EPiC Series in Computing
Volume 57, 2018, Pages 656–666
LPAR-22. 22nd International Conference on Logic for Programming, Artificial Intelligence and Reasoning

Rewriting Environment for Arithmetic Circuit Verification

Cunxi Yu[1], Atif Yasin[2], Tiankai Su[2], Alan Mishchenko[3], and Maciej Ciesielski[2]

[1] École Polytechnique Fédérale de Lausanne, Switzerland; cunxi.yu@epfl.ch
[2] University of Massachusetts, Amherst, MA, USA; {ayasin,tiankaisu,ciesiel}@umass.edu
[3] University of California, Berkeley, CA, USA; alanmi@berkeley.edu

Abstract

The paper describes a practical software tool for the verification of integer arithmetic circuits. It covers different types of integer multipliers, fused add-multiply circuits, and constant dividers - in general, circuits whose computation can be represented as a polynomial. The verification uses an algebraic model of the circuit and is accomplished by rewriting the polynomial of the binary encoding of the primary outputs (output signature), using the polynomial models of the logic gates, into a polynomial over the primary inputs (input signature). The resulting polynomial represents arithmetic function implemented by the circuit and hence can be used to extract functional specification from its gate-level implementation. The rewriting uses an efficient *And-Inverter Graph* (AIG) representation to enable extraction of the essential arithmetic components of the circuit. The tool is integrated with the popular ABC system. Its efficiency is illustrated with impressive results for integer multipliers, fused add-multiply circuits, and divide-by-constant circuits. The entire verification system is offered in an open source ABC environment together with an extensive set of benchmarks.

1 Introduction

Verification of arithmetic circuits can be viewed as a special case of *combinational equivalence checking*, in which the function implemented by the circuit is checked against its functional specification. Boolean methods, such as various canonical decision diagrams and SAT, that have been used extensively in logic synthesis and optimization, are computationally too expensive for arithmetic functions as they require "bit blasting", i.e., flattening the design to a bit-level netlist. The SAT and SMT competition results confirm that the verification of even small multipliers pose a real challenge to such solvers [15]. Similarly, the commercial tools cannot fully automatically handle full-size multipliers [17]. In general, the complexity of checking equivalence of large arithmetic circuits is too high for these methods [13][20].

The techniques that offer best solution in arithmetic circuits verification are formal methods based on *computer algebra* [20][10][12][17]. In this approach, the circuit specification and its implementation are represented as polynomials in binary signal variables. The verification problem is formulated as a proof that the implementation satisfies the specification. It is accomplished by reducing the specification modulo the implementation polynomials using theory of *Gröbner Basis*, which transform the verification problem into an *ideal membership testing* of the specification polynomial in the ideals [12][10][17][15]. Some of the authors [10][5] use Gaussian elimination, rather than explicit polynomial division, to speed up the reduction process.

An alternative, and more effective approach to accomplish the verification proof for gate-level arithmetic circuits is based on *algebraic rewriting* [20][21]. It transforms the polynomial at the primary outputs (called the *output signature*) into a polynomial in terms of primary inputs (the *input signature*) [20]. The resulting signature provides the functional specification of the circuit that can be compared with the expected specification; hence the method can also serve as *function extraction*. Although this approach has been successfully applied to large-scale multipliers and other arithmetic circuits, it still suffers from a potential memory explosion problem during rewriting due to the growing size of the intermediate polynomials. In particular, the method is very sensitive to the order in which rewriting is done, strongly affecting the verification performance.

The verification method and the tool presented in this paper offer an important step in finding and efficient solution to the arithmetic verification problem. The method is based on representing the circuit in a *functional*, rather than structural, gate-level domain, called the *And-Inverter Graph* (AIG) [11], in which the algebraic rewriting is done on the AIG representation of the circuit.

2 Algebraic Rewriting

Arithmetic circuit considered in this work is a circuit that computes polynomial expressed in the input variables. The circuit is modeled as a network of interconnected bit-level components (logic gates), each with a finite set of binary inputs and a single binary output. Each gate is modeled as a unique polynomial $f_i[X]$ with binary variables $X = \{x_1, ..., x_n\}$ and coefficients in $\mathbb{Z}_2$. Such a polynomial is also referred to as a *pseudo-Boolean polynomial*. Table 1 presents algebraic models of some of the basic Boolean operators [20].

Table 1: Boolean and algebraic models of basic logic functions.

