbner basis* [2], of $J(C)$ then applying multivariate polynomial reduction and checking whether the unique final result is zero. The set of polynomials $G(C) \cup B(X)$ is automatically a D-Gröbner basis with respect to RTTO for $J(C) \subseteq \mathbb{Z}[X]$ [12], and hence the correctness of the circuit can be verified by reducing $\mathcal{L}$ by these polynomials. Reductions by $B(X)$ are usually handled implicitly by immediately reducing exponents greater than one in order to speed up the computation.

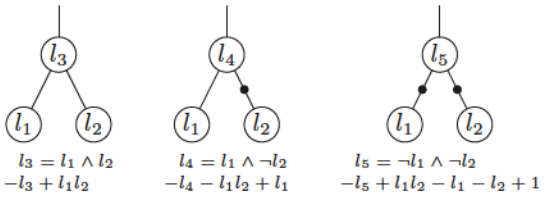

Figure 1: All polynomial encodings covered by AIG nodes

Proof certificates, such as those in PAC [14], allow monitoring the verification process. These proofs can be generated as a by-product of the reasoning technique. A PAC proof consists of three components: (i) the constraint set of polynomials S, (ii) the core proof, i.e., a sequence of proof steps P that model the properties of an ideal, and (iii) the target polynomial f.

The core of PAC uses two proof rules: ADD and MULT. These rules model the addition and multiplication properties of an ideal, e.g., in the ADD rule three polynomials p, q, r are provided such that $p + q = r$, and p and q are either contained in S or are derived in earlier proof steps. Thus we have $p, q \in \langle S \rangle$, hence $r \in \langle S \rangle$. PAC proofs over $\mathbb{Z}[X]$ can be checked by the proof checkers PACHECK and PASTÈQUE [14], where the latter is verified in Isabelle/HOL but slower.

III. Dual Variables

The number of monomials in a polynomial has a huge influence on the performance of polynomial operations such as addition or multiplication. The encoding of [12] uses only the expanded form of polynomials and the negation of an AIG node is encoded using the polynomial representation of an inverter $1 - l_i$. Figure 1 shows the gate polynomials for the AIG nodes l_3, l_4 and l_5 with inputs l_1 and l_2. It can be seen that the size of the gate constraint varies significantly, depending on the signs of the children l_1 and l_2.

The purpose of dual variables is to provide a shorthand notation for inverters. That is for each gate variable l_i we introduce a further variable f_i that encodes the negation of l_i.

Whenever variables $v, w \in \{0,1\}$ fulfill the relation $w = 1 - v$, we call w the *dual variable* of v and write $\mathrm{dual}(v) = w$. We further call $-w - v + 1$ the *dual constraint* for v.

We integrate dual variables in the polynomial encoding of the circuit as follows. For an internal gate variable l_i with $1 \leq i \leq k$ we introduce a dual variable $\mathrm{dual}(l_i) = f_i$. The dual variables are used in the polynomial representation of the AIG nodes to encode negation, e.g., we encode the gate $l_5 = \neg l_1 \wedge \neg l_2$ in Fig. 1 by the polynomial $-l_5 + f_1 f_2$.

Introducing separate variables for the negation of gate variables allows us to represent polynomials with fewer monomials as can be seen in the following example.

Example 1. Let $o = l_0 \vee l_1 \vee \ldots \vee l_{n-1}$, thus o is the output of an n-ary OR gate. Since $o = l_0 \vee l_1 \vee \ldots \vee l_{n-1} \Leftrightarrow \neg o = \neg l_0 \wedge \neg l_1 \wedge \ldots \wedge \neg l_{n-1}$, the polynomial representation

after flattening is $1 - o - (1 - l_0)(1 - l_1) \cdots (1 - l_{n-1})$ and contains 2^n monomials. Introducing dual variables $f_i = 1 - l_i$ for $0 \leq i < n$ allows us to represent the OR-gate using a polynomial with only three monomials $-o - f_0 f_1 \cdots f_{n-1} + 1$.

We show in the following that adding dual variables to X and adding the corresponding dual constraints to the polynomial encoding of the circuit maintains the D-Gröbner basis.

