Formal Methods in Arithmetic Circuit Verification: a Brief History and Challenges

Maciej Ciesielski,
University of Massachusetts Amherst, MA, USA
ciesiel@umass.edu

***Abstract*—**
The last few decades have witnessed an explosive growth in the development and use of cyber-physical and digital embedded systems. As more and more of those systems become security- and safety-critical, assuring correctness and dependability of digital hardware that implements those systems becomes of paramount importance. This keynote addresses formal verification of digital hardware and in particular of arithmetic circuits, essential components at the heart of those systems. It gives an overview of the methods used in arithmetic circuit verification, how they evolved from pure mathematical models to practical engineering solutions, and problems they face; and a brief look into the future and the inevitable challenges.

I. Introduction

The last few decades have witnessed an explosive growth in the development of cyber-physical and embedded systems. As more and more of those systems become security- and safety-critical, assuring functional correctness and dependability of digital hardware implementation of those systems became critical. Essential elements of those systems are arithmetic circuits: different types of adders, multipliers, multiply-accumulate, and divider circuits that need to be efficiently designed and optimized for area, delay and power. The increasing complexity of those circuits, containing millions of logic gates, makes them error-prone, which requires thorough verification and testing to guarantee their integrity. This keynote addresses some of these issues and concentrates on formal verification of hardware implementation of arithmetic circuits. It gives a brief overview of modern methods used in arithmetic circuit verification, including theorem proving, canonical diagrams, Boolean satisfiability, and symbolic computer algebra and its derivatives. It shows how these methods evolved from being based on pure mathematical models to practical engineering solutions. It discusses challenges they face and offers a look into the future.

II. Theorem Provers

The verification approach that gained particular attention in industry is Theorem Proving. Theorem Provers are highly interactive, inductive systems designed to prove that an implementation satisfies the specification using mathematical reasoning. The proof system is based on a set of axioms and inference rules, such as simplification, term rewriting, and induction [1] [2]. The basic element of theorem proving is *term rewriting*, a conceptually simple computational paradigm that uses repeated application of simplification rules and axioms, in which terms of one formula are replaced by other terms. The goal is to prove that the original formula of the implementation is syntactically identical to that of the specification.

Theorem prover systems are highly interactive, require intimate knowledge of the design domain and extensive user guidance and expertise. Success of the proof strongly depends on the choice and on the order in which the rules are applied, with no guarantee of a conclusive termination. Theorem Provers have been used in floating-point arithmetic verification [3][4], where other verification methods fail. Most of these systems, however, concentrate on proving correctness of the underlying algorithms to effect the computation and on the resulting architecture rather than on the actual hardware verification. Some of the most popular theorem proving tools are PVS [3], ACL2 [5], Coq [6] and Isabelle [7], among others.

The following fragment of Coq *Proof Assistant* illustrates the proof of correctness of a binary adder with inputs A, B, and C_{in}, and outputs S and C_{out}. It defines the implementation and Boolean properties of the adder, and proves that the properties hold for the implementation. Specifically, the goal is to prove that logic-level implementation satisfies logic properties of the adder: the sum output $S = A \oplus B \oplus C_{in}$; and the carry-out output $C_{out} = (A \wedge B) \vee (B \wedge C_{in}) \vee (A \wedge C_{in})$. Most of the theorem prover systems use this type of approach and format.

```
-------------------------------------------
(* Define the input and output signals as
   Boolean variables *)
Variable A B Cin S Cout : bool.

(* Define the logical equations for the adder *)
Definition eq1: Prop := S = xorb (xorb A B) Cin.
Definition eq2: Prop := Cout = orb (andb A B)
                (orb (andb B Cin) (andb A Cin)).
(* Define a theorem stating that the
implementation satisfies the properties *)
Theorem binary_adder_correct : eq1 /\ eq2.
Proof.
  (* Simplify the logical equations *)
     simpl. unfold eq1. unfold eq2.
    (* Prove that the properties hold for all
       possible input combinations *)
  destruct A; destruct B; destruct Cin;
  reflexivity.
QED.
-------------------------------------------
```

III. Canonical Diagrams

Binary Decision Diagrams (BDD) [8] and their derivatives, such as Binary Moment Diagrams (BMD) [9][10], Taylor Expansion Diagrams (TED) [11], and other hybrid diagrams [12] belong to the class of *canonical diagrams*. Of particular importance is BDD, a data structure constructed specifically for efficient representation and manipulation of Boolean functions.

A BDD is a rooted directed acyclic graph, whose internal nodes represent binary variables, with two terminal nodes representing constant $\mathbf{1}$ and constant $\mathbf{0}$. The BDD representation is based on Shannon expansion of the function at variable x onto its cofactors, $f(x) = x'f_{x'} + xf_x$. Each internal node has two children: one representing the function with variable x taking value 1 (positive cofactor, f_x), the other one with variable x taking value 0 (negative cofactor, $f_{x'}$). The BDD encodes a Boolean function in a set of paths from the root to terminal node 1 in a compact way. An example of a multi-rooted BDD is given in Figure 1 for a 3-bit adder, where each root represents one output of the adder. The solid lines represent positive cofactors and the dashed lines negative cofactors of the function w.r.t. the variable associated with the node.

For a given order of variables, a BDD is minimal (in the number of nodes), unique, and irredundant, i.e., there are no two internal nodes that represent the same function. An important feature of the reduced and ordered BDD is that it is canonical; two syntactically different Boolean function that are functionally equivalent will have the same BDD representation under the same ordering of variables. This feature makes it possible to check equivalence between two Boolean functions by constructing and comparing their BDDs.