Operation	Boolean model	Algebraic model
$INV(a)$	$\neg a$	$1 - a$
$AND(a,b)$	$a \wedge b$	ab
$OR(a,b)$	$a \vee b$	$a + b - ab$
$XOR(a,b)$	$a \oplus b$	$a + b - 2ab$
$XOR3(a,b,c)$	$a \oplus b \oplus c$	$a + b + c - 2ab - 2ac - 2bc + 4abc$
$MAJ3(a,b,c)$	$a \wedge (b \vee c) \vee b \wedge c$	$ab + ac + bc - 2abc$

By construction, each expression evaluates to a binary value (0,1) and correctly models the logic function of a Boolean 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, algebraic model for logic gate $g = a \vee (b \wedge c)$ can be derived as $g = a + bc - abc$, etc.

Algebraic rewriting relies on relating two pseudo-Boolean polynomials, called an output signature and an input signature. The *output signature*, Sig_{out}, is the the polynomial that represents the result stored as the binary encoding of the primary outputs. For example, an output signature of a signed 2's complement arithmetic circuit with n bits, $Sig_{out} = -2^{n-1}z_{n-1} + \sum_{i=0}^{n-2} 2^i z_i$. By construction, such a polynomial is unique. Similarly, the *input signature*, Sig_{in}, is the polynomial over the primary input variables that represents the arithmetic function performed by the circuit, i.e., its *functional specification*. For example, for an n-bit binary adder with inputs $\{a_0, \cdots, a_{n-1}, b_0, \cdots, b_{n-1}\}$, $Sig_{in} = \sum_{i=0}^{n-1} 2^i a_i + \sum_{i=0}^{n-1} 2^i b_i$. In our approach, the input specification need not to be known; it will be derived from the circuit implementation by algebraic rewriting.

Algebraic rewriting is the process of transforming Sig_{out} into Sig_{in} using algebraic models of the internal components (logic gates) of the circuit, such as those specified by Table 1. By definition, it is done in the 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* [20]. Intermediate expression obtained during rewriting is also represented as a polynomial, referred to as as *signature*, over the variables representing the internal signals of the circuit. By construction, each variable in a given signature polynomial (starting with Sig_{out}) represents an output of some logic gate. The rewriting transformation simply replaces that variable with the algebraic expression of the corresponding logic gate.

It has been shown that such a backward rewriting produces a unique input signature polynomial [20]. However, the rewriting performance strongly depends on the order in which the individual variables are rewritten. Two basic rules are used in determining the rewriting order: (1) Rewriting follows the reverse topological order; and (2) Signals that depend on common signals (fanins) are rewritten together (i.e., one immediately after the other). The first rule is obvious because of the direction in which the signature is propagated. Once a given variable (output of a gate) is rewritten, i.e., substituted by an algebraic expression of the gate inputs, it will be eliminated from the current expression and will never appear in the signature again. As a result, the final signature will be expressed in the primary inputs (PI) only. The second rule is dictated by the fact that rewriting the nodes with common fanins together maximizes the chance for potential term cancellation, hence minimizing the size of intermediate polynomials.

To illustrate the rewriting process consider the following example of a gate-level arithmetic circuit with inputs a, b, c_0, shown in Figure 1(a). The output signature of the circuit is $Sig_{in} = 2C + S$, determined by the weights of the two output signals dictated by the binary encoding. The goal is to determine the arithmetic function implemented by this circuit (or, equivalently to verify if it is a full adder) by rewriting Sig_{out} into an input signature (specification), Sig_{in}.

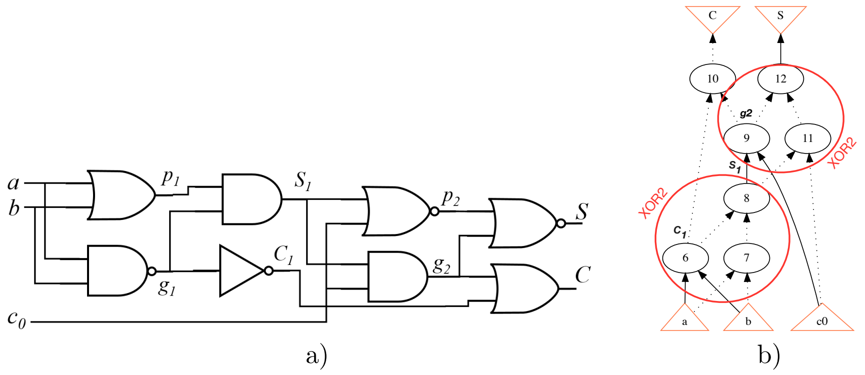

Figure 1: Gate-level arithmetic circuit (FA): a) circuit diagram; b) AIG representation

According to the rewriting algorithm [20] the optimum rewriting order is $\{(S, C), (p_2, g_2), (S_1, C_1), (p_1, g_1)\}$. The signals shown in brackets are the ones that depend on common inputs; they are to be rewritten together, i.e., one immediately after the other.

