July 2020

# Formal Verification of Divider and Square-root Arithmetic Circuits Using Computer Algebra Methods

Atif Yasin

# FORMAL VERIFICATION OF DIVIDER AND SQUARE-ROOT ARITHMETIC CIRCUITS USING COMPUTER ALGEBRA METHODS

A Dissertation Presented

by

ATIF YASIN

Submitted to the Graduate School of the
University of Massachusetts Amherst in partial fulfillment
of the requirements for the degree of

DOCTOR OF PHILOSOPHY

May 2020

Electrical and Computer Engineering

# FORMAL VERIFICATION OF DIVIDER AND SQUARE-ROOT ARITHMETIC CIRCUITS USING COMPUTER ALGEBRA METHODS

A Dissertation Presented

by

ATIF YASIN

Approved as to style and content by:

Maciej Ciesielski, Chair

Wayne Burleson, Member

Daniel Holcomb, Member

Namrata Shekhar, Member

Christopher V. Hollot, Department Head
Electrical and Computer Engineering

# ABSTRACT

## FORMAL VERIFICATION OF DIVIDER AND SQUARE-ROOT ARITHMETIC CIRCUITS USING COMPUTER ALGEBRA METHODS

MAY 2020

ATIF YASIN

B.S., LAHORE UNIVERSITY OF MANAGEMENT SCIENCES

M.S., UTAH STATE UNIVERSITY

Ph.D., UNIVERSITY OF MASSACHUSETTS AMHERST

Directed by: Professor Maciej Ciesielski

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. However, the verification of hardware dividers and square-root functions have received only a limited attention from the verification community, with a notable exception for theorem provers and other inductive, non-automated systems. Division, square root, and transcendental functions are all tied to the basic Intel architecture and proving correctness of such algorithms is of grave importance. Although belonging to the same iterative-subtract class of architectures, they widely differ from each other. IEEE floating point standard specifies square-rooting and division as basic arithmetic operation alongside the usual three basic operations. The difficulty of formally verifying hardware implementation of a divider/square-root can be attributed mostly to

the modeling of its characteristic function and the high memory complexity required by standard algebraic approach.

The work proposed in this thesis discusses formal verification of combinational divider and square-root circuits. Specifically, it addresses the problem of formally verifying gate-level circuits using an algebraic model. In contrast to standard verification approaches using satisfiability (SAT) or equivalence checking, the proposed method verifies whether the gate-level circuit actually performs the intended function or not, without a need for a reference design. Firstly, we present a verification methodology for a constant divider, where the divisor value is fixed to a constant integer. Albeit simpler case of verification, it provides us with the basic understanding of verification techniques and the underlying issues applicable to divider verification. Secondly, a layered verification approach is proposed for the verification of generic array dividers. Finally, the work proposed in this thesis will further analyze the divider and square-root circuits and aim to curb the memory explosion issue experienced by computer algebra based verification methods in order to successfully verify large bit-width divider-type arithmetic circuits. More specifically, a novel idea of "hardware rewriting" is introduced, which avoids the high memory complexity. The mentioned technique verifies a 256-bit gate-level square-root circuit with around 260,000 gates in just under 18 minutes and 127-bit gate-level divider circuit in under one minute.

# TABLE OF CONTENTS

# LIST OF TABLES

**Table** **Page**

# LIST OF FIGURES

**Figure** **Page**

# CHAPTER 1

# INTRODUCTION

## 1.1 VLSI Design Flow

What is hardware verification and more specifically, why is it important? Before we answer these questions, we need to understand the basic design flow of Very Large Scale Integrated circuits. Figure 1.1 shows the VLSI design flow, with the functional verification stage shown.

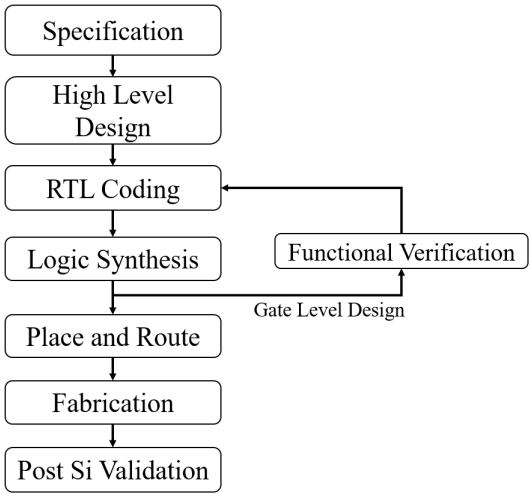



Figure 1.1: VLSI design flow

The design flow begins with an initial system-level specification written using a programming language, which can either be software (C, C++) or hardware language (Verilog, VHDL, etc.). This specification entails the functional behavior of the design, which is then converted into a register-transfer-level (RTL) description and translated

into Boolean expression for individual logic blocks and the interconnections logic by a process called synthesis. Synthesis is a process that maps the logic expressions of the design/circuit onto a gate-level netlist. Different optimization goals can be considered at this point to optimize the design for parameters, such as area, delay or power minimization, depending upon the target application and technology (ASIC or FPGA). Finally, the layout design tools, such as Cadence Encounter are used to perform physical synthesis, also known as place and route. Final sign-off of the design includes an extensively thorough verification before the circuit is sent out for fabrication. Many Electronic Design Automaton (EDA) tools provide seamless integration between all these steps and verify the functional correctness of the design between consecutive steps.

## 1.2 Hardware Verification

The main idea of these verification checks is to make sure that the specification of the design gets properly and accurately translated from previous state to the next and, ultimately, into a physical design without any errors. With contemporary processors containing over two billion transistors, it is imminent to employ design verification strategies to catch and correct the errors before the circuit gets fabricated on a physical chip. These multi-billion transistor chips can be generally sub-divided into datapath operators, memory unit, control unit, and other special purpose modules. Datapath operators are the main workhorses of computer systems, doing all the computations; they include arithmetic operations such as adders, subtractors, multipliers, dividers, square-rooters, etc. With an ongoing quest for minimum-area, high-speed and power-efficient design of these operators, these circuits/designs undergo several periodic improvements and optimizations. This consistent evolution demands a need for an efficient and consistent verification. Given the large input bit-width of these operators, enumerating the solution space by performing exhaustive simulation is sim-

ply infeasible. Hence, modern symbolic verification techniques (formal verification) are used to reason about the solution-sets without actually enumerating them. In this work, we use computer algebra techniques, which can reason about the solution space at both, the bit- and word-level, and naturally possess the required power of abstraction. Specifically, we target the functional verification of gate-level arithmetic designs (net-lists), and more specifically, dividers and square-rooters. Other kinds of hardware verification, such as model/property checking, physical verification, timing verification, clock domain crossing (CDC) verification, etc., are not subject of this work.

### 1.2.1 Canonical Diagrams

Binary decision diagrams (BDDs) [4] and all other variants BMDs [3], TEDs [8] etc. are data structures for representing Boolean functions, i.e., functions that take Boolean variables as input and produce Boolean outputs. Truth tables have been used to enumerate the logic described by a boolean function. However, truth tables require exhaustive enumeration, and are not memory efficient. Whereas, canonical diagrams, evolving from the Shannon expression, are compact graphs, where logic functions are encoded in graph paths efficiently, and some variants of BDDs are actually very efficient in crawling through the solution space of a Boolean function, addressing functional verification and equivalence checking of different arithmetic designs. These canonical diagrams are efficient to some degree in solving also arithmetic circuits, such as adders, subtractors, and multipliers. However, this representation has its limitations. Since the size of these diagrams increases exponentially with the size of bit-width, BDDs become prohibitive for larger designs and specifically for arithmetic circuits. Even though BDDs have been used to verify adders/subtractors [52] and to some extent multipliers [7], [23], the literature is rather scarce on verification of divider circuits. A notable exception in this domain is the work of Bryant [5].

Although, effective and being able to catch the infamous Pentium 4 floating point division 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.

### 1.2.2 SAT: Satisfiability Problem

In order to overcome the shortcomings of canonical diagram based verification, SAT (short for satisfiability) has emerged as a leading technique in Formal Verification. The main goal of SAT based verification is to find satisfying assignments to a formula, hence the name SAT. Typically, the formula is in the form of Cunjunctive Normal Form (CNF), a conjunction of one or more clauses, where a clause is a disjunction of literals. For example, the Boolean formula $\varphi = (a + \neg b)(\neg a + \neg b + c)$ can be satisfied by choosing $a = 1; b = 1; c = 1$ , which makes $\varphi = 1$, and hence provides a satisfiable solution. If the assignment of variables that makes the Boolean formula $\varphi = 1$ does not exist, the problem is called unsatisfiable or unSAT.

Since the seminal 1962 paper of Davis, Putnam, Logemann and Loveland [12], their DPLL algorithm has become a predominant algorithm to solve SAT instances. It is based on an intelligent space searching with a basic backtracking capability. Variables are selected according to some heuristic and assigned value 0 or 1. The newly assigned constants are propagated to the unsatisfied clauses by the process called Boolean Constraint Propagation (BCP) to identify implications, conflicts, and the satisfied clauses. An important enhancement to the basic DPLL algorithm is a Conflict-Driven Clause Learning (CDCL), which provides ability to learn new clauses that prevents the space search process from ending in an unsatisfying assignment. It also adds non-chronological backtracking from which a new search continues. Modern software SAT solvers are equipped with an efficient BCP, conflict resolution strategies,

and an improved decision heuristic that can rapidly and efficiently prune the search space.

However, regardless of all these innovations and developments, the SAT-based methods have a low scalability in verifying arithmetic circuits. For example, a state-of-art SAT solver, miniSAT [44], takes up to an hour to verify a 16-bit multiplier. Furthermore, it takes several hours to verify a 17-bit divider circuit. This is not an effective verification methodology since the current multiplication and divider units in core datapath are usually 32-bit or 64-bit wide. Hence, we need an effective and efficient methodology to verify these essential datapath operators.

### 1.2.3 Theorem Proving

The technique that received most attention in industry in arithmetic circuit verification is Theorem Proving. In this technique, the circuit is characterized by a set of rules, which are used to make complex formulas to represent the circuit [39][27][24][25][26]. However, the process of converting a circuit into a predefined set of rules to be applied sequentially requires a significant human effort and an intimate knowledge of the domain. The success of the proof relies on the choice of, and on the order in which the rules are applied, with no guarantee of a successful conclusion. Rager et al. [36] report that proving that the divider implemented by ORACLE is formally proven as equivalent of the SPARC ISA and IEEE 754 specifications, required "a sizable effort". Furthermore, these approaches cannot be fully automated or generalized. Regardless of these techniques, there is still a need to formally verify the actual hardware (typically, gate-level) implementation, which is addressed in this work.

# CHAPTER 2

# TECHNICAL BACKGROUND

This chapter provides a mathematical background of computer algebra method presented in this work and reviews the related work in the literature. This and the next chapter is written in collaboration with my colleague Tiankai Su [45] and is presented as a joint work to serve as a brief summary of computer algebraic background. Specifically, in order to build an algebraic model for an arithmetic circuit in the context of computer algebra, the following concepts are need to be revised: fields, polynomials, ideals, varieties and ideal membership, and Gröbner basis.

## 2.1 Fields, Polynomials, Ideals and Varieties

### 2.1.1 Fields

In mathematics, a field is a set $F$, containing at least two elements, on which two operations $+$ and $\cdot$ (called addition and multiplication, respectively) are defined, so that for each pair of elements $x, y \in F$ there are unique elements $x + y$ and $x \cdot y$ in $F$. A field is thus a fundamental algebraic structure, which is widely used in algebra, number theory, and many other areas of mathematics. To learn about fields, we start with the *commutative ring*, since field is a special class of ring. A commutative ring consists of a set R and two binary operations "$\cdot$" and "$+$" defined on R, for which the following conditions are satisfied:

(i) *Associativity*: $(a + b) + c = a + (b + c)$ and $(a \cdot b) \cdot c = a \cdot (b \cdot c)$ for all $a, b, c, \in R$.

(ii) *Commutativity*: $a + b = b + a$ and $a \cdot b = b \cdot a$ for all $a, b \in R$.

(iii) *Distributivity*: $a \cdot (b + c) = a \cdot b + a \cdot c$ for all $a, b, c \in R$.

(iv) *Identities*: There are elements $0, 1 \in R$ such that $a + 0 = a$ and $a \cdot 1 = a$ for all $a \in R$.

(v) *Additive inverse*: Given $a \in R$, there is $b \in R$ such that $a + b = 0$.

Two examples of commutative rings are the integers $\mathbb{Z}$ and the polynomial ring $k[x_1,...,x_n]$, with coefficients in an arbitrary field $k$. A *field* $\mathbb{F}$ is a commutative ring with unity, where every element in $\mathbb{F}$, except 0, has a *multiplicative inverse*: $\forall$ a $\in (\mathbb{F} - \{0\}), \exists\ \hat{a} \in \mathbb{F}$ such that $a \cdot \hat{a} = 1$. The most commonly used fields are $\mathbb{Q}$, $\mathbb{R}$ and $\mathbb{C}$. The set $\mathbb{Z}$, which is of particular interest to us, is a ring but not a field, since it does not have the attribution of a multiplicative inverse.

### 2.1.2 Polynomials

A polynomial is an expression consisting of variables and coefficients that involves the operations of addition, multiplication, and non-negative integer exponents of variables. In general, a **polynomial** $f$ in variables $x_1,...,x_n$ is a finite linear combination of monomials, with coefficients in some field $k$. A polynomial can always be written in a sum-of-product form $f = \sum a_i x_i^{\alpha_i}$ , where each product $x_i^{\alpha_i}$ is called *monomial* and $a_i$ is the coefficient. A monomial in variables $x_1,...,x_n$ is a product of the form $x_1^{\alpha_1} \cdot x_2^{\alpha_2} \cdots x_n^{\alpha_n}$, where all of the exponents $\alpha_1,...,\alpha_n$ are nonnegative integers. The *degree* of this monomial is the sum $\alpha_1 + \cdots + \alpha_n$. The total degree of polynomial $f$, denoted $\deg(f)$, is the maximum degree among all the monomials. A *term* of $f$ is the product of a nonzero coefficient and its monomial. As an example, polynomial $f = 2x^3y^2z + \frac{2}{3}y^3z^3 - 3xyz + y^2$ has four terms and total degree six. Note that there are two terms of maximal total degree, which is something that cannot happen for polynomials in one variable.

There are several ways to order monomials (referred to as *term order*), such as *lexicographic order* (LEX), *Degree reverse lexicographic order* (DEGREVLEX), and others. For instance, in LEX order, $2x^3y^2z > \frac{2}{3}y^3z^3$. The first, or greatest term of $f$

(in terms of the adapted term order), is called the *leading term* $lt(f)$ of the polynomial $f$. In the above example, the leading term is $2x^3y^2z$.

Leading terms play an important role in the proposed verification method, where logic gates of a circuit are described as polynomials. Specifically, the polynomial terms are ordered such that the leading term is a variable representing an output of a gate. This ordering makes a profound impact on the efficiency of the proposed verification technique. This issue will be discussed in detail in Chapter 3.

In this work, since all variables representing in the circuits are Boolean, we are particularly interested in polynomials with variables of degree 1. Such a polynomial is called *Pseudo-Boolean* polynomial. Formally, a Pseudo-Boolean function is a function $f : B^n \to \mathbb{R}$, where $B = \{0, 1\}$ is a Boolean domain and $n$ is a nonnegative integer called the arity of the function. It can be written as a multi-linear polynomial

$$f = a + \sum a_i x_i + \sum_{i<j} a_{ij} x_i x_j + \sum_{i<j<k} a_{ijk} x_i x_j x_k + \dots$$

with constant coefficients $a, a_i, ...$ in the given field.

### 2.1.3 Ideals and Varieties

Given a polynomial ring $R = k[x_1,...,x_n]$ with coefficients in some field k, a subset $I \subset R$ is an ***ideal*** if it satisfies:
(i) $0 \in I$. (ii) If $f, g \in I$ , then $f + g \in I$. (iii) If $f \in I$ and $h \in R$, then $hf \in I$.

In general, if $I \in k[x_1,...,x_n]$ consists of all the linear combinations of a set of polynomials $\{f_1, ..., f_s\} \in k[x_1,...,x_n]$, then $I$ is an ideal of the set $\{f_1, ..., f_s\}$, and the set of $\{f_i\}$ is called *generator* or *basis*.

$$J = \langle f_1, ..., f_s \rangle = h_1 f_1 + ... + h_s f_s : \ h_i \in R \tag{2.1}$$

We call $\langle f_1, ..., f_s \rangle$ the ideal generated by the basis $\{f_1, ..., f_s\}$.

Given an ideal $J = \langle f_1, ..., f_s \rangle$ generated by $f_1, ..., f_s, \in k[x_1, ..., x_d]$, the set of all solutions to: $f_1 = f_2 = \cdots = f_s = 0$ is called ***variety*** $V(f_1, ..., f_s)$ of $J$. While an ideal may have different bases, the variety depends only on the ideal and not on the basis (generator). That is, different bases that produce the same ideal will have exactly the same variety. In Section 2.3, we will introduce an especially useful basis for our verification, called Gröbner basis.

Let $\{f_1, ..., f_s\}$ and $\{g_1, ..., g_t\}$ be the bases of the same ideal in $k[x_1,...,x_n]$, i.e. $\langle f_1, ..., f_s \rangle = \langle g_1, ..., g_t \rangle$; then $V(f_1, ..., f_s) = V(g_1, ..., g_t)$. In the next section, we will show how the concept of ideal and variety is applied to circuit verification.

## 2.2 Ideal Membership Test

The symbolic algebra theories about polynomial rings, ideals and varieties we use in this work are all defined over a field, typically $\mathbb{Q}$. However, as described next and fully developed in the next section, the polynomials introduced in our work represent logic gates and are defined over ring $\mathbb{Z}$. However, these polynomials have a special structure, namely their leading term $lt(f_i)$ that represents a variable associated with a logic gate $g_i$ has coefficient 1. Subsequently, the process of polynomial division, which is an essential element of the verification process (to be described in detail later), will never introduce any coefficient outside of $\mathbb{Z}$. Consequently, this allows us to treat the polynomials as if they were in $\mathbb{Q}$.

Let $B = \{f_1, ...f_s\}$, with $f_i \in \mathbb{Z}[X]$, be a set of polynomials representing the circuit elements and let the ideal $J = \langle f_1, ..., f_s \rangle$ be generated by basis $\{f_1, ..., f_s\}$. In our case, each generator is a polynomial model of a circuit module (logic gate), and the set of generators can be viewed as the *implementation* of the circuit. Then, from the circuit perspective, the variety $V(J)$ of $J$, which is the set of all simultaneous solutions to a system of equations $f_1(x_1, ..., x_n) = 0; ..., f_s(x_1, ..., x_n) = 0$, contains all signal values of the circuit for all possible input valuations $\{x_i\}$.

Similarly, functional *specification* of the circuit is also defined as a polynomial in $\mathbb{Z}[X]$, where $X$ is a set of input and output variables. For example, the specification of a multiplier circuit, $Z = A \cdot B$, can then be written as a polynomial $F : Z - A \cdot B$. Here, $A, B$, and $Z$ are symbolic, word-level variables, each represented as a polynomial in their respective bit-level variables, e.g., $A = \sum_{i=0}^{n-1} 2^i a_i$, and similarly for $B$ and $Z$. In terms of *computer algebra*, the arithmetic circuit verification problem is then formulated as follows [34][30][41][38]:

Given a circuit represented by a set of generators (implementation), $B = \{f_1, ..., f_s\}$, and the specification $F$, the goal of functional verification is to prove that the implementation ($B$) satisfies the specification ($F$). Here, $B$ have the same notation as in the previous example, but it represents a set of gate polynomials. This means that for a functionally correct circuit, the solution to $F = 0$ agrees with $V(J)$, or, equivalently, that $F$ vanishes on $V(J)$[1]. Consequently, this problem has been termed as an *ideal membership test*, which decides whether the specification polynomial $F$ is a member of the ideal $J$ generated by $B$, i.e., if $F \in J$ [18][34][30].

