Formal Verification of Integer Dividers: Division by a Constant

Atif Yasin, Tiankai Su, Sébastien Pillement, Maciej Ciesielski

***Abstract*—Division is one of the most complex and hard to verify arithmetic operations. While verification of major arithmetic operators, such as adders and multipliers, has significantly progressed in recent years, less attention has been devoted to formal verification of dividers. A type of divider that is often used in embedded systems is divide by a constant. This paper presents a formal verification method for different divide-by-constant architectures and the generic restoring dividers based on computer algebra approach. Our experiments for different divider architectures and comparison with exhaustive simulation demonstrates the effectiveness and scalability of the method.**

I. Introduction

A Considerable progress has been made in recent years in verification of arithmetic circuits, such as multipliers, fused multiply-adders, multiply-accumulate, and other components of arithmetic datapaths, both in integer and finite field domain [1][2]. However, the verification of hardware dividers has received limited attention, with a notable exception for theorem provers and other inductive, non-automated systems. Division is one of the most complex arithmetic operators to implement and requires careful hardware implementation and verification [3][4]. The difficulty of formally verifying a hardware implementation of dividers can be attributed to the iterative nature of the division algorithm and the remainder computation.

An operation that comes up frequently in digital systems is a division of an integer by a constant. For example, such an operation is required in computer simulations which use Jacobi stencil algorithm to computes an average of three numbers; in arithmetic for base conversions, number theoretic codes, and graphics codes; in signal processing for computing the sample mean, the sample variance, or the automatic gain control. Finally divide-by-constant is useful for memory bank multiplexing which requires division by small integers, or to support compilers optimization to generate integer divisions to compute loop counts and subtract pointers [5]. The frequent appearance of such operation in many applications justifies creating a specialized operator in embedded systems design, referred to as "divider by a constant" [6]. In this paper, we concentrate on the verification of a division by a constant and conclude it by presenting a preliminary verification analysis of a restoring generic division algorithm.

While division by a constant 2^k can be efficiently implemented by shifters, division by other constants is more complex. Many algorithms for division by a constant use table-based approach and implement it using look-up tables (LUT). A notable example of such an implementation is a table-based SRT division implemented in an Intel Pentium Processor. The infamous *Pentium bug* in its floating point division (FDIV) instruction has galvanized the verification efforts [7][8] for divider circuits.

This paper describes the verification technique for a divide-by-constant circuit, generalized to the verification of a restoring generic divider. Our work is based on an algebraic rewriting model, which performs arithmetic *function extraction*, originally proposed and successfully applied to the verification of integer and Galois Field multipliers [9] [10]. The method has been suitably modified for dividers by identifying and taking advantage of the "vanishing monomials", which are an intrinsic property of table-based divide-by-constant architecture.

The rest of the paper is organized as follows. Section II provides the necessary background and literature survey for the implementation and verification of the divide-by-constant architecture. Section III shows the detailed verification methodology while Section IV presents the verification results and their analysis. We also compare our approach to an exhaustive simulation of the respective circuits. Preliminary results for a restoring generic divider are also presented, showing the applicability of our technique to a more generic case. Finally, Section V concludes the paper and briefly discusses the future work.

II. Background

A. Canonical Diagrams

Several techniques have been applied to formal verification of arithmetic circuits, such as adders, comparators, and multipliers. These techniques address functional verification and equivalence checking using various canonical diagrams, such as BDDs [11], BMDs [12], or TEDs [13]. BMDs have been useful in proving floating point multipliers, but the literature is rather scarce on divider circuits. A notable exception in this domain is the work of Bryant [8]. Although effective and being able to catch the Pentium bug, it requires generating a checker circuit, which itself needs to be proved. However, no reliable means were offered for the verification of the checker circuit itself.

The technique that received most attention in industry in arithmetic circuit verification is Theorem Proving. In this domain, a circuit is characterized by a set of rules, which are used to make complex formulas to represent the circuit [14][15][16]. However, generating these predefined set of rules for a particular circuit requires a significant human effort and cannot be easily automated.

B. Computer Algebra Approach

The most advanced approach to formal verification of arithmetic circuits is based on computer algebra. In this approach, the arithmetic function specification and its implementation are represented as polynomial rings in a given field; the verification problem is then posed as checking if the implementation satisfies the specification. This is achieved by reducing the specification polynomial modulo the implementation polynomials, given as canonical Groebner basis, known as *ideal membership testing* [17]. The method has been successfully adapted for the verification of Integer and Galois field multipliers [18][19][2].

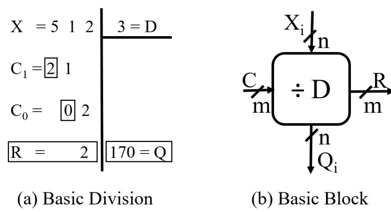
An alternative approach to arithmetic verification of gate-level circuits has been proposed in [1], using algebraic rewriting of the specification polynomial. In this approach, the polynomial representing the encoding of the primary outputs (the *output signature*) is transformed into a polynomial expressed in terms of the primary inputs (called the *input signature*) using algebraic models of circuit elements, such as logic gates. This method extracts an arithmetic function implemented by the circuit and hence is termed as *function extraction*. The method has been successfully applied to multipliers and complex adders [1][18] but not to divider circuits, mostly because of the difficulty of modeling the divider's specification. A thorough review of the state-of-the-art computer algebra methods for multiplier circuit verification can be found in [20], which also proves completeness and soundness of the algebraic technique of [1].