The following is a series of the rewriting steps, applied in the reverse topological order, for the gate-level circuit shown in Figure 1(a), using the algebraic models of the gates in Table 1. The terms shown in bold face are reduced to 0 during simplification. For brevity, the substitution is shown for each pair of variables at once. For example: (C,S) means rewriting using C and S variables.

$$
\begin{aligned}
Sig_{out} &= 2C + S \\
1.\ (S,\ C\ ) : &= 2(C_1 + g_2 - C_1g_2) + (1 - (p_2 + g_2 - p_2g_2)) \\
&= 2C_1 + g_2 - 2C_1g_2 - p_2 + p_2g_2 + 1 \\
2.(p_2,\ g_2) : &= 2C_1 + S_1c_0 - 2S_1C_1c_0 - (1 - (S_1 + c_0 - S_1c_0)) + (1 - (S_1 + c_0 - S_1c_0))S_1c_0 + 1 \\
&= 2C_1 + \mathbf{S_1c_0} - 2S_1C_1c_0 + S_1 + c_0 - \mathbf{S_1c_0} + \mathbf{S_1c_0} - \mathbf{S_1^2c_0} - \mathbf{S_1c_0^2} + \mathbf{S_1^2c_0^2} \\
&= 2C_1 - 2S_1C_1 + S_1 + c_0 \qquad (1) \\
3.(S_1, C_1) : &= 2(1 - g_1) - 2(1 - g_1)(p_1g_1)c_0 + p_1g_1 + c_0 \\
&= 2 - 2g_1 - 2(\mathbf{p_1g_1} - \mathbf{p_1g_1^2}) + p_1g_1 + c_0 \\
&= 2 - 2g_1 + p_1g_1 + c_0 \\
4.(p_1,\ g_1) : &= 2 - 2(1 - ab) + (a + b - ab)(1 - ab) + c_0 \\
&= \mathbf{2ab} + a + b - ab - \mathbf{a^2b} - \mathbf{ab^2} + \mathbf{a^2b^2} \ = \ a + b + c_0
\end{aligned}
$$

The resulting input signature is $Sig_{in} = a + b + c_0$, indicating that this is a full adder. During the rewriting two types of simplifications can be observed:

- Simplification of terms with same monomials; for example, $2g_2 - g_2 = g_2$, in Step 1.
- Lowering the term x^k with degree $k > 1$ to x, since the signal variables are binary, i.e., $x^k = x$. This can be seen in Step 3 of the rewriting, shown there in bold face: $(p_1g_1 - p_1g_1^2) = p_1g_1 - p_1g_1 = 0$. Similar simplifications appear also in steps 2 and 4.

Two modes of rewriting are possible: 1) Verification against the known specification, and 2) extracting the specification from the circuit structure. If the specification of the circuit is known, one needs to compare the computed input signature with this specification. While this can be done using canonical polynomial representations, such as TED or BMD, this comparison can be avoided altogether by rewriting the *difference* between the output and input signature, $Sig_{out} - Sig_{in}$ instead of Sig_{out}. The result of such a rewriting should be zero for a correct circuit. A non-zero result is an indication of a bug. In the case when the specification is not known, the computed input signature provides the function of the circuit (buggy or not).

In the case of a buggy circuit, the size of intermediate polynomials during rewriting may become prohibitively large, sometimes even preventing the computation from completing. This by itself can be used as a warning that the circuit is probably faulty. In general, concluding the the circuit is incorrect and identifying a bug is a challenging problem. Several attempts have been made to identify the bug(s), either by comparing the result of backward and forward rewriting [7] or by analyzing the difference between the computed input signature and the given specification [6]. With a notable exception of finite field (GF) arithmetic circuits [19] [8][14], the debugging remains an open problem.