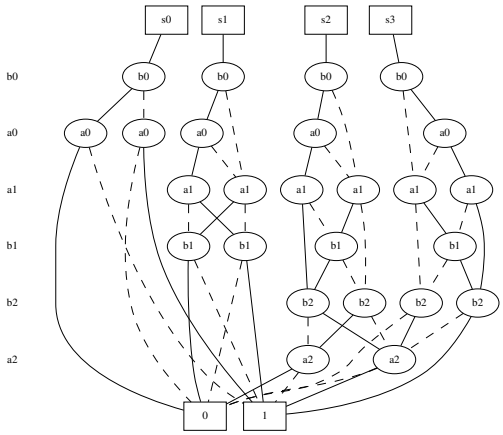


Fig. 1: BDD or a 3-bit adder.

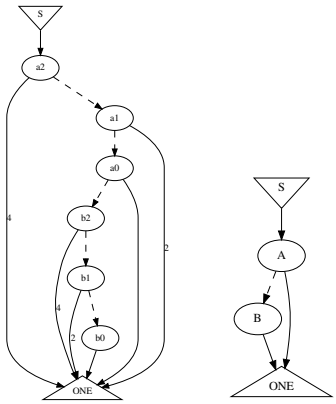
BDDs are commonly used in equivalence checking of logic circuits and some simple arithmetic circuits, such as adders/subtractors. However, their use for complex arithmetic circuits, such as multipliers and dividers, is limited by an exponential growth of their size with the number of bits of the circuit.

Another class of canonical methods includes Binary Moment Diagrams (called *BMD due to their multiplicative nature), whose representation is based on *moment decomposition*, $f(x) = f_{x'} + x(f_x - f_{x'})$ [13]. While a BDD represents mapping between Boolean inputs and Boolean outputs, $B \to B$, *BMDs represent mapping from Boolean inputs to Integer outputs, $B \to \mathbb{Z}$, and as such are better equipped to represent arithmetic functions. Attempts have been made to use *BMDs to verify multipliers [14][10], but even they do not scale well for those designs.

Figure 2a) shows a *BMD diagram for a 3-bit adder. Solid edges in this diagrams represent multiplication, while the dashed edges represent addition. The solid edges are labeled with constants to indicate a multiplicative factor of the respective subgraph. In this example the *BMD diagram represents the result of a 3-bit adder as a sum S of the binary encoded inputs A and B, namely: $S = (4a_2+2a_1+a_0)+(4b_2+2b_1+b_0)$.

Another representation, termed TED (for *Taylor Expansion Diagrams*) offers a higher level of abstraction by providing mapping between inputs and outputs both represented as Integers, $\mathbb{Z} \to \mathbb{Z}$ [15]. It uses finite Taylor expansion of a function w.r.t. to its inputs, and as such can be used to encode arithmetic functions whose specifications can be represented as polynomials.

Figure 2b) shows a TED for an adder with arbitrary bit-width; it simply encodes the adder function as $S = A + B$. We can see dramatic decrease in the complexity of the TED compared to BDD and *BMD diagrams, albeit at the cost of loosing the bit-level composition of the circuit.



a) b)

Fig. 2: A 3-bit adder represented as a) *BMD: $S = (4a_2 + 2a_1 + a_0) + (4b_2 + 2b_1 + b_0)$ and b) TED: $S = A + B$.

The example in Figure 3 illustrates the use of TED to verify correctness of the *resource sharing* transformation often used in industrial designs. Its goal is to maximize sharing of the resources, in this case the multipliers. Standard verification approach of identifying "similarities" between the internal points of the circuit cannot solve this problem because there are no internal equivalent points or they are lost during such transformation. In contrast, TEDs can solve this problem

efficiently for arbitrary bit-width of the operands by generating symbolic expressions for the original and the transformed form and proving that their TEDs are isomorphic.

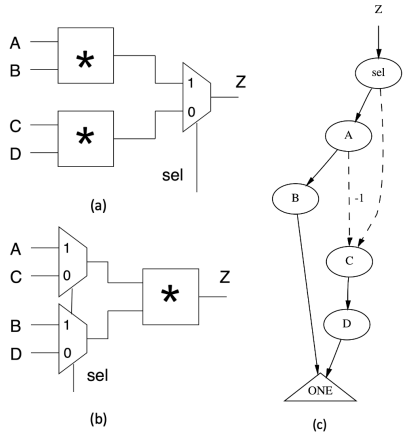


Fig. 3: Verification of resource sharing: using TED to prove equivalence of the two expressions: $Z = (A \cdot B \cdot sel + (C \cdot D) \cdot (1 - sel) = (A \cdot sel + C \cdot (1 - sel))(B \cdot sel + D \cdot (1 - sel))$.

IV. Boolean SAT and SMT Solvers

To mitigate the limitations of the decision diagrams, several techniques have been developed to reduce the complexity of equivalence checking and other verification tasks. The one that made a significant impact on the verification field is Boolean Satisfiability (SAT). The goal of SAT is to find an assignment of variables for which a given Boolean formula evaluates to 1. Typically the formula is given in a *conjunctive normal form* (CNF), a conjunction of clauses, where a clause is a disjunction of literals. For example, Boolean formula $\varphi = (a + \neg b)(\neg a + \neg b + c)$ can be satisfied by setting $a = 1, b = 0, c = 0$, which makes $\varphi = 1$. If the assignment of variables that makes the Boolean formula $\varphi = 1$ does not exist, the problem is called *unsatisfiable* (unSAT).