The work of [21] addresses the verification of array dividers by extracting a high-level arithmetic model from the low-level circuit implementation. The technique applies column-based XOR extraction, which relies on a regular structure of the adder trees and the presence of sum generation and carry propagation components. A lack of those components at the right places indicates a potential

bug. The limitation of the method is the requirement that adders be represented with XOR gates, which may not be the case in a synthesized circuit.

C. Divider Circuit Implementation

There are two main approaches to implementing arithmetic division: 1) division by addition/subtraction, such as SRT, restoring, and array dividers; and 2) division by reciprocation, or multiplication by the inverse via Newton-Raphson or Goldsmith algorithm [22]. A wide majority of practical division algorithms, such as SRT, resort to a look-up table (LUT) based implementation, a table-based combinational logic technique studied in [6][23][4]. These algorithms use a reference table, precomputed for a particular value of the divisor, implemented as a LUT. Such an implementation is particularly well-suited for the division by a constant. The dividend X is divided by the divisor D to produce quotient Q and the remainder R, which provides an input carry for the next block.



(a) Basic Division (b) Basic Block

Fig. 1: High school division operation and the basic divider block.

The author of [4] prove that this computation is still valid for any arbitrary radix of X. Thus the division can be implemented as a single block handling n bits, or n blocks handling one bit each, or any intermediate values. Another divider architecture analyzed in this paper is based on restoring algorithm, discussed in Section IV.

III. Verification

Our verification scheme is based on the functional extraction method of [1] [24], reviewed briefly in the next Section. We first illustrate our method for table-based divider with a single block of the divider, and then show how to verify the whole circuit unrolled by the required number of blocks. We will also discuss the case when the divider is faulty, with bugs injected in the implementation.

A. Function Extraction

Algebraic approach used in this work relies on polynomial representation of the circuit specification and its gate level model. In an arithmetic circuit, such as adder or a multiplier, the input-output relationship is well defined, with the outputs and inputs appearing on the opposite sides of the equation, e.g. $S = A + B$ for an adder, and $Z = A \cdot B$ for a multiplier. The operands $A,\ B$ are given as polynomials in a binary expansion form, e.g., $A = \sum_{i=0}^{n-1} a_i$. Right-hand side of those equations is referred to as the *Input Signature*, denoted Sig_{in}; it provides the *specification* of the arithmetic circuit. Similarly, the binary encoding of the result on the left-hand side of the equation (here S for the adder, or Z for the multiplier) is a polynomial called the *Output Signature*, and denoted Sig_{out}.

The goal of function extraction is to extract a unique bit-level polynomial function from the low-level hardware implementation (typically gate-level), in order to compare it to the expected specification Sig_{in}. This is accomplished by transforming the output signature Sig_{out} using the algebraic expressions of the circuit components (logic gates, etc.) into the input signature [9][10]. This process is referred to as the *backward rewriting*, since it performs rewriting in a reverse-topological order. It uses the following algebraic models for the basic logic gates:

$$\begin{aligned} \neg a &= 1 - a \\ a \wedge b &= a \cdot b \\ a \vee b &= a + b - a \cdot b \\ a \oplus b &= a + b - 2(a \cdot b) \end{aligned} \tag{1}$$

In the above equation and throughout the paper, we will use symbols $\neg$, $\wedge$, $\vee$, and $\oplus$ to indicate Boolean operations of the complement, AND, OR, and XOR, respectively. The signs $-$, $+$, and $\cdot$ will denote *algebraic* operations of subtraction, addition, and multiplication, respectively.

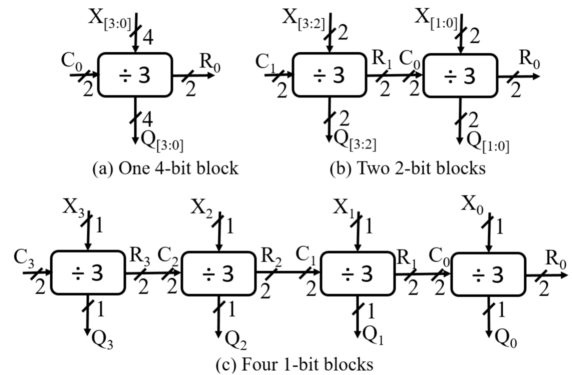
This model of *function extraction* cannot be directly applied to the generic dividers that do not have a close-form input/output relation. Specifically, they are governed by the following I/O relation:

$$X = D \cdot Q + R, \quad \text{with } R < D \tag{2}$$

However, in the case of a constant divider, input divisor D is a known constant making the analytical I/O relationship straightforward. In this context, $Sig_{in} = X$ and $Sig_{out} = D \cdot Q + R$.

