Preprint from Design Automation Conference (DAC), July 2020, San Francisco, USA

# Symbolic Computer Algebra and SAT Based Information Forwarding for Fully Automatic Divider Verification

Christoph Scholl
*Department of Computer Science*
*Albert-Ludwigs-University Freiburg*
Freiburg i. Br., Germany
scholl@informatik.uni-freiburg.de

Alexander Konrad
*Department of Computer Science*
*Albert-Ludwigs-University Freiburg*
Freiburg i. Br., Germany
konrada@informatik.uni-freiburg.de

***Abstract*—During the last few years Symbolic Computer Algebra (SCA) delivered excellent results in the verification of large integer and finite field multipliers at the gate level. In contrast to those encouraging advances, SCA-based divider verification has been still in its infancy and awaited a major breakthrough. In this paper we analyze the fundamental reasons that prevented the success for SCA-based divider verification so far and present SAT Based Information Forwarding (SBIF). SBIF enhances SCA-based backward rewriting by information propagation in the opposite direction. We successfully apply the method to the fully automatic formal verification of large non-restoring dividers.**

## I. Introduction

Today arithmetic circuits play a crucial role in many computationally intensive applications (like signal processing and cryptography). In order to satisfy the demands for high speed, low power, and low area designs, a large variety of architectures have been proposed for arithmetic units. These architectures take advantage of sophisticated algorithms to optimize different implementation aspects. As a result, they are usually extensively parallel and structurally complex which makes it extremely challenging to ensure the correctness of such arithmetic circuit implementations.

In this paper we look into fully automatic formal verification that goes beyond incomplete simulation-based approaches and semi-automatic approaches based on theorem proving which are still the state-of-the-art for arithmetic circuit verification in industrial practice. *Formal* verification is needed to provide rigorous guarantees concerning the correctness of arithmetic circuits. The need for *full automation* has substantially increased during the last years, since the design of circuits containing arithmetic is nowadays not only confined to the major processor vendors, but is also done by many different suppliers of special-purpose embedded hardware who cannot afford to employ large teams of specialized verification engineers being able to provide human-assisted theorem proofs.

Most arithmetic circuits are based on implementations of the four basic operations $+$, $-$, $\times$, and $\div$. While the automatic formal verification of adder circuits (and hence also subtractor circuits) turned out to be easier, multipliers and dividers formed a major problem for a long time. Fortunately, for the automatic formal verification of gate-level multipliers there has been great progress during the last few years. Methods based on Symbolic Computer Algebra (SCA) were able to verify large, structurally complex and highly optimized multipliers. Both finite field multipliers [1] and integer multipliers [2]–[12] have been considered. Here the verification task has been reduced to an ideal membership test for the specification polynomial based on so-called backward rewriting, proceeding from the outputs of the circuit in direction of the inputs.

Here we look into the formal verification of divider circuits where the situation is not such beneficial as for multipliers. After the famous Pentium bug [13] in 1994, where a subtle design error in the divider had not been detected by Intel's design validation (leading to erroneous Pentium chips brought to the market), a lot of research efforts went into divider verification. In one of the first approaches by Bryant [14], Decision Diagrams have been used to prove the correctness of an SRT divider [15]. However, the approach considered only a single stage of the divider (at the gate level). Methods based on word-level model checking [16] looked into SRT division as well, but considered only a special abstract and clean sequential (i.e., non-combinatorial) divider without gate-level optimizations. In [17] a very general (negative) result about the complexity of Decision Diagram representations for division has been proven, implying that for all known Word-level Decision Diagrams such as MTBDDs, EVBDDs, BMDs, *BMDs, K*BMDs and *PHDDs the size of the Decision Diagrams representing division and remainder is exponential in the number of input bits, independently from the used variable order. Other approaches like [18], [19], or [20] looked into fixed division algorithms and used semi-automatic theorem proving with ACL2, Analytica, or Forte to prove their correctness. Nevertheless, all those efforts did not lead to a fully automated verification method suitable for gate-level dividers. Related to modern SCA-based verification approaches, Hamaguchi et al. [21] mentioned already in 1995 a simple and fully automated method to verify integer dividers using *BMDs [22]. Assume that an integer divider for dividend $X$ and divisor $D$ computes the quotient $Q = \lfloor \frac{X}{D} \rfloor$ and the remainder $R = Q \times D - X$. Hamaguchi et al. start with a *BMD representing $Q \times D + R$ and use a backward construction to replace the bits of $Q$ and $R$ step by step by *BMDs representing the gates of the divider. (Note that Hamaguchi's proposal can be reinterpreted in terms of SCA-based verification as will become clear later on in this paper.) However, in their experiments they observed exponential blow-ups of *BMDs in the backward construction and thus the approach did not provide an effective way for verifying large integer dividers. Moreover, some first results on SCA-based divider verification have been published in [23] recently. However, they are mainly confined to division by constants and cannot handle general dividers due to a memory explosion problem.

The approach proposed in this paper uses Symbolic Computer Algebra (SCA) methods as well and enhances them with so-called SAT Based Information Forwarding (SBIF). It starts with a representation of the "specification polynomial" $Q \times D + R - X$ and (similar to Hamaguchi's method) tries to prove by "backward substitution" of gate output variables with the gates' specification polynomials (called backward rewriting) that the specification polynomial finally reduces to 0. We show that this approach does not work for large dividers due to an exponential blow-up of the sizes of intermediate polynomials and analyze fundamental reasons for this situation. Our main observation is that backward rewriting in direction from the outputs to the inputs will not be successful without adding information propagated in the opposite direction from the inputs to the outputs.

