Preprint from Proceedings of Design, Automation and Test in Europe (DATE), February 2021

Verifying Dividers Using Symbolic Computer Algebra and Don't Care Optimization

Christoph Scholl[1] Alexander Konrad[1] Alireza Mahzoon[2] Daniel Große[3] Rolf Drechsler[2]
[1]University of Freiburg, Germany [2]University of Bremen, Germany [3]Johannes Kepler University Linz, Austria
{scholl, konrada}@informatik.uni-freiburg.de, {mahzoon, drechsle}@informatik.uni-bremen.de, daniel.grosse@jku.at

***Abstract*—In this paper we build on methods based on Symbolic Computer Algebra that have been applied successfully to multiplier verification and more recently to divider verification as well. We show that existing methods are not sufficient to verify optimized non-restoring dividers and we enhance those methods by a novel optimization method for polynomials w. r. t. satisfiability don't cares. The optimization is reduced to Integer Linear Programming (ILP). Our experimental results show that this method is the key for enabling the verification of large and optimized non-restoring dividers (with bit widths up to 512).**

I. Introduction

Arithmetic circuits are crucial components in processor designs as well as in special-purpose hardware for computationally intensive applications like signal processing and cryptography. At the latest since the famous Pentium bug [1] 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), it has been widely recognized that incomplete simulation-based approaches are not sufficient for verification and formal methods should be used to verify the correctness of arithmetic circuits. 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, the interest in *fully automatic formal verification* of arithmetic circuits is growing more and more.

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. Early attempts to represent both multiplier specifications and implementations by canonical forms like BDDs [2] showed exponential space complexity [3]. SAT-based methods did not prove to be scalable either [4]. Nevertheless, for the automatic formal verification of gate-level multipliers there has been great progress during the last few years. Apart from rewriting based approaches like [5], methods based on *Symbolic Computer Algebra (SCA)* were able to verify large, structurally complex, and highly optimized multipliers. Both finite field multipliers [6] and integer multipliers [4], [7]–[17] have been considered in SCA-based approaches. 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.

Although the Pentium bug affected the divider unit, research efforts for divider verification were not as successful as for multipliers. Attempts to use Decision Diagrams for proving the correctness of an SRT divider [18] were confined to a single stage of the divider (at the gate level) [19]. Methods based on word-level model checking [20] looked into SRT division as well, but considered only a special abstract and clean sequential (i.e., non-combinatorial) divider without gate-level optimizations. Other approaches like [21], [22], or [23] 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. Hamaguchi et al. [24] mentioned in a side remark a simple and fully automated method to verify integer dividers using *BMDs [25] which is closely related to the SCA-based method to be presented in this paper. They start with a *BMD representing $Q \times D + R$ (where Q is the quotient, D the divisor, and R the remainder of the division) and use a backward construction to replace the bits of Q and R step by step by *BMDs representing the gates of the divider. The goal is to finally obtain a *BMD representation for the dividend $R^{(0)}$ which proves the correctness of the divider circuit. Unfortunately, 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.

We are aware of three recent approaches which try to take up the success of SCA to provide a fully automatic divider verification. The work in [26] is mainly confined to division by constants and cannot handle general dividers due to a memory explosion problem. [27] works at the gate level, but assumes that hierarchy information in a restoring divider is present. Using this hierarchy information it decomposes the proof obligation $R^{(0)} = Q \times D + R$ into separate proof obligations for each level of the restoring divider. Nevertheless, the approach scales only to medium-sized bit widths (up to 21 as shown in the experimental results of [27]). The approach of [28] works on the gate level as well, but it does not need any hierarchy information. It proves the correctness of non-restoring dividers by "backward rewriting" starting with the "specification polynomial" $Q \times D + R - R^{(0)}$. Here backward rewriting performs "backward substitutions" of gate output variables with the gates' specification polynomials. By finally obtaining the 0-polynomial they prove the dividers to be correct. To avoid an exponential blow-up during backward rewriting, their method uses information on equivalent and antivalent signals which is derived by forward propagating the allowed range of the divider inputs, i. e., the constraint $0 \leq R^{(0)} < D \cdot 2^{n-1}$, using SAT. The method scales very well, but it is restricted to a very clean implementation of non-restoring division. Using optimizations to the implementation (which are usually even contained in textbook designs) leads to a memory explosion problem for this approach, too.

Here we present a novel SCA-based verification method which is basically along the lines of [28], but completely avoids the problems mentioned above. Our method works on the gate-level netlist without using any hierarchy information which may have been lost during logic optimization. As [28] it does not use a "golden specification circuit" for division, but only uses the natural abstract specification from the definition of division. We consider optimized non-restoring dividers and show that the method from

This work was supported by the German Research Foundation (DFG) within the project VerA (SCHO 894/5-1, GR 3104/6-1 and DR 297/37-1).

[28] leads to memory explosion in this case. Whereas we agree with the authors of [28] in the analysis that backward rewriting cannot be efficient without forward propagation of information, we do not limit ourselves to the detection and exploitation of equivalences and antivalences as in [28]. Rather we compute satisfiability don't cares of atomic blocks in the divider circuit which are present due to the input constraint $0 \leq R^{(0)} < D \cdot 2^{n-1}$. We use those satisfiability don't cares to *optimize polynomials modulo don't cares*, i. e., from a polynomial P we compute a smaller polynomial P' which evaluates to the same values as P for all input combinations which do not satisfy the don't care conditions. The computation of optimized polynomials is reduced to suitable *Integer Linear Programming* (ILP) problems and makes use of progress in ILP solving made during the last decades. Since we change polynomials only w. r. t. satisfiability don't cares, i. e., w. r. t. value combinations which do not occur in the circuit as long as we stay in the allowed input range, it is clear that the final polynomial after backward rewriting is – within the allowed input range – identical to the polynomial which would have been computed without using don't care optimization.

