Functional Verification of Hardware Dividers using Algebraic Model

Atif Yasin*, Tiankai Su*, Sébastien Pillement†, Maciej Ciesielski*
*University of Massachusetts, Amherst, MA, USA
†Univ Nantes, CNRS, IETR UMR 6164, F-44000 Nantes, France
ayasin@umass.edu, tiankaisu@umass.edu, sebastien.pillement@univ-nantes.fr, ciesiel@umass.edu

***Abstract*—Division is one of the most complex arithmetic operations to implement and its hardware implementation requires thorough verification at the gate level. Dividers are difficult to verify using standard Boolean methods, such as equivalence checking or SAT-based techniques, as they require "bit-blasting" onto bit-level netlists. Other methods, such as theorem provers, concentrate mostly on proving correctness of the division algorithm. However, verification of low-level hardware implementations has received only a limited attention. This paper addresses the problem of verifying gate-level divider circuits by extending an algebraic model, successfully used to prove multipliers and other arithmetic circuits, to dividers. The method verifies whether the gate-level divider circuit actually performs a division, without a need for a reference design.**

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 the integer and finite field domain [1][2]. However, the verification of hardware dividers has received only a limited attention from the verification community. A notable exception are theorem provers, inductive reasoning systems that concentrate on proving correctness of the division algorithm and the resulting architectures, but not on the gate-level verification [3][4].

Division plays a major role in many domains, including computer arithmetic, computational geometry, embedded systems, and other special purpose applications. It is one of the most complex arithmetic operations to implement and requires careful hardware implementation and verification [5]. In this work, we concentrate on combinational dividers: divide by constant, integer divider, and the fractional fixed point divider, an essential component of the floating point division. Our approach is based on the algebraic rewriting model, which performs arithmetic *function extraction*, originally proposed and successfully applied to integer and Galois Field multipliers [1] [6].

The paper is organized as follows. Section II provides the necessary background and reviews the previous work in this field. Section III describes the algebraic rewriting technique that forms the core of our verification approach. Section IV briefly describes verification of a divide-by-constant divider, while Section V develops an algebraic technique for the fractional and integer dividers. Finally, Section VI presents some preliminary results and conclusions.

II. Background and Previous Work

A popular technique used by industry for arithmetic circuit verification is Theorem Proving. Theorem provers are inductive reasoning systems that use mathematical reasoning to verify correctness of arithmetic circuit algorithms and their architectures. They use rewriting rules and complex formulas to represent the circuit and require domain-specific user knowledge [3][4]. The success of the proof relies on the choice of the rules and on the order in which the rules are applied, with no guarantee of a successful conclusion. Independently, there is a need to formally verify the actual gate-level hardware implementation, which is addressed in this paper.

Most popular verification techniques use SAT-based approach, equivalence checking, and various canonical diagrams, such as BDDs [7], BMDs [8]. Some of them 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 [9]. Although effective and able to catch (post mortem) the infamous Pentium bug, it requires generating a checker circuit, which itself needs to be verified. However, no reliable means were offered for the verification of the checker circuit itself.

The most promising and novel 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 goal is then to check if the implementation satisfies the specification. It is accomplished by reducing the specification polynomial modulo the implementation polynomials, known as the *ideal membership testing* [10]. Several modifications have been reported in the literature that improve the efficiency of the technique for multipliers [2][11]. However, those techniques have not been applied in the verification of dividers due to the difficulty in finding a suitable polynomial model of the divider's specification.

An alternative approach to arithmetic verification of gate-level circuits has been proposed in [1], using algebraic rewriting of the specification polynomial. With this approach, the polynomial representing the encoding of the primary outputs (called the *output signature*) is transformed by a series of rewriting steps into a polynomial expressed in terms of the primary inputs (the *input signature*). The transformation uses algebraic models of the circuit elements, such as logic gates

978-1-7281-3915-9/19/$31.00 © 2019 IEEE

or bit-level arithmetic modules. This method, in fact, performs *function extraction* as it derives an arithmetic function from the gate-level implementation. It has been successfully applied to complex adders and multipliers [1][11] but it has not been applied to divider circuits, because of the difficulty of modeling the divider's specification.