This information propagation is performed by SBIF which is crucial for the efficiency of the method. It takes into account constraints on the allowed range of inputs as well as the structure of the circuit. We successfully use the method to verify large non-restoring dividers. In this context it turns out that a simple version of SBIF is sufficient to keep the sizes of intermediate polynomials small: Before adding a variable corresponding to a signal $a$ in the circuit through backward rewriting, we check (using SAT) whether there is an equivalent or antivalent signal $b$ (i.e. $a$ represents the same Boolean function as $b$ or its negation). If this is the case, we use a unique "representative variable" (or its negation) for all equivalent (or antivalent) signals, avoiding to introduce different variables for identical (or negated) Boolean functions. By this the occurring polynomials are simplified early on. The results of SBIF have to be exploited as early as possible during SCA-based backward rewriting to avoid an exponential growth of the polynomials. By using SBIF we do not compute a different polynomial in the end, but profit from "forward" information that would be used later on in the process. We expect that SBIF is able to improve the SCA based verification of other arithmetic circuits as well – just as the so–called AND-/EXOR-rule [6] which can be used to remove "vanishing monomials" early during the backward substitution process.

The verification of a divider is not complete, if we have not shown that the computed remainder $R$ is always smaller than the divisor $D$. To our great surprise, this did not form any problem and could be easily proven using BDDs [24].

Note that our method works on the gate level netlist without using any hierarchy information which may have been lost during logic optimization. Moreover, the correctness of the method does not rely on the availability of a "golden specification circuit" for division. It only uses the natural abstract specification from the definition of division.

The paper is structured as follows: In the next section we present preliminaries like pseudo-Boolean polynomials, SCA-based verification, and divider circuits. In Sect. III we look into the basic SCA-based verification method for dividers and analyze why it fails for divider verification. In Sect. IV we present SBIF to improve on the previous methods. Sect. V is devoted to the remaining task of showing that the remainder is always smaller than the divisor. We evaluate the approach by experiments in Sect. VI and conclude the paper by Sect. VII.

## II. Preliminaries

### A. SCA for Verification

Symbolic Computer Algebra (SCA) based approaches work with polynomials and reduce the verification task to an ideal membership test using a Gröbner base representation of the ideal. The ideal membership test is performed using polynomial division. While Gröbner base theory is very general and, e.g., can be applied to finite field multipliers [1] and truncated multipliers [12] as well, for integer arithmetic it boils down to substitutions of variables for gate outputs by polynomials over the gate inputs (in reverse topological order), if we choose an appropriate "term order" (see [6] or [9], e.g.). Due to lack of space and to keep the explanation as simple as possible we restrict ourselves to exactly this view. Experimental results for multipliers in [9] showed that those substitutions were much more efficient than polynomial divisions in computer algebra systems, even though the computed results were the same.

For integer arithmetic we consider polynomials over binary variables (from a set $X = \{x_1, \ldots, x_n\}$) with integer coefficients, i. e., a polynomial is a sum of terms, a term is a product of a monomial

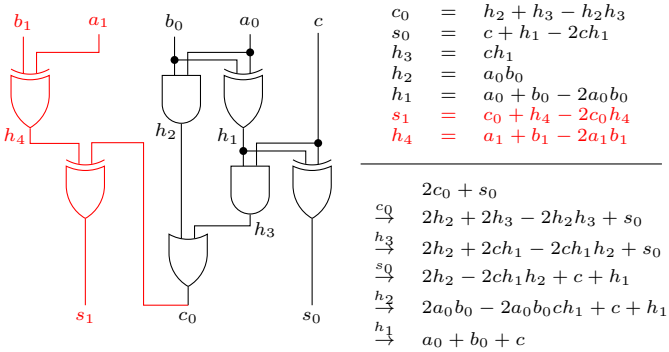


Fig. 1. Circuit with gate polynomials (upper right part) and series of substitutions (lower right part, only for black colored part of the circuit).

with an integer, and a monomial is a product of variables from $X$. Polynomials represent *pseudo-Boolean functions* $f : \{0,1\}^n \mapsto \mathbb{Z}$.

As a simple example consider the full adder from Fig. 1 (black colored part). The full adder defines a pseudo-Boolean function $f_{FA} : \{0,1\}^3 \mapsto \mathbb{Z}$ with $f_{FA}(a_0, b_0, c) = a_0 + b_0 + c$. We can compute a polynomial representation for $f_{FA}$ by starting with a weighted sum $2c_0 + s_0$ (called the "output signature" in [5]) of the output variables. Step by step, we replace the variables in polynomials by the so–called "gate polynomials". This replacement is performed in reverse topological order of the circuit, see Fig. 1. We start by replacing $c_0$ in $2c_0 + s_0$ by its gate polynomial $h_2 + h_3 - h_2h_3$ (which is derived from the Boolean function $c_0 = h_2 \vee h_3$). Finally, we arrive at the polynomial $a_0 + b_0 + c$ (called the "input signature" in [5]) representing the pseudo-Boolean function defined by the circuit. During this procedure (which is called *backward rewriting*) the polynomials are simplified by reducing powers $v^k$ of variables $v$ with $k > 1$ to $v$ (since the variables are binary), by combining terms with identical monomials into one term, and by omitting terms with leading factor 0. We can also consider $a_0 + b_0 + c = 2c_0 + s_0$ as the "specification" of the full adder. The circuit implements a full adder iff backward substitution, now starting with $2c_0 + s_0 - a_0 - b_0 - c$ instead of $2c_0 + s_0$ reduces the "specification polynomial" to 0 in the end. (This is the notion usually preferred in SCA based verification.)