3 AIG Rewriting

In contrast to the algebraic rewriting applied directly to a gate level circuit, as in Figure 1(a), the rewriting employed in our tool operates on the functional AIG representation of the circuit [21]. AIG (And-Inverter Graph) is a combinational Boolean network composed of two-input AND gates and inverters [1]. 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 to detect XOR and Majority (MAJ) functions with a common set of variables. Those nodes are essential in identifying half-adders (HA) and full-adders (FA), the basic components of an arithmetic circuit [21]. AIG rewriting then skips over the large portions of the circuitry, from the inputs to the outputs of the adders, significantly speeding up the rewriting process, as shown in Figure 1(b). The algorithm is outlined in Algorithm 1 [21].

Algorithm 1 Algebraic Rewriting in AIG

```
Input: Gate-level netlist $N$; Output signature $Sig_{out}$
Output: Pseudo-Boolean expression extracted by rewriting
 1: $G(V, E) \leftarrow$ structural hashing of $N$ into AIG.
 2: Detect all XOR3 and MAJ3 nodes in $G(V, E)$.
 3: $P \leftarrow$ pair(XOR3, MAJ3) nodes with common signals.
 4: Topological sort $G(V,E)$ considering each element in $P$ as one node.
 5: $i = 0$; $F_i = Sig_{out}$
 6: while there remain elements in $V$ do
 7:     Rewrite: $F_{i+1} \leftarrow F_i$ by variable substitution;
 8:     $i = i + 1$
 9: end while
10: return $F = F_i$ (to be compared with $Sig_{in}$)
```

The inputs to the algorithm are the gate-level netlist N and the output signature Sig_{out} and includes four basic steps: 1) converting the gate-level implementation into AIG; 2) detecting all pairs of (XOR3, MAJ3) functions with common AIG inputs[1]; 3) performing topological sorting of AIG nodes while treating the detected XOR and MAJ functions as a single element; and 4) applying algebraic rewriting from POs to PIs following the reverse topological order. As soon as the matching (XOR3, MAJ3) pairs are detected, a hybrid graph G is constructed, in which each XOR3 and MAJ3 function is considered as a single node. In the absence of XOR3, MAJ3 nodes, the two-input XOR2 and MAJ2(AND) functions are similarly detected. Algebraic rewriting is then applied to the modified graph G in a reverse topological order. The algorithm returns the extracted input signature Sig_{in}.

In the example of Figure 1(b), the groups of nodes (6,7,8) and (9,11,12) are identified as XOR2, and nodes 6 and 9 as the matching MAJ2 (AND) functions. Subsequently, the functions at node 12 (S) and node 10 (C) are identified as XOR3 and MAJ3, respectively, sharing the same inputs, a, b, c_0. At this point the entire graph G reduces to just two nodes, representing XOR3(a, b, c) and MAJ3(a, b, c). The rewriting of $Sig_{out} = 2C + S$ over the two nodes is trivial, with the nonlinear monomials cancelled as follows (refer to Table 1):

$$2C + S = 2(ab + ac_0 + bc_0 - 2abc_0) + (a + b + c_o - 2ab - 2ac_0 - 2bc_0 + 4abc_0) = a + b + c_o$$

As illustrated with this example, the AIG rewriting requires considerably fewer terms than the standard algebraic rewriting.

4 Results

The algebraic rewriting environment was implemented in C and integrated with the ABC tool [1], where it is available under command &*polyn*. Here we present an open source framework of Algebraic RewriTing (ARTi) system for verifying arithmetic circuits using the most recent version of ABC[2]. The results include some challenging nonlinear arithmetic circuits: large multipliers and divide-by-constant circuits. Comparisons are made w.r.t. the state-of-the art tools

[1]XOR2 and MAJ2(AND2) are special cases of XOR3 and MAJ3, with one of the inputs being constant zero.

[2]https://github.com/ycunxi/abc. This repository includes modifications in addition to original ABC https://github.com/berkeley-abc/abc.

in this domain, [15][16] and [17], which are all computer algebra based systems. The comparison with SAT, SMT, and commercial systems are not provided here since the computer algebraic approach has already been proved to be orders of magnitude faster than those techniques, as discussed in [20]. Other sources also report inadequate quality of these tools for arithmetic verification [17][15].