Several SAT solvers have been developed to solve Boolean decision problems, including GRASP [16], Chaff [17] and MiniSAT [18], among others. All SAT solvers are based on the basic *Davis-Putnam* (DPLL) search algorithm with backtracking [19]. A number of techniques, such as non-chronological backtracking, resolution, recursive learning, and conflict-driven clause learning, have been developed to improve the efficiency of SAT solver.

SAT methods are widely used in formal verification; they are particularly applicable to equivalence checking between two circuits: the *implementation* and the *reference* circuits. The equivalence check is accomplished by adding a *miter*, a cluster of XOR and OR gates, at the outputs of the two circuits and observing its output. An example of such a configuration is shown in Figure 4, with two circuits F, G connected by a miter. If the two circuits are functionally equivalent, then, for any input assignment, the same values should be observed at their outputs and the output of the miter should be 0. In case of a mismatch between the circuit outputs, the miter will generate 1. The equivalence checking problem is then solved by checking if the output of the miter is always 0; or, equivalently that it never evaluates to 1. This condition is known in the verification jargon as *unSAT*.

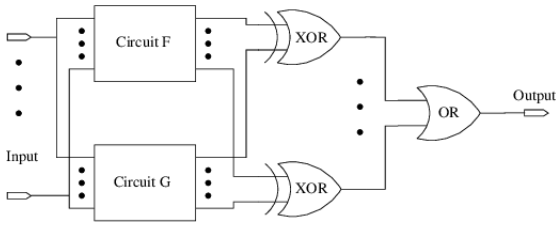


Fig. 4: Equivalence checking using SAT

SAT solvers can solve this efficiently by creating a Boolean expression of the miter structure in a CNF form and solve the corresponding SAT problem. If the problem is *satisfiable*, indicating that the circuits are *not equivalent*, the SAT solver returns a *counter-example*, an assignment of inputs for which the two circuits produce different outputs.

An extension of Boolean SAT is *Satisfiability Modulo Theories* (SMT) [20]. It covers Boolean and non-Boolean domains (bit vectors, integer and real arithmetic, linear inequalities, uninterpreted functions, etc.) and integrates them into a DPLL-style SAT decision procedure. Some of the common SMT solvers include Boolector [21], Z3 [22] and CVC [23]. However, SMT solvers still model the problem as a decision problem, and are not efficient at verifying large arithmetic circuits.

It should be noted that the Boolean methods based on BDDs and SAT require "bit blasting", flattening of the design into a bit-level netlists. Unfortunately, this makes it inefficient for complex arithmetic circuits, such as multipliers and dividers. Furthermore, equivalence checking approach supported by these tools relies on a trusted reference circuit, which may not be available. Next section provides a remedy to this problem by providing verification techniques based on symbolic computer algebra (SCA).

V. Computer Algebra Methods

A recent approach to formal verification of arithmetic circuits is mostly based on Symbolic Computer Algebra (SCA) [24], which represents arithmetic circuits in an algebraic rather than Boolean domain. There are two basic flavors of these techniques: a method based on polynomial reduction using Gröbner basis [20][25][26][27]; and one based on a more practical, engineering implementation, called *algebraic rewriting* [28]. The following sections briefly review these techniques and describe how they can be implemented using efficient synthesis tools, such as ABC [29].

A. Symbolic Computer Algebra (SCA) Approach

The SCA-based verification research was pioneered by [30] and [31], who applied concepts from computer algebra to formal verification of integer and Galois Fields arithmetic

circuits. In this approach, the circuit elements and the specification are represented as polynomial rings. Specifically, logic gates of the circuit are modeled by a set G of pseudo-Boolean polynomials, with binary variables and coefficients in $\mathbb{Z}_2$. Table I presents algebraic models of some of the basic logic gates [28]. The function computed by the circuit is represented as a *specification polynomial*, F, composed of two parts: 1) an *input signature*, Sig_{in}, which describes the arithmetic function in terms of its input variables; and 2) the *output signature*, Sig_{out}, which provides the encoding of the result in the output variables. With this, the specification polynomial can be expressed as $F = Sig_{out} - Sig_{in}$.

TABLE I: Boolean and algebraic models of basic logic gates.

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

For example, the specification of an n-bit unsigned integer multiplier, $Z = A \cdot B$ with inputs $A = [a_{n-1}, \cdots, a_0]$ and $B = [b_{n-l}, \cdots, b_0]$, is given by $F = \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} 2^{i+j} a_i b_j - \sum_{k=0}^{2n-1} z_k$. The first part of the specification polynomial is the *input signature*, and the second part is the *output signature*. The verification problem is then formulated as a proof obligation that the implementation (G) satisfies the specification (F) [20][25][26][32]. Mathematically, such a proof is achieved by reducing the specification polynomial F modulo G to a *normal form* and testing if it reduces to zero, known as the *ideal membership test* [20]. The reduction can be accomplished using a computer algebra system, such as Mathematica or Singular [33]. If the result of such a reduction is zero, the circuit satisfies the specification, proving that the circuit is functionally correct. However, for the result to be trusted, the set G must constitute a canonical form called *Gröbner basis* [24].

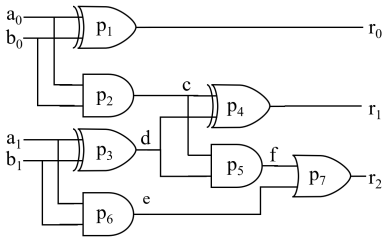
While computing a complete Gröbner basis is computationally expensive, it has been shown that for combinational circuits with logic gates given in reverse topological order, the resulting set G already constitutes a Gröbner basis (GB). However, an additional constraint must be imposed on the internal signals to restrict them to binary values. This is achieved by adding for each signal x a *field polynomial*, $\langle x^2 - x \rangle$. The zeros of this polynomial (0 or 1) automatically restrict variable x to binary.