Definition 1. Assume $D(C) = \{-f_i - l_i + 1 \mid l_i \text{ is an internal AIG node of } C,\ f_i = \text{dual}(l_i)\} \subseteq \mathbb{Z}[X]$ and let $G_D(C) \subseteq \mathbb{Z}[X]$ be the set of gate constraints that are generated using dual variables. Further let $J_D(C) = \langle G_D(C) \cup B(X) \cup D(C) \rangle \subseteq \mathbb{Z}[X]$. The polynomials in $G_D(C) \cup B(X) \cup D(C) \subseteq \mathbb{Z}[X]$ are ordered according to a RTTO, with $f_i > l_i$ for each gate variable l_i, i.e., dual variables representing inverter gates.

Proposition 2. Let $J_D(C) \subseteq \mathbb{Z}[X]$ be as in Def. 1. Then $G_D(C) \cup B(X) \cup D(C)$ is a D-Gröbner basis for $J_D(C)$.

Proof. Since $f_i > l_i$ for every gate, the leading term of all dual constraints is f_i. Thus all leading terms of $G_D(C) \cup B(X) \cup D(C)$ are unique and consist only of a single variable, and all leading coefficients are -1. The statement now follows by the same arguments as in the proof of Thm. 5 in [12]. □

During verification we always reduce the specification $\mathcal{L}$ by the dual constraint of a gate variable l_j before reducing by its gate constraint. This has the effect that all occurrences of f_j in the specification will be flipped to l_j before reducing l_j.

However, we need to address the duality of gate variables l_i and f_i in the tail of polynomials. A variable may occur as a literal l_i and a dual literal f_i within the same polynomial. The reduction algorithm per se is not aware of the connection of l_i and f_i. It will treat both variables as separate instances. Thus it may happen that a term in the resulting product contains a variable l_i and its dual variable f_i.

Proposition 3. For all Boolean variables l_i and their dual representation $\text{dual}(l_i) = f_i$ with $f_i = 1 - l_i$ we have $l_i f_i = 0$.

Proof. Since $l_i \in \{0, 1\}$, we have $l_i^2 = l_i$ and thus $-f_i - l_i + 1 = -l_i \cdot (-f_i - l_i + 1) = f_i l_i$. □

Proposition 3 encodes that a Boolean variable can never be zero and one at the same time. We include this property in the verification process. Whenever two terms are multiplied and the resulting term contains a variable and its dual counterpart, the resulting term is set to zero.

Secondly, intermediate polynomials may contain monomials with variable l_i as well as monomials with its dual f_i. Our goal is to merge some of these monomials, using $l_i + f_i = 1$, an immediate consequence of dual constraints. These monomials are characterized by having an identical coefficient and the same term except for the duality of a single variable.

Definition 2. Let m_1, m_2 be two monomials with $m_1 > m_2$. We call m_1 and m_2 *dual mergeable* iff $m_1 = c f_i \tau$ and $m_2 = c l_i \tau$ for c a constant, τ a term, and some index i. We call the monomial $\text{dmerge}(m_1, m_2) = c\tau$ their *dual merge*.

Algorithm 1: Merging monomials(p)

```
Input  : Polynomial p
Output : Simplified polynomial r
1  q ← sort-degree-lex(p); r ← 0;
2  while q ≠ 0 do
3      q_l ← lm(q); t ← tail(q); simplify ← ⊥;
4      while t ≠ 0 and deg(q_l) = deg(lt(t)) and ¬simplify do
5          q_t ← lt(t);
6          if q_l and q_t are dual mergeable then
7              q ← q − q_l − q_t + dmerge(q_l, q_t);
8              simplify ← ⊤;
9          else
10             t ← t − q_t;
11     if ¬simplify then
12         r ← r + q_l;
13         q ← q − q_l;
14 return sort-lex(r);
```

For example $2f_1l_2l_3$ and $2l_1l_2l_3$ are dual mergeable and we have $\text{dmerge}(2f_1l_2l_3, 2l_1l_2l_3) = 2l_2l_3$.

One possible merging algorithm is depicted in Alg. 1. It receives a polynomial $p \in \mathbb{Z}[X]$ as input and returns a polynomial r, whose dual mergeable monomials have been simplified by following a greedy strategy. In the first step (line 1) we re-sort the given polynomial p according to a degree lexicographic ordering.