Our experiments show that this approach to optimize polynomials in early stages (before an exponential blow-up occurs) is very efficient in restricting the sizes of polynomials. E.g. for the 512-bit divider the whole verification including backward rewriting and don't care based optimizations of polynomials could be performed in less than 162 CPU minutes.

Moreover, we make the observation that for a correct divider the final polynomial only has to evaluate to 0 for the inputs in the allowed input range ($0 \leq R^{(0)} < D \cdot 2^{n-1}$). Whereas for the clean divider from [28] the final polynomial evaluates to 0 for all input combinations ("by chance"), this is not the case for the optimized divider designs and thus we had to provide an efficient evaluation proving that the final polynomial is 0 for all inputs in the allowed input range $0 \leq R^{(0)} < D \cdot 2^{n-1}$.

The paper is structured as follows: In Sect. II we provide background on SCA and divider circuits. In Sect. III we motivate the need for our novel optimization method which is presented in Sect. IV. We evaluate the approach in Sect. V and conclude with final remarks in Sect. VI.

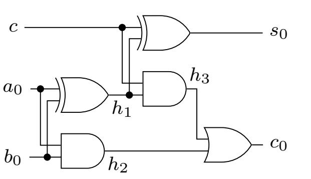
II. Preliminaries

A. SCA for Verification

For the presentation of SCA we basically follow [28]. 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 [6] and truncated multipliers [16] 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 [4] or [13], e.g.). Here we restrict ourselves to exactly this view.

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 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. 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 $2c_0 + s_0$



$$\begin{aligned} & 2c_0 + s_0 \\ \xrightarrow{c_0}\; & 2h_2 + 2h_3 - 2h_2h_3 + s_0 \\ \xrightarrow{h_3}\; & 2h_2 + 2ch_1 - 2ch_1h_2 + s_0 \\ \xrightarrow{s_0}\; & 2h_2 - 2ch_1h_2 + c + h_1 \\ \xrightarrow{h_2}\; & 2a_0b_0 - 2a_0b_0ch_1 + c + h_1 \\ \xrightarrow{h_1}\; & a_0 + b_0 + c \end{aligned}$$

Fig. 1. Circuit with series of substitutions.

(called the "output signature" in [10]) 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 [10]) 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 already follows from the following lemma:

Lemma 1. *Polynomials (with the above mentioned simplifications resp. normalizations) are canonical representations of pseudo-Boolean functions (up to reordering of the terms).*

The proof of Lemma 1 as well as a discussion of the relation between polynomials and *BMDs can be found in [28], e.g..

B. Divider Circuits

In the following we briefly review textbook knowledge on dividers. We use $\langle a_n, \ldots, a_0 \rangle := \sum_{i=0}^{n} a_i 2^i$ 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 let $0 \leq R^{(0)} < D \cdot 2^{n-1}$.*[*] *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 $R^{(0)} = Q \cdot D + R$ (verification condition 1 = "vc1") and $0 \leq R < D$ ("vc2").*

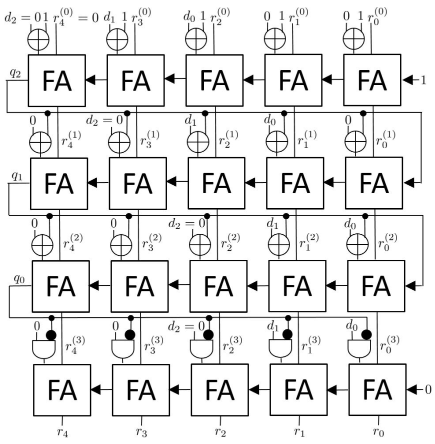
The simplest algorithm to compute quotient and remainder is *restoring division* which is the "school method" to compute quotient bits and "partial remainders" $R^{(j)}$. In each step it subtracts a shifted version of D. If the result is less than 0, the corresponding quotient bit is 0 and the shifted version of D is "added back",

[*]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 $0 \leq R^{(0)} < D \cdot 2^{n-1}$).

Algorithm 1 Non-restoring division.

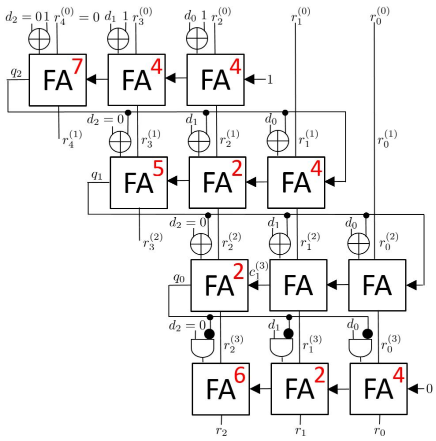
```
1: $R^{(1)} := R^{(0)} - D \cdot 2^{n-1}$; **if** $R^{(1)} < 0$ **then** $q_{n-1} := 0$ **else** $q_{n-1} := 1$;
2: **for** $i = 2$ to $n$ **do**
3:     **if** $R^{(j-1)} \geq 0$ **then** $R^{(j)} := R^{(j-1)} - D \cdot 2^{n-j}$
4:     **else** $R^{(j)} := R^{(j-1)} + D \cdot 2^{n-j}$;
5:     **if** $R^{(j)} < 0$ **then** $q_{n-j} := 0$ **else** $q_{n-j} := 1$;
6: $R := R^{(n)} + (1 - q_0) \cdot D$;
```

i. e., "restored". Otherwise the quotient bit is 1 and the algorithm proceeds with the next smaller shifted version of D. *Non-restoring division* optimizes restoring division by combining 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). More precisely, non-restoring division works according to Alg. 1.