4.1 Multipliers

The experiments were conducted on benchmarks released in [15][16][3]. For fair comparison, we recompiled their C code on our platform along the state-of-the-art computer algebra system, Singular v4.1.1 [4], used by those systems. The experiments were conducted on a PC with Intel(R) Xeon CPU E5-2420 2.20 GHz x24 with 1 TB memory. The memory out (MO) limit is 100 GB and timeout (TO) limit is 3600 seconds. Singular reports error state (ES) if the circuit contains more than 32,767 ring variables (limit imposed by Singular). The verification results for pre-synthesized multipliers are included in Table 2. The results in column ARTi are generated using three sets of commands, for *btor, sp-ar-rc*, and *abc* multipliers, as follows:

- *read btorXX.aig; &get; &polyn -o -v;* for the *btor*-XX multipliers;
- *read sp-ar-rcXX.aig; &get; &atree; &polyn -o -v;* for the *sp-ar-rc*-XX multipliers;
- *gen -N XXX -m abcXXX.blif; &get; &polyn -o;* for the *abc*-XXX multipliers.

The command *&polyn* includes various algebraic rewriting options, with *-o* flag indicating the use of the older version of the rewriting algorithm [20]. Command *&atree* invokes extraction of adder trees in the circuit.

Table 2: Verification time (sec) for pre-synthesized multipliers. ES = error reported by Singular. TO=Time out of 7200 sec. *Command *&aspec* are applied to Booth multipliers.

Designs	ARTi	[15]	[16]	Designs	ARTi	[15]	[16]
btor-16	0.01	0.5	0.01	sp-ar-rc16	0.01	1.1	0.01
btor-32	0.02	11.7	0.3	sp-ar-rc32	0.1	35.5	0.3
btor-64	0.1	725	4.0	sp-ar-rc64	0.4	1312	4.6
btor-128	0.5	ES	ES	sp-ar-rc128	1.6	ES	ES
abc-256	1.0	ES	ES	abc-512	4.5	ES	ES
abc-Booth-64*	1.0	TO	TO	abc-Booth-256*	1.0	ES	ES

Table 3 shows the results for for multipliers mapped onto standard cells with three different libraries, including industrial libraries of 14 nm and 7 nm technology nodes. The results of verifying the same set of benchmarks using the open source tools available from [15][16], are included. The results of the first seven designs in the Table are generated using command-**a** in Table 2. For the last two circuits, mapped onto industrial libraries, we executed several iterations of *dch* and *strash* commands before ARTi to eliminate extra logic introduced for the purpose of meeting the timing constraints.

To apply ARTi to Booth-encoded multipliers, such as radix-4 Booth multiplier, the design needs to be first preprocessed by extracting the adder trees using the XOR-MAJ extraction approach, described in Section 3, using a command *&atree*. In addition, a semi-canonical spectral approach that represents the arithmetic functions in an algebraic spectrum form can be used to further improve the ARTi for Booth multipliers [22]. The examples of such circuits, verified with this method, are included in our repository[4].

[3]http://fmv.jku.at/algeq/

[4]https://github.com/ycunxi/abc/blob/master/README.md

Table 3: Verification time (sec) of synthesized, technology mapped multipliers using different libraries. #GT = Number of gate types used. FI $\geq$5 = Number of gates with fanin $\geq$5.

Designs	ARTi	#GT	FI≥5	[15]	[16]
btor64-resyn3	0.1	-	-	711	4.2
abc64-resyn3	0.1	-	-	801	4.0
btor128-resyn3	0.3	-	-	ES	ES
abc128-resyn3	0.1	-	-	ES	ES
btor64-resyn3-map-simple	0.3	7	0	1073	418
abc64-resyn3-map-simple	0.1	7	0	1071	415
abc64-resyn3-map-14nm	35	15	17	TO	TO
abc64-resyn3-map-7nm	MO	24	9,791	TO	TO
abc128-resyn3-map-simple	1.8	7	0	ES	ES
abc128-resyn3-map-14nm	406	15	1,008	ES	ES
abc128-resyn3-map-7nm	MO	23	26,600	ES	ES

4.2 Complex Arithmetic Circuits

Table 4 shows the results of extracting word-level specifications from gate-level complex arithmetic circuits, constructed with multiplication and addition operations, and a three-operand multiplier. The multiplications in these datapaths are implemented using ABC-generated multipliers. Our approach can efficiently identify the word-level operations in the gate-level datapaths. In contrast, the approach of [18] could not detect the presence of multiplication or addition in these circuits; and our approach is much faster than [20].