Definition 3. A *degree lexicographic ordering* for terms σ_1 and σ_2 is defined as follows. If $\deg(\sigma_1) < \deg(\sigma_2)$ then $\sigma_1 < \sigma_2$ and if $\deg(\sigma_1) = \deg(\sigma_2)$ then $\sigma_1 < \sigma_2$ iff σ_1 is smaller than σ_2 in term lexicographic order.

Then (line 2), the algorithm repeatedly searches for potential merges of the leading monomial q_l by checking each tail monomial q_t in degree lexicographic order in turn to see if q_l and q_t are dual mergeable. If such a q_t is found (line 6), then q_l and q_t are merged and the dual merge is inserted in the appropriate place in the sorted order (line 7). The polynomial q will have a new leading monomial since q_l has been removed. Even before all tail monomials have been exhausted, the search can terminate early once $\deg(q_t) < \deg(q_l)$ since no such monomial can be merged with q_l.

If on the other hand we did not find a dual mergeable monomial for q_l, we know that q_l cannot be simplified, because rewriting dual mergeable monomials only adds monomials with a degree smaller than $\deg(q_l)$. Thus it is not possible that we generate a monomial in a later iteration that would be a dual mergeable counterpart for q_l. Hence q_l will be in the simplified polynomial r and can be excluded from our search space q.

Example 4. We apply Alg. 1 on $p = l_1f_2f_3 + l_1f_2l_3 + l_1l_2f_3 + f_1f_2 + l_2 \in \mathbb{Z}[l_1, l_2, l_3, f_1, f_2, f_3]$. We write q_i to denote the polynomial q after iteration i and indicate the dual merges.

$q_0 = l_1f_2f_3 + l_1f_2l_3 + l_1l_2f_3 + f_1f_2 + l_2$	$r = 0$
$q_1 = l_1l_2f_3 + f_1f_2 + \boxed{l_1f_2} + l_2$	$r = 0$
$q_2 = f_1f_2 + l_1f_2 + l_2$	$r = l_1l_2f_3$
$q_3 = \boxed{f_2} + l_2$	$r = l_1l_2f_3$
$q_4 = \boxed{1}$	$r = l_1l_2f_3$
$q_5 = 0$	$r = l_1l_2f_3 + 1$

IV. TAIL SUBSTITUTION

In GP adders all carries are computed in parallel to avoid a delay until the correct carry propagates. Although the parallel structures rely on the same input signals of the adder, the computation of the carries rarely shares internal nodes. This has the effect that cancellations in the intermediate reduction results are delayed until the reduction has reached the inputs of the GP adder. Even if two internal gates are equivalent we do not detect this commonality in the reduction until we have reached the adder inputs.

Example 5. Consider $p = f - g$ and $p_1, \dots p_6$ in $\mathbb{Z}[X]$:

$$p_1 := -f + h_1 h_2, \quad p_2 := -g + h_3 h_4 g_0 g_5,$$
$$p_3 := -h_1 + g_0 g_1 g_2, \quad p_4 := -h_3 + g_1 g_2,$$
$$p_5 := -h_2 + g_3 g_4 g_5, \quad p_6 := -h_4 + g_3 g_4$$

We have to reduce p by polynomials $p_1, \dots, p_6$ to obtain $p = 0$.

We present now a tail substitution approach, which allows us to share nodes on a higher topological level. In our approach we identify whether the tail of a polynomial is a subterm in the tail of a topologically larger polynomial. If so, we replace the tail of the smaller polynomial by the leading monomial of the gate constraint in the larger polynomial.

Example 6. Let $p_1, \dots p_6$ be as in Ex. 5. We see that $\mathrm{tail}(p_4) \mid \mathrm{tail}(p_3)$ and $\mathrm{tail}(p_6) \mid \mathrm{tail}(p_5)$ and thus are able to derive:
$p_3 := -h_1 + h_3 g_0, \quad p_5 := -h_2 + h_4 g_5$
In a second iteration we substitute the tails of p_3, p_5 in p_2:
$p_1 := -f + h_1 h_2, \quad p_2 := -g + h_1 h_2$
Hence we have to reduce p only by p_1 and p_2 to derive $p = 0$.