B. *Verification of the Divider*

In the iterative, divide-by-constant circuit, the divider is partitioned into a number of blocks, each having the structure shown in Figure 1(b). Figure 2 shows a generic configuration, where multiple blocks can be cascaded together.



(a) One 4-bit block (b) Two 2-bit blocks

(c) Four 1-bit blocks

Fig. 2: Generic divider block for X divided by const. d.

Let N be the number of blocks, and k the size in bits of the dividend X. If k/N is not an integer, the most significant block will have some inputs appended with the respective numbers of zeros. The bit size m of R_i and C_i is the same, and it is determined by the size of the divisor D.

Consider block B_i, shown in Figure 1. In the following, index i refers both to the block position and to the chunk of the respective word, C_i, X_i, Q_i, R_i, associated with the given block. The following summarizes the terms and parameters of the divider block:

- D - divisor (a hardwired constant), $D \neq 0$
- X_i, C_i - dividend and carry-in for block B_i
- Q_i, R_i - quotient and remainder for block B_i
- $n = \lceil k/N \rceil$ - number of bits of X_i and Q_i
- $m = (\lfloor \log_2 (D-1) \rfloor + 1)$ - bit-width of C_i and R_i

C. *Single Block Verification*

To explain the basic idea, consider a single-bit block architecture, $n = 1$, for the division by constant $D = 3$, with $m = 2$.

In the LUT-based division algorithm, each basic block is implemented as a lookup table with entries for all possible inputs, C_i, X_i, and the values of the corresponding outputs Q_i, R_i.

C	X	2C + X 3Q + R	Q	R
0	0	0	0	0
0	1	1	0	1
1	0	2	0	2
1	1	3	1	0
2	0	4	1	1
2	1	5	1	2
3	0	6	-	-
3	1	7	-	-

(a) Function Table

C_1	C_0	X	Q	R_1	R_0
0	0	0	0	0	0
0	0	1	0	0	1
0	1	0	0	1	0
0	1	1	1	0	0
1	0	0	1	0	1
1	0	1	1	1	0
1	1	0	-	-	-
1	1	1	-	-	-

(b) Truth table of LUT

Fig. 3: Divide-by-3 block specification tables

Figure 3 shows the specification tables (i.e., the function table and the LUT truth table) for the basic block of divide-by-3 divider. From the function table, one can derive the word-level input/output relation, shown in Equation 3, where R_i is fed to the next block as C_{i-1}. To verify the functionality of the basic block, we need to prove that the Equation 3 is correct for every input assignment.

$$2C_i + X_i = 3Q_i + R_i \tag{3}$$

The coefficient 2 of C_i comes from the fact that C_i and X_i form one word. In radix 2, the term $C_i = \sum_{k=0}^{n-1} 2^k C_{ik}$, where C_{ik} refers to bit k of block i. The coefficient 3 of Q_i is determined by the value of the divisor, $D = 3$ in our example.

Equation 4 shows the generic bit-level equation for an arbitrary block i. It is derived from Eq. 3 by substituting for a given block i, $C_i = 2C_{i1} + C_{i0}$ and $R_i = 2R_{i1} + R_{i0}$, since $m = 2$.

$$4C_{i1} + 2C_{i0} + X_{i0} = 3Q_{i0} + 2R_{i1} + R_{i0} \tag{4}$$

The left-hand-side of equation (4) is the *Input Signature*, Sig_{in} while the right-hand-side is the *Output Signature*, Sig_{out}, as defined above in Section III-B. In order to prove correctness, we just need to rewrite Sig_{out} using algebraic expressions of the logic-gate implementation and compare the resulting expression with Sig_{in}. If the two expressions are equal, the circuit is proved to be correct.

(a) Correct circuit

(b) Faulty circuit

Fig. 4: Gate level implementation of a single-block, one-bit architecture of a $X/3$ divider. Output signature $Sig_{out} = 3Q_0 + 2R_1 + R_0$; the expected input signature is $Sig_{in} = 4C_1 + 2C_0 + X_0$.

To illustrate the rewriting process, consider the one-block gate-level implementation of the division of X by constant 3, shown in Figure 4. The output signature for this circuit, using the index notation in the figure, is $Sig_{out} = 3Q_0 + 2R_1 + R_0$. Each of the output variables, Q_0, R_1, R_0 are successively replaced by the algebraic expression of their respective gates, as defined by Eq. 1. Each of the internal signals are in turn replaced by the expression of the logic gate they represent, etc., until the final expression contains only the primary input variables. In a functionally correct divider circuit, as in Figure 4(a), the resulting input signature should be

$$Sig_{in} = 4C_1 + 2C_0 + X_0 \tag{5}$$

However, the expression Sig obtained by rewriting Sig_{out} through the logic gates of the correct 1-bit divisor is actually equal to:

$$Sig = (4C_1 + 2C_0 + X_0) - 2C_0C_1X_0 \tag{6}$$

which does not match the expected specification Sig_{in} in Eq.(5).