The correctness of this statement can be derived from results of SCA or simply from the fact that polynomials (with the above mentioned simplifications resp. normalizations) are canonical representations of pseudo-Boolean functions (up to reordering of the terms). The canonicity easily results as follows: Let $p_1$ and $p_2$ be different polynomials. Then one of the two polynomials contains a term that is not included in the other. Let $t = zx_{i_1} \ldots x_{i_k}$ with $z \neq 0$ be the shortest term with this property. W. l. o. g. it belongs to $p_1$. Consider the valuation $x_{i_1} = \ldots = x_{i_k} = 1$ and $x_j = 0$ for all $x_j \in X \setminus \{x_{i_1}, \ldots, x_{i_k}\}$. This valuation evaluates $p_1 - p_2$ to $z - z'$, if $p_2$ contains $t' = z'x_{i_1} \ldots x_{i_k}$ with $z' \neq z$ and to $z$ otherwise. This shows that $p_1$ and $p_2$ represent different pseudo-Boolean functions. Polynomials can be seen as "flattened" *BMDs [22] which are canonical as well. In that way the *BMD based backward substitution method by Hamaguchi [21] is essentially the same as the backward rewriting methods from SCA. Whereas *BMDs can be more compact than polynomials due to factorization and sharing, the rather fine-granular and local optimization capabilities of polynomials make them more flexible to use and easier to optimize. In addition, the success of recent SCA-based methods builds on the exploitation of degrees of freedom in the backwards construction of polynomials that had never been used before.

### B. Divider Circuits

In the following we briefly review textbook knowledge on computer arithmetic and dividers in particular. We use the notions $\langle a_n, \ldots, a_0 \rangle := \sum_{i=0}^{n} a_i 2^i$ and $[a_n, \ldots, a_0]_2 := (\sum_{i=0}^{n-1} a_i 2^i) - a_n 2^n$ for interpretations of bit vectors $(a_n, \ldots, a_0) \in \{0,1\}^{n+1}$ as unsigned binary numbers and two's complement numbers respectively. The leading bit $a_n$ is called the sign bit. An unsigned integer divider is a circuit with the following property:

**Definition 1.** *Let $(r^{(0)}_{2n-2} \ldots r^{(0)}_0)$ be the dividend with sign bit $r^{(0)}_{2n-2} = 0$ and value $R^{(0)} := \langle r^{(0)}_{2n-2} \ldots r^{(0)}_0 \rangle = [r^{(0)}_{2n-2} \ldots r^{(0)}_0]_2$, $(d_{n-1} \ldots d_0)$ be the divisor with sign bit $d_{n-1} = 0$ and value $D := \langle d_{n-1} \ldots d_0 \rangle = [d_{n-1} \ldots d_0]_2$, and $0 \leq R^{(0)} < D \cdot 2^{n-1}$.* Then $(q_{n-1} \ldots q_0)$ with value $Q = \langle q_{n-1} \ldots q_0 \rangle$ is the quotient of the division and $(r_{2n-2} \ldots r_0)$ with value $R = [r_{2n-2} \ldots r_0]_2$ is the remainder of the division, if $R^{(0)} = Q \cdot D + R$ (vc1) and $0 \leq R < D$ (vc2).*

*Restoring division* is the simplest algorithm to compute quotient and remainder. It computes the quotient bits and "partial remainders" $R^{(j)}$ step by step. In each step it subtracts a shifted version of $D$. If the result is less than 0, the corresponding quotient bit is 0 and the shifted version of $D$ is "added back", i. e., "restored". Otherwise the quotient bit is 1 and we proceed with the next smaller shifted version of $D$. More precisely, restoring division works according to the following algorithm:

For $j = 1$ to $n$ do:

$$q_{n-j} = \begin{cases} 0, & \text{if } R^{(j-1)} - D \cdot 2^{n-j} < 0 \\ 1, & \text{if } R^{(j-1)} - D \cdot 2^{n-j} \geq 0 \end{cases}$$

$$R^{(j)} = R^{(j-1)} - q_{n-j}(D \cdot 2^{n-j}).$$

The final remainder $R$ is equal to $R^{(n)}$.

The idea of *non-restoring division* is to combine in case of a negative partial remainder two steps of restoring division: adding the shifted $D$ back and (tentatively) subtracting the next $D$ shifted by one position less. These two steps are replaced by just adding $D$ shifted by one position less (which obviously leads to the same result). A combinational circuit computing $Q$ and $R$ by non-restoring division is given in Fig. 2. It works with two's complement numbers and contains a subtractor, several combined adder/subtractors (denoted by *CAS* in Fig. 2), and an adder.

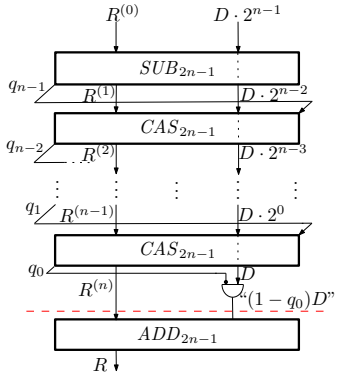


Fig. 2. Non-restoring divider.

The following well-known textbook result shows how addition of two's complement numbers can be reduced to addition of unsigned numbers:

**Lemma 1.** *Consider $a = (a_n, \ldots, a_0), b = (b_n, \ldots, b_0) \in \{0,1\}^{n+1}$, $c \in \{0,1\}$, $s = (s_n, \ldots, s_0) \in \{0,1\}^{n+1}$ with $\langle c_n, s \rangle = \langle a \rangle + \langle b \rangle + c$. (For $0 \leq i \leq n$ $c_i$ is the carry bit from the unsigned addition of $(a_i, \ldots a_0)$, $(b_i, \ldots b_0)$ with incoming carry $c$.) It holds $[a]_2 + [b]_2 + c \in \{-2^n, \ldots, 2^n - 1\}$ iff $c_n = c_{n-1}$. If this is the case, then we have $[a]_2 + [b]_2 + c = [s]_2$.*

*As in the case of multipliers where the number of product bits is two times the number of bits of one factor, we consider here the general case that the dividend has twice as many bits as the divisor. If both the dividend and the divisor are supposed to have the same length, we just set $r^{(0)}_{2n-2} = \ldots = r^{(0)}_{n-1} = 0$ and require $D > 0$ (which then implies $0 \leq R^{(0)} < D \cdot 2^{n-1}$).