$d_2 = 0$ 1 $r_4^{(0)} = 0$ d_1 1 $r_3^{(0)}$ d_0 1 $r_2^{(0)}$ 0 1 $r_1^{(0)}$ 0 1 $r_0^{(0)}$
FA
q_2
q_1
q_0
$r_4^{(1)}$ $r_3^{(1)}$ $r_2^{(1)}$ $r_1^{(1)}$ $r_0^{(1)}$
$r_4^{(2)}$ $r_3^{(2)}$ $r_2^{(2)}$ $r_1^{(2)}$ $r_0^{(2)}$
$r_4^{(3)}$ $r_3^{(3)}$ $r_2^{(3)}$ $r_1^{(3)}$ $r_0^{(3)}$
r_4 r_3 r_2 r_1 r_0

Fig. 2. Non-restoring divider, $n = 3$.



$d_2 = 0$ 1 $r_4^{(0)} = 0$ d_1 1 $r_3^{(0)}$ d_0 1 $r_2^{(0)}$ $r_1^{(0)}$ $r_0^{(0)}$
FA 7
FA 4
FA 4
FA 5
FA 2
FA 4
FA 2
FA
FA
FA 6
FA 2
FA 4
$c^{(3)}$
q_2
q_1
q_0
r_2 r_1 r_0

Fig. 3. Optimized non-restoring divider, $n = 3$. For simplicity, we present the circuit *before* propagation of constants (which is done however in the real implemented circuit). The red numbers in the upper right corners of full adders give the number of satisfiability don't cares resulting from the input constraint (for the circuit *before* constant propagation).

Fig. 2 shows a combinational circuit for $n = 3$ computing Q and R by non-restoring division as it was used in [28]. It works with two's complement numbers, the first row implements a subtractor, row j with $2 \leq j \leq n$ represents a combined adder/subtractor (CAS), depending on the quotient bit q_{n-j+1}, row $n + 1$ represents the backaddition in case $q_0 = 0$. The divider works with partial remainders $R^{(j)}$ of fixed lengths $2n - 1$, i. e., the width of a row is $2n - 1$. However, well-known textbook implementations make it with rows of width n only, leading to a non-restoring divider as shown in Fig. 3 for $n = 3$. This can be seen as follows: Due to shifting of D by $n - j$ bit positions to the left in rows 1 to n, the right-most $n - j$ full adders can be omitted in row j. The fact that the $j - 1$ left-most full adders may be omitted as well needs some deeper insight into non-restoring division and is due to range restrictions derived from the condition on the input range. We obtain the following result for the size of the partial remainders:

Lemma 2. *For all partial remainders in non-restoring division with* $0 \leq R^{(0)} < D{\cdot}2^{n-1}$ *we obtain* $-D{\cdot}2^{n-j} \leq R^{(j)} < D{\cdot}2^{n-j}$.

Proof. By induction using $0 \leq R^{(0)} < D \cdot 2^{n-1}$. □

Corollary 1. *For* $1 \leq j \leq n$ $R^{(j)}$ *may be represented by a two's complement representation with* $2n - j$ *bits.*

Proof. With $-D \cdot 2^{n-j} \leq R^{(j)} < D \cdot 2^{n-j}$ and $D \leq 2^{n-1} - 1 \leq 2^{n-1}$ we obtain $-2^{2n-j-1} \leq R^{(j)} < 2^{2n-j-1}$. □

From Corollary 1 we can conclude that $R^{(j)}$ can be represented by $(r^{(j)}_{2n-j-1} \dots r^{(j)}_0)$ and a circuit for non-restoring division does not need to compute any bits with higher indices than $r^{(j)}_{2n-j-1}$. However, in the adder/subtractor stage computing $R^{(j+1)}$, the leading bit $r^{(j)}_{2n-j-1}$ of $R^{(j)}$ does not need to be used as an input, since we know that the result $R^{(j+1)}$ can be represented by one bit less, i.e., $r^{(j+1)}_{2n-j-1}$ resulting from addition at bit position $2n-j-1$ would just be a sign extension of $(r^{(j+1)}_{2n-j-2} \dots r^{(j+1)}_0)$. A further analysis shows that for the allowed input range the carry output of the full adder computing $r^{(j)}_{2n-j-1}$ as a sum output is exactly equal to the negated sign bit $r^{(j)}_{2n-j-1}$ (which is in turn equal to the quotient bit q_{n-j}). Thus, $r^{(j)}_{2n-j-1}$ does not need to be computed at all. Altogether, the considerations sketched above lead to an optimized non-restoring divider implementation shown in Fig. 3.