The work of [12] addresses the verification of array dividers by extracting a high-level arithmetic model from the low-level circuit implementation. The resulting arithmetic operations are compared with the abstract model of the divider using structural matching. The technique applies column-based XOR extraction, which relies on a regular structure of the adder/subtracter trees and the presence of the sum generation and carry propagation components. A lack of those components at the right places indicates a bug. The method requires the divider circuit to have a well defined architecture, and the adders to be represented with XOR gates, which is often not the case in a synthesized circuit. Their method performs structural analysis to see if the circuit's structure matches that of a divider but it does not explicitly verify its actual arithmetic function. In contrast, the method described here actually verifies whether the circuit performs the division operation, regardless of its internal structure, and it does not require any reference circuit. In practice, there is a need for both approaches: the complete functional verification and equivalence checking against a known reference design.

III. Algebraic Rewriting

Our verification approach is based on the algebraic rewriting method of [1]. In this method the circuit is modeled in the algebraic domain, where both the circuit specification and its gate-level implementation are represented as *pseudo-Boolean* polynomials. Each gate is modeled as a unique polynomial $f_i[X]$ with binary variables $X = \{x_1, ..., x_n\}$ and integer coefficients. Table I presents algebraic models of some of the basic Boolean operators [1].

TABLE I: Boolean and algebraic models (characteristic functions) of basic logic operations.

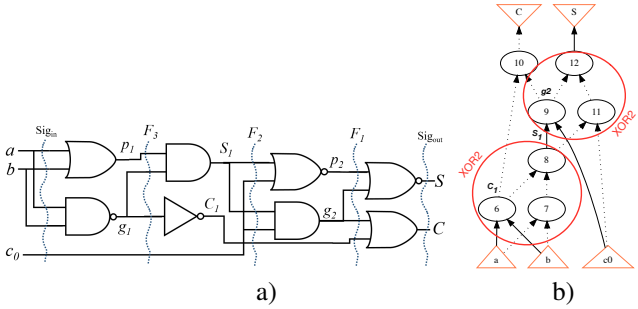
Operation	Boolean model	Algebraic model
$Inv(a)$	$\neg a$	$1 - a$
AND(a,b)	$a \wedge b$	ab
OR(a,b)	$a \vee b$	$a + b - ab$
XOR(a,b)	$a \oplus b$	$a + b - 2ab$

By construction, each expression evaluates to a binary value (0,1) and correctly models the Boolean function of the logic gate. Models for more complex AOI (And-Or-Invert) gates, used in standard cell technology, are readily obtained from these basic logic expressions. For example, an algebraic model for the logic gate $g = a \vee (b \wedge c)$ is $g = a + bc - abc$.

Algebraic rewriting relies on deriving two pseudo-Boolean polynomials, an output signature and an input signature, and transforming one into the other by a series of rewriting operations. The *output signature*, Sig_{out}, is the polynomial that represents the result using binary encoding of the primary outputs. For example, output signature of an usigned arithmetic circuit with n output bits is $Sig_{out} = \sum_{i=0}^{n-1} 2^i z_i$. By construction, such a polynomial is unique. Similarly, the *input signature*, Sig_{in}, is the polynomial over the primary input variables that represents an arithmetic function performed by the circuit, i.e., its *functional specification*. For example, the input signature for an n-bit unsigned multiplier is $Sig_{in} = \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} 2^{i+j} a_i b_j$.

Algebraic rewriting is the process of transforming Sig_{out} into Sig_{in} using algebraic models of the logic gates of the circuit, such as those given in Table I. It is done in a reverse topological order, from the primary outputs (PO) to the primary inputs (PI), hence referred to as *backward rewriting* [1]. The rewriting transformation simply replaces each variable in the polynomial with an algebraic expression of the corresponding logic gate. Once a given variable (output of a gate) is substituted by an algebraic expression of the gate inputs, it is eliminated from the expression. As a result, the final signature is unique and expressed in the primary inputs only. A set of ordering rules is imposed on the rewriting order to maximize its efficiency [1][11].