The reason for this mismatch can be understood by analyzing the truth table in Figure 3. Note that the table contains some entries, namely $\{C_1C_0X\} = 110, 111$, for which the quotient Q_0 cannot be encoded in one bit, with the remainder being strictly less than the divisor; hence those entries are considered *invalid*. In this case the combination $\{C_1C_0\} = 11$ is invalid. This can be translated into a logic constraint, expressed by expression $C_1C_0 = 0$ and used to simplify the resulting signature. Indeed, substituting $C_1C_0 = 0$ into Equation 6 reduces it to the expected Sig_{in}, proving that the circuit in Figure 4(a) is correct.

D. Vanishing Monomials

In our verification work, the monomials that correspond to invalid entries, like C_1C_0 above, are defined as *vanishing monomials*, since in the functionally correct implementation they always evaluate to zero. The vanishing monomials help to remove the redundant terms during the verification process. The vanishing monomials and the corresponding simplifying constraints can readily be derived from the architecture and the value of the divisor, see Figure 3, where invalid input assignments correspond to $don't\text{-}care$ conditions. The following theorem relates vanishing monomials to the input signature computed for the circuit.

Theorem: *The input signature Sig_{in} of the circuit contains vanishing monomials associated with the dc-set of the truth table, regardless whether these don't-care products are used during synthesis or not.*

Proof. Let F be some single-bit output of the arithmetic function, corresponding to one of the output columns of the truth table. It can be implemented as a disjunction (OR) of product terms. Since product term is a conjunction (AND) of literals of individual variables, it is represented in algebraic form as a product of the corresponding variables. This is also true for products that include complemented variables; for example $a \wedge \neg b = a \cdot (1-b) = a - a \cdot b$, and similarly for arbitrary variable polarities. Hence, any product from the valid entries of the truth table may contain vanishing monomials. The same argument applies to the case when the input variables (in this case C) appear in different product terms; a disjunction of those terms will also create a product of the respective literals, according to the Equation (1): $a \vee b = a + b - a \cdot b$, where a, b can be any product term. As a result, the signature expression generated during rewriting may contain product of variables that correspond to vanishing monomials. □

We illustrate this theorem for the circuit with the truth table in Figure 3. Assume that output Q is implemented without don't-cares as $Q = C_1 \wedge \neg C_0 \vee \neg C_1 \wedge C_0 \wedge X$. This can be represented algebraically using Equation 1 as

$$Q = C_1 + C_0X - C_1C_0 - C_1C_0X \tag{7}$$

The invalid product C_1C_0 (to be removed from the final expression) will therefore appear as a vanishing monomial, even if the circuit is synthesized without $don't\text{-}cares$. The result of the theorem can be readily extended to an arbitrary form, including product of sums and factored forms, which also include AND and OR operations.

The generation of vanishing monomials is illustrated here with an example of a single-bit block of the divide-by-5 circuit. For the divisor D = 5, the remainder R and carry-in C are strictly less than 5, and hence will be encoded with $m = 3$ bits. This means that the don't care entries 101, 110, 111 for variables $C_2C_1C_0$ are invalid and can be treated as $don't\text{-}care$. Even if they are not provided explicitly, they can be readily extracted knowing the bit size of the divisor.

We can compute the algebraic expression for the invalid entries using algebraic rewriting discussed in Section III-A. The logical sum of the three terms can be computed in the algebraic domain as

$$C_2(1-C_1)C_0 + C_2C_1(1-C_0) + C_2C_1C_0 \\ = C_2C_1 + C_2C_0 - C_2C_1C_0 \quad (8)$$

This is in fact an algebraic equivalent of the Boolean cover of the three terms, with prime implicants, $C_2C_0, C_2C_1, C_2C_1C_0$, or, equivalently $\{1-1, 11-, 111\}$. Of the three, only the first two suffice to represent the logic, since each of them dominates $C_2C_1C_0$, which can be removed. Hence only the first two monomials, C_2C_0 and C_2C_1 are needed and are identified as vanishing monomials.

In summary, the automatic generation of vanishing monomials (for single and multiple blocks) includes the following steps:

1) Extract the unused (dc) entries from the truth table.
2) Compute algebraic expression of the product terms associated with the dc entries.
3) Remove the negative and redundant monomials.

E. *Verifying Multiple Blocks*

Figure 5 shows a block level diagram of an $X/3$ divider using a two-block architecture, each with $n = 2$ and $m = 2$.

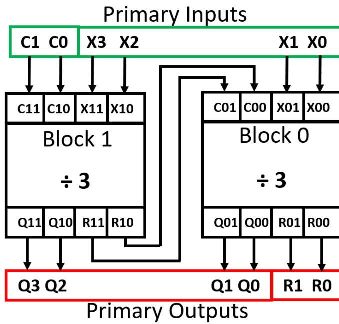


Fig. 5: Division of a 4-bit divide-by-3 in a two-bit block Divider Circuit. Rewriting is applied in the opposite direction to the data flow.