In the following theorem we show that tail substitution in the ideal generators does not affect the ideal.

Theorem 7. Let $P = \{p_1, p_2, \dots, p_m\} \subseteq \mathbb{Z}[X]$ and $I = \langle P \rangle$. Further let $p_i = \mathrm{lm}(p_i) + \mathrm{tail}(p_i)$, $p_j = \mathrm{lm}(p_j) + q\,\mathrm{tail}(p_i) \in P$ for some $q \in \mathbb{Z}[X]$. Assume that $h = \mathrm{lm}(p_j) - q\,\mathrm{lm}(p_i)$ and $J = \langle P \setminus \{p_j\} \cup \{h\} \rangle$. Then $I = J$.

Proof. The set of ideal generators for I and J is equal up to p_j and h. Per definition $p_j \in I$ and $h \in J$. To show that both ideals are equal we have to show that $h \in I$ and $p_j \in J$. First we show that $h \in I$: We know $p_i, p_j \in I$. Thus by the definition of an ideal also $(-q)p_i \in I$ and hence $(-q)p_i + p_j = \mathrm{lm}(p_j) - q\,\mathrm{lm}(p_i) = h \in I$.

Second, we show that $p_j \in J$: Per definition $p_i, h \in J$ and thus $qp_i + h = \mathrm{lm}(p_j) + q\,\mathrm{tail}(p_i) = p_j \in J$. □

Theorem 7 shows that whenever the tail of a polynomial p_j is a multiple of the tail of a polynomial p_i we are allowed to substitute the tail of a polynomial by its leading monomial, without affecting the generated ideal. Furthermore, it gives us a recipe for deriving the rewritten polynomial h.

Theorem 8. Let $P = \{p_1, p_2, \dots, p_m\} \subseteq \mathbb{Z}[X]$ be a D-Gröbner basis of $I = \langle P \rangle$. Let $p_i, p_j \in P$ with $\mathrm{lt}(p_j) > \mathrm{lt}(p_i)$ and $\mathrm{tail}(p_i) \mid \mathrm{tail}(p_j)$. Assume that $h = \mathrm{lm}(p_j) - q\,\mathrm{lm}(p_i)$. Then $P \setminus \{p_j\} \cup \{h\}$ is a D-Gröbner basis of I.

Algorithm 2: Carry-Rewriting in TELUMA

Input : Circuit C in AIG format with marked FSA gates
Output : Carry-rewritten Gröbner basis of C

1 $F \leftarrow$ Mark-final-stage-adder(C);
2 $G \leftarrow$ Dual-Polynomial-Encoding(F);
3 $H \leftarrow$ Polynomial-Encoding($C \setminus F$);
4 $G \leftarrow$ Eliminate-Pure-Positive-Variables(G);
5 $G \leftarrow$ Tail-Substitution(G);
6 $G \leftarrow$ Carry-Unfolding(G);
7 **return** $G \cup H$

Proof. Since $\mathrm{lt}(p_j) > \mathrm{lt}(p_i)$ we have $\mathrm{lm}(h) = \mathrm{lm}(p_j)$. Thus by definition $P \setminus \{p_j\} \cup \{h\}$ is a D-Gröbner basis of I [2]. □

We apply tail substitution steps in $G_D(C) \cup B(X) \cup D(C)$ according to the assumptions of Thm. 8 to maintain its D-Gröbner basis property. Due to the usage of dual variables the tails of p_i and p_j in $G_D(C) \cup B(X) \cup D(C)$ only consist of a single monomial with coefficient 1 and all exponents 1. Thus to check whether $\mathrm{tail}(p_i) \mid \mathrm{tail}(p_j)$, we simply validate whether all variables of $\mathrm{tail}(p_i)$ occur in $\mathrm{tail}(p_j)$.