Figure 1(a) illustrates the rewriting process for a gate-level circuit to prove that it is a full adder (FA). The output signature of the circuit is $Sig_{out} = 2C+S$, determined by the weights of the binary encoded output signals, carry C and sum S. During rewriting, the signature polynomial is iteratively modified by moving across the gates and substituting variables representing the gate outputs by the respective logic expressions in terms of their inputs.



a) b)

Fig. 1: Gate-level arithmetic circuit (FA): a) circuit diagram; b) AIG representation for half adder (HA) extraction.

Two types of simplifications are applied during rewriting: 1) Reduction of the terms with the same monomials, reducing some of them to zero; and 2) replacing the term x^k with degree $k > 1$ by x, since for binary signals $x^k = x$. The resulting input signature is $Sig_{in} = a + b + c_0$, indicating that this is a full adder (note that it is constructed without XORs).

In the case when the circuit contains a bug, the size of intermediate polynomials during rewriting may become prohibitively large, sometimes even preventing the computation from completing. In general, identifying a bug is a challenging problem. Several attempts have been made to identify the bug(s), either by comparing the result of backward and forward rewriting [13] or by analyzing the difference between the

computed input signature and the given specification [14]. With a notable exception of finite field (GF) arithmetic circuits [15] [16], circuit debugging remains an open problem.

The rewriting process can be improved by using a functional, And-Invert Graph (AIG) representation of the circuit [17]. This technique identifies half- and full adders in the circuit, shown in Figure 1(b), and performs the rewriting directly over those sub-circuits instead of its logic gate components.

It seems at first that such a rewriting model cannot be directly applied to the divider. The characteristic function of the divider can be described by the following expression:

$$X = D \cdot Q + R, \quad \text{with } R < D \tag{1}$$

where X (the dividend) and D (the divisor) are the inputs, and Q (the quotient) and R (the remainder) are the outputs. The problem is that the outputs, Q, R, cannot be directly expressed in terms of the inputs X, D. Hence it is not clear what the input signature and the output signature are. The remainder of the paper shows how to resolve this problem and apply the algebraic rewriting to the divider verification.

IV. Divide by Constant

First, we consider a special case of the divider, where the divisor D is a known integer constant. Such a divider is used in many practical application, such as base conversion, Jacobi stencil algorithm, computing the sample mean, memory bank multiplexing, and more [18]. The fact that the divisor is constant makes the analytical I/O relationship straightforward: $X = D \cdot Q + R$, where D is a *hardwired* constant. This implies that the input signature is $Sig_{in} = X$ and the output signature $Sig_{out} = D \cdot Q + R$, which allows us to perform algebraic rewriting from the outputs Q, R to the input X.

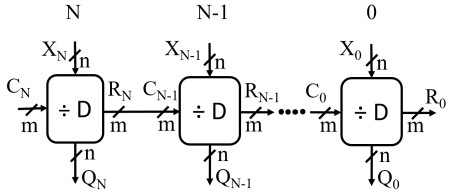


Fig. 2: Generic divider block for X divided by a constant D.

The constant divider is typically designed as an iterative circuit, composed of a number of blocks (N), as shown in Figure 2. The carry-in C_k and the n-bit dividend X_k are inputs of each block k. Similarly, the quotient Q_k and the remainder R_k are outputs of each block. For a single-bit block architecture (n=1) and the division by an integer constant D, we have $2C_k + X_k = DQ_k + R_k$. Typically, each basic block is implemented as a lookup table (LUT) with entries for all possible inputs, C_k, X_k, and the values of the corresponding outputs, Q_k, R_k. To verify the functionality of the basic block, we need to prove that the equation $2C_k + X_k = DQ_k + R_k$ is correct for every input assignment, where C_k and X_k form a one-word operand. The term $C_k = \sum_{i=0}^{n-1} 2^i C_i$, where C_i refers to bit i of block k.