Given an ideal $J = \langle f_1, ..., f_s \rangle$, in order to test if $F \in J$, polynomial $F$ is divided consecutively by $f_1, ..., f_s$. The goal of each division is to cancel the leading term of $F$ (with respect to a chosen term order) using one of the leading terms of $f_1, ..., f_s$. Such a reduction results in a polynomial remainder $r = F - \frac{lt(F)}{lt(f_i)} \cdot f_i$, in which the leading term $lt(F)$ has been canceled. If the remainder $r$ reduces to zero, the implementation satisfies the specification. However, if $r \neq 0$, such a conclusion cannot be drawn: $r$ can still be in $J$ but it is not divisible by any of the polynomials in $B = \{f_1, ..., f_s\}$. That is, the basis $B = \{f_1, ..., f_s\}$ may not be sufficient to reduce $F \longrightarrow 0$, and yet the circuit may be correct. To check if $F$ is reducible to zero for the given ideal $J$, one must compute a *canonical* set of generators, $G = \{p_1, ..., p_t\}$, called the *Gröbner basis*,

[1]Polynomial $f$ is said to vanish on a set $V$ if $\forall a \in V \ f(a) = 0$. Or, $V(f) \subseteq V(J)$.

with the same ideal $\langle p_1, ..., p_t \rangle = \langle f_1, ..., f_s \rangle$, the set $G = \{p_1, ..., p_t\}$ be the Gröbner basis for ideal $J$, then $F$ belongs to $J$ if and only if the remainder of the division of $F$ by the elements of $G$ is zero, denoted as $\forall F \in J$, $F \xrightarrow{G}_+ 0$ [1]. The sign + means that the division/reduction is done consecutively by using the elements of $G$ one by one. In short, the Gröbner basis is necessary to unequivocally answer the question whether $F \in J$.

## 2.3 Gröbner basis

A basis $\{p_1, ..., p_t\}$ of an ideal $J\langle p_1, ..., p_t \rangle$ is called a ***Gröbner basis*** (w.r.t. the monomial order $>$) if the leading term of every nonzero element of $J$ is a multiple of (at least) one of the leading term $lt(p_1), ..., lt(p_t)$. A known algorithmic procedure for computing a Gröbner basis is called Buchberger's algorithm [6]. Given some basis $B = \{f_1, ..., f_s\}$, it produces another basis $G = \{p_1, ..., p_t\}$, such that the ideals $\langle p_1, ..., p_t \rangle = \langle f_1, ..., f_t \rangle$ and hence have the same variety $V(\langle G \rangle) = V(\langle B \rangle)$. Buchberger's algorithm is computationally expensive, since it computes the so-called *S-polynomials (SPoly)* by performing reduction operations on all pairs of polynomials in $B$. The S-polynomial of polynomials $p$ and $g$ in a polynomial set $P$, is the combination $\text{Spoly}(p, g) = \frac{L}{lt(p)}p - \frac{L}{lt(g)}g$, where $L$ is the least common multiple $\text{LCM}(lm(p), lm(g))$. Note that $\text{Spoly}(p, g)$ cancels the leading terms of $p$ and $g$, and the remainder $r$ obtained in $\text{Spoly}(p, g)$ $F \xrightarrow{P}_+ r$ gives a new leading term.

The basic purpose of computing SPoly pairs is to compute polynomials with new leading terms, which can be used in the reduction step of the ideal-membership testing. These newly generated polynomials belong to the ideal $G$ which completely defines the system. To compute Gröbner basis $G = \{g_1, ..., g_l\}$ for an ideal $\langle p_1, ..., p_t \rangle$, Buchberger's algorithm computes $G$ in some finite number of steps by performing the $\text{Spoly}(p, g)$ $\xrightarrow{P}_+ r$ iteratively. The algorithm determines if $\text{Spoly}(p, g)$ $\xrightarrow{P}_+ 0$. In

this case, we also conclude that all polynomials are relatively prime to each other, with a distinct leading term.

This establishes that the generating set (generator) whose polynomials are relatively prime to each other is in fact a *Grobner* basis. This important fact will be used in developing the verification method in the upcoming sections. A number of other algorithms have been developed for computing a *Gröbner* basis, such as $F4$ [17], which in contrast to the basic Buchberger's algorithm, compute multiple SPoly pairs in each iteration. However, in general, the process of generating a *Gröbner* remains computationally expensive.

## 2.4 Related Work

The work in arithmetic circuit verification was pioneered by Shekhar et al. [43] and Wienand et al. [47], where some important concepts from computer algebra and algebraic geometry were applied to model the core verification problem. In [47] an arithmetic circuit is modeled as a network of arithmetic operators, such as half- and full-adders, comparators, and product generators, extracted from the gate-level implementation. These operators are modeled using *arithmetic bit-level* (ABL) expressions, $B = \{B_j\}$. The authors of [47] (and also of [30]) show that for an arbitrary combinational circuit, if the terms of the gate equations $B$ are ordered in reverse topological order, $\{\text{outputs}\} > \{\text{inputs}\}$, then all leading monomials of the polynomials in $B$ are relatively prime. As a result, the corresponding set $B$ already constitutes a Gröbner basis (GB), obviating the computation of the complete canonical Gröbner basis. The verification problem is solved by reducing the specification $F$ modulo $B$ to the *normal form* and testing if it vanishes over $\mathbb{Z}_{2^n}$. The restriction to binary variables is achieved by imposing Boolean constraints, $\langle x^2 - x \rangle$ for all the variable $x$ [34], and the problem is solved over quotient ring $Q = \mathbb{Z}_{2^n}[X]/\langle x^2 - x \rangle$ (for all variable $x$) using a popular computer algebra system, Singular [15]. This approach,

however, is limited to circuits composed entirely of half adders and full adders, which must first be extracted from the gate-level implementation. In practice, this is the most expensive part of the process, and it is not always possible to perform such extraction, especially in highly bit-optimized implementations.

In [30] the verification problem was similarly formulated as an *ideal membership test* but applied to Galois Field (GF or $\mathbb{F}_{2^q}$) arithmetic circuits. It has been shown that in GF, when the specification $F$ and the ideal $J$ of the circuit implementation are in $\mathbb{F}_{2^q}$, the problem can be reduced to testing if $F \in (J + J_0)$, over a larger ideal $(J + J_0)$ where $J_0 = \langle x^2 - x \rangle$ is an ideal of the field polynomials. Adding $J_0$ basically restricts the variety $V$ to solutions in $\mathbb{F}_2$, i.e. to $V(J) \cap V(J_0)$ [11]. The polynomials of $J_0$ are referred to as *field polynomials*. Similarly to [47], the authors of [30] derive the term order from the topological structure of the circuit, which renders the set of polynomials $B$ (circuit implementation) a Gröbner basis (GB), thus obviating the need to perform the expensive GB computation. The method uses a customized, F4-style polynomial reduction using a modified Gaussian elimination algorithm [17] under this term order.

A different approach to that of defined earlier has been proposed in [51], whereby the expensive polynomial reduction process has been replaced by a computationally simpler *algebraic rewriting* technique. The method introduces the concept of an *input signature*, a polynomial in the primary inputs, and an *output signature*, a polynomial derived from the encoding of the primary outputs. The verification is accomplished by rewriting the output signature, using algebraic expressions of the internal gates, into an input signature. This process de facto performs *function extraction*. Several ordering techniques have been described to make this method applicable to large arithmetic circuits, but the method still cannot handle the heavily optimized circuits.

A similar approach to arithmetic circuit verification, called *backward construction*, was proposed in 1995 in [23]. It uses BMDs to reconstruct functional, high level

representation from the gate-level structure of arithmetic circuits such as adders and multipliers. Experimental results show that time complexity of the tested circuits is in the order of $n^4$ for multipliers with $n$ bit operands. There is no clear indication if the BMD is an efficient data structure for this problem, and our experiments could not confirm its efficiency.

The basic approach of the ideal membership testing and Gröbner basis (GB) reduction has also been used in the works of [41][38], where it was applied to integer circuits. In [41] the following features have been added to make the reduction more efficient:

- *Logic reduction* with an AND-XOR vanishing rule, which analyzes the structure of the circuit to identify and remove vanishing monomials that correspond to the product of XOR, AND signals with shared input variables;
- An *XOR rewriting scheme*, which reduces the model of the circuit to consider only primary inputs, outputs, and fan-out points/XOR gates; and
- *Common rewriting*, which eliminates the nodes with a single parent. These techniques simplify the task of GB reduction by eliminating all the nodes which have exactly one parent, thus increasing the chance for early term cancellation during the rewriting process.

Another work [38] revisits the techniques from [51] and [41] and provides the proof of correctness for these approaches. It uses a column-wise technique to model and verify basic multiplier structures by computing the Gröbner basis incrementally for each column of the output bit, rather than for the entire circuit. The work is concluded by showing the efficacy of the technique by applying it to clean and "dirty", i.e., heavily optimized, multipliers. The paper justifies the use of the theory of ideal membership (in principle applicable to $\mathbb{Q}[X]$) to prove properties of integer arithmetic circuits in $\mathbb{Z}$. It points out that, since the leading coefficients of the gate polynomials forming

the Gröbner basis are +1 or -1, polynomial reduction never introduces fractional coefficients and their computation remains in $\mathbb{Z}$. This also explains why the dedicated implementations in [51] and [41] can rely on computation in $\mathbb{Z}$ only, while remaining sound and complete [38]. A follow-up paper [37] describes an enhancement to this column-wise technique by extracting half- and full-adder constraints to further reduce the size of Gröbner basis to speed up the reduction process.

In general, the problem of formally verifying complex integer arithmetic circuits (not just multipliers) remains open, and new solutions are being proposed. In the next chapter, an efficient and scalable approach, called *algebraic rewriting*, is introduced to address this issue. This approach has already been proposed by our group earlier, but it is further refined and formalized in this dissertation to be adequately applied to Divider and Square-root circuits. In addition, a *bit-flow* model is proposed to support the proof of the correctness of algebraic rewriting, and to offer a new insight into the problem of arithmetic circuit verification [9].

# CHAPTER 3

# ALGEBRAIC REWRITING AND DIVIDER MODEL

This chapter introduces the algebraic model used in circuit verification, which is the key to solve the verification problem in algebraic domain. Two flavors of computer algebra techniques that use this model will be discussed in detail: 1) Gröbner basis reduction techniques [34][41][38] and 2) algebraic rewriting [51]. Detailed algorithms for the reduction and the rewriting are given. We analyze the relation between these two computer algebra techniques and provide a comparison from the efficiency point of view.

## 3.1 Algebraic Model of Electronic Circuits

The arithmetic circuits considered in this dissertation are the ones which can be expressed as a polynomial in the input variables. These include adders, subtractors, multipliers, fused add-multiply circuits, dividers, and square-root. In this Section, we provide examples of existing solutions for multiplier verification. Later, we provide a detailed analysis of a divider verification methodology. Such arithmetic circuits are modeled as a network of interconnected bit-level components, each with a finite set of binary inputs with one or more binary outputs. In this work, we will focus on *gate-level* integer arithmetic circuits with single-output logic gates. However, the model can be extended to other, more complex, multiple-output circuit components such as dividers and square-rooters.

Each gate is modeled by a *pseudo-Boolean* polynomial $f_i \in \mathbb{Z}[X]$, with Boolean variables $X$ representing circuit signals associated with a logic gate. A pseudo-

Boolean polynomial is an integer-valued function $f : \{0,1\}^n \to \mathbb{Z}$. It is an algebraic expression with usual multiplication and addition operators over Boolean variables. The following expressions summarize the algebraic representation of basic Boolean operators NOT, AND, OR and XOR.

$$\begin{aligned} \neg a &= 1 - a \\ a \wedge b &= a \cdot b \\ a \vee b &= a + b - a \cdot b \\ a \oplus b &= a + b - 2a \cdot b \end{aligned} \tag{3.1}$$

By construction, each expression evaluates to a binary value {0,1} and hence correctly models the Boolean function of a 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, the algebraic model for the logic gate

- $g = a \vee (b \wedge c) => g = a + bc - abc$
- $OR3\ (a \vee b) => z = a + b + c - ab - ac - bc + abc$
- $XOR3\ (a \oplus b \oplus c) => z = a + b + c - 2ab - 2ac - 2bc + 4abc$
- $MAJ3\ (a \wedge b \vee a \wedge c \vee b \wedge c) => ab + bc + ac - 2abc$

Multiple output modules, such as single-bit adders, with binary inputs can be expressed similarly. For example, a half-adder (HA) and a full-adder (FA), can be expressed by the following expressions:

$$\begin{aligned} \text{HA}: &\quad 2C + S = a + b \\ \text{FA}: &\quad 2C + S = a + b + c_{in} \end{aligned} \tag{3.2}$$

where $a, b, c_{in}$ are binary inputs and $C, S$ are binary outputs.

The function computed by an arithmetic circuit is represented as a *specification* polynomial in the primary input variables, denoted $F_{spec}$. For example, the specification of an $n$-bit unsigned integer multiplier, $Z = A \cdot B$ with inputs $A = [a_0, \cdots, a_{n-1}]$ and $B = [b_0, \cdots, b_{n-1}]$, is described by $F_{spec} = \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} 2^{i+j} a_i b_j$. The result of the computation, stored in the primary output bits, is also expressed as a polynomial, called *output signature*, $S_{out}$. Typically, such a polynomial is linear, uniquely determined by the $m$-bit encoding of the output, provided by the designer. For example, for a signed 2's complement arithmetic circuit with $m$ output bits, $S_{out} = -2^{m-1} z_{m-1} + \sum_{i=0}^{m-2} 2^i z_i$. The circuit is implemented as a network of logic gates $G$, each modeled as a polynomial $g_i$ derived from Eqn.(3.1). The polynomial representing a given gate evaluates to zero for all the input and output combinations satisfied by this gate. As an example, a non-standard gate-level implementation of a full adder, is shown in Fig. 3.1.

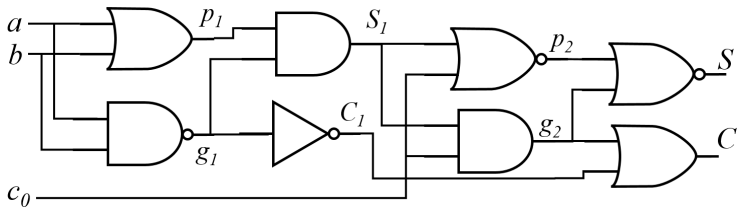



Figure 3.1: Gate-level arithmetic circuit (Full Adder)

The set of polynomials $G = \{f_i\}$ in Eqn. 3.3 represents the gate-level implementation of the full adder circuit. We refer to this set as $G$ to indicate that it is a Gröbner basis (or GB for short). It has been shown that if the polynomials in $G$ are ordered such that the leading term is the output of the gate, and the leading term of all the polynomials are relatively prime, the set $G$ forms Gröbner basis [35].

The set $G$ consists of two parts: *gate polynomials* $(f_1, ..., f_8)$ and *field polynomials* $(f_9, ..., f_{17})$. Each polynomial satisfies the relation $f_i = 0$. The gate polynomials have the form $f_i = v_i - tail(f_i)$, where the leading term $lt(f_i) = v_i$ is the output of

gate $f_i$, and $tail(f_i)$ is the logic specification of the gate in terms of its inputs. The leading terms under such ordering are relatively prime, which renders $G$ a Gröbner basis [34][30][38]. This feature is essential for both the GB reduction and algebraic rewriting, which will be discussed in the next sections.

$$
\begin{aligned}
f_1 &= p_1 - (-ab + a + b) \\
f_2 &= g_1 - (-ab + 1) \\
f_3 &= S_1 - p_1 g_1 \\
f_4 &= C_1 - (-g_1 + 1) \\
f_5 &= p_2 - (S_1 c_0 - S_1 - c_0 + 1) \\
f_6 &= g_2 - S_1 c_0 \\
f_7 &= S - (p_2 g_2 - p_2 - g_2 + +1) \\
f_8 &= C - (-C_1 g_2 + C_1 + g_2) \\
f_9 &= (a^2 - a) \\
f_{10} &= (b^2 - b) \\
&\cdots\cdots \\
f_{17} &= (g_2^2 - g_2)
\end{aligned}
\tag{3.3}
$$

Each field polynomials, $f_9, ..., f_{17}$, has the form $J_0 =< x^2 - x >$, where $x$ is one of the signals $\{a, b, c_0, p_1, g_1, S_1, C_1, p_2, g_2\}$. These field polynomials play an important role in polynomial reduction to maintain the Boolean property of each variable. However, they are handled differently in the GB reduction than in the algebraic rewriting approach, as discussed in the next sections.

## 3.2 Gröbner Basis Polynomial Reduction

In this method the reduction of $F$ modulo $G$ is accomplished by successively eliminating terms of $F$, one by one, by a leading term of some polynomial $f_i \in G$, using

Gaussian elimination. The reduction is performed over a Gröbner basis derived from $G$ and the field polynomials $J_0$. From the mathematical point of view, this means that the computation will be performed in the quotient ring, $\mathbb{Z}[X]/\langle x^2-x\rangle : x \in X$, the set of all variables (signals) of the circuit. The Gröbner basis (GB) reduction algorithm is given in Algorithm 1. First, the polynomial base $G=\{f_1, ..., f_m\}$ is derived from $\mathcal{N}$ using Equations (3.1), where $m$ is the number of logic components in $\mathcal{N}$. Each polynomial in $G$ has the form $f_i = v + tail(f_i)$, where $v$ is the the leading monomial $lm(f_i)$. All the variables in the circuit are ordered in reverse-topological order, from primary outputs to primary inputs, and for each gate polynomial from the gate output to its inputs.

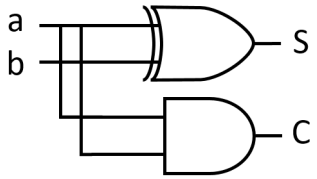



Figure 3.2: Half-Adder gate-level arithmetic circuit

Furthermore, the output signals of the gates that depend on common variables (fanins) should be ordered next to each other, as this will maximize the chance for a potential term cancellation and minimize the size of intermediate polynomials. For example, consider the reduction of a polynomial $F = 2C + S + ....$ in a circuit containing a half adder composed of an AND gate $C = ab$ and an XOR gate $S = a+b-2ab$, shown in Figure 3.2. Since both $C$ and $S$ depend on common variables, $a, b$, reducing them one immediately after the other will eliminate the product term $ab$ from the polynomial, resulting in $F = a + b + .....$ This is beneficial from the complexity point of view, and should be performed before the reduction of the remaining terms of the polynomial.

Considering these two basic ordering rules, one possible term order for the polynomial ring of the circuit in Figure 3.1 is shown below, where variables in curly brackets

can assume any relative order.

$$\{S, C\} > \{p_2, g_2\} > \{S_1, C_1\} > \{p_1, g_1\} > \{a, b, c_0\} \tag{3.4}$$

The expression $F$ to be reduced is initialized with the difference between the output signature $S_{out}$ and $F_{spec}$. In this case $F = 2C + S - (a + b + c_0)$. The goal is to reduce $F$ to 0 by $G$.