Table 4: Results of extracting word-level specification from complex arithmetic circuits. *TO* = TIME OUT (3600 s). *Error* = Unable to determine type of arithmetic operations. *TO**: finished in 23,760 s.

256-bit	[18]	[20]	**Ours**	
$F=A\times B+C$	Error	TO*	1×mult;1×add	44.7 s
$F=A\times(B+C)$	Error	TO	2×mult	45.1 s
$F=A\times B\times C$	Error	TO	1×mult3	68.5 s

4.3 Dividers (Divide by Constant)

This section presents the results for a special class of dividers, namely divide-by-constant circuits. The input to the circuit are the dividend X and the divisor constant D; the outputs are the quotient Q and the remainder R, concatenated to form an output word, $Z = [R.Q]$. In our experiment, the primary inputs and outputs are n-bit wide. Functional specification of such dividers can be expressed as $X = Q \cdot D + R$, where the divisor D is a constant.

Restoring Divider: First, we consider an architecture based on a standard *restoring divider* [9], in which the divisor D has been hardwired to a particular constant. The restoring divider has been implemented and synthesized using ABC [1]. During synthesis, the constant bits of the divider D have been propagated through the circuit and used to optimize the circuit.

Given a divider circuit to be verified, first step in the verification process is to create an output signature, $Sig_{out} = Q \cdot D + R$, determined by the number of output bits of Q, R and by the value of the constant divisor D. In the case of the divide-by-3 circuit, with X, D, Q, R all being three-bit words, the output signature is

$$Sig_{out} = 3(4Q_2 + 2Q_1 + Q_0) + 4R_2 + 2R_1 + R_0$$

or, alternatively, when expressed in terms of the output bits Z

$$Sig_{out} = 12Z_2 + 6Z_1 + 3Z_0 + 4Z_5 + 2Z_4 + Z_3$$

In our experiment, the initial divider circuit and the output signature were obtainted by the program written in python as follows:

```
python verify_constant_divider_abc.py -f gen-div.blif -divisor 011 -divexp 1+0
```

Here, the *gen-div.blif* file is the generic divider to be converted to a divide-by-constant circuit; -f indicates the required output format type; -*divisor* is the value of the divisor in binary format; and -*divexp* is the value of exponent in the required sum form (for example, constant 3 $= 011 = 2^1 + 2^0$, is written as 1+0). This program produces a divide-by-3 circuit *const-div.blif* and the output signature file, $S.out = 3{*}o0{+}4{*}o1{+}5{*}o2{-}1{*}o0{-}2{*}o1{-}3{*}o2{+}0{*}o3{+}1{*}o4{+}2{*}o5$. The following ABC command is then used to compute Sig_{in}:

```
read const-div.blif;  sweep; strash; dch; &get;
&polyn -v -w -S 3*o0+4*o1+5*o2-1*o0-2*o1-3*o2+0*o3+1*o4+2*o5;
```

with the output signature provided with a -S switch. The resulting input signature for this circuit obtained by ABC is $x_0 + 2x_1 + 4x_2$ (internally encoded in the exponent form as $0 * i0 + 1 * i1 + 2 * i2$). This result matches the primary input, dividend X, which confirms that the circuit correctly implements the division. Table 5 shows the verification CPU runtimes for different divisors for a 16-bit dividend X.

Table 5: Results of verifying the divide-by-constant **restoring divider** circuit for a 16-bit dividend X. Time-out of 20 minutes, Memory-out 24GB.

Divisor	# Rem. bits	Time (s) (No bug)	Divisor	# Rem. bits	Time (s) (No bug)
11	4	2.42	157	8	16.5
17	5	4.13	191	8	MO
31	5	10.7	223	8	317
43	6	9.22	241	8	125
53	6	3.83	251	8	2.20
61	6	9.22	257	9	16.9
73	7	5.96	263	9	223
89	7	11.0	277	9	30.7
101	7	2.14	283	9	22.3
131	8	15.1	311	9	326

Modular Divider: We also present an alternative, *modular divider architecture*, in which the divider is partitioned into a number of identical blocks, instantiated the required number of times, connected in series; each block has a fixed number of bits for the dividend X and quotient Q. A carry-in C into each block comes from the remainder R of the previous block. The number of bits of C and R is fixed and determined by the number of bits of the divisor D. The circuits were generated using an open-source hardware generator, FloPoCo [3], and synthesized using ABC tool [1] onto standard cell, gate-level circuits. For the reason of format incompatibility, these experiments applied the functional verification technique based on standard gate-level rewriting of [20], rather than AIG rewriting. The gate-level synthesized designs from ABC were converted into an algebraic equation format to perform block by block verification.