To verify the constant divider X/D with N blocks, a composite output signature $Sig_{out} = DQ+R$ is created, where D is a constant, $Q = \sum_{k=0}^{N-1} 2^k Q_k$, and $R = \sum_{k=0}^{N-1} 2^k R_k$. In a functionally correct divider circuit, the resulting input signature should be $X = \sum_{k=0}^{N-1} 2^k X_k$, where each block k has the form: $Q_k = \sum_{i=0}^{n-1} 2^i Q_i$, $R_k = \sum_{i=0}^{m-1} 2^i R_i$, $C_k = \sum_{i=0}^{m-1} 2^i C_i$, and $X_k = \sum_{i=0}^{n-1} 2^i X_i$.

Table II shows the verification run time for a modular divide-by-constant, 1-bit block architecture for a 32-bit dividend X and different values of the divisor D. It also shows the results for a constant divider implemented as a restoring divider circuit, shown in Figure 6, with a 21-bit dividend The experiments include both the correct (bug-free) and faulty circuits, emulated by randomly injecting multiple faults in the truth table of the LUT.

TABLE II: Verification results for a divide-by-constant divider circuit with a 32-bit dividend and a constant restoring divider circuit with a 21-bit dividend (refer to Figures 2, 6, resp.).

		Modular, 1-bit block, 32-bit Dividend				Restoring (constant div.) 21-bit Dividend	
Divisor	# Rem. Bits	#Gates	Time (s) No Bugs	#Bugs	Time (s) Bugs	#Gates	Time (s) No Bug
3	2	712	0.06	1	0.06	107	0.66
11	4	1919	1.15	2	1.11	241	2.42
17	5	1763	0.81	3	.75	252	4.13
31	5	1825	0.31	5	0.27	234	10.4
61	6	3715	3.50	8	3.56	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

We also simulated the divide-by-constant dividers for different sizes of divisors D and dividend X, ranging in size from 2^8 to 2^{32}, using Modelsim SE 10.0. The results demonstrate that the simulation for dividends larger than 28 bits required 15,264 seconds (4h 24m). It was impossible to simulate dividends with larger bit-widths without using some advanced simulation techniques. In conclusion, the verification of the divide-by-constant based on algebraic rewriting can successfully compete with simulation.

V. Fixed Point and Integer Dividers

This section describes our approach to verify two types of dividers: 1) the *fractional* divider, operating on fractional numbers, an essential component of the floating point divider; and 2) the *integer* divider, with the same structure as the divide-by-constant divider, to be used in algebraic rewriting.

Current divider verification methods model the divider with a series of controlled add/sub operations. The most advanced divider verification method to-date is probably that of [12]. It is based on reverse engineering of the gate-level implementation by creating a logic bit-level model of the circuit (LBLA), and matching it against the well-structured functional reference model (FBLA). The method relies on extracting essential components, such as carry propagation logic and sum generation logic that are expected to be present in some form in the divider. It also searches for XORs and specific logic patterns

present in the reference divider. An error is declared if such functions cannot be identified in the circuit.

While the CPU runtimes are very good, such a reverse engineering method, based on a strictly structural pattern matching, does not provide the *functional* verification per se. It may happen, that some components do not match the expected logic, but may work correctly as an ensemble. Or, that some logic is represented without XORs. As an example, Figure 1 shows a non-standard full adder implementation without XORs.

In contrast, in our work the divider is modeled in a single functional specification, $X = D \cdot Q + R$, to be compared to the signature computed by algebraic rewriting. This approach works for any combinational divider circuit, regardless of its internal structure.

A. Functional Verification Model

Fractional divider is an essential part of hardware for the floating point division. The dividend X and the divisor D are normalized by preshifting to comply with the IEEE 754 standard. Figure 3 shows the functional model of the divider verification considered in this work. The blue box below the divider is a "reverse division unit", RDU, which computes $D \cdot Q + R$. The verification goal is to check if it is equivalent to the dividend X.

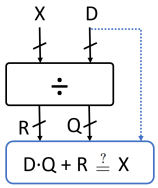


Fig. 3: Functional verification model of the divider.

One way to solve this problem is to apply a SAT or SMT technique; however instead of comparing the divider against a reference design we compare the RDU circuit ($D \cdot Q + R$) against the dividend X. As a proof of concept, we tested this method on both restoring and nonrestoring array dividers. A miter was added between the input X of the divider and the output of RDU. We used ABC system [17] to generate a CNF file for the SAT problem and solved it using $miniSAT$. While the solution required only 4.4 seconds to prove a 16-bit X/8-bit D divider, a 32-bit/16-bit divider timed out at 3600 seconds.