**Algorithm 1** Gröebner Basis Polynomial Reduction

**Input:** Specification polynomial $F_{spec}$;and Gate-level netlist $\mathcal{N}$
**Output:** Remainder *Rem*

```
1: Create base G={f_1,...,f_m} of N using Eq.(3.1)
2: Generate S_out from N
3: Define ring and specify term order
4: Initialize F ← S_out − F_spec
5: while F ≠ 0 do
6:     if ∃f_i ∈ G : lt(F)/lt(f_i) ≠ 0 then
7:     /* there exists f_i such that its leading term is divisible by lt(F) */
8:         F ← F − (lt(F)/lt(f_i)) · f_i // polynomial division
9:     else
10:    /* no leading term of f_i divides F, move lt(F) to Rem */
11:        F ← F − lt(F)
12:        Rem ← Rem + lt(F)
13:    end if
14: Maintain the term order imposed on the ring
15: end while
16: return Rem
```

The main part of the GB reduction is given in lines 5-15. The algorithm searches for a polynomial $f_i$ in $G$ such that the leading term of $f_i$ divides the current leading term $lt(F)$ of $F$. If such a polynomial exists, it will be used to reduce $F$, as shown in line 8. Otherwise, the $lt(F)$ will be moved to the remainder *Rem* (lines $11 - 12$). At any point, when new terms (containing new intermediate variables introduced by division) are added to polynomial $F$ (line 8), the procedure must maintain the term order imposed on the ring. The reduction process terminates when $F$ becomes empty,

either by being reduced or moved to *Rem*. The zero remainder is the evidence of a correct implementation, as discussed in Chapter 2.2.

We illustrate the GB reduction process with the example in Fig. 3.1. The initial polynomial for this circuit is:

$$F = 2C + S - (a + b + c_0) \tag{3.5}$$

Equation (3.6) gives the sequence of steps that reduces $F$ with the gate polynomials $f_i \in G$ for the circuit in Figure 3.1. At each step, $F$ represents the polynomial reduced by the previous reduction step. For brevity, the substitution is shown for a pair of variables at once. For example, $F/(C, S)$ means reducing variables $C$ and $S$ with polynomial $f_8$ followed by $f_7$. The term order given in Eqn. (3.4), imposed on the ring, is maintained throughout the entire reduction process.

$$\begin{aligned}
F &= 2C + S - (a + b + c_0) \\
1)\ F/(\ S\ ,\ C\ ) &= 2(-C_1 g_2 + g_2 + C_1) + (p_2 g_2 - p_2 - g_2 + 1) - (a + b + c_0) \\
&= p_2 g_2 - p_2 - 2g_2 C_1 + g_2 + 2C_1 - (a + b + c_0) + 1 \\
2)\ F/(\ p_2,\ g_2\ ) &= (S_1 c_0 - S_1 - c_0 + 1) S_1 c_0 - (S_1 c_0 - S_1 - c_0 + 1) - 2S_1 C_1 c_0 \\
&\quad + S_1 c_0 + 2C_1 - (a + b + c_0) + 1 \\
&= \mathbf{S_1^2 c_0^2 - S_1^2 c_0 - S_1 c_0^2 + S_1 c_0} - 2S_1 C_1 c_0 + S_1 + 2C_1 - (a + b) \\
3)\ F/(S_1^2 - S_1) &= -2S_1 C_1 c_0 + S_1 + 2C_1 - (a + b) \\
4)\ F/(\ S_1,\ C_1) &= -2(p_1 g_1)(-g_1 + 1)c_0 + p_1 g_1 + 2(-g_1 + 1) - (a + b) \\
&= -2(\mathbf{-p_1 g_1^2 + p_1 g_1}) \mathbf{c_0} + p_1 g_1 - 2g_1 - (a + b) + 2 \\
5)\ F/(\ g_1^2 - g_1) &= p_1 g_1 - 2g_1 - (a + b) + 2 \\
6)\ F/(\ p_1,\ g_1\ ) &= (-ab + a + b)(-ab + 1) - 2(-ab + 1) - (a + b) + 2 \\
&= \mathbf{a^2 b^2 - a^2 b - a b^2 + ab} \\
7)\ F/(\ a^2 -\ a) &= 0
\end{aligned} \tag{3.6}$$

The effect of field polynomials $J_0 =< x^2 - x >$, responsible for keeping each variable Boolean, can be observed during steps 2, 4, 6 and 7, shown in bold. The reduction terminates in $Rem = 0$, indicating that the circuit implements the function indicated by the specification, a full adder.

## 3.3 Algebraic Rewriting

Algebraic rewriting is the process of transforming the output signature $S_{out}$ into an input signature $S_{in}$ using algebraic models of the internal components (logic gates) of the circuit. The rewriting is done in reverse topological order: from the primary outputs (PO) to the primary inputs (PI); for this reason it is also referred to as a *backward rewriting* [51]. Intermediate expressions obtained during rewriting are also represented as polynomials, referred to as *signatures*, over the variables representing the internal signals of the circuit. By construction, each variable in a given signature (starting with $S_{out}$) represents an output of some logic gate.

The rewriting transformation simply replaces each variable with the corresponding algebraic expression of the logic gate. If the variable is part of a monomial involving other variables, the expression is multiplied by the remaining terms and expanded to a disjunctive normal form. This is followed by a standard polynomial simplification by combining terms with same monomials.

**Algorithm 2** Algebraic Rewriting

**Input:** Specification polynomial $F_{spec}$; and Gate-level netlist $\mathcal{N}$
**Output:** $(S_{in} == F_{spec})$, or the computed signature $S_{in}$

1: Derive $G$=$\{f_1,...,f_m\}$ from $\mathcal{N}$ using Eqn.(3.1)
2: Sort $G$ to maximize the cancellations // pre-processing
3: Generate $S_{out}$ from $\mathcal{N}$
4: Initialize $Sig \leftarrow S_{out}$
5: **for** $f_i$ in $G$ **do**
6: $v \leftarrow \text{lm}(f_i)$ // leading monomial of $f_i$ is output of a gate
7: **if** $v \in Sig$ **then**
8: /* replace $v$ with $tail(f_i)$ in $Sig$ */
9: $Sig \leftarrow Sig(v \leftarrow tail(f_i))$
10: $x \leftarrow x^2$ // for all $x$ in $Sig$
11: **end if**
12: **end for**
13: /* upon termination, $Sig$ is composed of PIs only */
14: **if** $Sig == F_{spec}$ **return True**
15: **else return** $S_{in} = Sig$

Algebraic Rewriting procedure is summarized in Algorithm 2. First, the polynomial base $G$=$\{f_1,...,f_m\}$ is derived from $\mathcal{N}$ using Eq.(3.1), as in the GB reduction. Then, the polynomials in $G$ are sorted in reverse-topological order (lines 1-2). Among several possible topological orders the one that maximizes the number of early cancellations during rewriting is sought. This has an effect of minimizing the size of the intermediate polynomials during rewriting (the "fat belly" effect) [51]. It is accomplished by keeping together the polynomials whose leading terms (gate outputs) depend on common variables, as in the GB reduction. The expression to be rewritten, $Sig$, is initialized with the given output signature $S_{out}$ of $\mathcal{N}$ (lines 3-4).

The main part of the rewriting, lines 5-12, iterates over the polynomials $f_i \in G$ and performs the required substitutions. Specifically, all occurrences of $v = lt(f_i)$ in $Sig$ are replaced by $tail(f_i)$, followed by possible expansion of the resulting term. To maintain Boolean values of the variables during rewriting, the degree of each variable in $Sig$ is reduced to 1 (line 10). This step is equivalent to dividing $Sig$ by a field polynomial $< x^2 - x >$, but it is achieved in a more efficient way. At the end, the algorithm returns $S_{in}$ as the derived signature of the circuit. If the terms of polynomials in $G$ are sorted in a reversed topological order, the returned polynomial $S_{in}$ contains only the primary input (PI) variables, so it can be compared with $F_{spec}$.

While the main goal of algebraic rewriting, as described by Algorithm 2, is to determine the arithmetic function implemented by the circuit, it can also be used to verify it against the known specification. This can be simply done by rewriting $F = S_{out} - F_{spec}$ and checking if it produces a zero. We will use this rewriting mode in order to compare it against the GB reduction method in Chapter 3.2.

We illustrate the rewriting process using the example of the gate-level full-adder circuit in Figure 3.1. The output signature of the circuit is $S_{out} = 2C + S$, determined by the binary encoding of the output. The specification for this circuit $F_{spec} = a+b+c_0$. Following the ordering rules described in [51], the best rewriting order which

$$
\begin{aligned}
F &= 2C + S - (a + b + c_0) \\
1)\ F/(S,\ C\ ) &= 2(C_1 + g_2 - C_1 g_2) + (1 - (p_2 + g_2 - p_2 g_2)) - (a + b + c_0) \\
&= 2C_1 + g_2 - 2C_1 g_2 - p_2 + p_2 g_2 + 1 - (a + b + c_0) \\
2)\ F/(p_2,\ g_2) &= 2C_1 + S_1 c_0 - 2S_1 C_1 c_0 - (1 - (S_1 + c_0 - S_1 c_0)) \\
&\quad + (1 - (S_1 + c_0 - S_1 c_0))S_1 c_0 + 1 - (a + b + c_0) \\
&= 2C_1 - 2S_1 C_1 c_0 + S_1 + \mathbf{S_1 c_0 - S_1^2 c_0 - S_1 c_0^2 + S_1^2 c_0^2} - (a + b) \\
&= 2C_1 - 2S_1 C_1 + S_1 - (a + b) \\
3)\ F/(S_1, C_1) &= 2(1 - g_1) - 2(1 - g_1)(p_1 g_1)c_0 + p_1 g_1 - (a + b) \\
&= 2 - 2g_1 - 2(\mathbf{p_1 g_1 - p_1 g_1^2}) + p_1 g_1 - (a + b) \\
&= 2 - 2g_1 + p_1 g_1 - (a + b) \\
4)\ F/(p_1,\ g_1) &= 2 - 2(1 - ab) + (a + b - ab)(1 - ab) - (a + b) \\
&= \mathbf{ab - a^2 b - ab^2 + a^2 b^2}\ =\ 0
\end{aligned}
\tag{3.7}
$$

minimizes the size of intermediate polynomials is $\{(S, C), (p_2, g_2),\ (S_1, C_1), (p_1, g_1)\}$, as in the GB reduction. The signals shown in brackets can be rewritten in any order as they depend on common inputs. Equation (3.7) shows the rewriting steps for the circuit. The terms shown in bold face indicate those that are reduced to zero during polynomial simplification. For brevity, the substitution is shown for each pair of variables applied at once. For example: $F/(C, S)$ means rewriting of $F$ using $C$ and $S$ variables of polynomials $f_8, f_7$.

During the rewriting, two types of simplifications can be observed:

- Simplification of the terms with same monomials; for example, $2g_2 - g_2 = g_2$, in Step 1. In the process, some polynomial terms are reduced to 0. This is a common simplification applied in GB reduction as well.

- Lowering the term $x^2$ to $x$, since the signal variables are binary. This can be seen in Steps 2, 3, and 4, shown in bold face. For example, in step 2 we have: $S_1 c_0 - S_1^2 c_0 - S_1 c_0^2 + S_1^2 c_0^2 = S_1 c_0 - S_1 c_0 - S_1 c_0 + S_1 c_0 = 0$. Similarly, in step 3: $(p_1 g_1 - p_1 g_1^2) = p_1 g_1 - p_1 g_1 = 0$, etc. This simplification is simpler and can be executed faster than dividing the polynomials by the respective field

polynomials ($x^2 - x$), as it is done in computer algebra approach. This is one of the main reasons for greater efficiency of the algebraic rewriting compared to GB reduction.

Subsequently, the final result reduces $F = S_{out} - F_{spec}$ to zero, indicating that the circuit correctly implements a full adder.

It should be noted that in addition to the two basic simplification rules mentioned above (rewriting the gates with common inputs, and the $x^2 \rightarrow x$ reduction), more sophisticated simplifications can be applied to the running polynomial $Sig$ during rewriting by analyzing the structure of the gate-level network. For example, recognizing that some signal $g$ is a product of XOR and AND signals with the same fanin inputs will reduce signal $g$ to zero. This simplification, called an *XOR-AND vanishing rule* has been used by [41], but for clarity of the illustration, it has not been taken into account here.

## 3.4 AIG Rewriting

The algebraic rewriting technique described in the previous section can be further improved by performing rewriting using the functional AIG (Add-Inverter Graph) representation of the circuit instead of its gate level structure. This section provides a brief overview how this is accomplished, with details provided in [50].

AIG (And-Inverter Graph) is a combinational Boolean network composed of two-input AND gates and inverters [2]. Each internal node of the AIG represents a two-input AND function; the graph edges are labeled to indicate a possible inversion of the signal. We use the *cut-enumeration* approach of ABC [2] to detect XOR and Majority (MAJ) functions with a common set of variables; they are essential components of adder trees that are present in most arithmetic circuits in some form [50]. After detecting the XOR and MAJ components of the adder's AIG, rewriting skips over the detected adders, significantly speeding up the rewriting process. Figure

3.3 illustrates the process for the full adder (FA) circuit from Figure 3.1. In Figure 3.3 the groups of nodes (6,7,8) and (9,11,12) correspond to half adders (HA). The functions rooted at nodes 6 and 9 are majority (AND) functions, and those at nodes 12 and 8 are XORs. Subsequently, the functions at node 12 ($S$) and node 10 ($C$) are identified as XOR3 and MAJ3, respectively, on the shared inputs, $a, b, c_0$. The AIG rewriting of $S_{out} = 2C + S$ over the extracted XOR3 and MAJ3 nodes is trivial, with the nonlinear monomials automatically cancelled, as shown in Eqn. 3.8.

$$\begin{aligned} 2C + S &= 2(ab + ac_0 + bc_0 - 2abc_0) \\ &\quad + (a + b + c_o - 2ab - 2ac_0 - 2bc_0 + 4abc_0) \\ &= a + b + c_o \end{aligned} \tag{3.8}$$

The resulting signature matches the specification, which clearly indicates that the circuit is a full adder. As illustrated with this example, the AIG rewriting requires considerably fewer terms than the standard algebraic rewriting.

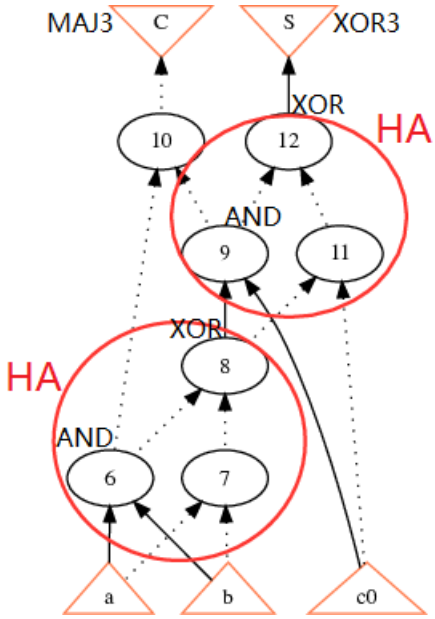



Figure 3.3: AIG rewriting of a full adder circuit from Figure 3.1.

**Data structure**: AIG rewriting is implemented in ABC with the polynomial data structure, type `Pln_Man_t`. Its main components include: 1) the AIG manager

(`Gia_Man`) that represents the input design; and 2) two vector hash tables using type `Hsh_VecMan_t` are used for storing the constants and monomials. The hash tables of monomials include coefficient vectors and monomial vectors. When substitution is applied to the leading term, new monomials are created and the substituted one removed. For example, when $ab + c + bd$ is substituted by $a = b + d$, the monomial $ab$ is removed first, and $b$ and $bd$ are added to `Pln_Man_t`. During the process of adding the new monomials, the program will first check if these monomials already exist in `Pln_Man_t`; in this case only the coefficient of these monomials will be changed accordingly. In this example, two new monomials are generated by the substitution, namely $b^2$, reduced to $b$, and $bd$. Since $bd$ already exists in the expression, the coefficient 1 of $bd$ is replaced by 2, resulting in $b + c + 2bd$.

## 3.5 Comparison between GB Reduction and Rewriting

It should be clear from the above discussion that both methods, the GB reduction and the algebraic rewriting, are equivalent in the sense that they both perform polynomial reduction. The GB reduction scheme achieves polynomial reduction by division, in fact, performing Gaussian elimination. In contrast, algebraic rewriting does it by substituting the gate output variable by the polynomial expression of the gate's function. While the goal of GB reduction scheme is to reduce $F = S_{out} - F_{spec}$ modulo the set of implementation polynomials $G$ to 0, it can also be used to extract the arithmetic function by reducing $S_{out}$ modulo $G$, and interpret the result as the functional specification of the circuit $F_{spec}$. In the algebraic rewriting scheme, the goal is to rewrite the output signature $S_{out}$ to $S_{in}$, the expression in the primary inputs, and check if it matches the expected specification $F_{spec}$. If $S_{in} = F_{spec}$, the circuit is correct; otherwise it is faulty. Alternatively, as illustrated above, algebraic rewriting can be also applied to $F = S_{out} - F_{spec}$, as in the GB approach.

Variable substitution of algebraic rewriting (line 9 of Algorithm 2) seems simpler than the main step of polynomial division of the GB reduction (line 8 of Algorithm 1). On the other hand, it requires additional multiplication of the terms and expansion into a sum of products. Hence, the complexity of these steps is comparable. Both methods avoid explicit computation of the Gröbner basis, but achieve it by different means. In the GB reduction it is done by setting the variable order in the ring so that all variables are in reverse topological order, which makes the implementation set $G$ a Gröbner basis. In the algebraic rewriting scheme on the other hand, the polynomials $f_i \in G$ are sorted in reverse topological order to effect the rewriting. As a result, both methods ensure that the polynomial base is a Gröbner basis. However, there are some essential differences between the two methods that affect their efficiency.

- The GB reduction scheme requires the *field polynomials* $J_0 =< x^2 - x >$ to be added to the base $G$ in order to keep the variables Boolean. This increases the size of the Gröbner basis and results in a larger search space in each iteration. Whereas in the rewriting scheme, the reduction by $< x^2 - x >$ is solved in a simpler way, namely by lowering $x^2$ to $x$ via a simple data structure (line 10 in Algorithm 2).

- In the algebraic rewriting scheme, the gate polynomials $f_i \in G$ are ordered in reverse topological order (line 5 in Algorithm 2) so that each gate polynomial $f_i$ is used exactly once. Furthermore, the selected polynomial is used to perform the rewriting by a simple string substitution and is never needed again. In contrast, in each iteration of the GB reduction one has to search for a polynomial $f_i$ that divides the leading term of $F$ under reduction. While in principle the GB reduction can also work over an ordered list of gate polynomials, this does not apply to the field polynomials $< x^2 - x >$, needed for the reduction. Since the appearance of intermediate signals in nonlinear terms $x^k$ is unpredictable, it is not possible to pre-order the list of field polynomials in GB reduction.

# CHAPTER 4

# FORMAL VERIFICATION OF INTEGER DIVIDERS: DIVISION BY A CONSTANT

Division is one of the most complex arithmetic operators to implement and requires careful hardware implementation and verification [16][46]. The difficulty of formally verifying hardware implementation of dividers can be attributed to the mathematical model of a divider: its characteristic function cannot be written as a closed-form expression, making it difficult to assign to it an output signature. In this chapter, we present a verification methodology for a constant divider, where the divisor value is fixed to a constant integer. Albeit a simpler case of verification, it provides us with the basic understanding of the underlying issues applicable to divider verification. Later, in the next chapter, we present a generalized approach for verifying generic array dividers using approach similar to that developed in this chapter.