TABLE I
PEAK SIZES OF POLYNOMIALS.

| $n$ | 2 | 4 | 8 | 16 |
|---|---|---|---|---|
| peak size | 27 | 1591 | 5,363,443 | MEMOUT |

The lemma basically says that the reduction of signed addition to unsigned addition only works, if there is no overflow, i. e., if the result can be represented as an $n+1$-bit signed binary number as well. Note that for the non-restoring divider it can be proven that there is no overflow during signed addition / subtraction by construction. As usual, subtraction of two's complement numbers can be reduced to addition by using $-[a]_2 = [\overline{a}]_2 + 1$ where $\overline{a}$ means bitwise negation of all $a_i$-bits. By this, it is also easy to design a combined adder/subtractor (CAS) for $a$ and $b$: If a control input $c$ is 0, we just add $a$ and $b$, if $c$ is 1 we add $a$, $\overline{b}$, and 1 (i. e., we subtract $b$ from $a$). This can be achieved by replacing in an adder for $a$ and $b$ the bits $b_i$ by $b_i \oplus c$ and the incoming carry by $c$.

## III. Limits of Existing SCA-based Techniques

Here we look into the question why existing SCA-based techniques for divider verification fail. To verify verification condition (vc1) from Def. 1 we start with a representation of $SP = Q \cdot D + R - R^{(0)} = (\sum_{i=0}^{n-1} q_i 2^i) \cdot (\sum_{i=0}^{n-2} d_i 2^i) + (\sum_{i=0}^{2n-3} r_i 2^i) - r_{2n-2} 2^{2n-2} - (\sum_{i=0}^{2n-3} r_i^{(0)} 2^i)$ by a polynomial and use the method summarized in Sect. II-A, i. e., we try to prove that after backward rewriting $SP$ finally reduces to 0. The backward rewriting is performed in reverse topological order from inputs to the outputs. To keep the sizes of the intermediate polynomials small, we use known heuristics from [10], [11].† We considered non-restoring dividers as in Fig. 2 with varying bit widths. Table I shows the peak sizes (measured in the numbers of terms) of the intermediate polynomials with increasing bit width. It clearly shows that the original method is not able to verify dividers with larger bit widths. For the 8-bit-divider already more than 5 million terms are needed, the 16-bit divider could not be verified with the available memory of 62 GiB. Fig. 3 depicts for the 8-bit divider how the polynomial sizes develop from substitution to substitution. Although in the end the polynomial reduces to 0, we observe huge peak sizes in between.

To analyze the reasons of this situation we consider a simpler problem first and look at a signed adder according to Lemma 1. Using induction and some algebraic manipulations we can prove the following lemma (the proof is omitted due to lack of space):

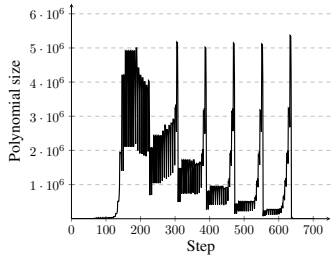


Fig. 3. Sizes of polynomials during verification of 8-bit divider.

**Lemma 2.** *The pseudo-Boolean function for the binary adder according to Lemma 1 is represented by the polynomial*

$$A_n = (\sum_{i=0}^{n-1} a_i 2^i - a_n 2^n) + (\sum_{i=0}^{n-1} b_i 2^i - b_n 2^n) + c - 2^{n+1} P_n$$

*where $P_n = C_n \cdot (1 - a_n - b_n + 2a_n b_n) - a_n b_n$ and $C_n$ represents the pseudo-Boolean function for the carry bit $c_{n-1}$ (expressed with input bits $a_{n-1}, \ldots, a_0$, $b_{n-1}, \ldots, b_0$, $c$). $C_n$ contains $\frac{1}{2}(3^{n+1} - 1)$ terms and thus $P_n$ contains $2 \cdot 3^{n+1} - 1$ terms.*

†In contrast to [11], we restrict ourselves just to the detection of half adders and full adders as atomic blocks and do not consider compressors.

This means that we arrive at a polynomial with exponential size, if we start with the polynomial $\sum_{i=0}^{n-1} s_i 2^i - s_n 2^n$ for the outputs of the binary adder and replace the gates of the circuit in reverse topological order. The reason why we do not arrive at $(\sum_{i=0}^{n-1} a_i 2^i - a_n 2^n) + (\sum_{i=0}^{n-1} b_i 2^i - b_n 2^n) + c$ is that the implementation is *only an adder, if the result can be represented with $n+1$ bits (as the operands), i. e., if no overflow occurs*. If we know for instance that one operand is positive and the other is negative, i. e., $a_n = \overline{b_n}$, then there will not be any overflow and $(1 - a_n - b_n + 2a_n b_n)$ as well as $P_n$ vanish.

Now we consider backward rewriting for the non-restoring divider shown in Fig. 2, started with polynomial $SP$. With similar techniques as in Lemma 2 we can see that the polynomial resulting after processing the final adder (see cut marked by the dashed red line in Fig. 2) *has exactly* $3^{n-1} + n^2 + n - 3$ *terms*. Now we can identify the reason why existing SCA-based methods for verifying dividers as in Fig. 2 have to fail: Whereas from the division algorithm it follows that overflows of adder, subtractor, and CAS stages cannot occur, this is not immediately visible from the circuit in Fig. 2. In particular, backward rewriting is not able to detect this when constructing polynomials from outputs to inputs, since *this information can only be obtained by propagating constraints from inputs to outputs*, i. e., in the opposite direction.

## IV. SAT Based Information Forwarding

In the last section we observed that a simple backward rewriting without using information propagated in direction from inputs to outputs will not be successful and fails already after processing the last divider stage. Such "forward information propagation" can be interpreted as a restricted form of computing don't cares. Here we will make use of the strength of modern SAT solvers to compute the needed information. This "don't care information" will be used to simplify polynomials as early as possible. We call this method *SAT Based Information Forwarding (SBIF)*.