A more promising method to verify the divider is based on the algebraic rewriting using the structure shown in Figure 3. In this approach the output signature polynomial, $Sig_{out} = QD + R$, based on the outputs Q, R and the divisor D is created and algebraically rewritten through the divider network to the primary inputs, where in the correct circuit it should be equal to the dividend X.

For the illustration purposes we consider here an unsigned nonrestoring divider, a preferred hardware implementation that can be easily extended to signed division. In fact, both the restoring and nonrestoring dividers satisfy $X = D \cdot Q + R$, with $R < |D|$, but the nonrestoring divisor requires a minor correction when the remainder R and the dividend X have opposite signs, to make sure that $R < D$ [19].

B. Fractional vs. Integer Divider

We now demonstrate that the fractional divider can be used for integer division [19]. In fact, it is only a matter of interpretation of the result, whether a fractional or integer division is performed by the hardware, as demonstrated by the example below (see Figure 4). This will allow us to perform algebraic rewriting on the divider's circuit, while working only with integers.

In the following example we consider unsigned fractional numbers, with 0 in the leading bit position before the fractional dot, i.e., $X = 0.\{x_i\}$, in accordance with the IEEE standard. We assume that the bit-widths are sized as required to avoid overflow or underflow, i.e., X has size $2n + 1$ and D, Q, R are of size $n + 1$, including 0 before the fractional dot [19].

Fractional Divider (Figure 4(a)): The dividend and the divisor are preshifted, such that $X < D$, so that the result Q is also a fraction. The following representation is used: $X = 0.x_1....x_6, D = 0.d_1d_2d_3, Q = 0.q_1q_2q_3, R = 0.r_1r_2r_3$.

Let $X = (0.100000)_2 = 1/2$ and $D = (0.110)_2 = 3/4$, which satisfies a non-overflow condition, $X < D$. The result is: $Q = (0.101)_2 = 5/8, R = (0.010)_2 = 1/4$, Figure 4(a).

The computed remainder R needs to be multiplied by 2^{-3} (determined by its number of bits) to obtain the final remainder $R' = 2^{-3} \cdot 1/4 = 1/32$. Hence, $X = D \cdot Q + R' = 5/8 \cdot 3/4 + 1/32 = 1/2$, which is a correct result.

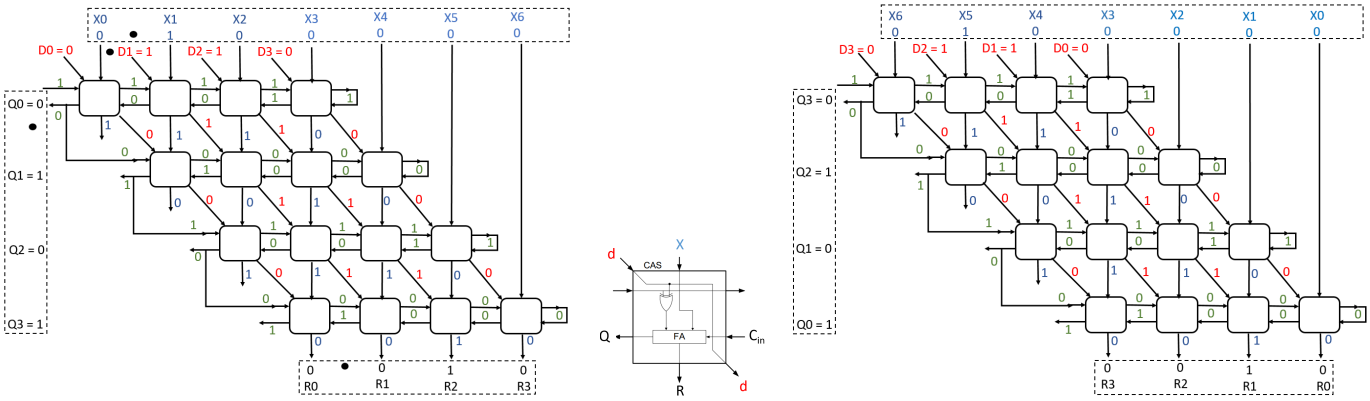
Integer Divider (Figure 4c): The result in the integer domain can be obtained with exactly the same hardware, but with the bits of the operand and the results ordered in the opposite direction. In this case, $X = 0x_6...x_2x_1 = (0100000)_2 = 32$, $D = 0d_3d_2d_1 = (0110)_2 = 6$. The result is: $Q = 0q_3q_2q_1 = (0101)_2 = 5$ and $R = 0r_3r_2r_1 = (0010)_2 = 2$. The result is correct: $X = D \cdot Q + R = 6 \cdot 5 + 2 = 32$; no adjustment of R is necessary in the integer case.