## 4.1 Introduction

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 that use Jacobi stencil algorithm to compute 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 compiler optimization to generate integer divisions to compute loop counts and subtract pointers [19]. 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" [32]. In this chapter, we concentrate on the verification of a division by a constant and conclude it by presenting a preliminary verification analysis of a restoring generic divider circuit.

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 [42][5] for divider circuits. The work by [20],[21] address the verification of array dividers by applying reverse engineering methods to obtain 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 extracted components include logical bit level adders with sum generation logic (SGL), carry propagation logic (CPL) and controlling logic (CL). The technique applies column-based XOR extraction, which relies on a regular structure of the adder trees and on the presence of sum generation and carry propagation components. Lack of those components at the right places indicates a potential bug. The limitation of the method is that it assumes a known, well structured architecture and that adders are represented with XOR gates, which may not be the case in a synthesized circuit.

This chapter describes the verification technique for a divide-by-constant circuit, later generalized for 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 [10] [53]. The method has been suitably modified for dividers by iden-

tifying and taking advantage of the "vanishing monomials", which are an intrinsic property of table-based divide-by-constant architecture.

The rest of the chapter is organized as follows. Section 4.2 provides the necessary background and the survey of the implementation and verification of the divide-by-constant architectures. Section 4.3 shows the detailed verification methodology, while Section 4.4 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.

## 4.2 Background

The algebraic based verification methods described in earlier chapter 3 have been successfully applied to complex adders and multipliers, including Booth, [51][41] but have not been applied to divider circuits, because of the difficulty of modeling the divider's specification. 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:

$$X = D \cdot Q + R, \quad \text{with } R < D \tag{4.1}$$

where $X$ (the dividend) and $D$ (the divisor) are the inputs, and $Q$ (the quotient) and $R$ (the remainder) are the outputs where $R < D$. 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. 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 $Sig_{out} = D \cdot Q + R$. The remainder of the thesis presents how algebraic rewriting is utilized for verification of divider and square-root circuits.

### 4.2.1 Divider Circuit Implementation

There are two main approaches to implementing arithmetic division: 1) division by addition/subtraction, such as SRT, restoring, non-restoring; and 2) division by reciprocation, or multiplication by the inverse via Newton-Raphson or Goldsmith algorithm [29]. 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 [32][13][46]. 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.

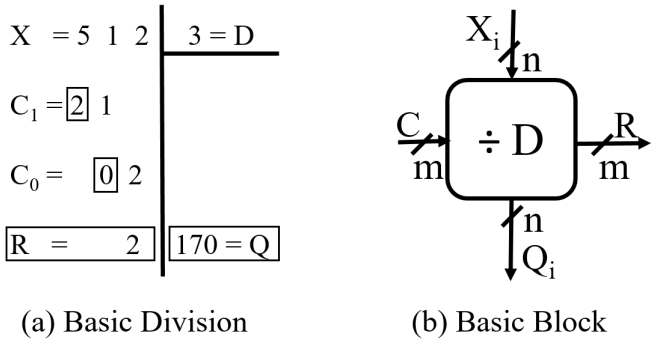



Figure 4.1: Pencil and Paper division operation and the basic divider block.

The author of [46] 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 work is based on the restoring algorithm, discussed in Section 4.4.3 and shown in Figure 4.7 in the later part of the chapter.

## 4.3 Verification

Our verification scheme is based on the functional extraction method of [51] [50], reviewed in detail in Chapter 2. 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.

### 4.3.1 Verification of the Constant Divider

In the iterative, divide-by-constant circuit, the divider is partitioned into a number of blocks, each having the structure shown in Figure 4.1(b). Figure 4.2 shows a generic configuration, where multiple blocks can be cascaded together.

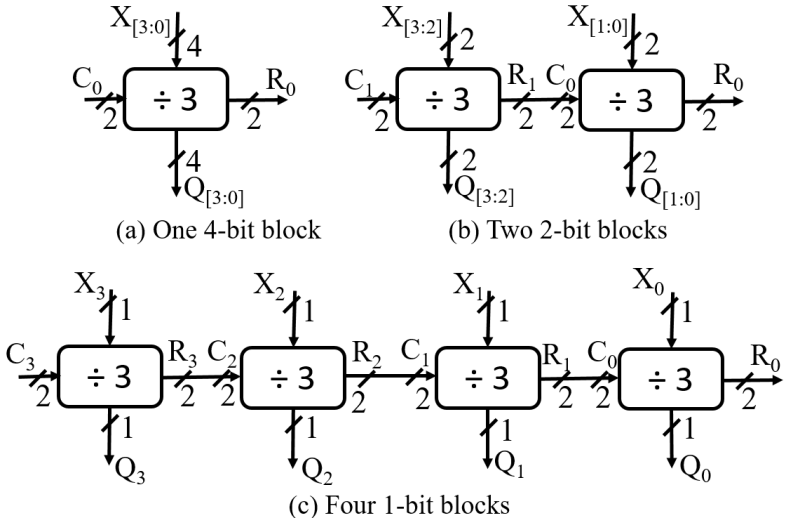



Figure 4.2: Generic divider block for $X$ divided by const. d = 3

Let $N$ be the number of blocks, and $k$ the number bits of the dividend $X$. If $k/N$ is not an integer, the most significant block will have some inputs appended with the required 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 4.1. 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 $R_i$

### 4.3.2 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<br>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

Figure 4.3: Divide-by-3 block specification tables

Figure 4.3 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 4.2, 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 Equation 4.2 is correct for every input assignment.

$$2C_i + X_i = 3Q_i + R_i \tag{4.2}$$

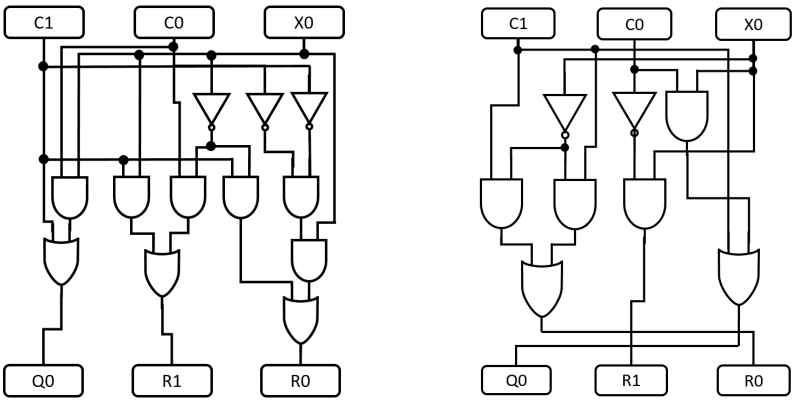
The coefficient 2 of $C_i$ comes from the fact that $C_i$ and $X_i$ form one word. In radix 2, the term $C_i = \sum_{k=0}^{n-1} 2^k C_{ik}$, where $C_{ik}$ refers to bit $k$ of block $i$. The coefficient 3 of $Q_i$ is determined by the value of the divisor, $D = 3$ in our example.

Equation 4.3 shows the generic bit-level equation for an arbitrary block $i$. It is derived from Eq. 4.2 by substituting for a given block $i$, $C_i = 2C_{i1} + C_{i0}$ and $R_i = 2R_{i1} + R_{i0}$, since $m = 2$.

$$4C_{i1} + 2C_{i0} + X_{i0} = 3Q_{i0} + 2R_{i1} + R_{i0} \tag{4.3}$$

The left-hand-side of Equation (4.3) is the *Input Signature*, $Sig_{in}$ while the right-hand-side is the *Output Signature*, $Sig_{out}$, as defined above in Section 4.3.1. In order to prove the functional correctness of the divider, we 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. The comparison of the polynomial expressions can be done using TEDs [8], BMDs [3], or similar canonical representations capable to compare two polynomials.

To illustrate the rewriting process, consider the one-block gate-level implementation of the division of $X$ by constant 3, shown in Figure 4.4. 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




a ) Correct circuit b ) Faulty circuit

Figure 4.4: 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 $Sig_{in} = 4C_1 + 2C_0 + X_0$.

of their respective gates, as defined by Eq. 3.1. 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 4.4(a), the resulting input signature should be

$$Sig_{in} = 4C_1 + 2C_0 + X_0 \tag{4.4}$$

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

$$Sig = (4C_1 + 2C_0 + X_0) - 2C_0C_1X_0 \tag{4.5}$$

which does not match the expected specification $Sig_{in}$ in Eq.(4.4).

The reason for this mismatch can be understood by analyzing the truth table in Figure 4.3. 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 4.5 reduces it to the expected $Sig_{in}$, proving that the circuit in Figure 4.4(a) correctly implements a divby3 computation.

### 4.3.3 Vanishing Monomials

For the purpose of this work, the monomials that correspond to invalid entries, such as $C_1C_0$ above, are defined as *vanishing monomials*, since in the functionally correct implementation they always evaluate to zero. The vanishing monomials help remove the redundant terms during the verification process. The vanishing monomials and the corresponding simplifying constraints can readily be derived from the architecture for a given value of the divisor, where invalid input assignments correspond to *don't-care* conditions; refer to Figure 4.3. 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 don'tcare set of the truth table, regardless whether these don't-care products are used during synthesis or not.*

*Proof.* Let $F$ be a 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 an 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 vari-

ables (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 (3.1): $a \vee b = a + b - a \cdot b$, 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 4.3. 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 3.1 as

$$Q = C_1 + C_0X - C_1C_0 - C_1C_0X \tag{4.6}$$

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-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 are 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-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 3.3. The logical sum of the three terms can be computed in the algebraic domain as

$$\begin{aligned} &C_2(1 - C_1)C_0 + C_2C_1(1 - C_0) + C_2C_1C_0 \\ &= C_2C_0 + C_2C_1 - C_2C_1C_0 \end{aligned} \tag{4.7}$$

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 (*don't care*) entries from the truth table.

2. Compute algebraic expression of the product terms associated with the *don't care* entries.

3. Remove the negative and redundant monomials.

### 4.3.4 Verification of a Multiple-Block Architecture

Figure 4.5 shows a block level diagram of an $X/3$ divider using a two-block architecture, each with $n = 2$ and $m = 2$.

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 $i$ and 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

$$Sig_{out} = 3(8Q_3 + 4Q_2 + 2Q_1 + Q_0) + 2R_1 + R_0 \tag{4.8}$$

The expression 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

$$Sig_{in} = 32C_1 + 16C_0 + 8X_3 + 4X_2 + 2X_1 + X_0 \tag{4.9}$$

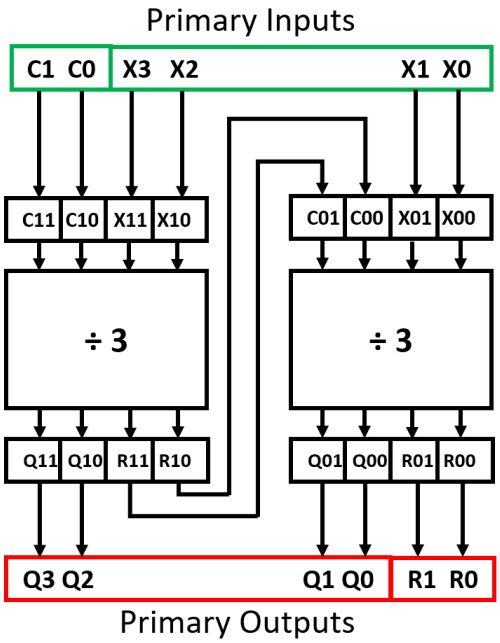



Figure 4.5: Division of a 4-bit divide-by-3 in a two-bit block divider circuit. Rewriting is applied in the opposite direction to the flow of the data.

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 4.3.3. 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 4.9.

$$Sig = F(V, C, X) + 32C_1 + 16C_0 + 8X_3 + 4X_2 + 2X_1 + X_0 \tag{4.10}$$

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 $C_{01}, C_{00}$ 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; it is often referred to as a *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 4.3.3, unless those signals are renamed by the synthesis process. The extraction, detection, and removal of vanishing monomials is fully automated for this methodology.

### 4.3.5 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 4.3. Then the gate-level implementation will be different, as shown in Figure 4.4(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 $R_{11}R_{10} = 1$, 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 4.5 is faulty and block $B_0$ is correct. The 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 4.5, 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.

## 4.4 Results and Analysis

The program implementing the described verification method for the constant divider 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 [14], and synthesized using ABC tool [31] 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.

### 4.4.1 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 4.1 shows the verification run time for the divide-by-constant iterative architecture for several block sizes with a 32-bit dividend $x$. The results are shown for divisors value of up to 283 and a 9-bit remainder.

Table 4.1: Verification results for the divide-by-constant divider circuit using our technique for: (1) Modular 1-bit block, 2-bit block; and 2) 4-bit block architecture with a 32-bit dividend $X$ (Figure 4.2). Time-out TO = 1200 s, Memory-out MO = 16 GB

| | | Modular Unroll | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | 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 |
| 3 | 2 | 712 | 0.06 | 1 | 0.06 | 665 | 2.26 | 895 | 0.90 |
| 11 | 4 | 1919 | 1.15 | 2 | 1.11 | 1917 | 2.23 | 4045 | MO |
| 17 | 5 | 1763 | 0.81 | 3 | .75 | 2236 | 5.83 | 2492 | MO |
| 31 | 5 | 1825 | 0.31 | 5 | 0.27 | 1676 | 0.85 | 10163 | MO |
| 61 | 6 | 3715 | 3.50 | 8 | 3.56 | Memory Out | | | |
| 89 | 7 | 4520 | 13.5 | 5 | 16.71 | | | | |
| 113 | 7 | 3652 | 6.68 | 7 | 7.21 | | | | |
| 139 | 8 | 5542 | 27.9 | 7 | 94.75 | | | | |
| 191 | 8 | 4736 | 9.67 | 5 | 11.36 | | | | |
| 251 | 8 | 6410 | 110.4 | 5 | 113.5 | | | | |
| 257 | 9 | 6549 | 22.56 | 7 | 23.0 | | | | |
| 283 | 9 | 8951 | 643.8 | 9 | 638.4 | | | | |

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 the 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 4.1. 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 circuit, 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 4.1 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.

### 4.4.2 Flat Unroll Architecture

In the *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 $2^9$ shown in Table 4.2.

### 4.4.3 The Restoring Constant Divider

We also tested an alternative architecture based on a standard *restoring divider* [29], in which the divisor $D$ has been hardwired to a particular constant, Figure 5.4. The restoring divider has been implemented and synthesized using ABC [31] as a tool. Constants from the bits of $D$ are propagated through and used to optimize the

Table 4.2: Verification results for the divide-by-constant divider circuit using our technique for Flat-Unroll architecture with a 9-bit dividend $X$ (Figure 4.2). Time-out TO = 1200 s, Memory-out MO = 16 GB;

| | | Flat Unroll | |
|---|---|---|---|
| Divisor | # Rem. Bits | #Gates | Time (s) No Bugs |
| 3 | 2 | 105 | 2.96 |
| 11 | 4 | 300 | 42.6 |
| 17 | 5 | 192 | 6.68 |
| 31 | 5 | 282 | 169 |
| 61 | 6 | Memory Out | |
| 89 | 7 | | |

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 22 bits, resulting in segmentation fault over 24 GB. Table 4.3 shows the results for such a restoring divide-by-constant circuit for the 22-bit dividend.

Table 4.3: Verification results for the divide-by-constant divider circuit using our technique for Restoring Constant Divider with a 22-bit dividend $X$. TO = 1200s, MO = 24GB. SF = Segmentation Fault.

| | | Restoring | |
|---|---|---|---|
| Divisor | # Rem. Bits | #Gates | Time (s) No Bugs |
| 3 | 2 | 183 | 0.01 |
| 11 | 4 | 488 | 2.42 |
| 17 | 5 | 538 | 4.13 |
| 31 | 5 | 638 | 10.4 |
| 61 | 6 | 726 | 9.12 |
| 89 | 7 | 726 | 10.9 |
| 113 | 7 | 765 | 3.82 |
| 139 | 8 | 766 | 71.1 |
| 191 | 8 | 892 | SF |
| 251 | 8 | 880 | 14.53 |
| 257 | 9 | 786 | 57.2 |
| 283 | 9 | 871 | 89.9 |

### 4.4.4 Simulation Based Verification

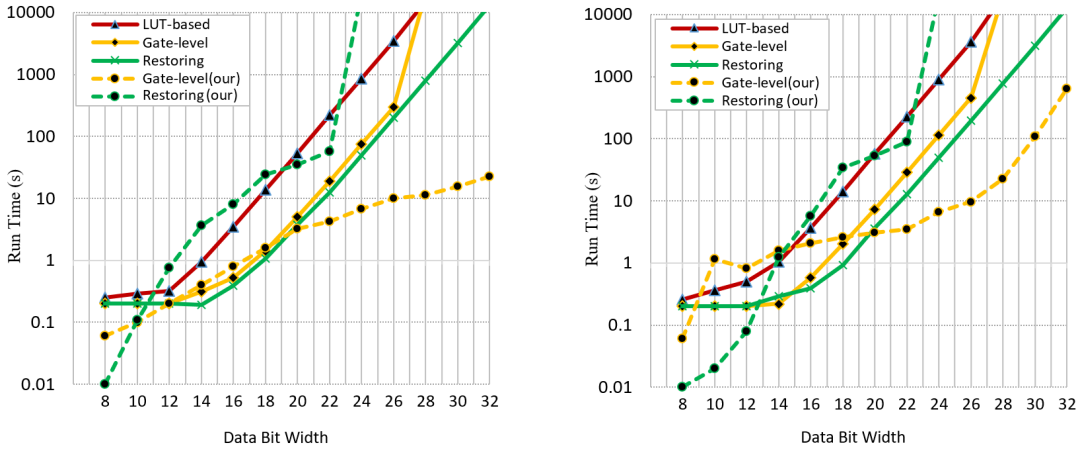
We simulated the divide-by-constant dividers(different architectures) for different size of divisors $D$ and dividend $X$, ranging from $2^8$ to $2^{32}$. We used Modelsim SE 10.0 on a Xeon 5650 processor with 6 cores (2.67GHz), 24 GB of RAM, and 350 GB free hard disk space.

Figure 4.6 shows the simulation results for $D = 257$ and 283 (bold lines) and compares their results with the rewriting approach (dotted line) described in this section. The following cases are considered:

- LUT-based implementation generated by FloPoCo [14];
- Gate-level implementation of a LUT, synthesized with ABC; and
- Restoring constant divider implemented with ABC.

As shown in Figure 4.6, 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 (Gate-level) 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 for LUT-based and gate-level but lacks in performance, compared to the simulation results for the restoring constant divider.




a ) Divisor = 257 b ) Divisor = 283

Figure 4.6: Exhaustive simulation run time for divisors D=257 and D=283 for different implementations, as a function of the dividend bit-width. Dotted Lines show equivalent for our rewriting technique.