In the following, to simplify the notation, a single-letter index i represents the bit position of the entire circuit, rather than the block number. The internal signals are indexed by a pair, ij, referring to block i, bit j. The rewriting starts with the primary outputs $Q_3, Q_2, Q_1, Q_0, R_1, R_0$, with the output signature

$$Sig_{out} = 3(8Q_3 + 4Q_2 + 2Q_1 + Q_0) + 2R_1 + R_0 \quad (9)$$

It propagates through both blocks, B_1, B_0 until all the primary inputs, $C_1, C_0, X_3, X_2, X_1, X_0$ have been reached. The expected input signature at the primary inputs of the divider circuit is

$$Sig_{in} = 32C_1 + 16C_0 + 8X_3 + 4X_2 + 2X_1 + X_0 \quad (10)$$

In this particular stand-alone two-block configuration, C_1 and C_0 are set to zero, but in general they are coming from a 2-bit remainder of the higher level block. However, as explained earlier, the rewriting process will generate additional terms related to the product of the carry-in signals, the *vanishing monomials*, defined in Section III-D. The actual input signature obtained by the rewriting contains additional terms, denoted below as $F(V, C, X)$, in addition to the expected input signature of Equation 10.

$$Sig = F(V, C, X) + 32C_1 + 16C_0 + 8X_3 + 4X_2 + 2X_1 + X_0 \quad (11)$$

The term $F(V, C, X)$ is a polynomial containing the terms associated with the vanishing monomials V, in this case C_{11}, C_{10}, and with the redundant terms containing C and X. In a correct circuit, $F(V, C, X)$ will reduce to zero, proving that the circuit meets the specification.

Vanishing monomials expressed in terms of C_{11}, C_{10} for block 0 gets transformed in terms of C_{11}, C_{10} and X for block 1. Hence the size of these terms may grow during the successive rewriting steps over multiple blocks, which in a correct circuit will evaluate to zero. This build-up of a vanishing expression can be large, and it may significantly decelerate the performance of function extraction, often known as fat-belly effect. In a large circuit, this effect is even more pronounced since the vanishing monomials may be rewritten into more complex (yet redundant) monomials, causing a potential blow-up in the size of the computed signature.

One way to address this problem is to remove redundancy (vanishing monomials and boundary conditions) at the boundary of a given block before propagating the signature to the next block. However, in an unrolled circuit, synthesized across the block boundaries, it may be impossible to determine the boundary between the adjacent blocks. Fortunately, in the case of the division by a constant this information is readily available from the invalid (don't-care) entries in the lookup table, as explained in Section III-D, unless those signals are renamed by the synthesis process. The extraction, detection, and removal of vanishing monomials is fully automated in our methodology.

F. *Faulty Circuit Verification*

Let us consider the divide-by-3 circuit discussed in the previous sections. The output signature is $Sig_{out} = 3Q_0 + 2R_1 + R_0$ and the expected input signature of the correct circuit is $Sig_{in} = 4C_1 + 2C_0 + X_0$. Assume that the fault is caused by swapping the second and third entries in the truth-table of Figure 3. Then the gate-level implementation will be different, as shown in Figure 4(b), causing the algebraic transformations also to be different. As a result, the input signature obtained by backward rewriting, after removing the vanishing monomials, is $Sig_{in} = -C_1X_0 + C_0 + 2X_0 + 4C_1$. The mismatch between such obtained expression and the expected specification indicates that the circuit is faulty.

It should be noted that if there is a fault in the circuit that causes $R_{11}R_{10} = 1$, then the removal of this product as a vanishing monomial will not result in a wrong conclusion about the correctness of the circuit. Assume that block B_1 in Figure 5 is faulty and block B_0 is correct. Rewriting process starts from signals $Q_{01}, Q_{00}, R_{01}, R_{00}$ and transforms them into signals C_{01}, C_{00}, X_{01}, X_{00}. Since block B_0 is correct, the output signature across this block is also correct and linear after removing the vanishing monomials. In the next step, even when the vanishing monomial $C_{01}C_{00}$ (which in block B_1 becomes $R_{11}R_{10}$) is set to zero, the individual signals R_{11}, R_{10} are *not* removed from the expression and they are rewritten up to primary inputs, regardless of what their actual value is. This is also apparent by examining Equation 6, where the product C_0C_1 is removed as vanishing monomial, but the individual variables C_0 and C_1 are not! If block B_1 is faulty, the final computed signature will not match the correct signature/specification, because the expressions for R_{11}, R_{10}, propagated to the PIs, are faulty. Therefore, removing the vanishing monomials in any of the earlier stages will not affect the correctness of the signature in the subsequent blocks; and they never appear as a product in the output signature for a given block since the output signature is linear.

IV. Results and Analysis

The program implementing the described verification method was coded in Python and C++ and the experiments were conducted on a 64-bit Intel Core i7-7600 CPU, 2.80GHz × 2, with 31.0 GB of memory. The circuits were generated using an open-source hardware generator, FloPoCo [26], and synthesized using ABC tool [27] onto standard cell, gate-level circuits.

Four sets of results are presented, including: two types of unrolling schemes (modular and unrolled), verification of a restoring constant divider architecture, and a numerical simulation. We also show the results for a generic, restoring divider architecture.