Note that, as long as the operands and the result registers are of correct size, the integer divider will always compute the correct value, with the difference between X and $Q \cdot D$ being compensated by the remainder R. The equivalence between the fractional divider and the integer divider, as illustrated above, gives us a right to use our algebraic rewriting technique on the integer divider to prove the fractional divider circuit.

C. Layered Rewriting

When rewriting an output Q or R of the divider, the final polynomial at the input, Sig_{in}, will be expressed in terms of the primary inputs, X, D. In a correct circuit, the composition of the resulting polynomials, $Q(X, D) \cdot D + R(X, D)$ should result in the dividend X, with variables of D eliminated. This is an ultimate test if the circuit correctly implements the divider.

One possible way to accomplish this is to generate the polynomial $Sig_{out} = D \cdot Q + R$ expressed in terms of the respective bits of Q, R, D, and rewrite it all the way to the primary inputs



(a) (b) (c)

Fig. 4: Nonrestoring 7-4 divider ($n = 3$): a) Fractional divider; b) Controlled Add/Subtract (CAS) block; c) Integer divider

(PI). The resulting Sig_{in} should produce a polynomial in bits of X only, representing the dividend. An alternative approach would be to express Q and R separately, each in their own bits, rewrite them to the PI, and then compose the resulting signatures as $Sig_{in} = Sig_{in}(Q) \cdot D + Sig_{in}(R)$. The result for a correct circuit should also be X, with D eliminated. Our initial experiments, however, suggest that these methods, when applied directly to the entire circuit, are inefficient, since the size of the intermediate polynomials becomes prohibitively large.

To address this problem, we developed a *layered* technique, which rewrites each row corresponding to one bit q_i of the quotient at a time. In this case, the output signature is $q_i D + P_i$, where P_i is the intermediate (partial) remainder, with the boundary condition $P_0 = R$. The expected input signature of row i is P_{i+1}, and $P_n = X$, where n is the number of (fractional) bits of R, Q and D.

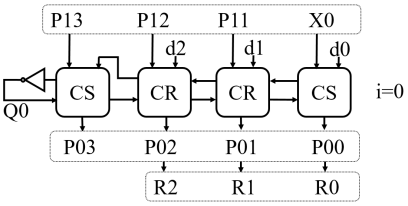


Fig. 5: Single layer of the restoring divider used in rewriting.

This approach can be justified by the observation that logic between two adjacent rows will not be optimized by a synthesis tool and the partial remainder signals are preserved during synthesis (refer to Theorem 2 of [12]). Synthesis tools, such as Synopsys DC, typically apply the *maintain hierarchy* directive, which is beneficial for physical synthesis. The circuit is synthesized across the add/sub modules of each layer, but not vertically across the rows.

Let P_i denote a *partial remainder* associated with the row corresponding to the quotient bit q_i (starting with $i = 0$). At the bottom of the array, $P_0 = R$, the final remainder; and at the top of the array, $P_n = X$, the dividend. Rewriting starts at the remainder output R and rewrites one row of the add/subtract circuitry at a time, using one bit of the quotient q_i and the entire divisor D to compute the partial remainder P_i. That is,

$$Sig_{out}(i) = q_i \cdot D + P_i \quad \text{and} \quad Sig_{in}(i) = 2P_{i+1} + X_i$$