The program was coded in Python and C++ and the experiments were conducted on a 64-bit Intel Core i5-3470 CPU, 3.20GHz $\times$ 2, Ivy-Bridge with 15.6 GB of memory. Each basic block for a given divisor D is implemented as a lookup table (LUT).

Table 6: Verification results of divide-by-constant divider circuits for a **one-bit block architecture** and a 32-bit dividend X. Time-out of 20 minutes.

Divisor	# Rem. bits	# Gates	Time (s) (No bugs)	# Bugs	Time (s) (With bugs)
17	5	1763	0.81	3	0.75
61	6	3715	3.50	8	3.56
113	7	3652	6.68	7	7.21
241	8	4891	21.7	7	30.42
251	8	6410	110.4	5	113.5
263	9	8114	29.3	8	39.1
277	9	8238	T/O	-	-
283	9	8951	643.8	9	638.4

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 LUT. Table 6 includes the verification time for the divide-by-constant, one-bit (of X) block architecture. The results are shown for a 32-bit dividend X, divisors D value up to 283, and a 9-bit remainder R. The non-monotonic behavior of the verification time as a function of the divisor size can be explained by examining the content (on-set) of the truth table for the corresponding division and its dependence on the value of the divisor.

4.4 Interactive Examples

The following example shows the script and the results of verifying, i.e., deriving the specification of a 64-bit multiplier, using the ABC system with &*polyn* command.

```
abc 01> gen -N 64 -m mult-abc-64.blif; strash; &get; &ps; &polyn -w > mult64.log
```

The results are shown below in two formats: (option 1) implicitly, by listing the number of coefficients appeared in the computed polynomial; and (option 2) explicitly, by listing all the monomials; only a small subset is listed here for brevity.

```
abc 01> gen -N 64 -m mult64-abc.blif;st;strash;ps
Hierarchy reader flattened 8256 instances of logic boxes and left 0 black boxes.
Multi64  : i/o =  128/  128  lat =    0  and =  32064  lev =501
abc 04> &get;&ps;
Multi64  : i/o =  128/  128  and =   32064  lev =  501 (312.05)  mem = 0.37 MB
```

Verbose option 1

```
abc 04> &polyn -w
Polynomial with 4096 monomials:
| +2^0 * i0 * i64
| +2^1 * i0 * i65
  +2^1 * i1 * i64
...
```

Verbose option 2

```
abc 04> &polyn -v
Input signature with 4096 monomials:
  +2^0 appears 1 times
  +2^1 appears 2 times
  +2^2 appears 3 times
  ...
```

The following log shows the usage of the tool by explicitly providing the output signature Sig_{out} in terms of the weights of the output bits, for a 2-bit unsigned integer multiplier. $Sig_{out} = 1z_0+2z_1+4z_2+8z_3$ is coded showing only exponents of the coefficients: *0*o0+1*o1+2*o2+3*o3*, with symbol o_k referring to the kth output bit with coefficient 2^k.

```
abc 01> gen -N 2 -m mult-abc-2.blif; strash; &get; &polyn -w -S 0*o0+1*o1+2*o2+3*o3
```

```
Hierarchy reader flattened 10 instances of logic boxes and left 0 black boxes.
HashC = 7. HashM = 25.  Total = 40. Left = 4.  Used = 4.  Time =     0.00 sec
Input signature with 4 monomials:
  +2^0 appears 1 times
  +2^1 appears 2 times
  +2^2 appears 1 times
Polynomial with 4 monomials:
| +2^0 * i0 * i2
| +2^1 * i0 * i3
  +2^1 * i1 * i2
| +2^2 * i1 * i3
```

The computed input signature is: $Sig_{in} = 1i_0i_2 + 2i_0i_3 + 2i_1i_2 + 4i_1i_3$, which matches the specification of the two-bit unsigned multiplier, $(i_0 + 2i_1)(i_2 + 2i_3)$.