V. CARRY REWRITING

In this section we present how we use dual variables and tail substitution to rewrite and simplify the encoding of complex GP adders. In a typical GP adder with inputs $c_{in}, x_0, \dots, x_m, y_0, \dots, y_m$ and outputs $s'_0, \dots, s'_m, c_{m+1}$, the outputs s'_i are calculated as $s'_i = p_i \oplus c_i$ with $p_i = x_i \oplus y_i$. The carries c_i are recursively generated as $c_{i+1} = g_i \vee (c_i \wedge p_i)$ where $g_i = x_i \wedge y_i$. The precise computation of the carries c_i (recursively, unrolled or mixed) depends on the circuit architecture. For example, in a pure carry look-ahead adder, the calculation of the carries is completely unrolled:

$$c_0 = c_{in}$$
$$c_1 = g_0 \vee (c_0 \wedge p_0)$$
$$c_2 = g_1 \vee (g_0 \wedge p_1) \vee (c_0 \wedge p_0 \wedge p_1)$$

Our goal is to rewrite the carries and restore the recursion of c_i; i.e., express the carries c_i in terms of the previous carry c_{i-1} and the generate and propagate bits g_{i-1}, p_{i-1}. Therefore, the carry look-ahead unit is rewritten into a ripple-carry unit, which can easily be verified using computer algebra.

We use our tool AMULET 2.0 [11] as a basis for our implementation. AMULET 2.0 is able to detect complex final stage (FS) adders and identifies their components. In the approach of [11], complex FS adders are replaced by the AIG encoding of simple RC adders. We integrate our novel carry rewriting algorithm that uses dual variables and tail substitutions into AMULET 2.0 and yield the tool TELUMA. The outline of carry rewriting can be seen in Alg. 2.

In a first step (line 1), we use the functionality of AMULET 2.0 to identify and mark the components of the FS adders. Secondly, we generate the gate constraints for the AIG nodes (line 2). For all nodes that are marked as belonging to the FS adder, we apply the encoding including dual variables. This has the effect that the tail of all gate constraints of the FS adder consists only of a single monomial. All other nodes are encoded using positive variables only (line 3).

The preprocessing and rewriting steps (line 4–6) are only executed for nodes of the FS adder. After each polynomial operation we apply Alg. 1. In the first preprocessing step (line 4) all variables that occur only positively are substituted. Since the tails of all polynomials consist of single monomials, and we replace only positive variables by their tail, all tails of rewritten polynomials will remain single monomials.

We apply tail substitution as proposed in Sect. IV. In the data structure of AMULET 2.0 all terms are internally shared using reference counters. If we encounter a tail term that occurs more than once, we check the parent nodes of the first variable to identify nodes that use the same tail.

We notice that after applying tail substitution the tails of the adder carries contain only dual variables.

Example 9. The following polynomials are an excerpt of a carry-lookahead adder after tail substitution, with x_i, y_i being the i-th inputs of the adder, c_{i+1}, c_i denoting carries and p_i being the polynomial encoding of $x_i \oplus y_i$:

$$-c_{i+1} + f_4 f_5 f_6 f_7, \quad -l_7 + x_i y_i, \quad -l_6 + p_i l_3,$$
$$-l_5 + p_i l_2, \quad -l_4 + p_i l_1, \quad -c_i + f_1 f_2 f_3$$

Proposition 10. Let $-l_i + \sigma\tau_i$ for $1 \leq i \leq k$ be a given set of polynomials, with $l_i \in X$ and $\sigma, \tau_i \in [X]$. Assume $\forall_{i=0}^{k} f_i = \text{dual}(l_i)$. Then $\prod_{i=0}^{k} f_i = 1 - \sigma(1 - \prod_{i=0}^{k}(1-\tau_i))$.

Proof. We have $\prod_{i=0}^{k} f_i = \prod_{i=0}^{k}(1-l_i) = \prod_{i=0}^{k}(1-\sigma\tau_i) = \prod_{i=0}^{k}(1-\sigma+\sigma-\sigma\tau_i) = \prod_{i=0}^{k}((1-\sigma)+\sigma(1-\tau_i))$.