where $D = \sum_{k=0}^{n} 2^k d_k$, $P_i = \sum_{k=0}^{n} 2^i P_{i,(i+k)}$, where $P_{i,j}$ denotes a bit of partial remainder in row i, column $j = i + k$, with the following boundary conditions:
$P_{0,k} = R_k$ ($k = 0,\ldots,n\text{-}1$); $P_{i,n+i} = 0$ ($i = 0,\ldots,n$); $P_{n,k} = X_k$ ($k = n,\ldots,2n$). Hence, at each level (row) i, we have

$$2^i(q_i D + P_i) = 2^{i+1} P_{i+1} + 2^i X_i$$

After n steps, the expected signature of the divider is X.

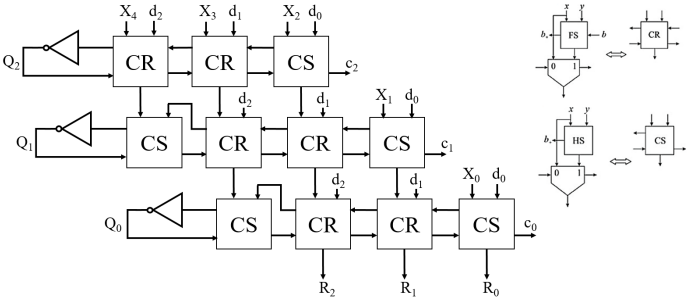


Fig. 6: Restoring integer divider [20].

To illustrate the idea, the following rewriting is applied to the restoring divider shown in Figure 6.

$$q_0 D + 4R_2 + 2R_1 + R_0 = 8P_{13} + 4P_{12} + 2P_{11} + X_0$$

$$2q_1 D + 8P_{13} + 4P_{12} + 2P_{11} = 16P_{24} + 8P_{23} + 4P_{22} + 2X_1$$

$$4q_2 D + 16P_{24} + 8P_{23} + 4P_{22} = 16X_4 + 8X_3 + 4X_2$$

By adding the above equations, we obtain:

$$(4q_2 + 2q_1 + q_0) \cdot D + (4R_2 + 2R_1 + R_0) =$$

$$16X_4 + 8X_3 + 4X_2 + 2X_1 + X_0$$

or, equivalently $Q \cdot D + R = X$, which is the ultimate proof that the circuit is a divider.

In this approach, the input signature computed for a given layer becomes an output signature for the next layer. Such a layered rewriting approach significantly speeds up the verification process and avoids the problem of a potential memory explosion, especially when there is a bug in the circuit. Furthermore, the method enables *debugging* by observing the signature at each rewriting step. If the result of local rewriting does not match the polynomial representing the partial remainder, P_i, we conclude that the bug exists in the current layer. This process can be easily done in a speculative parallel manner, since the form of each polynomial at the row boundary is known, and can be stopped when one of the layers does not produce the expected result. The source of error is constrained to the particular layer and the propagation of rewriting will stop there to examine the bug. The same procedure can be used to prove the nonrestoring dividers.

VI. Results

The verification technique described here was implemented in the ABC environment as a rewriting command $\&polyn$. The experiments were conducted on a 64-bit Intel Core i7-7600 CPU, 2.80 GHz $\times$ 2, with 31 GB of memory. The circuits were generated by a restoring divider generator tool and synthesized onto standard cells by the Synopsys Design Compiler (DC).

Table III shows the results for two verification methodologies: one, for fully rewriting the entire circuit, which (as explained earlier) does not offer promising results; and the other based on the layered verification described in this paper. The results are also compared against: 1) exhaustive simulation using Modelsim 10.5b on an Intel Core i7, 2.2 GHz with 16 GB memory; and 2) equivalence checking using miniSAT. For the SAT experiment, the synthesized divider circuits are compared against the dividers instantiated by the Synopsys DesignWare (DW) library. As one can see from the table, neither the simulation nor the SAT results can compete with the layered verification. While the time of 780 sec for a 21-bit restoring divider seems excessive compared to those presented in [12], it gives the time to verify the *function* of the divider circuit against its functional specification. This is a significantly harder task than checking its equivalence w.r.t. a reference design.