III. Analyzing SCA for Divider Verification

With only a high level view on the algorithm for non-restoring division one could assume that backward rewriting starting with the specification polynomial $Q \times D + R - R^{(0)}$ would lead to small polynomials – at least if the backward rewriting would always hit the boundaries between the stages of the divider and there would not be any significant peaks in polynomial sizes in between: With $R = R^{(n)} + (1 - q_0) \cdot D$ the polynomial obtained after processing stage $n{+}1$ would be $(\sum_{i=1}^{n-1} q_i 2^i + 2^0){\cdot}D + R^{(n)} - R^{(0)}$. For $j = 2$ to n the algorithm implies $R^{(j)} = R^{(j-1)} - q_{n-j+1}(D \cdot 2^{n-j}) + (1 - q_{n-j+1})(D \cdot 2^{n-j}) = R^{(j-1)} + (1 - 2q_{n-j+1})(D \cdot 2^{n-j})$ and thus the polynomial after processing stage j with $j = n$ to 2 would be (by induction) equal to $(\sum_{i=n-j+2}^{n-1} q_i 2^i + 2^{n-j+1}) \cdot D + R^{(j-1)} - R^{(0)}$. Finally, after processing stage 1 using the equation $R^{(1)} = R^{(0)} - D \cdot 2^{n-1}$, the polynomial would reduce to 0.

It has been observed already in [28] that even for the unoptimized implementation from Fig. 2 the polynomials represented at the cuts between stages are different from this high-level derivation. The reason lies in the fact that the stages do not really implement signed addition / subtraction. In general, signed addition / subtraction of two $(2n - 1)$-bit numbers leads to a $2n$-bit number. The leading bit of the result can only be omitted, if "no overflow occurs". Since this fact cannot be seen by backward reasoning, backward rewriting leads to exponential intermediate polynomials. As a main effect of SAT-based information forwarding (SBIF), [28] could derive from the input constraint $0 \leq R^{(0)} < D \cdot 2^{n-1}$ that in each stage from 1 to n the most significant bits are antivalent, and thus the stages really implement addition / subtraction under this constraint. Using this information as early as possible in backward rewriting avoids blow-ups in the sizes of polynomials.

In the optimized implementation from Fig. 3 the situation is completely different. The most significant bits in stages 1 to n are neither equivalent nor antivalent. Thus, we cannot expect that SBIF with exploiting equivalent / antivalent signals helps much for the design from Fig. 3. In fact, we did not only experimentally observe memory blow-ups during verification of non-restoring dividers as in Fig. 3 with the method from [28], but we were able to *prove* that the canonical polynomials for pseudo-boolean functions occurring

Algorithm 2 Optimized backward rewriting.

```
Input: Specification polynomial $SP^{init}$, Input constraint $IC$, Circuit $CUV$ with atomic
    blocks $a_1 \prec_{top} \dots \prec_{top} a_m$ in topological order $\prec_{top}$ and signals $s_1, \dots, s_n$
Output: 1 iff specification holds for all inputs satisfying $IC$
 1: $SP_m := SP^{init}$; $oldsize :=$ size($SP_m$); $i := m$; $ST := \emptyset$;
 2: $(dc(a_1), \dots, dc(a_m))$ := Compute_DC($CUV$, $IC$);
 3: $(rp(s_1), \dots, rp(s_n))$ := SBIF($CUV$, $IC$, $(dc(a_1), \dots, dc(a_m))$);
 4: while $i > 0$ do
 5:     $SP_{i-1}$ := Rewrite($SP_i$, $a_i$);
 6:     if size($SP_{i-1}$) $> threshold \cdot oldsize$ and $ST \neq \emptyset$ then
 7:         ($SP$, $j$, $type$) = pop($ST$);
 8:         $i := j$; $SP_{i-1} := SP$;
 9:         if $type =$ dc then $SP_{i-1}$ := Opt_DC($SP_{i-1}$, $dc(a_i)$);
10:         if $type =$ eq then $\forall s \in in(a_i)$: $SP_{i-1}$ := Replace($SP_{i-1}$, $s$, $rp(s)$);
11:     else
12:         if $dc(a_i) \neq \emptyset$ then push($ST$, ($SP_{i-1}$, $i$, dc)); $oldsize$ := size($SP_{i-1}$);
13:         if $\exists s \in in(a_i)$ with $rp(s) \neq s$ then
14:             push($ST$, ($SP_{i-1}$, $i$, eq)); $oldsize$ := size($SP_{i-1}$);
15:     $i := i - 1$;
16: return evaluate($SP_0$);
```

Algorithm 3 evaluate(SP_0).

```
1: for $i = n - 2$ to 0 do
2:     if $SP_0|_{d_i=1, r^{(0)}_{i+n-1}=0} \neq 0$ then return 0;        ▷ divider incorrect
3:     $SP_0 := SP_0|_{r^{(0)}_{i+n-1}=d_i}$;
4: return 1;                                                            ▷ divider correct
```

at cuts between stages in Fig. 3 have exponential sizes. (The proof is omitted due to lack of space.) Thus, the memory explosion we observed in our implementation of [28] is not due to unfavorable heuristic decisions, but cannot be avoided – if the backward rewriting from [28] is not enhanced by a stronger propagation and exploitation of constraints in the opposite direction from inputs to outputs.

IV. SCA with don't care optimization

In this section we present our algorithm which uses don't care optimization of polynomials as an essential ingredient. The algorithm starts with a gate level netlist and detects atomic blocks [15], resulting in a circuit with non-trivial atomic blocks (like full adders, half adders etc.) and trivial atomic blocks (original gates not included in non-trivial atomic blocks). We compute a topological order $\prec_{top}$ on the atomic blocks with heuristics from [14], [15]. Assume m atomic blocks with $a_1 \prec_{top} \dots \prec_{top} a_m$. The remaining algorithm is given in Alg. 2.