Here we restrict the SAT based information propagation to the question whether there are pairs of signals $a$ and $b$ that are equivalent or antivalent (i. e., $a$ represents the same Boolean function as $b$ or its negation). Alg. 1 summarizes the approach.

**Input :** Input constraint $C$, Circuit $CUV$ with set of signals $S = \{a_1, \ldots, a_n\}$, maximal window depth $d_{max}$
**Output:** Partition $E$ of signals (or their negations) into equivalence classes

1 Choose set $V$ of input vectors satisfying $C$;
2 Simulate $CUV$ with vectors from $V$ leading to simulation vectors $sim(a)$ and $sim(\overline{a})$ for each $a \in S$;
3 Choose topological order $\prec_{top}$ for $S$;
4 $E = \{\{a_1\}, \ldots, \{a_n\}\}$;
5 **foreach** $a \in S$ *in topological order* **do**
6  **foreach** $b \prec_{top} a$ *with* $sim(a) = sim(b^\varepsilon)$ **do**
7  **if** $a \notin [b]$ **then**
8  **for** $s \in \{a, b\}$ **do**
9  $W_s = \{g \mid g \in$ $S$ is an $i$-step predecessor of $s$ with $i \leq d_{max}\}$;
10  **if** $UNSAT(CNF(a \oplus b^\varepsilon, W_a, W_b, C))$ **then**
11  $E = (E \setminus \{[a], [b]\}) \cup \{[a] \cup [b^\varepsilon]\}$;
12 **return** $E$;

**Algorithm 1:** SAT based information propagation.

The algorithm works with a constraint $C$ for the input variables of the circuit under verification $CUV$. In our case $C = (0 \leq R^{(0)} < D \cdot 2^{n-1})$ (or $D > 0$ for a dividend with only $n$ bits). First, we simulate the circuit with input vectors satisfying $C$. To take antivalence into account as well, we consider both the simulation vector $sim(a)$ and its bitwise negation $sim(\overline{a})$ for each signal $a$. Then we check using a SAT solver whether the candidate equivalences / antivalences are real ones. For this we partition the set of signals into equivalent or antivalent signals. In the beginning each signal is in its own equivalence class (line 4). We process the signals $a$ in topological order and check whether there are equivalent / antivalent predecessor signals $b$; candidates are found by considering simulation vectors (line 6). (We use the notion $b^1 = b$ and $b^0 = \overline{b}$, i. e., in case $\varepsilon = 0$ our candidate equivalence of $(a, \overline{b})$ is actually a candidate antivalence of $(a, b)$.) To be able to handle large circuits we do not consider the complete logic cones of the signals $a$ and $b$ in the comparison $a \oplus b^\varepsilon$, but only restricted "windows" $W_a$ and $W_b$ up to a certain maximal depth $d_{max}$. Note that in the computation of those windows we replace the predecessors $pred(t)$ of signals $t$ by the topologically minimal representatives of their corresponding equivalence classes $[pred(t)]$. This also explains why we examine the candidate pairs in topological order: We collect information from the inputs to the outputs and the handling in topological order enables to use already computed information (since we only consider input cones of signals in our SAT checks). The counterexamples to equivalence have to satisfy the given input constraint $C$, of course. If an equivalence is detected, the corresponding equivalence classes are merged (in case of antivalence all signals in $[b]$ have to be negated (line11)).

It is interesting to note that Alg. 1 is able to prove e.g. $\overline{q_{n-j}} = r^{(j)}_{2n-2}$ for all $j \in \{1, \ldots, n\}$ in Fig. 2 with windows of depth 4. Without using already computed information in the following SAT checks this would not have been possible, since those antivalences cannot be detected by local reasoning.

Alg. 1 forwards information that results both from the input constraints and from the circuit implementation. In the next step we make use of this information in SCA-based backward rewriting. In Alg. 2 a simplified version of backward rewriting is presented that starts with the specification polynomial and performs substitutions of variables by gate polynomials in reverse topological order. First of all, signals in the specification polynomial are replaced by the topologically minimal representatives in their equivalence classes that have been computed by Alg. 1 (lines 2–4). In the same way, before replacing a signal $r_i$ with its gate polynomial $p_{r_i}$, we in turn replace the signals in the gate polynomial with the topologically minimal representatives in their equivalence classes (lines 6–8). It is crucial for the success of the approach that those replacements are done *as early as possible*, such that polynomials are not reduced after an exponential blow-up in size has happened, but such a blow-up is prevented *before* it can occur.

**Input :** Circuit $CUV$ with topological order $\prec_{top}$ on signals.
Let $E = \{e_1, \ldots, e_m\}$ be returned by Alg. 1.
$\forall 1 \leq i \leq m$ let $r_i^{\varepsilon_{r_i}} \in e_i$ with $r_i$ minimal wrt. $\prec_{top}$.
Let $r_1 \prec_{top} \ldots \prec_{top} r_m$, $p_{r_i}$ is the gate polynomial of $r_i$.
**Output:** 1 iff specification holds

1 $SP_m := SP$; $i = m$;
2 **while** $SP_m$ *depends on* $a$ *with* $a^{\varepsilon_a} \in e_j$, $a \neq r_j$ **do**
3  **if** $\varepsilon_a = \varepsilon_{r_j}$ **then** $SP_m = SP_m|_{a \leftarrow r_j}$;
4  **else** $SP_m = SP_m|_{a \leftarrow (1 - r_j)}$ ;
5 **while** $i \geq 0$ **do**
6  **while** $p_{r_i}$ *depends on* $a$ *with* $a^{\varepsilon_a} \in e_j$, $a \neq r_j$ **do**
7  **if** $\varepsilon_a = \varepsilon_{r_j}$ **then** $p_{r_i} = p_{r_i}|_{a \leftarrow r_j}$;
8  **else** $p_{r_i} = p_{r_i}|_{a \leftarrow (1 - r_j)}$ ;
9  $SP_{i-1} = SP_i|_{r_i \leftarrow p_{r_i}}$; $i = i - 1$;
10 **return** $(SP_0 = 0)$;