TABLE III: Verification results for a bug-free restoring divider. #Bits = Dividend bit-width. MO = Memory-out 20 GB, TO = Time-out 3600 s

# Bits	# Gates	Time (s) Full-rewrite	Time (s) This work	Time (s) Simulation	Time (s) SAT
5	201	0.08	0.01	0.45	0.14
7	352	4.78	0.01	0.97	0.24
11	415	MO	0.01	1.23	10.68
13	570	MO	0.01	8.3	19.16
17	970	MO	4.72	552.5	1584.32
19	1207	MO	51.7	TO	TO
21	1470	MO	780	TO	TO
23	1750	MO	MO	TO	TO

VII. Conclusion and Future work

This paper presents a method to verify the arithmetic function of the gate-level implementation of an integer divider using an algebraic rewriting technique. The main advantage of this method is that it verifies the divider circuit against its *functional specification* and does not require a reference circuit. As such, it can be used to prove newly developed architectures and certify them as a reference (golden model); or it can be used for designs that do not have a well defined or trusted reference. The method can be easily parallelized and applied to other arithmetic circuits. It also enables debugging of the circuit at the single layer granularity.

ACKNOWLEDGMENT: This work has been supported by a grant from the National Science Foundation, Award No. CCF-1617708.

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*, vol. 35, no. 12, pp. 2131–2142, 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] E. M. Clarke, S. M. German, and X. Zhao, "Verifying the SRT division algorithm using theorem proving techniques," in *(CAV)*. Springer, 1996, pp. 111–122.

[4] R. Kaivola and M. Aagaard, "Divider circuit verification with model checking and theorem proving," *Theorem Proving in Higher Order Logics*, pp. 338–355, 2000.

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

[6] C. Yu and M. Ciesielski, "Efficient parallel verification of Galois field multipliers," *ASP-DAC 2017*, 2017.

[7] R. E. Bryant, "Graph-based algorithms for Boolean function manipulation," *IEEE Trans. on Computers*, vol. 100, no. 8, pp. 677–691, 1986.

[8] R. E. Bryant and Y.-A. Chen, "Verification of Arithmetic Functions with Binary Moment Diagrams," in *Design Autom. Conf.*, 1995, pp. 535–541.

[9] R. E. Bryant, "Bit-level analysis of an SRT divider circuit," in *33rd (Design Automation Conference)*. ACM, 1996, pp. 661–665.

[10] E. Pavlenko, M. Wedler, D. Stoffel, and W. Kunz, "Stable: A new QF-BV smt solver for hard verification problems combining Boolean reasoning with computer algebra," in *DATE*, 2011, pp. 155–160.

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

[12] M. H. Haghbayan and B. Alizadeh, "A dynamic specification to automatically debug and correct various divider circuits," *INTEGRATION, the VLSI journal*, vol. 53, pp. 100–114, 2016.

[13] S. Ghandali, C. Yu, D. Liu, W. Brown, and M. Ciesielski, "Logic debugging of arithmetic circuits," in *ISVLSI'15*, July 2015.

[14] F. Farahmandi and P. Mishra, "Automated test generation for debugging multiple bugs in arithmetic circuits," *IEEE Trans. on Computers*, 2018.

[15] T. Su, A. Yasin, C. Yu, and M. Ciesielski, "Computer algebraic approach to verification and debugging of Galois field multipliers," in *ISCAS'18*, 2018, pp. 1–5.

[16] U. Gupta, I. Ilioaea, V. Rao, , A. Srinath, P. Kalla, and F. Enescu, "On the rectifiability of arithmetic circuits using Craig interpolants in Finite Fields," *IFIP Intl. Conference on VLSI (VLSI-SOC)*, Oct. 2018.

[17] A. Mishchenko, "ABC: A System for Sequential Synthesis and Verification," *URL http://www.eecs.berkeley.edu/˜ alanmi/abc*, 2007.

[18] T. Granlund and P. L. Montgomery, "Division by invariant integers using multiplication," *SIGPLAN Not.*, vol. 29, no. 6, pp. 61–72, Jun. 1994.

[19] I. Koren, *Computer Arithmetic Algorithms*. Universities Press, second edition, 2002.

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