For divider verification the algorithm is started with the specification polynomial SP^{init} representing $Q \times D + R - R^{(0)}$ and the input constraint $IC = 0 \leq R^{(0)} < D \cdot 2^{n-1}$.

a) Computation of satisfiability don't cares: In a first step we compute satisfiability don't cares at the inputs of atomic blocks (like full adders) that result from the input constraint IC (Line 2). We start by using simulation (taking IC into account) to detect candidates for satisfiability don't cares. Whereas in principle we could prove those candidates to be really satisfiability don't cares using SAT, preliminary experiments showed that for large dividers it was not possible to confirm a sufficient number of don't care candidates by SAT due to lack of resources. SAT-solving restricted to windows of moderate size was not successful either, since it seems that for the non-restoring divider designs existing don't cares cannot be confirmed by local reasoning. However, it turned out in our experiments (see Sect. V) that a series of BDD-based image computations [29] was able to derive all satisfiability don't cares at the inputs of atomic blocks. We start with a BDD for representing input constraint IC. Then we identify the first atomic block a_i in the topological order which has a non-empty set of don't care candidates (computed by simulation). Atomic blocks $a_1, \dots, a_{i-1}$ form the first slice sl_1 in the circuit. The output signals of sl_1 are exactly the signals connecting sl_1 with atomic blocks $a_i, \dots, a_m$, i. e., by construction, the inputs of a_i are outputs of sl_1. We use BDD-based image computation to compute the BDD for the image IMG_1 of IC under sl_1. Then we evaluate IMG_1 wrt. to the don't care candidates at the inputs of a_i. If the evaluation results in constant 0 for some candidate, then it is not included in the image and it is confirmed as a don't care. Then, we consider the next atomic block a_j with a non-empty set of don't care candidates, choose $a_i, \dots, a_{j-1}$ to form the next slice, compute the image IMG_2 of IMG_1 under this slice etc.. In the end we have classified all don't care candidates to be real don't cares or not. As an example see Fig. 3 showing (in red) the numbers of don't cares computed for the full adders in the optimized 3-bit divider before constant propagation.

During our BDD-based computations we just use a static variable order corresponding to the initial order in [28]. The variable order is chosen such that the bits r_i and d_i as well as $r^{(0)}_{i+n-1}$ and d_i are arranged side by side and bits with higher indices are evaluated first. Other variables corresponding to internal signals are ordered according to [30], extended to the case that the relative order of the previously mentioned variables has already been fixed.

Note that, as a side effect, those image computations can also be used to check whether verification condition (vc2) from Def. 1 holds. As a last step we perform an image computation with the remaining atomic blocks and obtain the complete image IMG_f of IC under the whole circuit under verification. Now we just have to check whether IMG_f implies $0 \leq R < D$.[†]

b) Optimization: During backward rewriting we are using two optimization methods: (1) Exploitation of equivalent / antivalent signals from [28] and (2) our novel don't care optimization for polynomials. For (1) we compute classes of equivalent / antivalent signals (Line 3). The signals s in each class are represented by the unique signal $rp(s)$ that is minimal w. r. t. $\prec_{top}$. It is important to note that for the divider design from Fig. 3 the original SBIF version with simulation and restricted window-based SAT-solving does not work, since it seems that existing antivalences / equivalences cannot be proven by local reasoning. For this reason, it is essential to enhance the SAT-problems with the information that signal combinations corresponding to satisfiability don't cares (as computed in Line 2) cannot occur.

Our experiments in Sect. V show that it is essential to use our novel don't care optimization method for polynomials $P(x_1, \dots, x_n)$ which represent pseudo-boolean functions $f(x_1, \dots, x_n) : \{0,1\}^n \to \mathbb{Z}$. If certain valuations for the inputs of a polynomial P cannot occur, since they are satisfiability don't cares, then we can modify the pseudo-boolean function for those valuations. Our goal is to find an assignment to the don't cares that minimizes the size of $P(x_1, \dots, x_n)$. For a polynomial $P(x_1, \dots, x_n)$ with don't care cubes $dc_1, \dots, dc_n$ the method consists in the following steps:

- Introduce a new *integer* variable v_i for each don't care cube dc_i.
- Add for all $1 \leq i \leq n$ "$v_i \cdot dc_i$" to P.
- Multiply out, combine terms with the same monomial etc..
- Use Integer Linear Programming to minimize the size of P.

Example 1. *As an example we choose $P(x_1, x_2, x_3) = 1 - x_1 - x_2 - x_3 + 2x_1x_2 + 2x_1x_3 + 2x_2x_3 - 4x_1x_2x_3$ with don't care cubes $\neg x_1x_2x_3$ and $x_1\neg x_2\neg x_3$. For $\neg x_1x_2x_3$ we choose the integer*

[†]This is in contrast to [28] where the verification of (vc2) starts with $0 \leq R < D$ and performs backward substitutions.

variable v_1, for $x_1 \neg x_2 \neg x_3$ we choose v_2. Since $v_1(1-x_1)x_2x_3$ and $v_2x_1(1-x_2)(1-x_3)$ reduce to 0 for all care vectors, we can add them to $P(x_1, x_2, x_3)$ without changing the polynomial within the care set. By multiplying out and combining terms with the same monomial we arrive at the polynomial $1 + (v_2 - 1)x_1 - x_2 - x_3 + (2 - v_2)x_1x_2 + (2 - v_2)x_1x_3 + (2 + v_1)x_2x_3 + (v_2 - v_1 - 4)x_1x_2x_3$. Minimizing the number of terms in the polynomial means choosing integer values for v_1 and v_2 such that the constant of a maximum number of terms is 0 (thus eliminating a maximum number of terms). Thus, in the equation system in Fig. 4 we try to satisfy a maximum number of equations. It is easy to see that the optimal solution is $v_1 = -2$, $v_2 = 2$, leading to the polynomial $1 + x_1 - x_2 - x_3$.