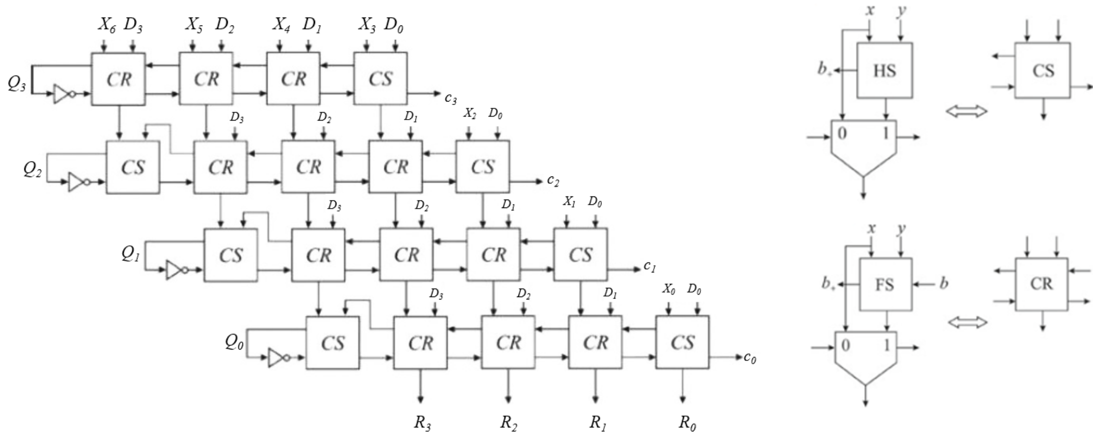


Fig. 6: Restoring Generic Divider [25].

A. Modular Architecture

In the *Modular* architecture, each block is instantiated the required number of times (depending upon the dividend bit-width). In this scheme, the boundary between adjacent blocks is known and the vanishing monomials are extracted and removed from the signature at each block, before rewriting the next block in series. The experiments include both correct (bug-free) and faulty circuits. The faults were emulated by randomly injecting multiple faults in the truth table into the valid portion of the look-up table. The invalid part of the table is not affected since it is used as a don't-care in synthesis.

Table I shows the verification run time for the divide-by-constant, one-bit block architecture using Modular scheme for a 32-bit dividend x. The results are shown for divisors value of up to 283 and a 9-bit remainder.

The table shows that the verification time does not change monotonically with the size of the divisor and can be explained by the content of look-up table. This non-monotonic behavior can be explained by examining the content of the truth tables for the corresponding divisions and its dependence on the value of the divisor. Consider, for example, a Divide-by-17 in Table I. The size of the LUT is 6 bits (one bit for the dividend X and 5 bits for the carry-in C, same as the size of the remainder). Of the 64 entries in the LUT only 35 are used, while the remaining 29 entries are invalid and treated as don't-cares. Whereas in the Divide-by-31, with the same size of the remainder and the LUT table, 62 out of 64 entries are used, and only two entries are redundant.

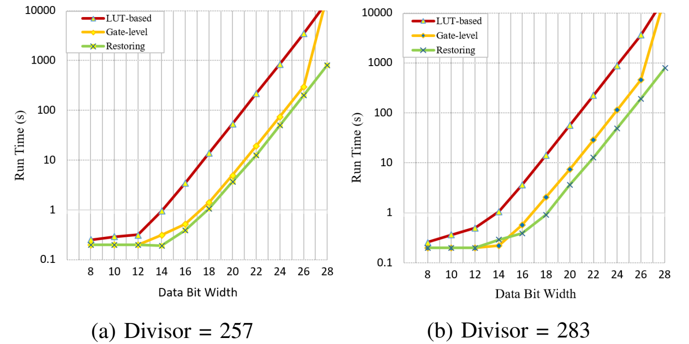
Table I also shows the results for the Modular, two-bit and four-bit block architectures for different divisors. The lower verification performance for these circuits compared to a one-bit architecture is caused by a drastically larger number of gates per block, preventing efficient removal of vanishing monomials during rewriting.

B. Flat Unroll

In *Flat Unroll* architecture the circuit is unrolled and synthesized (optimized) across the block boundaries. This causes any hierarchical information about the block boundaries to be lost, making the verification process harder. Since under this scheme the vanishing monomials are not removed at the block boundaries, the verification problem is significantly more memory intensive. As a result, the largest value of the dividend verified under this methodology is 2^9.

C. Restoring Constant Divider

We also tested an alternative architecture based on a standard *restoring divider* [22], in which the divisor D has been hardwired to a particular constant. The restoring divider has been implemented and synthesized using ABC. Constants from the bits of D are propagated through and used to optimize the overall circuit. Our rewriting-based verification technique has been integrated with the ABC data structure as a customized polynomial rewriting command &$polyn$. Unfortunately, ABC was unable to verify the circuits beyond 16 bits, resulting in a segmentation fault over 24 GB.



(a) Divisor = 257 (b) Divisor = 283

Fig. 7: Exhaustive simulation run time for divisors D=257 and D=283 for different implementations, as a function of the dividend bit-width.