B. Algebraic Rewriting

A more practical approach to solve this problem has been proposed in [28], whereby the expensive polynomial reduction modulo GB has been replaced by a computationally simpler *algebraic rewriting*.

Algebraic rewriting is the process of transforming the output signature Sig_{out} into an input signature, Sig_{in}, using algebraic models of the logic gates given in Table I. To satisfy the requirement of the implementation set G to be Gröbner basis, the logic gates of the circuit represented by polynomials in G are ordered in reverse topological order. In addition, in order to restrict the signals to binary values, whenever variable x is raised to x^2, it is lowered to x, so it never appears in higher degree and the polynomial remains multi-linear. This operation is equivalent to a division by the field polynomial $< x^2 - x >$ but simpler to implement. In a functionally correct circuit, the resulting Sig_{in} should match the expected functional specification of the circuit (adder, multiplier, etc). In fact, the specification needs not to be known; in this case the resulting Sig_{in} provides the arithmetic function computed by the circuit. In this sense, the rewriting performs a *functional extraction*, and the method can be used as a reverse engineering tool for arithmetic function extraction.

Algebraic rewriting is illustrated with a 2-bit adder circuit shown in Figure 5. The output signature of the circuit, $Sig_{out} = 4r_2 + 2r_1 + r_0$, is rewritten into an input signature Sig_{in}. It is then compared with the circuit specification of an adder $(2a_1 + a_0 + 2b_1 + b_0)$ to check if the circuit performs this arithmetic function.



$$
\begin{aligned}
p_1 &= r_0 - (a_0 + b_0 - 2a_0b_0)\\
p_2 &= c - a_0b_0\\
p_3 &= d - (a_1 + b_1 - 2a_1b_1)\\
p_4 &= r_1 - (c + d - 2cd)\\
p_5 &= f - cd\\
p_6 &= e - a_1b_1\\
p_7 &= r_2 - (e + f - ef)
\end{aligned} \tag{1}
$$

Fig. 5: A two-bit adder circuit and gate polynomials G.

The rewriting starts with $F = Sig_{out} = 4r_2 + 2r_1 + r_0$. Each variable in F that represents an output of a logic gate p is replaced with the respective input expression for that gate, denoted as F/p. For example, F/p_7 means that in the starting polynomial F the output r_2 of gate p_7 is replaced by the algebraic expression of its inputs, $r_2 = e + f - ef$. This is equivalent to dividing polynomial F by the gate polynomial p, as done in the standard CSA approach.

$$
\begin{aligned}
&F = Sig_{out} = 4r_2 + 2r_1 + r_0\\
&F/p_7 = 4e + 4f - 4ef + 2r_1 + r_0\\
&F/...p_4 = 4e + 4f - 4ef + 2c + 2d - 4cd + r_0\\
&F/...p_5 = 4e - 4edc + 2c + 2d + r_0\\
&F/...p_1 = 4e - 4edc + 2c + 2d + a_0 + b_0 - 2a_0b_0\\
&F/...p_2 = 4e - 4eda_0b_0 + 2d + a_0 + b_0\\
&F/...p_3 = 4e - 4ea_0b_0(a_1 + b_1 - 2a_1b_1) + 2(a_1 + b_1 - 2a_1b_1) + a_0 + b_0\\
&F/...p_6 = 4a_1b_1 - 4a_1b_1a_0b_0(a_1 + b_1 - 2a_1b_1) + 2(a_1 + b_1) - 4a_1b_1\\
&\qquad + a_0 + b_0 = 2a_1 + 2b_1 + a_0 + b_0\\
&F/(G) = Sig_{in} = 2a_1 + 2b_1 + a_0 + b_0
\end{aligned} \tag{2}
$$

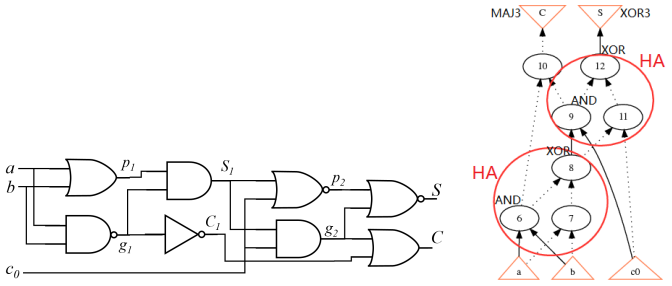
Here, the computed Sig_{in} matches the expected specification of a 2-bit adder, $(2a_1 + a_0 + 2b_1 + b_0)$, proving that the circuit correctly implements the function of an adder.

Algebraic rewriting is plagued, however, by an excessive number of intermediate polynomials generated during rewrit-

ing. Most of them disappear during the rewriting process and some, called *vanishing monomials*, eventually evaluate to 0 but take a large amount of space before reaching the state in which they are removed. The goal is to determine early in the process how to eliminate the vanishing polynomial and how to efficiently carry out the rewriting. A number of rewriting ordering strategies have been employed to reduce the number and size of intermediate polynomials. They include: processing the polynomials of gates with same inputs close to each other; and removing the vanishing monomials, early in the process [34]. Another simplification performed during rewriting occurs due to the reduction of monomials containing x^2 to x, whenever a nonlinear term x^2 is generated during the rewriting. As mentioned earlier, this is more efficient than performing division by polynomials $< x^2 - x >$, associated with each internal signal x.