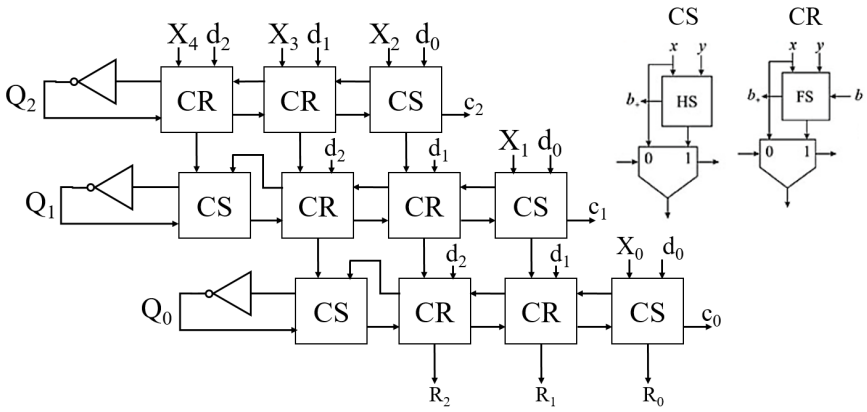



Figure 4.7: Restoring Generic 3-bit Divider [40].

### 4.4.5 The Restoring Generic Divider

This section demonstrates the applicability of our approach to the implementation of constant divider by a generic restoring divider, shown in Figure 4.7. Table 4.4 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. The reason being

that the underlying vector data-structure is not able to contain the prohibitively large polynomial. Under this methodology, the divider circuit is heavily optimized and hence any boundary information between different modular blocks is lost, as shown in Table 4.4.

Our constant divider methodology is not scalable to generic dividers as of yet and currently the simulation based verification outperforms our technique. However, it still demonstrates the significance of its applicability to generic divider circuits. Chapter 5 presents a layered rewriting strategy to avoid this memory explosion issue to some extent.

Table 4.4: 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 |
| 19 | 1048576 | 4325 | SF:240 | TO |

# CHAPTER 5

# FORMAL VERIFICATION OF HARDWARE DIVIDERS USING LAYERED VERIFICATION STRATEGY

## 5.1 Introduction

In this work, we concentrate on combinational dividers: integer divider, and the fractional fixed point divider, essential components of floating point division unit. This chapter is organized as follows. Section 5.2 develops an algebraic verification technique for the fractional and integer dividers. Section 5.3 presents some preliminary results and conclusions.

## 5.2 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 [22], as briefly reviewed previously. 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 a 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 impressive, such a reverse engineering method, based on a strictly structural pattern matching, does not accomplish the *functional* verification per se. It may happen that some components do not match the expected logic, but the circuit may work correctly as an ensemble. Or, that some logic is represented without XORs. As an example, Figure 3.1 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.

### 5.2.1 Functional Verification Model

Fractional divider is an essential part of hardware for floating point division. The dividend $X$ and the divisor $D$ are normalized by pre-shifting to comply with the IEEE 754 standard. Figure 6.7 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 the expression is equivalent to (or reduces by algebraic rewrting to) the dividend $X$.

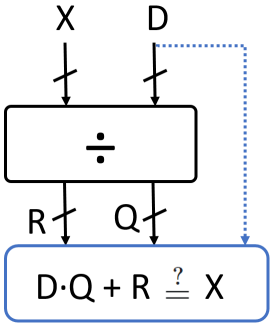



Figure 5.1: Functional verification model of the divider.

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 and the ABC system [31] was used 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 6.7. In this approach the output signature polynomial, $Sig_{out} = D \cdot Q + R$, 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 $R < D$ [29].

### 5.2.2 Fractional vs. Integer Divider

We now demonstrate that the fractional divider can be used for integer division [29]. In fact, it is only a matter of interpretation of the result, whether a fractional or an integer division is performed by the hardware, as demonstrated by the example below (see Figure 5.2). 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., $X = 0.x_{-1}x_{-2}... \; x_{-n}$, in accordance

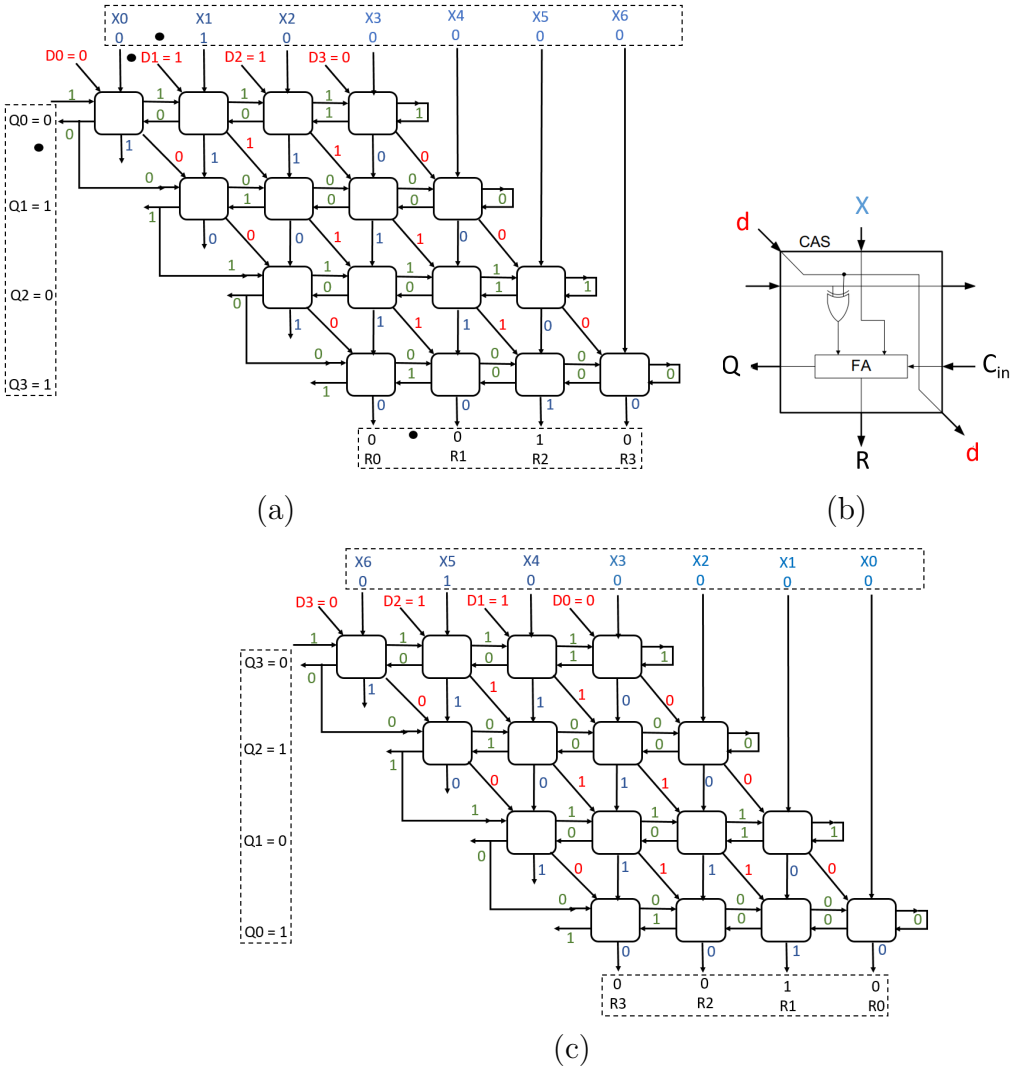



Figure 5.2: Nonrestoring 7-4 divider ($n = 3$): a) Fractional divider; b) Controlled Add/Subtract (CAS) block; c) Integer divider

with the IEEE standard. We assume that the bit-widths are sized as required to avoid an overflow or an underflow, i.e., $X$ has size $2n + 1$ and $D, Q, R$ are of size $n + 1$, including 0 before the fractional dot [29].

- **Fractional Divider** (Figure 5.2(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:

$$X = 0.x_1....x_6, \quad D = 0.d_1d_2d_3, \quad Q = 0.q_1q_2q_3, \quad R = 0.r_1r_2r_3.$$

To illustrate this issue, consider the following example:

$$X = (0.100000)_2 = 1/2 \text{ and } D = (0.110)_2 = 3/4$$

which satisfies a non-overflow condition, $X < D$. The result is:

$$Q = (0.101)_2 = 5/8, R = (0.010)_2 = 1/4,$$

as shown in Figure 5.2(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 5.2c): 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,

$$X = 0x_6...x_2x_1 = (0100000)_2 = 32,$$

$$D = 0d_3d_2d_1 = (0110)_2 = 6,$$

$$Q = 0q_3q_2q_1 = (0101)_2 = 5,$$

$$R = 0r_3r_2r_1 = (0010)_2 = 2.$$

The result is correct: $X = D \cdot Q + R = 6 \cdot 5 + 2 = 32$ and 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.

### 5.2.3 Layered Rewriting

When rewriting output $Q$ or $R$ of the divider, the final polynomial at the primary inputs, $Sig_{in}$, 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 (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, one row 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 $D$.

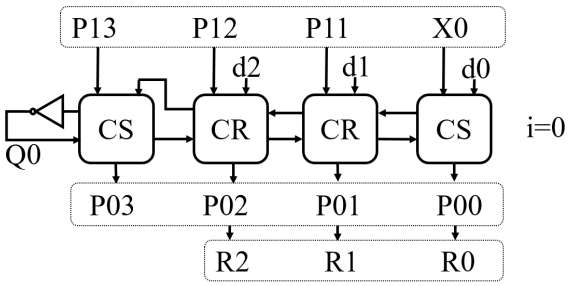



Figure 5.3: Single layer of the restoring divider used in rewriting.

This approach can be justified by observing that the 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 [22]). 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,. . . ,$n$-1); $P_{i,n+i} = 0$ ($i$ = 0,. . . ,$n$); $P_{n,k} = X_k$ ($k$ = $n$,. . . ,2$n$). 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 $X$.

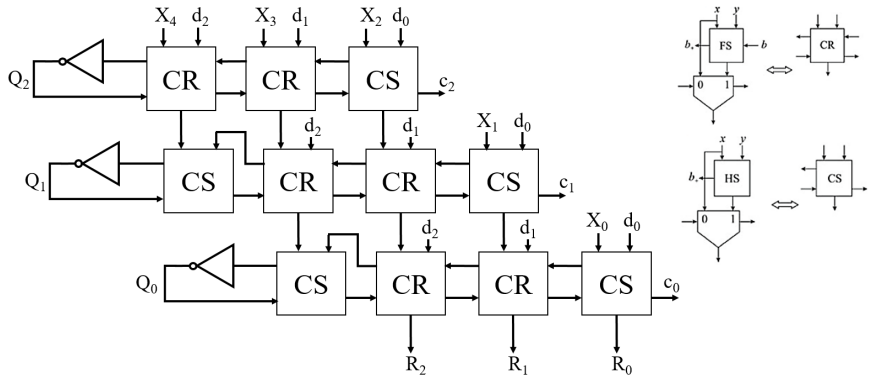



Figure 5.4: Restoring integer divider [40].

To illustrate the idea, the following rewriting is applied to the restoring divider shown in Figure 4.7.

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

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

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

By adding the above equations, we obtain:

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

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

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 then 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.

## 5.3 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 × 2, with 31 GB of memory. The circuits were generated by an in-house restoring divider generator tool and synthesized onto standard cells by the Synopsys Design Compiler (DC).

Table 6.3 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) An 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 [22], 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, especially when both the circuit under verification and the reference design exhibit similar structure.

Table 5.1: 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 |

## 5.4 Conclusion

This chapter presented a verification method for gate-level integer divider circuits based on algebraic rewriting. The verification relies on creating a "reverse divider unit" that defines the output signature needed for algebraic rewriting. This approach can, in principle, be used for the entire circuit, or be applied to individual layers of the divider circuit in the array architecture. It has been shown that the same approach can be used for fractional arithmetic, since the integer and fractional dividers share the same architecture.

A notable advantage of this method is that it verifies the divider 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 reference (golden model); alternatively, it can be used for designs that do not have a well defined or trusted reference. The layered technique can be easily parallelized and applied to other arithmetic circuits with similar architectures. It also enables debugging of the circuit at the single-layer granularity.

The algebraic rewriting advocated in this work provides an efficient method for verifying multipliers, adders, subtractors, and multiply-accumulate circuits. However, the method is not scalable for the verification of divider circuits, because of their "non-standard" characteristic function, $X = Q \cdot D + R$, which is *not* a closed-form expression. Specifically, input $X$ is expressed in terms of outputs and another input $D$, which, combined with its non-linearity, limits the efficacy of the rewriting. As a result, dividers do not benefit from rewriting as a means of verification.

In order to learn how to overcome this limitation and devise a method for efficient verification, we digress for a moment and look at a class of arithmetic circuits similar to that of divider, but with a different and an easier to handle characteristic function. In the next chapter, we consider a square-root array circuits (SQRT) that belongs to the same iterative/subtract architecture family as divider, with a characteristic function $X = Q^2 + R$. It has a single input $X$ and outputs $(Q, R)$, which provides a clean output signature, $Sig_{out} = Q^2 + R$, containing only the output terms. We analyze the architecture of such circuits and present another original method to verify large bit-width SQRT circuits, avoiding the limitation of the standard algebraic rewriting technique. We will then return to dividers and apply similar technique for the divider circuits.

# CHAPTER 6

# SQUARE-ROOT AND DIVIDER CIRCUIT VERIFICATION USING HARDWARE REWRITING

Square-root algorithm plays a major role in many domains, including computer arithmetic, computational geometry, embedded systems, and other special purpose applications. It belongs to the class of dividers and it is one of the most complex arithmetic operation to implement and verify [25]. Square-root computation of a positive number has numerous applications, including Euclidean Norm as well as in the generalizations of the Hilbert Spaces. It defines an important concept of standard-deviation (root of a variance), and has a major application in quadratic formula to compute roots for quadratic equations and fields [29][33]. In this chapter, we first present a verification methodology for square-root circuits using standard rewriting, and then introduce a new concept of *HardwareRewriting*, and then extend it to divider.

## 6.1 Characteristic Function of Square-Root

In order to apply algebraic rewriting to the SQRT circuit, we need to define the input and output signatures for the circuit and the characteristic function of the square rooter. The obvious inputs and outputs of such a circuit are $X$ (the radicand) and $Q$ (the root). The characteristic function of the SQRT operation, $Q = \sqrt{X}$, can then be described by:

$$X = Q^2 + R, \quad \text{with } R \leq 2Q \tag{6.1}$$

where $R$ is the remainder (or residue).[1] The remainder $R$ is needed in the expression so that the arithmetic function of the circuit can be represented as a strict equality (characteristic function) rather than an approximation. Its role in Equation (6.1) is similar to that in the division, $X = QD + R$, where $Q$ is the quotient, $D$ the divisor, and $R$ the remainder. Depending on the square-root extraction algorithm implemented by the circuit, the remainder $R$ can be positive (in the restoring algorithm) or of any sign (in the non-restoring algorithm). Using a simple integer example with $X = 13$, the solution to $Q = \sqrt{(13)} \approx 3.6055$ can be either $Q = 3$ and $R = 4$; or $Q = 4$ and $R = -3$. Typically, $R > 0$ is preferred as a standard solution, but negative remainders can also be used, depending on the required precision (as explained in more detail in Section 6.2.1.) However, it is important to note that in both cases, equation (6.1) is satisfied: $X = Q^2 + R = 3^2 + 4 \ = \ 4^2 - 3 = 13$.

Unfortunately, most hardware SQRT implementations do not provide the remainder, so it needs to be generated for the purpose of our verification approach. Section 6.4.1 describes how such a remainder is generated and used in the verification.

## 6.2 Integer vs. Fractional SQRT

Consider an *integer* SQRT circuit, with radicand $X$ and root $Q$ being integers. To verify the SQRT circuit, we need to prove that the equation $X = Q^2 + R$ is satisfied for every input assignment and that $R \leq 2Q$. It can be shown that $R$ is also integer and result Q is unique. Furthermore, for the restoring algorithm, $R$ is positive. In principle, verification of $X = Q^2 + R$ can be achieved by performing algebraic rewriting discussed in Section 3.1, by rewriting $Sig_{out} = Q^2 + R$ at the outputs $Q, R$, into $Sig_{in} = X$ at the input $X$. The word-level symbols $X, Q, R$ are represented as polynomials in binary variables (bits) with integer coefficients $2^i$. Specifically, for n+1

[1]The reason for the requirement $R \leq 2Q$ is that for $R \geq 2Q+1$, we have $X = Q^2+R \geq (Q+1)^2$, hence the result would be incorrect [33].

bit vectors: $X = \sum_{i=0}^{n} 2^i x_i$, $Q = \sum_{i=0}^{n} 2^i q_i$, and $R = \sum_{i=0}^{n} 2^i r_i$. We will return to the verification of the requirement $R \leq 2Q$ later in Section 6.4.2.

It has been shown in the literature on computer arithmetic [29] and [33] that the integer SQRT circuit will also perform the SQRT function with the *fractional* radicand $X$ and root $Q$. In fact both circuits use exactly the same algorithm and the same architecture. Such fractional circuits are routinely used for floating point calculations and hence it is important to develop a method for their verification. The two designs will only differ in the representation of the operand X and the result Q, with the fractional circuit having binary representation $X = [x_0.x_{-1}....x_{-n}] = \sum_{i=0}^{-n} 2^i x_i$, and similarly for $Q$. In the fractional case, the number of bits of $R$ is $2n+1$. The radicand of the integer circuit is $X = [x_0 x_{-1}....x_{-n}] = 2^0 X_0 + 2^{-1} X_{-1} + ... + 2^{-n} X_{-n} = \sum_{i=0}^{-n} 2^i x_i$, where $x_0$ is the most significant bit and $x_{-n}$ the least significant bit; and similarly for $Q$ and $R$.

To avoid fractional coefficients, the fractional representation of $X$ and $Q$ can be simply multiplied by $2^n$, where $n$ is an even number of bits ($n = 2k$) of $X$ and $Q$. That is,

$$\sqrt{(}x_0 \ . \ x_{-1}....x_{-n}) = \sqrt{(}x_0 x_{-1}....x_{-n} \cdot 2^{-n})$$

$$= 2^{-k} \sqrt{(}x_0 x_{-1}....x_{-n}) \cdot$$

This simple normalization gives us the right to apply the algebraic rewriting concept (as well as the hardware rewriting developed later) to fractional SQRT circuits using polynomials with positive coefficients.

### 6.2.1 Restoring vs Nonrestoring SQRT Verification

We close this section by making remarks about the application of our approach to all versions of SQRT designs, including integer and fractional, both restoring and nonrestoring.

We illustrate this point with the following example:

$$X = 118/64 = 1.110110, \text{ where } \sqrt{(}118/64) \approx 1.3578....$$

- **Fractional Restoring**:

$$Q = 86/64 = 1.010110; R = 156/64^2 = 0.000010011100.$$

Equation (6.1) is satisfied: $X = (86/64)^2 + 156/64^2 = 118/64$

- **Fractional Nonrestoring**:

Without correction,

$$Q = 87/64 = 1.010111,\ R = -17/64^2 = 1.111111101111$$

and Equation (6.1) is satisfied:

$$X = (87/64)^2 - 17/64^2 = 118/64$$

With correction:

$$Q = 86/64 = 1.010110, \quad R = 156/64^2 = 0.000010011100$$

In this case Equation (6.1) is also satisfied: $X = (86/64)^2 + 156/64^2 = 118/64$. This result is obtained by rounding down (truncating) the initial result, while the original one without correction (with $R = -17/64^2$) is actually closer to the real solution and may be more desirable. In any case, regardless of the correction or rounding, the characteristic equation $X = Q^2 + R$ is satisfied and will be used as an invariant in our verification method. Recall that the **integer** solution for $\sqrt{(}118) = \sqrt{(}01110110)_2$, with $Q = 10_{10} = (1010)_2$ and $R = 18 = (10010)_2$, also satisfies Equation (6.1).