D. Simulation Based Verification

We simulated the divide-by-constant dividers for different size of divisors D and dividend X, ranging from 2^8 to 2^{32}. We used Modelsim SE 10.0 on a Xeon 5650 processor with 6 cores (2.67GHz), 24 GB of RAM, and 350 GB free hard disk space.

Figure 7 shows the simulation results for $D = 257$ and 283 for the following three cases: 1) LUT-based implementation generated by FloPoCo [26]; 2) Gate-level implementation synthesized with ABC; and 3) Restoring constant divider implemented with ABC. As shown in Figure 7, the simulations are faster for gate-level than the LUT-based implementation. In contrast to our rewriting approach, the value of divisor D does not have significant impact on the simulation time.

The results show that the simulation approach is competitive for dividend bit-widths up to 22 bits (simulation time slightly longer than of our approach, for constant dividers). With higher bit-widths however, simulation time becomes prohibitive. For example, the simulation for dividends larger than 28 bits required 15,264 seconds (4h24m), with memory out of 24GB for larger bit-widths. Furthermore, the simulation based experiments shown here are run on a Xeon 5650 with six cores, which is a much more powerful machine compared to all of the other results (Core i7-7600 CPU with two cores). Regardless, our technique still outperforms simulation based verification schemes.

E. Restoring Generic Divider

This section demonstrates the applicability of our approach to the implementation of constant divider by a generic restoring divider,

TABLE I: Verification results for the divide-by-constant divider circuit using our technique for (1) Modular 1-bit block, 2-bit block, and 4-bit block architecture with a 32-bit dividend X (Figure 2). TO = 1200s, MO = 16GB; (2) Restoring Constant Divider with a 16-bit dividend X. TO = 1200s, MO = 24GB. SF = Segmentation Fault.

		Modular Unroll								Flat Unroll		Restoring (constant)	
		1-bit Block				2-bit Block		4-bit Block					
Divisor	# Rem. Bits	#Gates	Time (s) No Bugs	#Bugs	Time (s) Bugs	#Gates	Time (s) No Bugs	#Gates	Time (s) No Bugs	#Gates	Time (s) No Bug	#Gates	Time (s) No Bug
3	2	712	0.06	1	0.06	665	2.26	895	0.90	105	2.96	107	0.66
11	4	1919	1.15	2	1.11	1917	2.23	4045	MO	300	42.6	241	2.42
17	5	1763	0.81	3	.75	2236	5.83	2492	MO	192	6.68	252	4.13
31	5	1825	0.31	5	0.27	1676	0.85	10163	MO	282	169	234	10.4
61	6	3715	3.50	8	3.56	Memory Out				Time Out		263	9.12
89	7	4520	13.5	5	16.71							324	10.9
113	7	3652	6.68	7	7.21							284	3.82
139	8	5542	27.9	7	94.75							342	71.1
191	8	4736	9.67	5	11.36							316	SF
251	8	6410	110.4	5	113.5							295	14.53
257	9	6549	22.56	7	23.0							297	16.9
283	9	8951	643.8	9	638.4							336	22.3

shown in Figure 6. Table II shows the preliminary data for the verification run-time of a restoring divider over an AIG. As the complexity of the design increases beyond 1000 gates, the abc tool crashes with a segmentation fault, with a memory consumption of 20GB. Under this methodology, the divider circuit is heavily optimized and hence any boundary information between different modular blocks is lost, as shown in (Figure II). Our constant divider methodology is not scalable to generic dividers as of yet and currently the simulation based verification outperforms our technique. However, it demonstrates the significance of its applicability to generic divider circuits. We aim to empirically analyze these dividers in more depth in future to find the vanishing monomials, if any, to overcome the memory explosion problem.

TABLE II: Verification run time for the Restoring generic Divider. #Bits show the bit-width of dividend. SF = segmentation fault.

# Bits	Divisor Max.Value	# Gates	Time (s) This work	Time (s) Simulation
3	8	119	0.00	0.05
4	16	216	0.01	0.15
5	32	341	0.08	0.19
6	64	494	0.59	0.45
7	128	675	4.78	0.60
8	256	884	36.96	0.97
9	512	1121	SF:264	3.4
10	1024	1386	SF:232	13.55
20	1048576	4325	SF:240	MO

V. Conclusion and Future work

The paper presents a methodology to formally verify gate-level implementation of divide-by-constant divider architectures using algebraic rewriting verification. To the best of our knowledge this is the first attempt to accomplish such a verification without resorting to more labor-intensive methods, such as inductive systems and theorem provers. Our method can be automated and applied to a wide range of divider circuits. The results show that it is more efficient than the exhaustive simulation for circuits with operands larger than 22 bits(for constant dividers). We aim to improve the scalability of our technique for generic restoring division algorithm by exploiting block-level modularity (Figure 6) and detecting potential vanishing monomials. This initial work on the verification of constant dividers will offer insight into other divider architectures: non-restoring, SRT, and for floating point division algorithms. The future work will include a more challenging issue of bug identification and removal in faulty dividers.