C. AIG Rewriting

The process of algebraic rewriting can be significantly improved by using a functional And-Invert Graph (AIG) representation employed by the ABC system [29]. ABC uses cut enumeration to detect the XOR and Majority (MAJ) functions with a common set of variables. Those nodes are essential in identifying half-adders (HA) and full-adders (FA), basic components of arithmetic circuits. This makes it possible to skip over the significant portions of the circuitry, from the inputs to the outputs of the adders, as shown in Figure 6, significantly speeding up the rewriting process.



a) b)

Fig. 6: Full Adder: a) circuit diagram; b) AIG representation

D. Verification under Input Constraints

While algebraic rewriting works reasonably well for basic arithmetic functions, including standard multipliers, it becomes unmanageable when applied to complex arithmetic circuits, such as Booth multipliers and dividers. The rewriting in such circuits is plagued by an excessive number of intermediate polynomials and vanishing monomials generated by the rewriting, causing memory overload. This problem is particularly acute in divider circuits.

The difficulty comes from the fact that typical divider architectures are optimized under the input constraint $0 < X \leq 2^{n-1}D$, which is in line with the fact that the dividend X is twice as large in the number of bits than the divisor D, quotient Q and remainder R. As a result, each row of the divider is half the length of the dividend X, as shown in Figure 7 for a restoring divider. This makes it "unclean" for a straightforward rewriting: a large number of monomials become redundant and need to be removed from propagating further. Authors of [35] recognize this problem and propose a method to remove such monomials early in the rewriting process using a combination of simulation and SAT. The detection of such monomials is supported by signal propagation from the primary inputs, hence termed "SAT-based Information Forwarding" (SBIF).

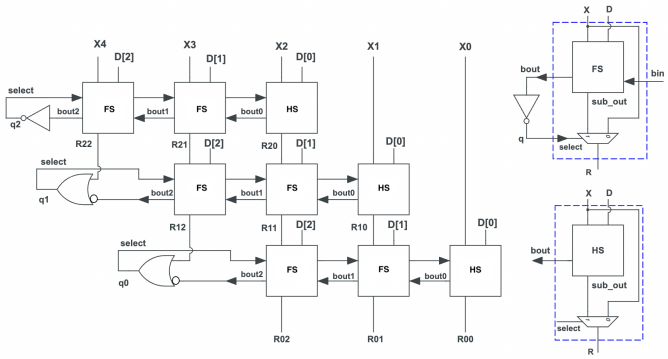


Fig. 7: Architecture of an optimized restoring divider.

The redundant monomials are analogous to don't cares (DC) in logic circuits and are modeled in [35] as *satisfiability don't cares*. The technique proposed there minimizes the number of polynomials propagating through the circuit using ILP optimization. When the size of the polynomial increases by some predefined rate they stop the propagation of polynomials, backtrack to the previous step, apply the ILP optimization to reduce the number of vanishing monomials and continue with the rewriting. One must note, however, that this method depends on reverse engineering for the extraction of the HA/FA blocks in order to gain access to the inputs on which to apply don't cares.

VI. Hardware Rewriting

Recently an original technique for verifying arithmetic circuits has been proposed, in which algebraic rewriting is replaced by hardware synthesis [36]. This approach has been motivated by a need to verify integer and fractional dividers that don't have well defined input/output signatures. In contrast to other arithmetic circuits, such as adders or multipliers, divider does not have a closed form formula to express its output as a function of the inputs. Instead, its functionality is governed by the equation: $X = Q \cdot D + R$, with $R < D$, where X is the dividend, D the divisor, and Q, R are the quotient and remainder, respectively. In principle, one can apply algebraic rewriting to a divider circuit by treating the expression $Q \cdot D + R$ as the output signature and comparing it with X as the input signature [37]. It turns out, however, that this approach is inefficient; the size of intermediate polynomials becomes prohibitively large, causing serious memory overload.

Another way to solve this problem is to create an "inverse" circuit $Z = Q \cdot D + R$ and check the equivalence between its

output Z and the dividend X. A miter is created between the dividend input X and output $Z = Q \cdot D + R$ of the circuit, shown in Figure 8, to check if the CNF formula of the resulting miter circuit is unsatisfiable (unSAT). Unfortunately, the dividers greater than 16 bits also could not be verified using this method.

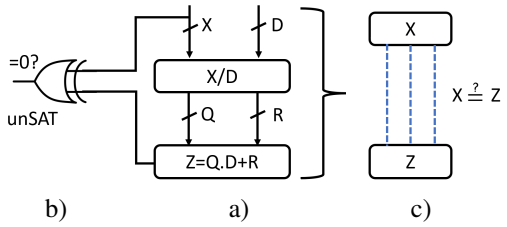


Fig. 8: Divider verification approaches: a) Divider verification model; b) solution with SAT, c) result of hardware reduction.