## 6.3 SQRT Verification Technique

Figure 6.1 shows a 7-bit modular square-root circuit architecture with the two outputs, $Q$ and $R$. The basic components of the circuit are the Controlled-Subtract CS and CR blocks. Here, CS is basically a half-subtractor and CR a full-subtractor (similar to a half adder and a full adder). The circuit has an iterative-subtract architecture composed of a number of blocks organized in an iterative fashion: each row computes a single bit of $Q$ on the left, and the partial remainder $R$ that is fed to the lower row. To verify the SQRT design, we need to prove that the equation $X = Q^2 + R$ is satisfied for every input assignment. The word-level symbols $X, Q, R$ are represented in terms of their bits as: $X = \sum_{i=0}^{2k-1} 2^i X_i$, $Q = \sum_{i=0}^{k-1} 2^i Q_i$, and $R = \sum_{i=0}^{k-1} 2^i R_i$.

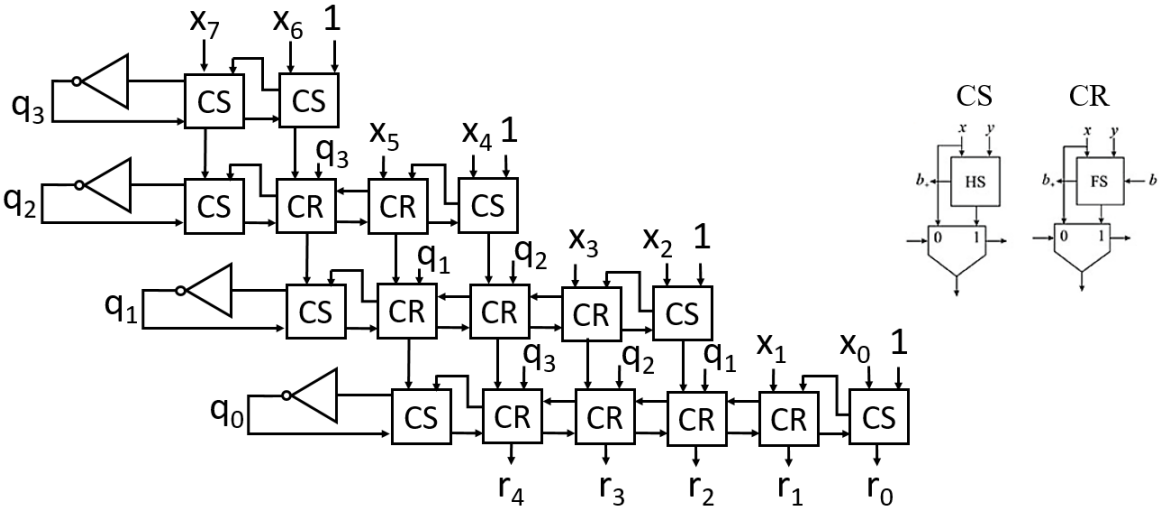



Figure 6.1: A restoring SQRT circuit with a 7-bit radicand, 4-bit quotient, and a 5-bit remainder.

In this chapter, we consider an *integer* SQRT circuit. However, it has been shown in [29] that exactly the same architecture will also perform the SQRT function on a *fractional* SQRT circuit. This is similar to the case of the divider as already shown in Section 5.2. The two designs will only differ in the representation of the input and

the results, with the fractional circuit having representation $X = [0.x_1....x_{n-1}]$ and in the integer case as $X = [x_{n-1}....x_0]$; and similarly for $Q$ and $R$.

The internal working of the SQRT circuit is similar to the divider: it uses iterative subtraction, except that the divisor is changing at each step. We can therefore apply the verification approach used in the divider verification, by rewriting the output signature $Q^2 + R$ to the input $X$.

To test this idea, we generated a square root circuit with outputs $Q, R$ using the ABC tool [31], so that the equation X = $Q^2 + R$ should hold for a correct circuit.

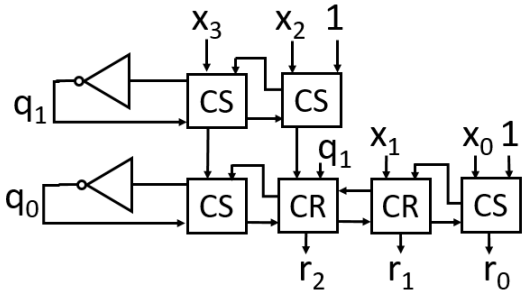



Figure 6.2: A restoring SQRT circuit with a 4-bit radicand, 2-bit quotient, and a 3-bit remainder.

For example, the circuit with a 4-bit radicand $X$ and a 2-bit quotient, $Q$, as shown in Figure 6.2, has the following output signature:

$$Sig_{out} = (2q_1 + q_0)(2q_1 + q_0) + (8r_3 + 4r_2 + 2r_1 + r_0), \tag{6.2}$$

which in a reduced form is

$$Sig_{out} = (4q_1 + 4q_0q_1 + q_0) + (8r_3 + 4r_2 + 2r_1 + r_0). \tag{6.3}$$

This signature is then rewritten to the outputs, with expected expression being:

$$Sig_{in} = (8X_3 + 4X_2 + 2X_1 + X_0) = X \tag{6.4}$$

While conceptually the verification can be accomplished by algebraic rewriting described by Equations 6.2, 6.3, and 6.4, the resulting $Sig_{in}$ can still be too complex or the computation may not terminate. To address this problem, we reconstruct the circuit by explicitly creating the circuit that computes the residual $R$. The reconstructed part is obtained by generating the residue circuit $R = X - Q^2$, whose inputs are $X, Q$ and output is $R$, further explained in Section 6.4.1. The reconstructed circuit is shown in red in the upper-right part of Figure 6.5. At this point the rewriting could potentially be done on the circuit with the original output $Q$ and the newly generated $R$. However, such a "standard" rewriting is not efficient, mostly because the output signature $Sig_{out} = Q^2 + R$ is nonlinear and the intermediate polynomials can become prohibitively large causing memory overload.

Table 6.1 shows the verification runtime for the square-root restoring architecture, using the rewriting tool originally developed for multipliers and also applied to dividers [54]. As expected, the designs with a radicand greater than 8 bits cannot be verified with this naïve approach due to memory issue. The non-linear signature $(Q^2 + R)$ causes the intermediate polynomial to become so large that the rewriting does not converge even with 22 GB of memory. The next section describes an original method to fix this problem.

Table 6.1: Square Root verification results using standard-Style rewriting

| Radicand Bits | Time (s) |
|---|---|
| 3 | 0.00 |
| 4 | 0.01 |
| 5 | 0.08 |
| 6 | 1.93 |
| 8 | 712 |
| 10 | T/O |

Currently, as can be seen from the preliminary results, the memory explosion during the rewriting process prevents the verification to succeed for large bit-width

circuits. In order to overcome this issue, a novel concept of hardware-based rewriting is introduced next.

## 6.4 Hardware Rewriting for SQRT Verification

### 6.4.1 Remainder Generation

We now come to the critical issue: in order to prove the circuit using equation $X = Q^2 + R$, we need the remainder $R$. However, a typical SQRT circuit provides only a single output $Q$. To solve this problem, we restore the "missing" residual output by constructing a circuit $R_{ref} = X - Q_{ref}^2$, with inputs $X$ and $Q_{ref}$, and the needed output $R_{ref}$, as shown in Figure 6.3. Input $Q_{ref}$ is provided by the reference SQRT design, labeled $Ref\sqrt{X}$ in the figure, a golden model known to be correct.

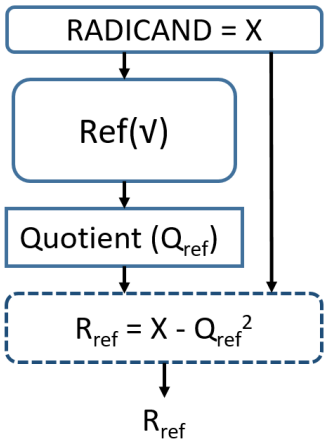



Figure 6.3: Residue generation using a Reference Design.

The reconstructed circuit is shown in Figure 6.4. At this point the rewriting can be done on the modified circuit with the original output $Q$ and the newly generated $R_{ref}$. However, such a simple-minded, standard rewriting is not efficient, mostly because the output signature $Sig_{out} = Q^2 + R_{ref}$ is nonlinear and the intermediate polynomials can become prohibitively large. Our experiments show that the designs with a radicand greater than 8 bits cannot be verified with this approach due to a

memory overload. The main culprit seems to be the non-linearity of the signature $(Q^2 + R)$, which causes the intermediate polynomial to grow fast, so the rewriting does not converge even with 22 GB of memory.

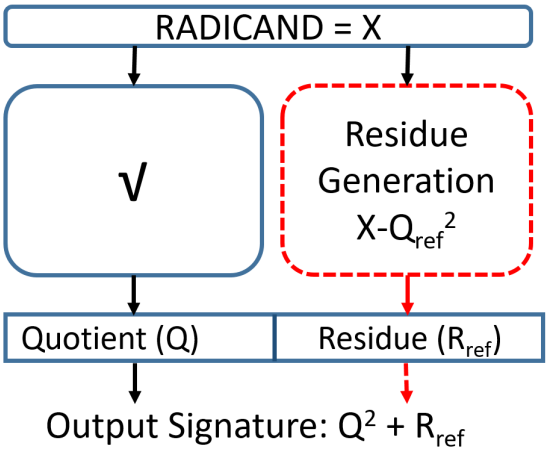



Figure 6.4: Conceptual standard rewriting.

The failure of applying algebraic rewriting is actually not surprising; algebraic rewriting has been successfully applied to complex adders and multipliers, which are characterized by a linear output signature in its output bits $z_i$, determined by the encoding of the output bits. For example, a multiplier $Z = A \cdot B$ has a clearly defined linear signature $Sig_{out} = Z = \sum_{i=0}^{n-1} 2^i z_i$. However, the square-root circuit has a non-linear signature, $X = Q^2 + R$. Rewriting a non-linear signature is much more memory intensive, as demonstrated above, and a standard algebraic rewriting proves largely ineffective for these circuits.

The reader may ask at this point, and rightly so: "why not compare $Q_{ref}$ directly to output $Q$ of the design under verification"? In fact we tried it using SAT technique, but the results were very disappointing, as shown in Table 6.2 Section 6.6. The next section describes an original method to solve this problem by extending algebraic rewriting to synthesis to be performed directly on the hardware.

### 6.4.2 Hardware Rewriting

We now introduce the concept of a *Signature Linearizer*, a circuit that transforms a nonlinear signature $Q^2+R_{ref}$ into a linear one, in an attempt to enable the rewriting.

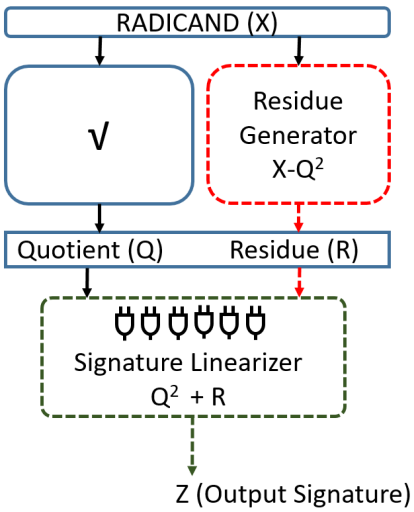



Figure 6.5: Hardware rewriting

Figure 6.5 shows the basic concept of our approach. The upper part of the figure is the circuit that computes $Q^2 + R_{ref}$ described earlier and shown in Figure 6.4. Since algebraic rewriting of such a polynomial is inefficient, we introduce another circuit that computes function $Z = Q^2 + R_{ref}$, derived from the output $Q$ of the original circuit, and the reference remainder $R_{ref}$, generated from the reference $Q_{ref}$. Such constructed output $Z$ of the circuit is a *linear* polynomial in its bit variables, $Z = \sum_{i=0}^{n} 2^i z_i$.

As shown in Figure 6.5, the combined circuit has input $X$ and output $Z$, which for the correct SQRT circuit should satisfy $Z = X$. In principle, this equivalence could be checked by algebraic rewriting of the linear polynomial $Z$ all the way to the primary inputs, in an attempt to obtain $X$. However, it turns out that such an algebraic rewriting is still inefficient, since the internal polynomials can become prohibitively large.

Instead, we perform an implicit *hardware rewriting* by resynthesizing the entire circuit that computes $Z$ as a function of $X$. This resynthesis process uses a state-of-the art synthesis tool, ABC [31], that includes structural and functional hashing ($strash$), functional simplifications ($fraig$), and a final resynthesis step ($dch$), all using an AIG data structure. In a functionally correct SQRT circuit, the newly constructed circuit should become *redundant* and reduced to a set of wire connections between $X$ and $Z$, provided that the added parts (residue generator $R = X - Q^2$ and the linearizer, $Z = Q^2 + R$) are correct.

We cannot, however, rely on resynthesis as a formal proof; if the synthesis does not simplify the design to a redundant state (wires/buffers), we cannot conclude that the circuit is incorrect. In this case, those portions of the circuit that are not reduced to wires can be verified using SAT. Specifically, for each bit $Z_i$ that does not trivially reduce to $X_i$, we create an XOR/miter and check if the result is unSAT (or, equivalently if the output of an XOR for each pair of bits $X_i, Z_i$ is 0).

If the result is unSAT, the circuit is correct; otherwise, the SAT solution provides a counter example that can be used to identify the bug. This "hardware rewriting" idea is illustrated in Figure 6.6.

It should be emphasized that this verification method is sound only if all parts of the circuit, including the added residual and linearizer circuits, are functionally correct. If the result is unSAT, one may safely conclude that the added circuits are correct as well. The chance of the add-ons being faulty in such a way that they mask the error in the original SQRT circuit is highly unlikely. On the other hand, the presence of an error in any part of the circuit will result in a satisfiable solution to the SAT problem, and it wouldn't be clear if the error comes from the tested circuit or from the added components. However, correctness of these "add-ons" is guaranteed by construction: first the reference remainder, $R_{ref}$, is constructed from a reference result $Q_{ref}$ of a golden reference SQRT circuit. Then the linearizer circuit

$Z = Q^2 + R_{ref}$ is obtained using some golden (certified) squarer $Q^2$ or a multiplier $Q \cdot Q$ of input $Q$, and a golden adder that adds $Q^2$ to $R_{ref}$. In this case, a satisfiable solution will correctly indicate that the SQRT circuit is faulty, avoiding false negatives.

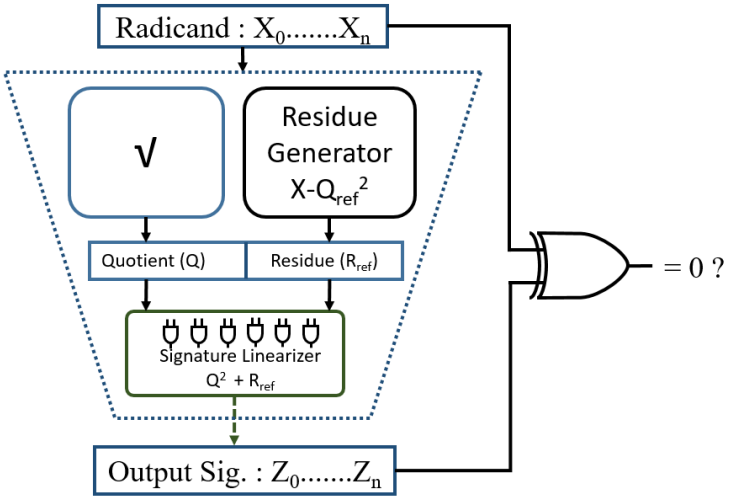



Figure 6.6: Final verification using SAT: check if $\forall i, X_i = Z_i$.

Similar argument applies to verifying the constraint $R \leq 2Q$ in equation (6.1), since the circuit for $R_{ref}$ in Figure 6.3 is derived from a correct reference design $Q_{ref}$. If this constraint is not satisfied by the SQRT under verification, the output $Q$ of the circuit in conjunction with the correct reference remainder $R_{ref}$ would not match $X$.

The results shown in Section 6.6 demonstrate that our methodology works well on both bug-free and buggy designs. Finding a source of a bug and performing the debugging is a separate and challenging problem, which will be considered in future work.

## 6.5 Divider Verification

In this section, we analyze a typical architecture of an arithmetic divider. Its internal structure is similar to that of a SQRT circuit (both are based on a standard shift and subtract algorithm), except that here the divisor $D$ remains fixed during

each shift/subtract iteration. The division can also be based on a restoring or a non-restoring algorithm, as in SQRT. The essential difference between the two, however, comes from their mathematical model, the *characteristic function* used in the verification. Specifically, SQRT function $Q = \sqrt{X}$ has a simple closed-form formula $X = Q^2 + R$, with the input operand $X$ on the left-hand side of the equation and the outputs $Q, R$ on the right, as developed in Section 6.3. In this case, the output signature is clearly defined as $(Q^2 + R)$, and can be directly used in the algebraic or hardware rewriting. In contrast, the characteristic function of a divider is more complex, as it is governed by the following equation:

$$X = D \cdot Q + R, \quad \text{with } R < D \tag{6.5}$$

where $X$ is the dividend, $D$ the divisor, and $Q, R$ the quotient and remainder, respectively. Note that the right-hand side of the equation, in addition to the outputs $Q, R$ also contains the input $D$; that is, the inputs $X, D$ appear on both sides of the equation. This makes it difficult to determine what the output signature is, hence questioning the very algebraic rewriting approach advocated here. To address this issue, we propose the method in [49], which considers the entire right-hand side of Equation (6.5), including the input $D$, as the output signature and attempts to verifies if it reduces to $X$. The remainder of this section describes how to effect this verification.

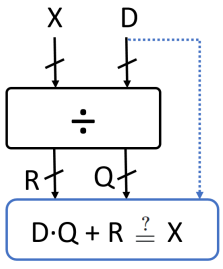



Figure 6.7: Divider verification model.

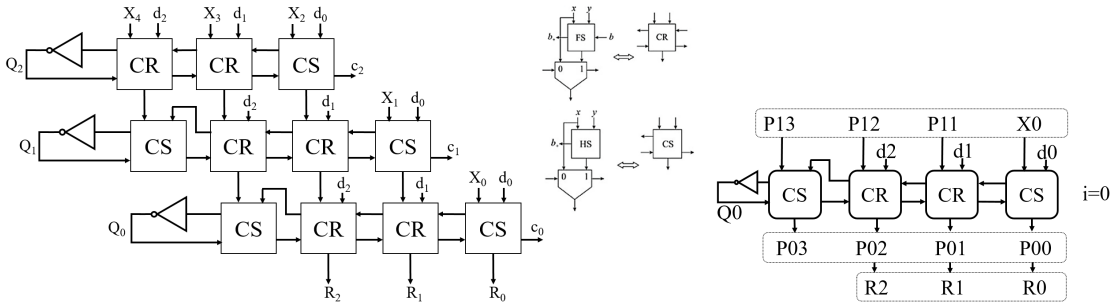



Figure 6.8: Restoring integer: divider [40]; a) Layered architecture b) Single layer used in rewriting [48].

### 6.5.1 Verification Model: SAT-based vs Algebraic Rewriting

Figure 6.7 shows an abstract model of the divider verification employed in this work. The upper part of the diagram is the divider under verification. The lower box "reverses" the division $X/D$ by computing Q·D + R from the quotient $Q$ and the remainder $R$, produced by the divider. The goal is to prove that the computed result matches the original dividend $X$, i.e. X = Q·D + R, and the condition that $R < D$ is satisfied.