**Algorithm 2:** Modified backward rewriting.

**Example 1.** *Consider the circuit in Fig. 1 again, now also with the*

*parts shown in red. $c_0$ now becomes an internal signal. Suppose that we start backward rewriting with the polynomial $s_0 - 2s_1$. After substituting the 7 gate polynomials we arrive at the polynomial $a_0 - 2a_1 + b_0 - 2b_1 + c - 4a_0b_0 - 4a_0c - 4b_0c + 8a_0b_0c + 4a_0a_1b_0 + 4a_0a_1c + 4a_1b_0c - 8a_0a_1b_0c + 4a_0b_0b_1 + 4a_0b_1c + 4b_0b_1c - 8a_0b_0b_1c - 8a_0a_1b_0b_1 - 8a_0a_1b_1c - 8a_1b_0b_1c + 16a_0a_1b_0b_1c + 4a_1b_1$. If the circuit in Fig. 1 is part of a larger circuit which implies that $b_1 = \overline{a_1}$, then all 17 terms shown in red will vanish. Now we consider what happens, if we apply Alg. 1 and Alg. 2. Suppose that Alg. 1 is able to prove $b_1 = \overline{a_1}$. The first replacement of $s_1$ leads to $s_0 - 2c_0 - 2h_4 + 4c_0h_4$. Before the replacement of $h_4$ by its gate polynomial $a_1 + b_1 - 2a_1b_1$ we notice that $\{b_1, \overline{a_1}\} \in E$, thus (assuming $a_1 \prec_{top} b_1$) the gate polynomial is simplified to 1 by substituting $b_1$ with $1 - a_1$. Replacing $h_4$ by the modified gate polynomial 1 results in $2c_0 + s_0 - 2$ and the remaining substitutions are the same as in Fig. 1 (apart from the additional term $-2$). During the modified backward rewriting we never observe more than 5 terms in a polynomial. If we were to use the knowledge gained by Alg. 1 only after substituting all 7 gates however, the 17 terms shown in red would be introduced, of course.*

## V. Verifying the Size of the Remainder

Verifying condition (vc1) from Def. 1 is not sufficient to prove that a circuit implements a divider. We have to prove as well that for the remainder $0 \leq R < D$ holds (condition (vc2)). Unfortunately, backward rewriting starting with a polynomial for $0 \leq R < D$ fails, since already the polynomial representation for $0 \leq R < D$ has exponential size. In contrast, there is a BDD [24] of *linear size* that represents this constraint. The variable ordering is chosen such that bits of $R$ and $D$ having the same index are arranged side by side and bits with higher indices are evaluated first. Starting from the BDD for the predicate $0 \leq R < D$ we perform a backward traversal of the circuit using the same reverse topological order as for the backward rewriting of Sect. IV. Now we substitute BDD variables for gate outputs with BDDs for the gate functions. Finally, we obtain a BDD for a predicate $WPC$ depending only on input variables. It is easy to see that $WPC$ represents the weakest precondition on the input variables that implies that $0 \leq R < D$ holds at the outputs of the circuit. Then condition (vc2) holds, iff the constraint $C = (0 \leq R^{(0)} < D \cdot 2^{n-1})$ on the input variables implies $WPC$, i. e., iff the BDD for $\overline{C} \vee WPC$ is constant 1. For computing an initial order on the BDD variables we use [25], extended to the case that the relative order of certain variables (in our case the variables of $R$ and $D$ as mentioned above) has already been fixed. During the computation of $WPC$ we use symmetric sifting [26] as a dynamic variable reordering method. To our great surprise, our experiments showed that those simple measures were sufficient to make the building of the needed BDDs possible for large bit widths $n$, see Sect. VI.

## VI. Experimental Results

For our experiments we have used one core of an Intel Xeon CPU E5-2643 with 3.3 GHz and 62 GiB of main memory. The runtime of all experiments was limited to 72 CPU hours. All run times in our result table (Table II) are given in CPU seconds.

We consider the verification of non-restoring dividers as in Fig 2. Note that we do not use any hierarchy information during verification. We just use the flat gate level netlist and employ heuristics for detecting atomic blocks (restricted to half and full adders) and for finding a good substitution order [10], [11].

We start with two experiments for comparison. In both experiments we check the equivalence of the divider circuit with a "golden specification". For this we construct a miter circuit between the divider and its golden specification and conjoin it with a circuit for $C = (0 \leq R^{(0)} < D \cdot 2^{n-1})$ to ensure that counterexamples are restricted to the allowed range of inputs.

In the first experiment we use a SAT solver (MiniSat 2.2.0 [27]) to solve the corresponding satisfiability problems. In Table II the results are presented in col. 2. SAT solving for non-trivial arithmetic circuits is hard and the SAT problems with bit widths larger than 8 could not be solved due to a timeout.

In the second experiment we use the combinational equivalence checking (CEC) approach of ABC [28], [29]. In contrast to the pure SAT approach, ABC is based on And-Inverter-Graph (AIG) rewriting via structural hashing, simulation, and SAT. This technique reduces the overall complexity of checking equivalence between two designs by finding equivalent internal AIG nodes. Based on those techniques, ABC is able to verify dividers up to 32 bit. However, as already observed in [5] for multiplier verification, finding internal equivalent nodes in non-trivial arithmetic designs is difficult and thus ABC is unable to verify larger dividers, see col. 3 in Table II.