To address this problem, an approach, originally proposed for SQRT circuits [36], has been investigated, where algebraic rewriting has been replaced by synthesis, informally referred to as "hardware rewriting". It has been demonstrated, however, that this method also cannot handle large circuits. Instead, for the divider architectures similar to those shown in Figure 7, verification can be applied to each row (layer) of the divider. This approach is justified by noting that logic between two adjacent rows of the divider is not optimized during synthesis and the partial remainders R_i are preserved during synthesis. This important feature has been confirmed by [38], and verified by analyzing the dividers synthesized with Synopsys DC and academic tools, such as ABC [29] and Yosys [39]. Such a layer-based verification can also be done in a speculative, parallel manner, since the form of each polynomial at the layer boundary is known, and the verification can be halted 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 main idea of layer-based hardware rewriting is similar to that shown in Figure 8 but applied to a single layer. For layer i computing $R_i = R_{i+1} - q_i D$, an "inverse" circuit Z_i is constructed at the bottom of that layer, which computes $Z_i = q_i D + R_i$. This output should obviously match the partial remainder R_{i+1} on the top of the layer. The goal is then to prove that the n-bit output Z_i matches the n-bit input R_{i+1} bit-by bit. This is accomplished by synthesizing the circuit of the layer and checking the equivalence between its input and output bits. Experiments show that for layers up to 24-bit wide, the result is a redundant circuit composed of a set of direct wires/buffers, connecting the bits of Z_i with bits of R_{i+1}. For larger circuits, where the synthesis does not reduce the structure to such a redundant state, this can be trivially verified using bit-by bit SAT, or by a simple XOR comparison. The verification of the entire circuit is then accomplished by *composing* the verification results of individual layers.

The technique of layered verification shows to be efficient and scalable and can verify dividers of up to 255-bit dividends in single minutes. This is significantly faster than SBIF-based verification [35], or a SAT approach, which times out after one hour on a single 32-bit layer of the divider.

VII. Casing-based Approach

A novel, recently proposed approach to divider verification [40] avoids reverse engineering to determine the layer boundaries, needed by other techniques, and instead relies on a low-level functional analysis of the circuit. Specifically, it uses a "casing" approach by dividing the verification problem into smaller and easier to perform verification tasks. It has been applied to a restoring array divider with an $(2n-1)$-bit dividend X and the n-bit divisor D, quotient Q and remainder R, like the one shown in Figure 7. Recall that the divider is composed of n layers, each performing computation $R_i = R_{i+1} - q_i D$, where R_{i+1} is the partial remainder associated with the top boundary of layer i, and R_i is the partial remainder at the bottom of the layer. In addition, layer i produces a quotient bit q_i. At the boundaries of the entire circuit ($i = n-1$ for the top, and $i = 0$ for the bottom) we have $R_n = X$ and $R_0 = R$, consistent with the computation performed by the circuit described by equation $X = Q \cdot D + R$.

The verification procedure is based on the following observation: the quotient bit q_i, produced as the MSB of each subtraction step, serves as an internal select signal sel_i, which determines whether the input vector R_{i+1} or the difference $R_{i+1} - D$ is produced at the output R_i of the layer. Specifically, 1) For $sel_i = q_i = 0$, the signals at the top of the layer are connected directly to those at the bottom: $R_i = R_{i+1}$; this case is referred to as the *vertical* flow, shown in Figure 9 by color lines. 2) For $sel_i = q_i = 1$, the layer performs subtraction, $R_i = R_{i+1} - q_i D$, referred to as the *horizontal flow*.

The sel_i signal of layer i can be identified in the synthesized logic as a signal originating at q_i entering as input to the controlled subtractor. Vertical verification is then accomplished by setting $sel_i = 0$; and the horizontal verification by $sel_i = 1$. It is important to note that the vertical verification makes it also possible to identify the layer boundaries, without any need for reverse engineering. This is shown in Figure 9 with the directed dotted lines.

Verification of the divider considers both the vertical and horizontal flow. By setting sel_i = 0 and resynthesizing the circuit, the vertical flow of layer i is verified by checking if the condition R_i = R_{i+1} holds for that layer. The horizontal flow is verified using *combinational equivalence checking* (CEC) between the layer synthesized with $sel_i = 1$ and a generic reference subtractor. A global proof is then conducted to demonstrate that once each layer has been verified to implement a controlled subtractor, the entire array correctly implements a divider, satisfying the divider's characteristic equation: $X = Q \cdot D + R$, with $0 \leq R < D$.

The verification approach conducted at the logic level, as described here, offers a significant advantage over the complex rewriting techniques and SAT [40].

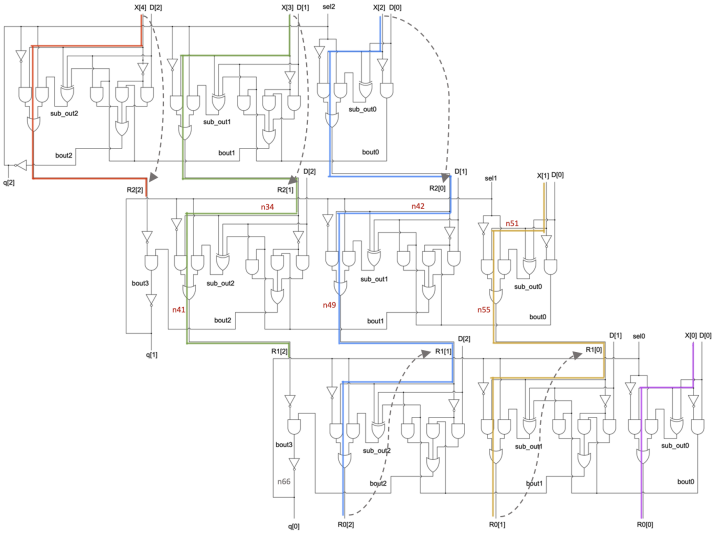


Fig. 9: Restoring divider: dotted arcs show identification of boundary signals for Layer 1 by setting $sel_0, sel_2 = 0$; vertical verification shown by color lines.