One can solve this problem by creating a circuit Y = Q·D + R and checking the equivalence between its output $Y$ and the dividend $X$. In principle, this can be done using a standard SAT technique: create a miter between the dividend input $X$ and output $Y = QD + R$ of the circuit and check if the CNF formula of the resulting miter circuit is unsatisfiable (unSAT). As a proof of concept, we tested this method on both restoring and non-restoring array dividers using the ABC system [31], with *MiniSAT* [44] as the underlying SAT engine. The solution required only 4.4 seconds to prove a 16-bit $X$ with a 8-bit $D$ divisor, but the computation timed out after 3600 sec for larger instances. Dividers with dividend bit-widths greater than 16 bits could not be verified using this method.

Another approach is to use algebraic rewriting, described in Section 3.1. It uses polynomial $(QD + R)$ as the output signature and transforms by rewriting all the

way to the primary inputs $X$ of the divider. However, as shown in Section 4.4.5, this technique is still inefficient and suffers from high memory complexity. To address this problem, a *layered rewriting* technique, described earlier in Section 5.2.3 has been briefly described in the next section for convenience.

### 6.5.2 Layered Algebraic Rewriting

Layered rewriting is a technique of algebraic rewriting applied to each row (layer) associated with a singe output bit $q_i$ of the divider; refer to Figure 6.8a). The layered approach can be justified by noting that the logic between two adjacent rows is typically not optimized during synthesis and the partial remainder signals are preserved during synthesis (Theorem 2 of [22]). Synthesis tools, such as Synopsys DC, allow IC designers to impose a *maintain hierarchy* directive to make physical synthesis more efficient. Even if the design is optimized during synthesis, one can revert the changes by using data contained in the setup verification file (e.g., SVF, maintaned by Synopsys DC). The circuits can be synthesized across the add/sub modules of each layer, but not vertically across the rows.

We explain the layered approach for an architecture of the divider circuit with $2n+1$ bits of $X$ and $n+1$ bits of $D, Q$ and $R$, as shown in Figure 6.8a). Algebraic rewriting is applied to each layer corresponding to one bit $q_i$ at a time. 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. The output signature for a given layer is $q_i D + P_i$, and the expected input signature of row $i$ is $P_{i+1}$, with the boundary condition $P_n = X$, where $n$ is the number of (fractional) bits of $R, Q$ and $D$.

Rewriting starts at the remainder output $R$ and rewrites one row of the add/subtract circuitry at a time. The input signature computed for a given layer becomes an output signature for the next layer. Specifically,

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

where $D = \sum_{k=0}^{n} 2^k d_k$, and $P_i = \sum_{k=0}^{n} 2^k P_{i,(i+k)}$. Here $P_{i,j}$ denotes a bit of the partial remainder $P_i$ in row $i$ and 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 layer $i$, 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 $X$. To illustrate the idea, the following rewriting is applied to the restoring divider shown in Figure 6.8.

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

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

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

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 $QD + R = X$, which proves that the circuit is a divider.

Such a layered rewriting approach significantly speeds up the verification process and partially avoids the problem of a potential memory explosion, which can be especially severe in the presence of a bug. Furthermore, it 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 current layer contains a bug. This process can also be 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. This way the source of an error is constrained to a particular layer and the propagation of rewriting will stop there to examine the bug. The same procedure can be used to prove non-restoring dividers.

However, such a layered algebraic rewriting is still non-scalable and fails for circuits with dividends beyond 21 bits. The next section proposes a method to remedy the problem by applying the idea of *hardware-based* rewriting described in Section 6.4 to array dividers.

### 6.5.3 Layered Hardware Rewriting

The concept of hardware rewriting, initially introduced in the context of the SQRT circuits in Section 6.4, can be extended to array dividers by applying it to individual layers of the array divider. The main idea is to add a circuit that reverses the computation of a partial remainder implemented in a given layer $i$ from its output $P_i$ and the given bit of the quotient, $q_i$. The circuit is termed *a signature linearizer*, since its output signature can be represented as a multi-linear polynomial. Figure 6.9 shows a single layer appended with such a linearizer circuit which computes $Dq_i + P_{ik}$. For simplicity it is shown here for the bottom layer, $i = 0$. The linearizer simply creates a single polynomial $Z_0 = Dq_0 + P_{0k}$ expressed in terms of its individual bits, i.e., $Z_0 = \sum_{k=0}^{n} 2^k z_{0k}$. This technique of layered hardware rewriting is much more efficient and scalable compared to all other techniques mentioned so far, verifying dividers of up-to 127-bit in just 19 seconds, shown in Section 6.6.2.

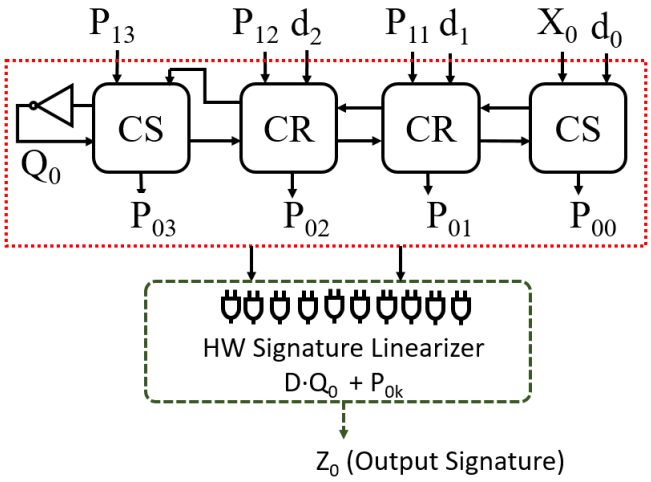



Figure 6.9: Layered hardware rewriting for dividers.

### 6.5.4 Verifying Output Constraint, $R < D$

For the verification of a divider to be complete, one must also verify that the remainder and the divisor satisfy the constraint $R < D$ (integer case) and $D \neq 0$. It should be emphasized that this constraint applies to the integer divider circuits in which the bit-widths of all the operands, $X, D$, and the outputs, $Q, R$, are the same, as in Figure 6.8(a). However, for the divider with size of $D, Q, R$ being roughly half ($n+1$ bits) of that of the dividend $X$ ($2n+1$ bits) an additional *user* constraint must be imposed on the dividend vs divisor, namely $X < 2^n D$, or equivalently $D > 2^{-n} X$, in order to avoid overflow of the quotient $Q$ [29] . Similarly, for the case of fractional divider the constraint on the inputs is $X < D$, for otherwise the result would not be fractional. These constraints must be taken into account during verification, otherwise the result cannot be correctly verified. Here, we describe the solution to the integer case, where $X, D, Q, R$ all have the same bit-width, in which case the constraint on the inputs is simply $X > D > 0$. Later, we present the solution for the constraint verification for the layered architecture as well.

The basic idea of the proposed verification of this constraint is shown in Figure 6.10. The goal is to check if the constraint $R < D$ is *always* satisfied by the circuit. This is done by appending the circuit with a comparator, $z = (R \geq D)$ and synthesizing the resulting circuit. Its output evaluates to 1 if the condition $R \geq D$ is satisfied, or, equivalently when $R < D$ is not satisfied. In addition we also consider the case of divider value $D = 0$, which should be disallowed. The condition $D \neq 0$ needs to be checked, otherwise $X = DQ + R$ makes $R = X$, which for an unsigned integer divider clearly violates the condition $R < D$. This condition is coded using signal $Divby0$, which indicates if $D = 0$; such a signal is provided by the circuit (as in the case of Synopsys DesignWare) or can be derived directly from the input $D$. The combined goal is now to prove that

$$(R < D) \wedge (D \neq 0)$$

is satisfied or, equivalently, if

$$(R \geq D) \vee (D = 0) \tag{6.6}$$

is unsatisfiable (unSAT). The resulting hardware (gate-level netlist) of such constructed circuit is converted to the CNF format and subjected to SAT. If the solution is unsatisfiable, unSAT, this proves that the constraint $R < D$ and $D \neq 0$ holds. We tested this idea on dividers with dividends up to 21 bits using ABC tool [31] and confirmed the validity of this approach. Unfortunately, the constraint verification for a complete divider design could not be completed for dividends with bit-widths beyond 21 bits. In particular, we could not verify $R < D$ on a 32-bit Synopsys DW divider design via this SAT method; the experiment timed out in 3600 seconds; specific information is given in the results Section 6.6 in Table 6.4. In the following section, we propose a method to solve the problem for large dividers using case-splitting.

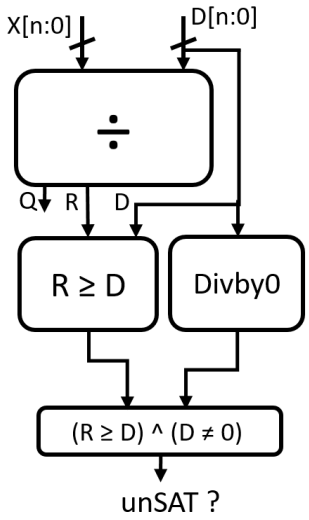



Figure 6.10: Verifying condition $R < D$ of a complete divider.

### 6.5.5 Verifying constraint $R < D$ by Case Splitting

To make the problem manageable, we divide it into sub-problems and verify the constraint (6.6) using SAT approach for individual ranges of $D$, each of them being sufficiently small for the SAT to handle. It suffices to only impose a case on $D$. This is accomplished by appending the divider circuit with a comparator $R \geq D$ and applying it to individual ranges of $D$. For instance, when verifying the constraint for a 32-bit divider, we split the process into four cases. The first case verifies the range of $D \leq 2^8$ by adding a constraint (another comparator) which imposes this restriction. This restriction is appended with the restrictions on $R, D$ discussed above, the resulting circuit is synthesized and subjected to a satisfiability test (SAT), as shown in Figure 6.11. Similar test needs to be performed for each range of $D$.

The range splitting of $D$ makes the constraint verification tractable. Without this casing strategy, the SAT problem is complex and for the case of 32-bit Synopsys DW divider the SAT verification times out after 3600s, as shown in Section 6.5.4. By splitting the verification into multiple cases, the constraint verification is reduced to take only a couple of seconds for dividends up-to 64-bits. The designer has to

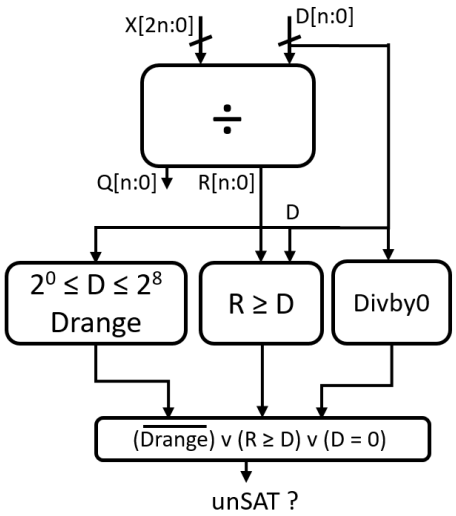



Figure 6.11: Verifying condition $R < D$ of a complete divider using case-splitting strategy for a given range of $D$.

decide what granularity to choose for case-splitting, depending on the number of core processors and the number of jobs that can run simultaneously. For the 32-bit divider constraint verification, we split the verification into four cases.

### 6.5.6 Constraint Verification for Layered Divider

While, in principle, the case splitting strategy for constraint verification simplifies the problem, the complete verification of the divider functionality and $R < D$ constraint still does not scale, as shown later in Table 6.4. To address this issue, we adopt the *layered approach*, described in Section 6.5.3, and apply the constraint analysis to the individual layers of the divider. In this case we also need to take care of the bounds on $X$ and $D$, such as $X < 2^n D$, to address the overflow issue. To do this, we add another comparator to the design to implement the constraint of $X < 2^n D$. This approach is illustrated in Figure 6.12.

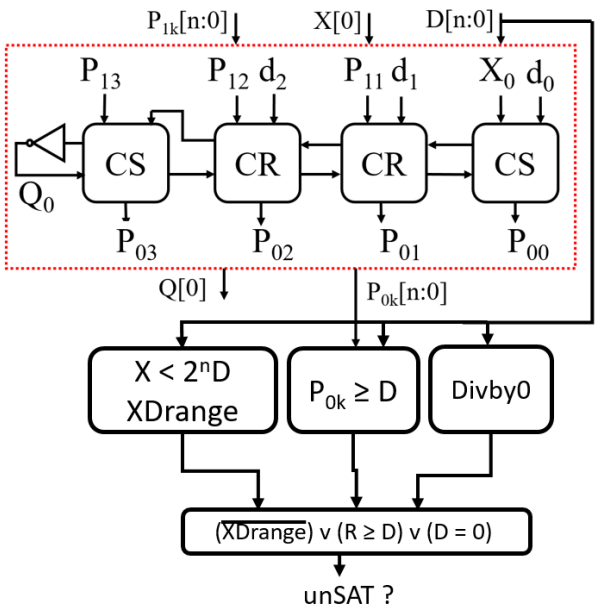



Figure 6.12: Verifying condition $R < D$ for the layered verification strategy, layer 0, using case-splitting

Table 6.2: Verification run times for SQRT circuits. #Bits = Radicand bit-width; MO = Memory-out 20GB; TO = Time-out 3600s

| # Bits | # Gates | Full-Rewrite (Software) (s) | SAT (s) | Sim. (s) | This Work (Hardware-Rewrite) (s) | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | | | | Residue Generation | Re-syn. | HR-SAT Bug-free | Total Time (Bug-free) | Total Time (Buggy) |
| 6 | 78 | 1.93 | 0.01 | 0.25 | 0.01 | 0.03 | 0.01 | 0.05 | 0.06 |
| 12 | 381 | MO | 0.01 | 0.25 | 0.01 | 0.03 | 0.01 | 0.05 | 0.06 |
| 18 | 897 | MO | 0.13 | 1.9 | 0.04 | 0.11 | 0.01 | 0.15 | 0.16 |
| 24 | 1584 | MO | 2.37 | 115 | 0.10 | 0.24 | 0.02 | 0.36 | 0.38 |
| 32 | 2794 | MO | 146.9 | TO | 0.77 | 1.06 | 0.02 | 1.85 | 1.91 |
| 64 | 10994 | MO | TO | TO | 1.70 | 6.07 | 0.02 | 7.79 | 7.85 |
| 96 | 24570 | MO | TO | TO | 3.78 | 6.88 | 0.95 | 11.61 | 8.26 |
| 128 | 43522 | MO | TO | TO | 6.43 | 10.26 | 2.92 | 19.52 | 21.10 |
| 256 | 263377 | MO | TO | TO | 9.81 | 73.53 | 983.33 | 1067.3 | 91.42 |

## 6.6 Results

### 6.6.1 SQRT Circuits

The verification technique described in this paper was implemented in Python and C++ as a stand-alone program, which uses ABC [2] at the back-end for synthesis. The program was tested on a number of SQRT circuits with radicand bit-widths varying

from 6 to 256. The square-root circuits used in the experiments were generated from Synopsys DWare library with the add-ons (the residual circuit and the linearizer) generated using the ABC tool [2]. Each design was appended with a residual circuit and a linearizer, and synthesized using ABC (*strash*, *dfraig*, *dch*). It should be pointed out that the architecture of the tested SQRT circuits and the add-ons generated by ABC are different and do not exhibit structural similarities, which proves the efficiency of our technique.

In the experiments, circuits with radicands smaller than 24 bits were synthesized down to bare wires, proving that the circuit indeed performs a SQRT function. The larger circuit, beyond 24 bits, required formal verification using SAT.

Table 6.2 compares the verification time of our technique with those obtained with the following techniques: 1) standard rewriting; 2) SAT (used for equivalence checking between Synopsys DW circuits and the reference design generated by ABC); and 3) simulation. The SAT experiments (column 4) were performed by creating a miter between the SQRT circuit and a reference design obtained from Synopsys DesignWare library and ABC tool respectively. The test for satisfiability was performed using miniSAT [44]. Despite its renown efficiency, miniSAT was unable to handle circuits with radicand bit-widths greater than 32. Similarly, standard algebraic rewriting of [9] (col 3) could not verify designs beyond 8-bit radicands because of memory overload; it used over 22 GB of memory in a matter of minutes.

The table also shows CPU time required for all phases of our experiment, including: the residue generation (col. 6); resynthesis (col. 7); and hardware-rewrite SAT to prove hardware rewriting (col. 8). Total verification time is given in column 9. As we can see from the table, SPEAR outperforms all of the tested techniques.

We also performed experiments on buggy circuits by inserting a number of bugs (5) in random places in the design. The verification results are shown in the last column (col. 10) of the table. Experiments show that for buggy circuits, solving

the satisfiability problem (and hence proving the bug) was easier than proving that the functionally correct, bug-free circuit is unSAT. This is not surprising, since in general the unSAT problems are harder to solve (in the worst case the entire solution space may need to be examined). In addition, the solution provided by SAT provides a counter-example that can be used to identify the bug. The debugging is a challenging problem, and it is part of future work.

We also performed exhaustive simulation experiments. The simulation was unable to handle circuits with more than 24-bit radicand and was aborted after running for over 10 hours of CPU time and consuming over 10 GB of memory for storing the intermediate simulation results. In contrast to all these schemes, hardware rewriting was able to verify square-root designs with up to 256-bit radicands, containing over 260,000 gates in less than 18 minutes, while using less than 4 GB of memory.

Table 6.3: Verification of a bug-free restoring divider. MO = Memory-out 20 GB, TO = Time-out 3600 s.

| Dividend bits | # Gates | Full-Rewrite SW [9] Time (s) | Layered-Rewrite SW [48] Time (s) | Simulation Time (s) | SAT [44] Time (s) | [22] | This Work Time (s) |
|---|---|---|---|---|---|---|---|
| 5 | 201 | 0.08 | 0.01 | 0.45 | 0.14 | - | 0.09 |
| 7 | 352 | 4.78 | 0.01 | 0.97 | 0.24 | - | 0.12 |
| 11 | 415 | MO | 0.01 | 1.23 | 10.68 | - | 0.18 |
| 13 | 570 | MO | 0.01 | 8.30 | 19.16 | 5.30 | 0.21 |
| 17 | 970 | MO | 4.72 | 552.50 | 1584.32 | 12.20 | 0.27 |
| 19 | 1207 | MO | 51.70 | TO | TO | - | 0.40 |
| 21 | 1470 | MO | 780.00 | TO | TO | - | 0.44 |
| 23 | 1750 | MO | MO | TO | TO | - | 0.48 |
| 33 | 3700 | MO | MO | TO | TO | 24.50 | 0.68 |
| 63 | 13446 | MO | MO | TO | TO | 40.50 | 4.48 |
| 95 | 28200 | MO | MO | TO | TO | - | 12.96 |
| 127 | 51200 | MO | MO | TO | TO | - | 18.56 |

### 6.6.2 Divider Circuits

The experiments were conducted on a 64-bit Intel Core i7-7600 CPU, 2.80 GHz $\times$ 2, with 30 GB of memory. The circuits were generated by a restoring divider generator tool and synthesized onto standard cells by the Synopsys Design Compiler (DC). The results of our work are stated in column "This Work". The consolidated

results are shown in Table 6.3 and include multiple verification methodologies: 1) Algebraically rewriting the entire circuit using full-rewrite software [9]; as mentioned earlier, this method is not effective in divider verification; 2) Based on the proposed layered verification [48] using layered-rewrite software. Our results are also compared with: 3) exhaustive simulation using Modelsim 10.5b on an Intel Core i7, 2.2 GHz with 16 GB memory; and 4) equivalence checking using miniSAT. In the SAT experiment, the synthesized divider circuits were compared against the dividers instantiated by the Synopsys DesignWare (DW) library. We also compare our results with those in [22] and show that our tool performs orders of magnitude faster; the results include the time for constraint verification as well.