The term σ consists only of Boolean variables, thus $\sigma^2 = \sigma, (1-\sigma)^2 = 1-\sigma$, and $(1-\sigma)\sigma = 0$. Hence the product calculates to $\prod_{i=0}^{k}((1-\sigma)+\sigma(1-\tau_i)) = (1-\sigma)+\sigma\prod_{i=0}^{k}(1-\tau_i) = \prod_{i=0}^{k} f_i = 1-\sigma(1-\prod_{i=0}^{k}(1-\tau_i))$. □

We use Prop. 10 to unfold all carries (line 6), starting with the most significant carry. At a first glance the right side of the simplification proposed in Prop. 10 seems to be more complex than $\prod_{i=0}^{k} f_i$. However, in practice the term τ_i is almost always a single variable. Thus, $(1-\tau_i)$ can be expressed using $\text{dual}(\tau_i)$. Applying tail substitution, we are typically able to replace $\prod_{i=0}^{k}\text{dual}(\tau_i)$ by a single gate variable—e.g., in Ex. 9 we have that $f_1 f_2 f_3 = c_i$. We derive $-c_{i+1} + f_7 p_i c_i - f_7 p_1 + f_7$, with $g_i = l_7$ which is the desired expression.

If not all τ_i are single variables, e.g., in Brent-Kunge adders, we repeat Prop. 10 and tail substitution on those polynomials $1-\tau_i$ that share a variable until we have unfolded all τ_i.

It may happen in the first iteration from the most significant carry to the least significant carry that the tail substitution is not successful, as the smaller carries are not yet rewritten and we cannot detect shared equalities. We keep those carries for which rewriting fails on a stack. In subsequent iterations we apply carry rewriting from the least significant carry on the stack to the most significant carry until completion.

After rewriting all carries of the FS adder, our proposed preprocessing technique is finished and returns the rewritten polynomial encoding of the FS adder together with the polynomial encoding of the remaining parts of the multiplier. The specification $\mathcal{L}$ is reduced by these polynomials using the verification method of AMULET 2.0 [11].

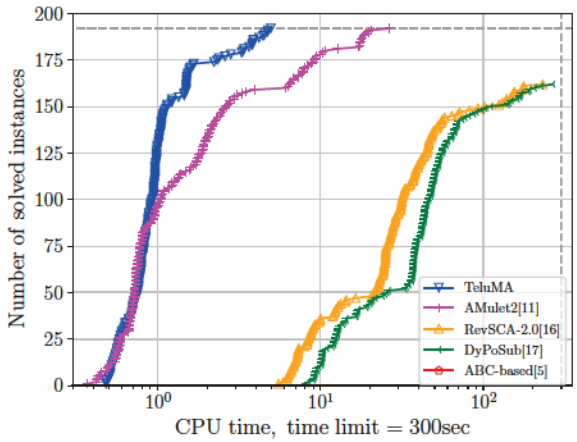

Figure 2: Verification time of 192 unsigned multipliers

VI. Proof Generation

AMULET 2.0 generates certificates in PAC. In the following we describe how we generate proof steps for our new carry rewriting technique. First, we include dual variables in PAC by adding the set of dual constraints $D(C)$ to the constraint set of the proof. That is, the constraint set of polynomials is $S = G_D(C) \cup D(C)$. $B(X)$ is treated implicitly in PAC.

In the core proof we have to include steps that model Prop. 3. That is, whenever polynomials are multiplied we generate a multiplication step where the resulting polynomial is not reduced and may contain monomials of the form $cf_i l_i \tau$ for $c \in \mathbb{Z}$, $f_i, l_i \in X$, and $\tau \in [X]$. We generate a multiplication step that calculates $(cl_i\tau)(-f_i - l_i + 1) = -cf_i l_i \tau$. Using an addition step yields the cancellation of $-cf_i l_i \tau$. We proceed in a similar fashion for the simplifications executed by Alg. 1. We multiply the dual constraint of the variable v by q_l and add the resulting polynomial to p to yield the desired reduction.

Tail substitution as described in Thm. 7 can be monitored using a multiplication step for calculating $-qp_i$, which is then added to p_j to derive the desired polynomial h.

To unfold carries as in Prop. 10 we have to monitor the rewriting of the product of dual variables. That is we first flip the dual variables using dual constraints. Second, we substitute leading terms of gate constraints l_i by their tails. Since PAC proofs consider only polynomials in expanded form, the product is expanded. The reductions of exponents in σ is implicit during product expansion. The tail substitution of the (unfolded) τ_i is monitored as described above.