VIII. Challenges

The techniques described in the paper concentrate on the verification problem, whose goal is to confirm that the circuit implements the desired function. However, very little work has been done on *debugging*, the detection and correction of *faulty* circuits. New methods need to be developed to address the debugging issue that should work on the lower, gate level circuits. The debugging approaches offered in the literature either consider standard stack-at faults or target the gate replacement, but are unable to handle arbitrary logic faults. The techniques are needed to pin-point the bug to a small logic area that can be easily corrected by the designer of automatically replaced by the correct logic, but not just on a gate-by-gate manner.

Efficient techniques are also needed to handle floating point (FP) arithmetic, so they can verify all components of the FP divider, including normalization, exponent and mantissa blocks, in addition to the integer portion described here. Furthermore, new techniques are needed to handle drastically different division schemes, such as non-array divisors, table-based SRT, or Goldsmidth division. These methods need new insight into the underlying division schemes translated into lower level hardware. Similarly, dedicated verification and debugging methods need to be developed for custom hardware accelerators, with non-standard arithmetic components.

Finally, new methods are needed to support *modular arithmetic* used in cryptography. Recent advances in cryptography, needed to support cloud computing with adequate security, confidentiality and data privacy lead to the development of advanced encryption techniques. One of the recent and most ambitious inventions in cryptography is fully homomorphic encryption (FHE), which makes it possible to perform computation directly on an encrypted data without an intermediate decryption. Several advanced architectures based on a number-theoretic transform (NTT) have been developed to support this new technology [41][42]. These architectures, while offering significant computing speedup in addition to the required security, pose a considerable challenge for verification. Cryptographic circuits used in those systems have a high burden of proof for mathematical correctness that elevates the need for full circuit verification. In particular, verification of modular multipliers which perform computations in polynomial ring $R = \mathbb{Z}[x]$ modulo some reduction polynomial $f(x)$, with bit-widths exceeding 4,096 bits, is a hard mathematical problem.

Figure 10 shows the architecture of a single NTT arithmetic core taken from [42] with large integer multiplication and the modular reduction block. Verification tools should be able to verify the reduction algorithm, such as Barrett or Montgomery, applied in the system, as well as the circuit itself to prove functional correctness of the computed result. The modular multiplier is the costliest one in terms of the CPU computation time and hardest to verify from the mathematical point of view. Novel approaches need to be develop to deal with these challenging tasks.

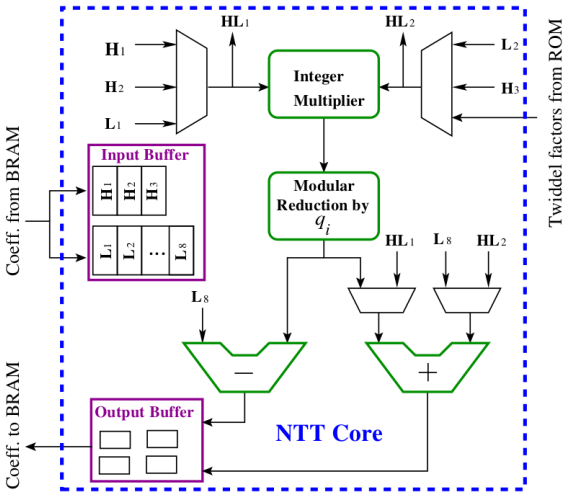


Fig. 10: Architecture of NTT core [42].

In general, the verification problem for complex systems composed of arithmetic circuits cannot be solved with a single verification technique in polynomial space and time w.r.t. to the size of the circuit. Instead, a hybrid and hierarchical approach is needed to handle its complexity. Such an approach should be able to mix formal, simulation and test-like methods to make it possible to verify the overall system composed of pre-verified components in polynomial computation time. As stated in a recent paper "With additional knowledge about the boundaries of components, polynomial verification becomes possible through the step-wise verification of sub-components and the use of different formal proof engines" [43]. Those techniques should target large systems, composed of large arithmetic components, combined with higher level models to prove the functional correctness of the overall system.

IX. ACKNOWLEDGMENT

The work summarized here has been supported by grants from the National Science Foundation, Awards No. CCF-1617708 and CCF-2006465. It is the result of years of research by a number of talented PhD students, including: Serkan Askar, Priaynk Kalla, Cunxi Yu, Atif Yasin, Tiankai Su, and Jiteshri Dasari, who worked diligently on these problems during their doctoral studies at the University of Massachusetts, Amherst.

REFERENCES

[1] S. Vasudevan, V. Viswanath, R. W. Sumners, and J. A. Abraham, "Automatic Verification of Arithmetic Circuits in RTL using Stepwise Refinement of Term Rewriting Systems," *IEEE Trans. on Computers*, vol. 56, no. 10, pp. 1401–1414, 2007.

[2] R. Kaivola, R. Ghughal, N. Narasimhan, A. Telfer, J. Whittemore, S. Pandav, A. Slobodova, C. Taylor, V. Frolov, E. Reeber, and A. Naik, "Replacing testing with formal verification in intel coretm i7 processor execution engine validation." in *CAV*, 06 2009, pp. 414–429.

[3] D. M. Russinoff, *Formal Verification of floating-point hardware design: a mathematical approach*. Springer, 2018.

[4] M. Temel, A. Slobodova, and W. A. Hunt, "Automated and scalable verification of integer multipliers," in *International Conference on Computer Aided Verification*. Springer, 2020, pp. 485–507.

[5] ACL2, "Acl2 software," http://www.cs.utexas.edu/users/moore/acl2/v8-3/combined-manual/index.html?topic=ACL2TUTORIAL.

[6] Coq, "Coq, Proof Assistant," https://coq.inria.fr/documentation.