An example of the gate-level rewriting is demonstrated with an earlier (non-AIG based) version of our tool, *petBoss* [2][5]. This tool takes an equation file (an example given in the source directory) and produces the input signature polynomial. The Sig_{out} must be appended as the last line of the netlist equation file.

```
:~/abc/petBoss/petBoss-source: ./petBoss -b < ../mult4-syn.eqn
>>>>>>>>>>> a0*b0+2*a0*b1+2*a1*b0+4*a0*b2+...+32*a2*b3+32*a3*b2+64*a3*b3
```

5 Conclusions

The paper describes a practical tool for functional verification of integer arithmetic circuits. It uses algebraic representation of the circuit and performs an algebraic backward rewriting either structurally, on a gate-level netlist, or functionally, on its AIG representation. The algebraic model explicitly considers finite bit-width words, and as such naturally handles modular, integer modulo 2^n, signed and unsigned arithmetic circuits. Experimental results show the effectiveness of the tool in proving functional verification of add/subtract and fused add-muliply circuits, multipliers, including some Booth-encoded circuits, and a special case of dividers, namely divide-by-constant. Extensions to other arithmetic circuits, including restoring and non-restoring array dividers is currently under work.

References

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

[2] M Ciesielski, C Yu, W Brown, D Liu, and André Rossi. Verification of Gate-level Arithmetic Circuits by Function Extraction. In *DAC 2015*, pages 1–6. ACM, 2015.

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

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

[5] Farimah Farahmandi and Bijan Alizadeh. Groebner basis based formal verification of large arithmetic circuits using gaussian elimination and cone-based polynomial extraction. *Microprocess. Microsyst.*, 39(2):83–96, March 2015.

[5]https://github.com/ycunxi/abc/tree/master/petBoss

[6] Farimah Farahmandi and Prabhat Mishra. Automated test generation for debugging multiple bugs in arithmetic circuits. *IEEE Transactions on Computers*, 2018.

[7] S. Ghandali, C. Yu, D. Liu, W. Brown, and M. Ciesielski. Logic debugging of arithmetic circuits. In *ISVLSI'15*, pages 113–118, July 2015.

[8] Utkarsh Gupta, Irina Ilioaea, Vikas Rao, Arpitha Srinath, Priyank Kalla, and Florian Enescu. On the rectifiability of arithmetic circuits using craig interpolants in finite fields. *Intl. Conf. on VLSI (VLSI-SOC'18)*.

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

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

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

[12] E. Pavlenko, M. Wedler, D. Stoffel, W. Kunz, A. Dreyer, F. Seelisch, and G.M. Greuel. Stable: A new qf-bv smt solver for hard verification problems combining boolean reasoning with computer algebra. In *DATE*, pages 155–160, 2011.

[13] T. Pruss, P. Kalla, and F. Enescu. Efficient symbolic computation for word-level abstraction from combinational circuits for verification over finite fields. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 35(7):1206–1218, July 2016.

[14] Vikas Rao, Utkarsh Gupta, Irina Ilioaea, Arpitha Srinath, Priyank Kalla, and Florian Enescu. Post-Verification Debugging and Rectification of Finite Field Arithmetic Circuits using Computer Algebra Techniques. *FMCAD'18*.

[15] Daniela Ritirc, Armin Biere, and Manuel Kauers. Column-wise verification of multipliers using computer algebra. In *FMCAD'17*, 2017.

[16] Daniela Ritirc, Armin Biere, and Manuel Kauers. Improving and extending the algebraic approach for verifying gate-level multipliers. In *DATE'18*, 2018.

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

[18] Mathias Soeken, Baruch Sterin, Rolf Drechsler, and Robert Brayton. Simulation graphs for reverse engineering. In *Proceedings of the 15th Conference on Formal Methods in Computer-Aided Design*, pages 152–159. FMCAD Inc, 2015.

[19] T. Su, A. Yasin, C. Yu, and M. Ciesielski. Computer algebraic approach to verification and debugging of galois field multipliers. In *2018 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 1–5, May 2018.

[20] Cunxi Yu, Walter Brown, Duo Liu, André Rossi, and Maciej J. Ciesielski. Formal verification of arithmetic circuits using function extraction. *TCAD*, 35(12):2131–2142, 2016.

[21] Cunxi Yu, Maciej J. Ciesielski, and Alan Mishchenko. Fast Algebraic Rewriting Based on And-Inverter Graphs. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 37(9):1907–1911, 2018.

[22] Cunxi Yu, Tiankai Su, Atif Yasin, and Maciej J. Ciesielski. Spectral approach to verifying non-linear arithmetic circuits. *ASP-DAC'19*, 2019.