$$\begin{array}{rcl} v_2 - 1 & = & 0 \\ 2 - v_2 & = & 0 \\ 2 - v_2 & = & 0 \\ 2 + v_1 & = & 0 \\ v_2 - v_1 - 4 & = & 0 \end{array}$$

Fig. 4. Minimization

Satisfying a maximum number of linear integer equations can be reduced to integer linear programming by standard methods (replacing each equation $\ell_i(x_1, \dots, x_n) = 0$ by $\ell_i(x_1, \dots, x_n) \le Md_i$ and $-\ell_i(x_1, \dots, x_n) \le Md_i$ with a sufficiently large constant M and a binary deactivation variable d_i for each equation, then minimizing the number of deactivation variables assigned to 1).

In Lines 4 to 15 of Alg. 2 we perform backward rewriting processing the atomic blocks in reverse topological order. We apply our optimization methods only if needed, i. e., if we observe a significant growth in size of the polynomial SP_{i-1}. Whenever we arrive at an atomic block with a non-empty don't care set $dc(a_i)$ or at an atomic block whose set of inputs $in(a_i)$ contains a signal with a topologically smaller equivalent / antivalent signal (indicated by $rp(s) \neq s$ in Line 13) we save this situation as a backtrack point on a stack ST. If the polynomial SP_{i-1} grows too much (Line 6, we use a growth factor *threshold* $= 2.0$ in our implementation), we backtrack to the last backtrack point and perform an optimization of the saved polynomial – either by applying don't care optimization of the polynomial (Line 9) or by replacing input variables of the replaced atomic block by their topologically minimal equivalent / antivalent representative signals (of course in the right polarity, Line 10).

c) Evaluating the final polynomials: We observed that simple and non-optimized SCA-based methods applied to optimized non-restoring dividers as in Fig. 3 finally result in a non-zero polynomial (for small bit widths where the verification finishes). Whereas this is in contrast to the typical assumption in SCA-based verification ("the implementation is correct iff backward rewriting reduces the specification polynomial to 0"), it does not indicate an error. The divider has only to be correct for input combinations satisfying the input constraint $IC = 0 \le R^{(0)} < D \cdot 2^{n-1}$, i. e., the divider is correct iff the final polynomial evaluates to 0 for all inputs satisfying $IC = 0 \le R^{(0)} < D \cdot 2^{n-1}$. Of course, the number of inputs satisfying the input constraint is exponential and thus it is not advisable to evaluate the polynomial for all admissible input combinations. Fortunately, in the special case of a divider the input constraint IC has a form that allows a decomposition into a linear number of cases to be evaluated, following the idea of bitwise comparing $R^{(0)}$ and $D \cdot 2^{n-1}$ starting with the most significant bit (see Alg. 3).

V. Experimental Results

Our experimental evaluation has been performed on 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 24 CPU hours. To solve the ILP problems for don't care optimization of polynomials we used the ILP solver Gurobi [31]. For image computations we used the BDD package CUDD 3.0.0 [32]. The run times in Table I are given in CPU seconds.

We consider the verification of optimized non-restoring dividers as in Fig. 3 with different bit widths (Col. 1 in Table I). During verification we did not use any hierarchy information. We just used the flat gate-level netlist (numbers of gates are shown in Col. 2) and employed heuristics for detecting atomic blocks as well as for finding a good substitution order [14], [15].

We start with three experiments for comparison. In those 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 $0 \le R^{(0)} < D \cdot 2^{n-1}$ to ensure that counterexamples are restricted to the allowed range of inputs.

In the first experiment we used a SAT-solver (MiniSat 2.2.0 [33]) to solve the corresponding satisfiability problems. In Table I the results are presented in Col. 3. The results show that 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 time-out. In the second experiment we considered the combinational equivalence checking (CEC) approach of ABC [34], [35] which is based on And-Inverter-Graph (AIG) rewriting via structural hashing, simulation, and SAT and reduces the overall complexity of equivalence checking between two designs by finding equivalent internal AIG nodes. As for SAT-solving, ABC cannot verify the dividers with bit widths larger than 8, see Col. 4 in Table I. In a third experiment we considered the commercial verification tool from OneSpin [36]. As Col. 5 in Table I shows, the OneSpin tool performs better on the non-restoring dividers, but it needs already more than 1 CPU hour for the verification of 24-bit-dividers and runs into a timeout for larger dividers.

Col. 6 of Table I shows that the SCA-based method using SBIF from [28] is not suitable for verifying large optimized non-restoring dividers either. The method exceeds the available memory for all dividers with bit widths larger than 8 bits.