VII. Evaluation

Our experiments on an Intel Xeon E5-2620 v4 CPU running at 2.10 GHz (with turbo-mode disabled) use a memory limit of 128 GB. The time is listed in seconds (wall-clock time). We evaluate our approach on the aoki-benchmark set [9], which contains 192 64-bit unsigned multipliers. The time limit is set to 300 seconds. Full experimental results including source code are available at `http://fmv.jku.at/teluma`.

Table I: Proof Generation and Checking

architecture	n	[12] DRUP gen (s)	[12] DRUP check (s)	[12] DRUP #rules	[12] PAC gen (s)	[12] PAC check (s)	[12] PAC #rules	[12] total (s)	[13] PAC gen (s)	[13] PAC check (s)	[13] PAC total (s)	[13] PAC #rules	Our approach PAC gen (s)	Our approach PAC check (s)	Our approach PAC total (s)	Our approach PAC #rules
sp-ar-cl	32	0	0	14 927	0	1	33 834	1	133	31	164	1 597 897	0	0	0	60 336
sp-bd-ks	32	0	0	17 528	0	1	34 958	1	20	8	28	817 956	0	0	0	54 116
sp-dt-lf	32	0	0	3 138	0	1	33 451	1	2	3	5	321 720	0	0	0	47 835
bp-ct-bk	32	0	0	2 276	0	1	27 312	1	1	2	3	217 128	0	0	0	36 356
bp-wt-cl	32	1	1	46 502	0	1	30 561	2	3 133	242	3 375	5 536 176	1	1	2	114 665
sp-ar-cl	64	2	1	65 317	2	3	139 338	8	TO	-	-	-	1	3	4	289 632
sp-bd-ks	64	1	0	44 921	2	3	142 138	6	56	18	74	1 440 943	1	2	3	214 378
sp-dt-lf	64	0	0	28 772	2	3	138 539	6	10	10	19	816 572	1	1	2	192 805
bp-ct-bk	64	0	0	19 891	2	3	105 579	5	8	7	15	459 262	1	1	2	136 703
bp-wt-cl	64	8	6	42 199	2	3	118 573	19	TO	-	-	-	7	17	24	774 044

PPG: simple (sp), Booth (bp) PPA: array (ar), Balanced delay tree (bd), Dadda tree (dt), compressor tree (ct), Wallace tree (wt) TO = 3600 sec
FSA: carry look-ahead (cl), Kogge-Stone (ks), Ladner-Fischer (lf), Brent-Kung (bk) Benchmarks are generated by the Arithmetic Model Generator [9].

First we evaluate the time our new tool TELUMA (cf. Sect. V) needs to verify the benchmarks and compare to recently developed tools AMULET 2.0 [11], RevSCA-2.0 [16], and DYPOSUB [17] as well as to the ABC-based work of [5]. Results in Fig. 2 show that AMULET 2.0 and TELUMA are both able to verify the complete benchmark set. The ABC-based approach produces a segmentation fault for all benchmarks. TELUMA provides a significant speedup for those benchmarks where AMULET 2.0 uses more than one second. These are all multipliers where the FS adder is a GP adder.

In our second experiment, we evaluate the proof generation of TELUMA on the same instances as used in [13], with results in Tbl. I. In the first block we show the proof generation ("gen") and checking time ("chk") in seconds of DRUP and PAC proofs using the toolflow of AMULET 2.0 [11] and provide the number of generated proof rules in DRUP as well as PAC. The second block shows the approach of [13] where the DRUP and PAC proofs of the first block are merged to yield a single PAC proof. The third block shows the PAC proofs directly generated by TELUMA. It can be seen that the unified PAC proofs of [13] are between 3 and 48 times larger than the proofs generated by TELUMA, which is also noticeable in the time needed for generating and checking the proofs e.g., the proof generation time for 32-bit bp-wt-cl multipliers takes around an hour, whereas our approach is able to generate a single PAC certificate in less than a second.

VIII. CONCLUSION

In this paper we presented how to include dual variables that represent negative literals in the polynomial encoding of gate-level circuits. We introduced a novel tail substitution scheme and discussed a carry rewriting technique that simplifies complex final stage adders. The combination of these methods allowed us to verify complex multiplier circuits up to an order of magnitude faster than existing techniques. Furthermore, we were able to provide a uniform PAC proof certificate without causing any extra overhead of simulation.