[7] Isabelle, "Isabelle, Proof Assistant," http://citebay.com/how-to-cite/isabelle-hol.

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

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

[10] Yimg-An Chen and Bryant, "*phdd: an efficient graph representation for floating point circuit verification," in *1997 Proceedings of IEEE ICCAD*, Nov 1997, pp. 2–7.

[11] M. Ciesielski, P. Kalla, and S. Askar, "Taylor Expansion Diagrams: A Canonical Representation for Verification of Data Flow Designs," *IEEE Trans. on Computers*, vol. 55, no. 9, pp. 1188–1201, Sept. 2006.

[12] B. Becker, R. Drechsler, and R. Werchner, "On the relation between bdds and fdds," *Inf. Comput.*, vol. 123, pp. 185–197, 1995.

[13] R. E. Bryant and Y. Chen, "Verification of arithmetic functions with binary moment diagrams," Pittsburgh, PA, USA, Tech. Rep., 1994.

[14] K. Hamaguchi, A. Morita, and S. Yajima, "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.

[15] M. Ciesielski, D. Gomez-Prado, Q. Ren, J. Guillot, and E. Boutillon, "Optimization of Data-Flow computation using Canonical TED Representation," *IEEE Trans. on Computers*, vol. 28, no. 9, pp. 1321–1333, September 2009.

[16] J. P. Marques-Silva and K. A. Sakallah, "Grasp: A new search algorithm for satisfiability," *Proceedings of International Conference on Computer-Aided Design*, pp. 220–227, 1996.

[17] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik, "Chaff: Engineering an efficient SAT solver," in *Proceedings of the 38th Design Automation Conference, DAC 2001, Las Vegas, NV, USA, June 18-22, 2001*, 2001, pp. 530–535.

[18] N. Sörensson and N. Eén, "MiniSat 2.1 and MiniSat++ 1.0 - SAT race 2008 editions," *SAT*, p. 31, 2009.

[19] G. L. D. Davis, Martin; Logemann, "Communications of the acm," in *A Machine Program for Theorem Proving*, 1962.

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

[21] A. Niemetz, M. Preiner, and A. Biere, "Boolector 2.0," *Journal on Satisfiability, Boolean Modeling and Computation*, vol. 9, 2015.

[22] L. De Moura and N. Bjørner, "Tools and algorithms for the construction and analysis of systems," in *Z3: An efficient smt solver*, 2008.

[23] A. Stump, C. W. Barrett, and D. L. Dill, "CVC: A Cooperating Validity Checker," in *14th International Conference on Computer Aided Verification (CAV)*, ser. Lecture Notes in Computer Science, E. Brinksma and K. G. Larsen, Eds., vol. 2404. Springer-Verlag, 2002, pp. 500–504, copenhagen, Denmark.

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

[25] J. Lv, P. Kalla, and F. Enescu, "Efficient Gröbner Basis Reductions for Formal Verification of Galois Field Arithmatic Circuits," *TCAD*, vol. 32, no. 9, pp. 1409–1420, September 2013.

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

[27] D. Ritirc, A. Biere, and M. Kauers, "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.

[28] C. Yu, W. Brown, D. Liu, A. Rossi, and M. J. Ciesielski, "Formal verification of arithmetic circuits using function extraction," *TCAD*, vol. 35, no. 12, pp. 2131–2142, 2016.

[29] A. Mishchenko *et al.*, "Abc: A system for sequential synthesis and verification," *URL http://www. eecs. berkeley. edu/˜ alanmi/abc*, 2007.

[30] N. Shekhar, P. Kalla, and F. Enescu, "Equivalence Verification of Polynomial Data-Paths Using Ideal Membership Testing," *TCAD*, vol. 26, no. 7, pp. 1320–1330, July 2007.

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

[32] D. Ritirc, A. Biere, and M. Kauers, "Column-wise verification of multipliers using computer algebra," in *FMCAD'17*, 2017.

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

[34] A. Mahzoon, D. Große, and R. Drechsler, "REVSCA-2.0: SCA-based formal verification of non-trivial multipliers using reverse engineering and local vanishing removal," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2021.

[35] C. Scholl, A. Konrad, A. Mahzoon, D. Große, and R. Drechsler, "Verifying dividers using symbolic computer algebra and don't care optimization," in *2021 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2021, pp. 1110–1115.

[36] A. Yasin, T. Su, S. Pillement, and M. Ciesielski, "SPEAR: hardware-based implicit rewriting for square-root circuit verification," *2020 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pp. 532–537, 2020.

[37] A. Yasin, T. Su, S. Pillement, and M. Ciesielski, "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.

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

[39] C. Wolf, "Yosys open synthesis suite," https://yosyshq.net/yosys/.

[40] J. Dasari and M. Ciesielski, "Formal verification of restoring dividers made fast and simple," in *60th Design Automation Conference (DAC)*. IEEE, 2023.

[41] S. Riazi, K. Laine, B. Pelton, and W. Dai, "Heax: An architecture for computing on encrypted data," 03 2020, pp. 1295–1309.

[42] S. Sinha Roy, F. Turan, K. Jarvinen, F. Vercauteren, and I. Verbauwhede, "Fpga-based high-performance parallel architecture for homomorphic computing on encrypted data," in *2019 IEEE Intl. Symposium on High Performance Computer Architecture (HPCA)*, 2019, pp. 387–398.

[43] R. Drechsler and A. Mahzoon, "Preserving design hierarchy information for polynomial formal verification," in *2022 IFIP/IEEE 30th Intl. Conference on Very Large Scale Integration (VLSI-SoC)*, 2022, pp. 1–7.