In contrast, from Col. 7 we can see that dividers up to 512 bits do not form any problem for our method. The verification of 512-bit dividers needs less than 162 CPU minutes. Remember that (as in [28]) we do not need any golden specification circuits for the verification, since we prove that the divider circuits fulfill their abstract specification from Def. 1. We split the overall run time into different sub-tasks for a more detailed analysis. Col. 8 shows the run times for reading the circuit design, Col. 9 the run times for simulations that produce candidates for satisfiability don't cares. In Col. 10 the run times for BDD-based image computation confirming don't care candidates are given. Col. 11 shows the corresponding peak sizes for the number of BDD nodes. Col. 12 gives the run times for computing equivalent / antivalent signals (SBIF, [28]) with a window size of 2 for the SAT-problems, making use of the computed don't cares as well. Finally, Col. 13 gives the run times for SCA-based backward rewriting, Col. 14 the run times for don't care optimization of polynomials, Col. 15 the number of backtrackings our algorithm performs, Col. 16 the peak sizes of polynomials, and Col. 17 gives the size of the final polynomial after rewriting. In almost all cases our optimization was strong enough to minimize the final polynomial to 0 and thus the run times for evaluating the final polynomial were negligible. Verifying condition (vc2) from Def. 1 has been integrated into the BDD-based image computation and is thus included in Col. 10. The results clearly show that don't care optimization of polynomials is fast and effective. By using it *before* the polynomial sizes have

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

						Our method										
n	#Gates	SAT time	ABC time	Commercial time	[28] time	overall time	read time	sim time	BDD time	BDD nodes	SBIF time	rewrite time	DCOP time	#bt	peak poly.	final poly.
3	54	0.27	<0.01	2.44	0.08	0.10	0.05	0.02	<0.01	216	<0.01	<0.01	0.02	4	74	6
4	100	0.82	0.01	2.56	0.16	0.16	0.09	0.02	<0.01	366	<0.01	<0.01	0.03	7	79	0
8	404	47.17	17.60	2.65	904.30	0.48	0.37	0.05	0.01	1,228	<0.01	0.02	0.04	11	272	0
16	1,588	TO	TO	165.89	MO	1.91	1.59	0.13	0.03	3,617	<0.01	0.08	0.07	19	859	0
24	3,540	TO	TO	4,319.20	MO	3.82	3.12	0.22	0.11	6,881	<0.01	0.25	0.11	27	2,046	0
32	6,260	TO	TO	TO	MO	6.79	5.62	0.29	0.22	11,041	0.01	0.51	0.14	35	2,864	0
48	14,004	TO	TO	TO	MO	14.19	11.07	0.53	0.75	22,049	0.02	1.54	0.25	51	7,509	0
64	24,820	TO	TO	TO	MO	28.86	21.51	0.87	1.92	36,641	0.03	4.11	0.40	67	9,706	0
96	55,668	TO	TO	TO	MO	70.22	44.40	1.85	7.19	76,577	0.09	15.80	0.85	99	24,724	0
128	98,804	TO	TO	TO	MO	148.18	78.38	3.02	20.25	130,849	0.16	44.52	1.78	131	54,920	0
256	394,228	TO	TO	TO	MO	989.91	335.40	8.62	294.84	491,297	0.68	343.91	6.49	259	203,838	0
512	1,574,900	TO	TO	TO	MO	9,668.70	1,258.88	26.21	5,595.47	1,900,321	2.78	2,762.31	23.02	515	676,521	0
1024	6,295,540	TO	TO	TO	MO	TO	-	-	-	-	-	-	-	-	-	-

increased to a large extent we are able to successfully verify dividers up to 512 bits with small peak sizes of polynomials.

To check whether don't care optimization of polynomials could also be replaced by don't care optimization at the bit level, we considered the following variant of our algorithm: We forbid to use exactly the don't cares (whose number is two for $n \geq 4$) that occur at the inputs of one atomic block, the block computing output q_1. Instead, we replace this block by all possible circuit functions that result from the 4 possible assignments of don't cares in its function table (the resulting variants are called $v1$, $v2$, $v3$, and $v4$ in Fig. 5, $v1$ is the version which does not change the atomic block). Fig. 5 shows that for the 32-bit divider, e.g., all four variants lead to an exponential growth in polynomial sizes, no matter how the don't care assignment at the bit level is chosen. (The curves for $v1$ to $v4$ are not exactly equal, but unfortunately they overlap at the given resolution.) In contrast, for our original method the growth behaviour of the polynomials is moderate. (The observed small peaks always occur before backtracking to an optimization step.) This result clearly shows that optimization of polynomials at the *word level* is essential and cannot be replaced by bit-level don't care optimization.

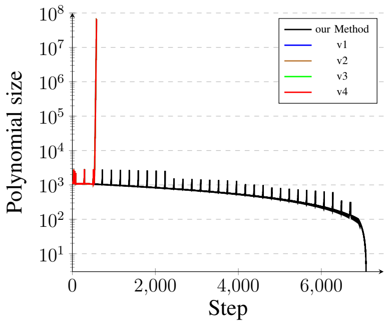


Fig. 5. Growth behaviour of polynomials for 32-bit divider.

VI. CONCLUSIONS AND FUTURE WORK

We presented a novel ILP-based method for optimizing polynomials w. r. t. satisfiability don't cares. Using this method has been essential to successfully verify large optimized non-restoring dividers. For the verification we had to take into account that usually divider designs are only correct under given input constraints. This observation typically holds for other designs as well and (to the best of our knowledge) has not been used in SCA-based verification approaches so far. We strongly believe that the combination of forward information propagation (leading to the computation of satisfiability don't cares) and backward computation of polynomials that will be optimized using this information is not only applicable to non-restoring dividers, but will be the key to move forward the verification of other divider architectures as well as other arithmetic circuits.