References

[1] C. Yu, W. Brown, D. Liu, A. Rossi, and M. J. Ciesielski, “Formal verification of arithmetic circuits using function extraction,” *IEEE Trans. on CAD of Integrated Circuits and Systems*, 2016.

[2] T. Pruss, P. Kalla, and F. Enescu, “Equivalence Verification of Large Galois Field Arithmetic Circuits using Word-Level Abstraction via Gröbner Bases,” in *DAC’14*, 2014, pp. 1–6.

[3] R. W. Doran, “Special cases of division,” in *J. UCS The Journal of Universal Computer Science*. Springer, 1996, pp. 176–194.

[4] H. F. Ugurdag, F. De Dinechin, Y. S. Gener, S. Goren, and L.-S. Didier, “Hardware division by small integer constants,” *IEEE TC*, 2017.

[5] T. Granlund and P. L. Montgomery, “Division by invariant integers using multiplication,” *SIGPLAN Not.*, pp. 61–72, Jun. 1994.

[6] A. Paplinski, “Cse2306/1308 digital logic lexture notes, lecture 8,” *Lecture Notes*, 2006.

[7] H. Sharangpani and M. Barton, “Statistical analysis of floating point flaw in the pentium processor,” *Intel Corporation*, 1994.

[8] R. E. Bryant, “Bit-level analysis of an srt divider circuit,” in *33rd (DAC)*. ACM, 1996, pp. 661–665.

[9] M. Ciesielski, C. Yu, W. Brown, D. Liu, and A. Rossi, “Verification of Gate-level Arithmetic Circuits by Function Extraction,” in *52nd DAC*. ACM, 2015, pp. 52–57.

[10] C. Yu and M. J. Ciesielski, “Efficient parallel verification of galois field multipliers,” *ASP-DAC’17*, 2017.

[11] R. E. Bryant, “Graph-based algorithms for boolean function manipulation,” *Computers, IEEE Transactions on*, vol. 100, no. 8, pp. 677–691, 1986.

[12] R. E. Bryant and Y.-A. Chen, “Verification of Arithmetic Functions with Binary Moment Diagrams,” in *DAC’95*.

[13] M. Ciesielski, P. Kalla, Z. Zeng, and B. Rouzeyre, “Taylor Expansion Diagrams: A Compact Canonical Representation with Applications to Symbolic Verification,” in *(DATE-02)*, 2002, pp. 285–289.

[14] H. Rueß, N. Shankar, and M. K. Srivas, “Modular verification of srt division,” in *(ICCAD)*. Springer, 1996, pp. 123–134.

[15] R. Kaivola and K. Kohatsu, “Proof engineering in the large: formal verification of pentium® 4 floating-point divider,” *International Journal on STTT*, vol. 4, no. 3, pp. 323–334, 2003.

[16] J. Harrison, “Formal verification of ia-64 division algorithms,” *Theorem Proving in Higher Order Logics*, pp. 233–251, 2000.

[17] E. Pavlenko, M. Wedler, D. Stoffel, W. Kunz *et al.*, “Stable: A new qf-bv smt solver for hard verification problems combining boolean reasoning with computer algebra,” in *DATE*, 2011, pp. 155–160.

[18] A. Sayed-Ahmed, D. Große, U. Kühne, M. Soeken, and R. Drechsler, “Formal verification of integer multipliers by combining grobner basis with logic reduction,” in *DATE’16*, 2016, pp. 1–6.

[19] J. Lv, P. Kalla, and F. Enescu, “Efficient groebner basis reductions for formal verification of galois field arithmetic circuits,” *IEEE Trans. on CAD*, vol. 32, no. 9, pp. 1409–1420, 2013.

[20] D. Ritirc, A. Biere, and M. Kauers, “Column-wise verification of multipliers using computer algebra,” in *Formal Methods in Computer-Aided Design (FMCAD)*, 2017.

[21] M. Haghbayan, B. Alizadeh, P. Behnam, and S. Safari, “Formal verification and debugging of array dividers with auto-correction mechanism,” in *VLSI Design and 2014 13th ICES*. IEEE, 2014, pp. 80–85.

[22] I. Koren, *Computer Arithmetic Algorithms*. Universities Press, 2002.

[23] F. De Dinechin and L.-S. Didier, “Table-based division by small integer constants,” in *(ISARC*. Springer, 2012, pp. 53–63.

[24] C. Yu, M. Ciesielski, and A. Mishchenko, “Fast algebraic rewriting based on and-inverter graphs,” *IEEE TCAD of ICS*, vol. 37, no. 9, pp. 1907–1911, Sep. 2018.

[25] A. L. Ruiz, E. C. Morales, L. P. Roure, and A. G. Ríos, *Algebraic Circuits*. Springer, 2014.

[26] F. De Dinechin and B. Pasca, “Designing custom arithmetic data paths with flopoco,” *IEEE Design & Test of Computers*, vol. 28, no. 4, pp. 18–27, 2011.

[27] A. Mishchenko *et al.*, “ABC: A System for Sequential Synthesis and Verification,” *URL http://www. eecs. berkeley. edu/˜ alanmi/abc*, 2007.