Now we look into results for our method. We use SCA-based methods for verifying that the divider circuits fulfill their abstract specification from Def. 1. In particular, for the proofs we do not need golden specification circuits. As already demonstrated in Sect. III, SCA is not successful without using SBIF. Fig. 4 shows the peak sizes of the polynomials with increasing bit widths. Without SBIF we observe an exponential increase in the peak size of the polynomials during backward rewriting (measured in the number of terms), leading to more than 5 million terms already for the 8-bit divider. With SBIF however the growth in size is moderate, leading to peak sizes of 9,510 and 16,774 for the 96-bit and 128-bit dividers, respectively.‡

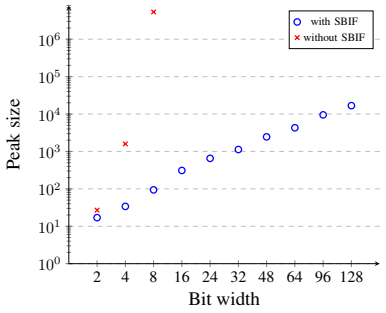


Fig. 4. Peak sizes for $n$-bit dividers.

For a more detailed analysis, we show run times split into different sub-tasks in Table II. In col. 4 we show run times for reading the circuit design, in col. 5–6 we consider forward information propagation using SAT (SBIF, col. 5 gives the number of equivalences / antivalences found by SAT and col. 6 the time needed), and in col. 7 we show run times for backward rewriting making use of the results of forward propagation. The results clearly show that the method using SBIF is able to verify large dividers in short time. E.g. for the 128-bit divider the complete SCA-based verification needs less than 4 CPU minutes. It is interesting to observe that SCA-based rewriting needs only 0.88 CPU seconds, after SBIF has been performed before. The remaining run time is distributed between reading the circuit and SAT solving time for SBIF. Remember that CEC with ABC already needs more than 72 CPU hours (our timeout) for dividers with bit widths of 48 or more.

Finally columns 8 and 9 of Table II give results for the BDD-based verification of condition (vc2) from Def. 1 on the size of the final remainder. We use the BDD package CUDD 3.0.0 [30]. Col. 8 shows peak sizes for the number of BDD nodes during the verification and col. 9 shows the needed run times. The fact that the peak sizes on the number of BDD nodes do not monotonically increase

‡Note that our results do not contradict the negative result from [17], since we do not start backward rewriting with a polynomial for the quotient $Q$ or for the remainder $R$, but we start with a polynomial for $Q \cdot D + R - R^{(0)}$.

TABLE II
VERIFYING NON-RESTORING DIVIDERS, TIMES IN CPU SECONDS.

| $n$ | SAT | ABC | SCA-SBIF | | | | | |
|---|---|---|---|---|---|---|---|---|
| | | | read | SBIF | | rewrite | vc2 | |
| | time | time | time | #equiv | time | time | nodes | time |
| 2 | 0.23 | 0.03 | 0.05 | 21 | $< 0.01$ | $< 0.01$ | 23 | $< 0.01$ |
| 4 | 1.48 | 0.04 | 0.20 | 40 | $< 0.01$ | $< 0.01$ | 106 | 0.01 |
| 8 | 83.06 | 2.07 | 0.80 | 120 | 0.01 | $< 0.01$ | 397 | 0.02 |
| 16 | TO | 8.49 | 3.24 | 376 | 0.05 | 0.01 | 630 | 0.23 |
| 24 | TO | 69.27 | 6.83 | 760 | 0.15 | 0.03 | 1,502 | 1.16 |
| 32 | TO | 171.83 | 13.61 | 1272 | 0.31 | 0.05 | 83,392 | 4.16 |
| 48 | TO | TO | 26.53 | 2680 | 1.67 | 0.11 | 591,365 | 27.53 |
| 64 | TO | TO | 46.57 | 4600 | 4.43 | 0.20 | 843,594 | 71,67 |
| 96 | TO | TO | 110.73 | 9976 | 17.50 | 0.47 | 737,660 | 510.29 |
| 128 | TO | TO | 188.21 | 17400 | 37.16 | 0.88 | 80,911 | 1226.39 |

with increasing bit widths can be explained by the use of dynamic variable reordering which is automatically triggered at favourable or less favourable times during the BDD construction. Nevertheless, the times needed for BDD construction increase monotonically and we obtain surprisingly small CPU times (e.g. about 8.5 CPU minutes for a bit width of 96 and about 20.5 CPU minutes for a bit width of 128). Note that the moderate CPU times for BDD-based verification can only be observed for the verification of condition (vc2). Using BDDs in the context of CEC for single output bits of dividers or for backward rewriting starting with a circuit for $Q \cdot D + R$ (with BDDs instead of polynomials) is not a good idea and immediately fails with exponential numbers of BDD nodes.

## VII. CONCLUSIONS AND FUTURE WORK

With the fully automatic formal verification of divider circuits we solved a problem that has been open for a long time. Moreover, we precisely characterized the reasons why SCA-based approaches for divider verification did not work so far and we presented forward information propagation as a means to enable backward verification of dividers. Our results for non-restoring dividers are only a first step and our next steps will be to evaluate and extend the approach for different divider designs such as SRT division [15] which is based on the same principles as non-restoring division but works with redundant number systems and lookup tables to enable quotient bit determination and addition in constant time, or the Goldschmidt algorithm (also called IBM method) which is based on iterated multiplication [31]. We expect that those architectures will need (possibly extended) forward information as well, which is derived both from the circuit and the allowed input range and is used for optimizing intermediate polynomials.

## REFERENCES


[1] J. Lv, P. Kalla, and F. Enescu, "Efficient Gröbner basis reductions for formal verification of Galois field arithmetic circuits," *IEEE Transactions on Computer Aided Design of Circuits and Systems*, vol. 32, no. 9, pp. 1409–1420, Sept 2013.