REFERENCES

[1] T. Coe, "Inside the Pentium FDIV bug," *Dr. Dobbs J.*, vol. 20, no. 4, pp. 129–135, 1995.
[2] R. E. Bryant, "Graph-based algorithms for Boolean function manipulation," *TC*, vol. 35, no. 8, pp. 677–691, 1986.
[3] J. R. Burch, "Using BDDs to verify multipliers," in *DAC*, 1991, pp. 408–412.
[4] 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 *DATE*, 2016, pp. 1048–1053.
[5] M. Temel, A. Slobodová, and W. A. Hunt, "Automated and scalable verification of integer multipliers," in *CAV*, 2020, pp. 485–507.
[6] J. Lv, P. Kalla, and F. Enescu, "Efficient Gröbner basis reductions for formal verification of Galois field arithmetic circuits," *TCAD*, vol. 32, no. 9, pp. 1409–1420, 2013.
[7] O. Wienand, M. Wedler, D. Stoffel, W. Kunz, and G. Greuel, "An algebraic approach for proving data correctness in arithmetic data paths," in *CAV*, 2008, pp. 473–486.
[8] F. Farahmandi and B. Alizadeh, "Gröbner basis based formal verification of large arithmetic circuits using gaussian elimination and cone-based polynomial extraction," *MICPRO*, vol. 39, no. 2, pp. 83–96, 2015.
[9] M. Ciesielski, C. Yu, D. Liu, and W. Brown, "Verification of gate-level arithmetic circuits by function extraction," in *DAC*, 2015, pp. 52:1–52:6.
[10] C. Yu, W. Brown, D. Liu, A. Rossi, and M. Ciesielski, "Formal verification of arithmetic circuits by function extraction," *TCAD*, vol. 35, no. 12, pp. 2131–2142, 2016.
[11] D. Ritirc, A. Biere, and M. Kauers, "Column-wise verification of multipliers using computer algebra," in *FMCAD*, 2017, pp. 23–30.
[12] C. Yu, M. Ciesielski, and A. Mishchenko, "Fast algebraic rewriting based on And-Inverter graphs," *TCAD*, vol. 37, no. 9, pp. 1907–1911, 2017.
[13] D. Ritirc, A. Biere, and M. Kauers, "Improving and extending the algebraic approach for verifying gate-level multipliers," in *DATE*, 2018, pp. 1556–1561.
[14] A. Mahzoon, D. Große, and R. Drechsler, "PolyCleaner: clean your polynomials before backward rewriting to verify million-gate multipliers," in *ICCAD*, 2018, pp. 129:1–129:8.
[15] ——, "RevSCA: Using reverse engineering to bring light into backward rewriting for big and dirty multipliers," in *DAC*, 2019, pp. 185:1–185:6.
[16] D. Kaufmann, A. Biere, and M. Kauers, "Verifying large multipliers by combining SAT and computer algebra," in *FMCAD*, 2019, pp. 28–36.
[17] A. Mahzoon, D. Große, C. Scholl, and R. Drechsler, "Towards formal verification of optimized and industrial multipliers," in *DATE*, 2020, pp. 544–549.
[18] J. E. Robertson, "A new class of digital division methods," *IRE Trans. Electronic Computers*, vol. 7, no. 3, pp. 218–222, 1958.
[19] R. E. Bryant, "Bit-level analysis of an SRT divider circuit," in *DAC*, 1996, pp. 661–665.
[20] E. M. Clarke, M. Khaira, and X. Zhao, "Word level model checking - avoiding the Pentium FDIV error," in *DAC*, 1996, pp. 645–648.
[21] 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.
[22] E. M. Clarke, S. M. German, and X. Zhao, "Verifying the SRT division algorithm using theorem proving techniques," *Form Methods Syst. Des.*, vol. 14, no. 1, pp. 7–44, 1999.
[23] 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.
[24] K. Hamaguchi, A. Morita, and S. Yajima, "Efficient construction of binary moment diagrams for verifying arithmetic circuits," in *ICCAD*, 1995, pp. 78–82.
[25] R. E. Bryant and Y. A. Chen, "Verification of arithmetic circuits with binary moment diagrams," in *DAC*, 1995, pp. 535–541.
[26] A. Yasin, T. Su, S. Pillement, and M. J. Ciesielski, "Formal verification of integer dividers: Division by a constant," in *ISVLSI*, 2019, pp. 76–81.
[27] ——, "Functional verification of hardware dividers using algebraic model," in *VLSI-SoC*, 2019, pp. 257–262.
[28] C. Scholl and A. Konrad, "Symbolic computer algebra and sat based information forwarding for fully automatic divider verification," in *DAC*, 2020.
[29] O. Coudert and J. C. Madre, "A unified framework for the formal verification of sequential circuits," in *ICCAD*, 1990, pp. 126–129.
[30] 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 *ICCAD*, 1988, pp. 6–9.
[31] Gurobi Optimization, LLC, "Gurobi optimizer reference manual," 2020. [Online]. Available: http://www.gurobi.com
[32] F. Somenzi, "Efficient manipulation of decision diagrams," *STTT*, vol. 3, no. 2, pp. 171–181, 2001.
[33] N. Eén and N. Sörensson, "An extensible SAT-solver," in *SAT*, 2003, pp. 502–518.
[34] R. K. Brayton and A. Mishchenko, "ABC: an academic industrial-strength verification tool," in *CAV*, 2010, pp. 24–40.
[35] "ABC: A system for sequential synthesis and verification," available at https://people.eecs.berkeley.edu/~alanmi/abc/, 2019.
[36] OneSpin Solutions GmbH, 2020. [Online]. Available: https://www.onespin.com