As one can see from the table, neither the simulation nor the SAT results can compete with the layered HR-SAT verification. The seventh column presents the most scalable to-date divider verification results. The method of [22] relies on a heavy structural matching and it is only scalable up to 64-bit dividers. The sign " $-$ " in some entries shows that the data for that entry was not available from the reference paper. Our methodology can verify dividers of up to twice the bit-width.

Table 6.4: Detailed analysis of verification of a bug-free restoring divider using Hardware-based rewriting for Full vs. Layered strategies. MO = Memory-out 20 GB, TO = Time-out 3600 s.

| Dividend bits | # Gates | Verification Time (s) Hardware Rewrite - Full Divider | | | | Verification Time (s) Hardware Rewrite - Layered | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | Re-syn. | HR-SAT | R < D Case-based | Total (s) | Re-syn. | HR-SAT | R < D | Total (s) |
| 5 | 201 | 0.01 | 0.01 | 0.01 | 0.03 | 0.01 | 0.01 | 0.01 | 0.09 |
| 7 | 352 | 0.19 | 0.01 | 0.01 | 0.21 | 0.01 | 0.01 | 0.01 | 0.12 |
| 11 | 415 | 0.80 | 0.55 | 0.01 | 1.36 | 0.01 | 0.01 | 0.01 | 0.18 |
| 13 | 570 | 0.89 | 1.59 | 0.01 | 2.49 | 0.01 | 0.01 | 0.01 | 0.21 |
| 17 | 970 | 1.10 | 92.72 | 0.05 | 95.09 | 0.01 | 0.01 | 0.01 | 0.27 |
| 19 | 1207 | 1.26 | 1044.28 | 0.11 | 1045.65 | 0.01 | 0.02 | 0.01 | 0.40 |
| 21 | 1470 | - | TO | 0.14 | TO | 0.01 | 0.02 | 0.01 | 0.44 |
| 23 | 1750 | - | TO | 0.17 | TO | 0.01 | 0.02 | 0.01 | 0.48 |
| 33 | 3700 | - | TO | 1.01 | TO | 0.02 | 0.03 | 0.01 | 0.68 |
| 63 | 13446 | - | TO | 10.06 | TO | 0.05 | 0.08 | 0.01 | 4.48 |
| 95 | 28200 | - | TO | 1343.00 | TO | 0.10 | 0.15 | 0.02 | 12.96 |
| 127 | 51200 | - | TO | TO | TO | 0.11 | 0.16 | 0.02 | 18.56 |

Table 6.4 presents the detailed results for this work, showing the verification times of different stages of the verification process. We also compare the HW-based rewriting for a full divider vs. layered divider. A Full-Rewrite (hardware) column presents data for dividers, attempting a complete (non-layered) design verification. The synthesis tools are not powerful enough to remove all redundancies, even after simplifying the design with an appended hardware linearizer, leaving some redundant logic and making the resulting SAT problem not solvable. In contrast, the layered based hardware-rewriting is efficient and scalable.

Our methodology not only verifies the functionality of the design, but it also verifies whether the design satisfies the intrinsic constraint $R < D$. Column labeled "$R < D$" in Table 6.4 (Full Divider) is for the architecture with same bit-widths for all operands $(X, D, Q, R)$. For the architecture with smaller bit-widths of $D, Q, R$ than the bit-width of $X$, such as shown in Figure 6.8, the situation is more complex: the (seemingly obvious) condition $R < D$ is satisfied only if $Q$ does not overflow, that is for $D > 2^{-n}X$ [29]. We implemented these constraints as well in the context of layered rewriting. Since each layer is small compared to the complete design, functional verification of each layer, as well as verifying the constraints on $X, R, D$, is very effective in terms of memory and time complexity. This is done by appending gate-level comparators for $D > 2^{-n}X$ to each layer to prove that $R < D$. Our results for layered-based HR-SAT are shown in the right column of Table 6.4 (Layered).

# CHAPTER 7

# SUMMARY, CONTRIBUTION, AND PUBLICATIONS

It may seem surprising that adding more hardware ($R_{ref}$, $Q^2 + R_{ref}$, and linearizers) to the design actually simplifies the verification problem at hand. While this obviously increases the circuit complexity in terms of the hardware involved, the resulting circuit, as long as it is bug-free, should become *redundant*. The described technique relies on the synthesis and SAT tools to prove this redundancy. As clearly demonstrated by the experiments, even if the synthesis is unable to reduce the resulting, inherently redundant circuit to wires/buffer, the standard SAT-based verification has a much easier task to prove the equivalence. In particular, SAT can be applied independently to a single-input logic cone, in parallel, as shown in Figure 6.6.

In principle, since the proposed technique does not depend on the internal structure of the circuit (or the algorithm it implements), it should also handle other types of square-root and divider circuits, such as those based on the convergence algorithm [28]. In general, the described hardware verification technique can be applied to arithmetic circuits with non-linear polynomial characteristic function. To the best of our knowledge, this is the first work that was able to successfully verify large integer square-root and divider circuits using formal methods.

## 7.1 Contribution

- Presented an algebraic model for the verification of constant and generic array dividers. Introduced an engineering way of verifying layered-based divider circuits to enhance scalability.

- Proposed and implemented a novel technique of hardware rewriting for SQRT circuits. This entails a complete automated verification methodology, including automatic generation of the residue circuit (for SQRT) and the linearizer circuit, and their integration with the original circuit, followed by re-synthesis with a synthesis tool, such as ABC.

- Integrated a reverse-division unit to help perform a "cleaner" layered rewriting strategy, as shown in Section 5.1. The layered rewriting makes the rewriting more efficient, specially for a hardware-based rewriting approach. With this, one can now verify the dividers with up to 127-bit dividend operand using the hardware rewriting combined with layered approach.

- Discussed the applicability of the underlying methodology to floating point divider circuits implemented in fractional arithmetic.

- Created an extensive set of benchmarks representative of real industrial designs, including the instantiated elements of the Design Ware library of Synopsys Design Compiler (DC). While the Verilog code for these circuits is encrypted, the optimized gate-level netlists are available and were used for this purpose.

- Implemented python scripts for automating the verification process, including the netlist conversion between Synopsys DC and ABC tool.

### 7.1.1 Future Directions

There is a number of interesting research directions, not discussed in this work that can be further explored to enhance the efficiency and applicability of the proposed technique to other architectures.

- Apply the verification to arithmetic circuits in fractional arithmetic.

- We demonstrated that hardware rewriting works well for the complete square-root design, whereas on dividers, we have to rely the layered architecture. We

analyzed the reason for it, namely the difference in the characteristic function. However, one can analyze in more detail the gate level logic to try to infer what makes the rewriting hard. Understanding the architectural difference between SQRT and DIV presented in this work might help answer this question.

- Explore architectures of SQRT and DIV using hardware rewriting and/or algebraic rewriting. Other arithmetic circuits like modulo arithmetic operation, iterative subtractors/adders, and other structures might be worth looking into as well.

- Lastly, for checking algorithmic correctness, loop-invariant equations can be used to determine the characteristic equations. Once these well-defined equations are derived, hardware rewriting can be applied for verification.

## 7.2 Publications

The following research articles have been published over the course of this research.

- Atif Yasin, Tiankai Su, Sébastien Pillement, Maciej Ciesielski. **Verifying Square-root and Divider Circuits by Hardware Rewriting** (Submitted TCAD2020)

- Atif Yasin, Tiankai Su, Sébastien Pillement, Maciej Ciesielski. **SPEAR: Hardware-based Implicit Rewriting for Square-root Verification** (DATE 2020)

- Atif Yasin, Tiankai Su, Sébastien Pillement, Maciej Ciesielski. **Functional Verification of Hardware Dividers using Algebraic Model**. IFIP/IEEE International Conference on Very Large Scale Integration VLSI-SOC, Oct 2019, Cusco, Peru.

- Atif Yasin, Tiankai Su, Sébastien Pillement, Maciej Ciesielski. **Formal Verification of Integer Dividers: Division by a Constant**. IEEE Symposium on VLSI (ISVLSI), IEEE, Jul 2019, Miami, USA.

- Cunxi Yu, Atif Yasin, Tiankai Su, Alan Mishchenko, Maciej Ciesielski, **Rewriting Environment for Arithmetic Circuit Verification**, 22nd International Conference on Logic Programming, Artificial Intelligence and Reasoning (LPAR-22), EPIC Series in Computing, 2018.

- T. Su, A. Yasin, C. Yu and M. Ciesielski, "**Computer Algebraic Approach to Verification and Debugging of Galois Field Multipliers**," 2018 IEEE International Symposium on Circuits and Systems (ISCAS), Florence, Italy, 2018.

- M. Ciesielski, T. Su, A. Yasin and C. Yu, "**Understanding Algebraic Rewriting for Arithmetic Circuit Verification: a Bit-Flow Model**," in IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, April 2019 (Early Access)

- T. Su, C. Yu, A. Yasin and M. Ciesielski, "**Formal Verification of Truncated Multipliers Using Algebraic Approach and Re-Synthesis**," 2017 IEEE Computer Society Annual Symposium on VLSI (ISVLSI), Bochum, Germany, 2017.

- Cunxi Yu, Tiankai Su, Atif Yasin, and Maciej Ciesielski. 2019. **Spectral Approach to Verifying Non-linear Arithmetic Circuits**. In Proceedings of the 24th Asia and South Pacific Design Automation Conference (ASPDAC '19). ACM, New York, NY, USA.

# BIBLIOGRAPHY


[1] Adams, W.W., and Loustanau, P. *An Introduction to Gröbner Bases*. American Mathematical Society, 1994.

[2] Brayton, R., and Mishchenko, A. ABC: An Academic Industrial-Strength Verification Tool. In *Proc. Intl. Conf. on Computer-Aided Verification* (2010), pp. 24–40.

[3] Bryant, R. E., and Chen, Y-A. Verification of Arithmetic Functions with Binary Moment Diagrams. In *DAC'95*.

[4] Bryant, Randal E. Graph-based algorithms for boolean function manipulation. *Computers, IEEE Transactions on 100*, 8 (1986), 677–691.

[5] Bryant, Randal E. Bit-level analysis of an srt divider circuit. In *33rd (DAC)* (1996), ACM, pp. 661–665.

[6] Buchberger, B. *Ein algorithmus zum auffinden der basiselemente des restklassenringes nach einem nulldimensionalen polynomideal*. PhD thesis, Univ. Innsbruck, 1965.

[7] Burch, J. R. Using bdds to verify multipliers. In *28th ACM/IEEE Design Automation Conference* (June 1991), pp. 408–412.

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

[9] Ciesielski, M., Su, T., Yasin, A., and Yu, C. Understanding algebraic rewriting for arithmetic circuit verification: a bit-flow model. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* (2019), 1–1.

[10] Ciesielski, M, Yu, C, Brown, W, Liu, D, and Rossi, André. Verification of Gate-level Arithmetic Circuits by Function Extraction. In *52nd DAC* (2015), ACM, pp. 52–57.

[11] Cox, D., Little, J., and O'Shea, D. *Ideals, Varieties, and Algorithms*. Springer, 1997.

[12] Davis, Martin, Logemann, George, and Loveland, Donald. A machine program for theorem-proving. *Commun. ACM 5*, 7 (July 1962), 394–397.

[13] De Dinechin, Florent, and Didier, Laurent-Stéphane. Table-based division by small integer constants. In *(ISARC* (2012), Springer, pp. 53–63.

[14] De Dinechin, Florent, and Pasca, Bogdan. Designing custom arithmetic data paths with flopoco. *IEEE Design & Test of Computers 28*, 4 (2011), 18–27.

[15] Decker, W., Greuel, G.-M., Pfister, G., and Schönemann, H. SINGULAR 3-1-6 A Computer Algebra System for Polynomial Computations. Tech. rep., 2012. http://www.singular.uni-kl.de.

[16] Doran, Robert W. Special cases of division. In *J. UCS The Journal of Universal Computer Science*. Springer, 1996, pp. 176–194.

[17] Faugere, Jean-Charles. A New Efficient Algorithm for Computing Gröbner Bases (F4). *Journal of Pure and Applied Algebra 139*, 1–3 (1999), 61 – 88.

[18] Gao, Sicun. Counting zeros over finite fields with gröbner bases. *Master's thesis, Carnegie Mellon University* (2009).

[19] Granlund, Torbjörn, and Montgomery, Peter L. Division by invariant integers using multiplication. *SIGPLAN Not.* (June 1994), 61–72.

[20] Haghbayan, M., Alizadeh, B., Rahmani, A., Liljeberg, P., and Tenhunen, H. Automated formal approach for debugging dividers using dynamic specification. In *2014 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT)* (Oct 2014), pp. 264–269.

[21] Haghbayan, M. H., Alizadeh, B., Behnam, P., and Safari, S. Formal verification and debugging of array dividers with auto-correction mechanism. In *2014 27th International Conference on VLSI Design and 2014 13th International Conference on Embedded Systems* (Jan 2014), pp. 80–85.

[22] Haghbayan, Mohammad Hashem, and Alizadeh, Bijan. A dynamic specification to automatically debug and correct various divider circuits. *INTEGRATION, the VLSI journal 53* (2016), 100–114.

[23] Hamaguchi, K., Morita, A., and Yajima, S. Efficient construction of Binary Moment Diagrams for verifying arithmetic circuits. In *Proceedings of IEEE International Conference on Computer Aided Design (ICCAD)* (Nov 1995), pp. 78–82.

[24] Harrison, John. Formal verification of ia-64 division algorithms. *Theorem Proving in Higher Order Logics* (2000), 233–251.

[25] Harrison, John. Formal verification of square root algorithms. In *Formal Methods in Systems Design* (2003), p. 2003.

[26] Kaivola, Roope, Ghughal, Rajnish, Narasimhan, Naren, Telfer, Amber, Whittemore, Jesse, Pandav, Sudhindra, Slobodova, Anna, Taylor, Christopher, Frolov, Vladimir, Reeber, Erik, and Naik, Armaghan. Replacing testing with formal verification in intel coretm i7 processor execution engine validation. In *CAV* (06 2009), pp. 414–429.

[27] Kaivola, Roope, and Kohatsu, Katherine. Proof engineering in the large: formal verification of pentium® 4 floating-point divider. *International Journal on STTT 4*, 3 (2003), 323–334.

[28] Kong, I., and Swartzlander, E. E. A goldschmidt division method with faster than quadratic convergence. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems 19*, 4 (April 2011), 696–700.

[29] Koren, Israel. *Computer Arithmetic Algorithms*. Universities Press, 2002.

[30] Lv, J., Kalla, P., and Enescu, F. Efficient Grobner Basis Reductions for Formal Verification of Galois Field Arithmatic Circuits. *IEEE Trans. on CAD 32*, 9 (September 2013), 1409–1420.

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

[32] Paplinski, A. Cse2306/1308 digital logic lexture notes, lecture 8. *Lecture Notes* (2006).

[33] Parhami, Behrooz. *Computer Arithmetic: Algorithms and Hardware Designs*. Oxford University Press, Inc., New York, NY, USA, 2000.

[34] Pavlenko, E., Wedler, M., Stoffel, D., Kunz, W., 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.

[35] Pruss, T., Kalla, P., and Enescu, F. Equivalence Verification of Large Galois Field Arithmetic Circuits using Word-Level Abstraction via Gröbner Bases. In *DAC'14* (2014), pp. 1–6.

[36] Rager, D. L., Ebergen, J., Nadezhin, D., Lee, A., Chau, C. K., and Selfridge, B. Formal verification of division and square root implementations, an oracle report. In *2016 Formal Methods in Computer-Aided Design (FMCAD)* (Oct 2016), pp. 149–152.

[37] Ritirc, D., Biere, A., and Kauers, M. Improving and extending the algebraic approach for verifying gate-level multipliers. In *2018 Design, Automation Test in Europe Conference Exhibition (DATE)* (March 2018), pp. 1556–1561.

[38] Ritirc, Daniela, Biere, Armin, and Kauers, Manuel. Column-wise verification of multipliers using computer algebra. In *Formal Methods in Computer-Aided Design (FMCAD)* (2017).

[39] Rueß, Harald, Shankar, Natarajan, and Srivas, Mandayam K. Modular verification of srt division. In *(ICCAD)* (1996), Springer, pp. 123–134.

[40] Ruiz, Antonio Lloris, Morales, Encarnación Castillo, Roure, Luis Parrilla, and Ríos, Antonio García. *Algebraic Circuits*. Springer, 2014.

[41] Sayed-Ahmed, Amr, Große, Daniel, Kühne, Ulrich, Soeken, Mathias, and Drechsler, Rolf. Formal verification of integer multipliers by combining grobner basis with logic reduction. In *DATE'16* (2016), pp. 1–6.

[42] Sharangpani, HP, and Barton, ML. Statistical analysis of floating point flaw in the pentium processor. *Intel Corporation* (1994).

[43] Shekhar, N., Kalla, P., and Enescu, F. Equivalence Verification of Polynomial Data-Paths Using Ideal Membership Testing. *TCAD 26*, 7 (July 2007), 1320–1330.

[44] Sorensson, Niklas, and Een, Niklas. Minisat v1. 13-a sat solver with conflict-clause minimization. *SAT 2005* (2005), 53.

[45] Su, Tiankai. *Analysis and verification of arithemtic circuits using computer algebra approach*. PhD thesis, University of Massachusetts Amherst, 2019.

[46] Ugurdag, H Fatih, De Dinechin, Florent, Gener, Y Serhan, Goren, Sezer, and Didier, Laurent-Stéphane. Hardware division by small integer constants. *IEEE TC* (2017).

[47] Wienand, O., Wedler, M., Stoffel, D., Kunz, W., and Greuel, G.-M. An Algebraic Approach for Proving Data Correctness in Arithmetic Data Paths. *CAV* (July 2008), 473–486.

[48] Yasin, A., Su, T., Pillement, S., and Ciesielski, M. Functional verification of hardware dividers using algebraic model. In *2019 IFIP/IEEE 27th International Conference on Very Large Scale Integration (VLSI-SoC)* (Oct 2019), pp. 257–262.

[49] Yasin, Atif, Su, Tiankai, Pillement, Sebastien, and Ciesielski, Maciej. Formal verification of integer dividers:division by a constant. In *2019 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)* (2019), IEEE.

[50] Yu, C., Ciesielski, M., and Mishchenko, A. Fast algebraic rewriting based on and-inverter graphs. *IEEE TCAD of ICS 37*, 9 (Sep. 2018), 1907–1911.

[51] Yu, Cunxi, Brown, Walter, Liu, Duo, Rossi, André, and Ciesielski, Maciej J. Formal verification of arithmetic circuits using function extraction. *IEEE Trans. on CAD of Integrated Circuits and Systems* (2016).

[52] Yu, Cunxi, and Ciesielski, Maciej. Analyzing imprecise adders using BDDs–a case study. In *VLSI (ISVLSI), 2016 IEEE Computer Society Annual Symposium on* (2016), IEEE, pp. 152–157.

[53] Yu, Cunxi, and Ciesielski, Maciej. Efficient parallel verification of Galois field multipliers. *ASP-DAC 2017* (2017).

[54] Yu, Cunxi, Yasin, Atif, Su, Tiankai, Mishchenko, Alan, and Ciesielski, Maciej. Rewriting environment for arithmetic circuit verification. EasyChair Preprint no. 662, EasyChair, 2018.