In future work, we want to apply our general techniques of dual variables and tail substitution to more general circuit verification. A further goal is to determine a minimal representation of pseudo-Boolean polynomials using dual variables.

REFERENCES

[1] M. Alekhnovich, E. Ben-Sasson, A. A. Razborov, and A. Wigderson. Space complexity in propositional calculus. *SIAM J. Comput.*, 31(4):1184–1211, 2002.

[2] T. Becker, V. Weispfenning, and H. Kredel. *Gröbner Bases*. Springer, 1993.

[3] A. Biere. Collection of combinational arithmetic miters submitted to the SAT Competition 2016. In *SAT Competition 2016*, volume B-2016-1 of *Dep. of CS Series of Pub. B*, pages 65–66. Univ. Helsinki, 2016.

[4] R. E. Bryant and Y. Chen. Verification of arithmetic circuits using binary moment diagrams. *STTT*, 3(2):137–155, 2001.

[5] M. J. Ciesielski, T. Su, A. Yasin, and C. Yu. Understanding algebraic rewriting for arithmetic circuit verification: a bit-flow model. *IEEE TCAD*, pages 1–1, 2019.

[6] D. Cox, J. Little, and D. O'Shea. *Ideals, Varieties, and Algorithms*. Springer-Verlag New York, 1997.

[7] S. F. de Rezende, M. Lauria, J. Nordström, and D. Sokolov. The power of negative reasoning. In *CCC*, volume 200 of *LIPIcs*, pages 40:1–40:24. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021.

[8] M. J. H. Heule and A. Biere. Proofs for satisfiability problems. In *All about Proofs, Proofs for All*, volume 55, pages 1–22, 2015.

[9] N. Homma, Y. Watanabe, T. Aoki, and T. Higuchi. Formal design of arithmetic circuits based on arithmetic description language. *IEICE Transactions*, 89-A(12):3500–3509, 2006.

[10] D. Kaufmann. *Formal Verification of Multiplier Circuits using Computer Algebra*. PhD thesis, Informatik, Johannes Kepler University Linz, 2020.

[11] D. Kaufmann and A. Biere. Amulet 2.0 for verifying multiplier circuits. In *TACAS (2)*, volume 12652 of *LNCS*, pages 357–364. Springer, 2021.

[12] D. Kaufmann, A. Biere, and M. Kauers. Verifying large multipliers by combining SAT and computer algebra. In *FMCAD 2019*, pages 28–36. IEEE, 2019.

[13] D. Kaufmann, A. Biere, and M. Kauers. From DRUP to PAC and back. In *DATE 2020*, pages 654–657. IEEE, 2020.

[14] D. Kaufmann, M. Fleury, and A. Biere. Pacheck and Pastèque, Checking Practical Algebraic Calculus Proofs. In *FMCAD 2020*, volume 1 of *FMCAD*, pages 264–269. TU Vienna Academic Press, 2020.

[15] J. Lv, P. Kalla, and F. Enescu. Efficient Gröbner basis reductions for formal verification of Galois field arithmetic circuits. *IEEE TCAD*, 32(9):1409–1420, 2013.

[16] A. Mahzoon, D. Große, and R. Drechsler. RevSCA: Using reverse engineering to bring light into backward rewriting for big and dirty multipliers. In *DAC*, pages 185:1–185:6. ACM, 2019.

[17] A. Mahzoon, D. Große, C. Scholl, and R. Drechsler. Towards formal verification of optimized and industrial multipliers. In *DATE 2020*, pages 544–549. IEEE, 2020.

[18] B. Parhami. *Computer Arithmetic - Algorithms and Hardware designs*. Oxford University Press, 2000.

[19] A. A. R. Sayed-Ahmed, D. Große, M. Soeken, and R. Drechsler. Equivalence checking using gröbner bases. In *FMCAD 2016*, pages 169–176. IEEE, 2016.

[20] M. Temel, A. Slobodová, and W. A. Hunt. Automated and scalable verification of integer multipliers. In *CAV (1)*, volume 12224 of *LNCS*, pages 485–507. Springer, 2020.