[2] O. Wienand, M. Wedler, D. Stoffel, W. Kunz, and G. Greuel, "An algebraic approach for proving data correctness in arithmetic data paths," in *Computer Aided Verification*, 2008, pp. 473–486.

[3] F. Farahmandi and B. Alizadeh, "Gröbner basis based formal verification of large arithmetic circuits using gaussian elimination and cone-based polynomial extraction," *Microprocessors and Microsystems*, vol. 39, no. 2, pp. 83–96, 2015.

[4] M. Ciesielski, C. Yu, D. Liu, and W. Brown, "Verification of gate-level arithmetic circuits by function extraction," in *Design Automation Conf.*, 2015, pp. 52:1–52:6.

[5] C. Yu, W. Brown, D. Liu, A. Rossi, and M. Ciesielski, "Formal verification of arithmetic circuits by function extraction," *IEEE Transactions on Computer Aided Design of Circuits and Systems*, vol. 35, no. 12, pp. 2131–2142, 2016.

[6] 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 *Design, Automation and Test in Europe*, 2016, pp. 1048–1053.

[7] D. Ritirc, A. Biere, and M. Kauers, "Column-wise verification of multipliers using computer algebra," in *Int'l Conf. on Formal Methods in CAD*, 2017, pp. 23–30.

[8] C. Yu, M. Ciesielski, and A. Mishchenko, "Fast algebraic rewriting based on And-Inverter graphs," *IEEE Transactions on Computer Aided Design of Circuits and Systems*, vol. 37, no. 9, pp. 1907–1911, 2017.

[9] D. Ritirc, A. Biere, and M. Kauers, "Improving and extending the algebraic approach for verifying gate-level multipliers," in *Design, Automation and Test in Europe*, 2018, pp. 1556–1561.

[10] A. Mahzoon, D. Große, and R. Drechsler, "PolyCleaner: clean your polynomials before backward rewriting to verify million-gate multipliers," in *International Conference on Computer-Aided Design*, 2018, pp. 129:1–129:8.

[11] ——, "RevSCA: Using reverse engineering to bring light into backward rewriting for big and dirty multipliers," in *Design Automation Conf.*, 2019.

[12] D. Kaufmann, A. Biere, and M. Kauers, "Verifying large multipliers by combining SAT and computer algebra," in *Int'l Conf. on Formal Methods in CAD*, 2019.

[13] T. Coe, "Inside the Pentium FDIV bug," *Dr. Dobbs Journal*, vol. 20, no. 4, pp. 129–135, 1995.

[14] R. E. Bryant, "Bit-level analysis of an SRT divider circuit," in *Design Automation Conf.*, 1996, pp. 661–665.

[15] J. E. Robertson, "A new class of digital division methods," *IRE Trans. Electronic Computers*, vol. 7, no. 3, pp. 218–222, 1958.

[16] E. M. Clarke, M. Khaira, and X. Zhao, "Word level model checking - avoiding the Pentium FDIV error," in *Design Automation Conf.*, 1996, pp. 645–648.

[17] C. Scholl, B. Becker, and T. M. Weis, "On WLCDs and the complexity of word-level decision diagrams – a lower bound for division," *Formal Methods in System Design*, vol. 20, no. 3, pp. 311–326, 2002.

[18] D. M. Russinoff, "A mechanically checked proof of IEEE compliance of the floating point multiplication, division and square root algorithms of the AMD-K7 processor," *LMS Journal Comput. Math.*, vol. 1, pp. 148–200, 1998.

[19] E. M. Clarke, S. M. German, and X. Zhao, "Verifying the SRT division algorithm using theorem proving techniques," *Formal Methods in System Design*, vol. 14, no. 1, pp. 7–44, 1999.

[20] J. O'Leary, X. Zhao, R. Gerth, and C.-J. H. Seger, "Formally verifying IEEE compliance of floating point hardware," *Intel Technology Journal*, vol. Q1, pp. 1–10, 1999.

[21] K. Hamaguchi, A. Morita, and S. Yajima, "Efficient construction of binary moment diagrams for verifying arithmetic circuits," in *International Conference on Computer-Aided Design*, 1995, pp. 78–82.

[22] R. E. Bryant and Y. A. Chen, "Verification of arithmetic circuits with binary moment diagrams," in *Design Automation Conf.*, 1995, pp. 535–541.

[23] A. Yasin, T. Su, S. Pillement, and M. J. Ciesielski, "Formal verification of integer dividers: Division by a constant," in *IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, 2019, pp. 76–81.

[24] R. E. Bryant, "Graph-based algorithms for Boolean function manipulation," *IEEE Trans. on Comp.*, vol. 35, no. 8, pp. 677–691, 1986.

[25] S. Malik, A. R. Wang, R. K. Brayton, and A. L. Sangiovanni-Vincentelli, "Logic verification using binary decision diagrams in a logic synthesis environment," in *International Conference on Computer-Aided Design*, 1988, pp. 6–9.

[26] S. Panda, F. Somenzi, and B. Plessier, "Symmetry detection and dynamic variable ordering of decision diagrams," in *International Conference on Computer-Aided Design*, 1994, pp. 628–631.

[27] N. Eén and N. Sörensson, "An extensible SAT-solver," in *Theory and Applications of Satisfiability Testing*, 2003, pp. 502–518.

[28] R. K. Brayton and A. Mishchenko, "ABC: an academic industrial-strength verification tool," in *Computer Aided Verification*, 2010, pp. 24–40.

[29] "ABC: A system for sequential synthesis and verification," available at https://people.eecs.berkeley.edu/ alanmi/abc/, 2019.

[30] F. Somenzi, "Efficient manipulation of decision diagrams," *STTT*, vol. 3, no. 2, pp. 171–181, 2001.

[31] I. Koren, *Computer Arithmetic Algorithms*, 2nd ed. A. K. Peters, Ltd., 2001.