Arithmetic Bit-Level Verification Using Network Flow Model

Maciej Ciesielski[1], Walter Brown[1], and André Rossi[2]

[1] University of Massachusetts
ECE Department
Amherst, MA 01003, USA
ciesiel@ecs.umas.edu,webrown@umass.edu
[2] Université de Bretagne-Sud
Lab-STICC UMR 6285
56321 Lorient Cedex France
andre.rossi@univ-ubs.fr

Abstract. The paper presents a new approach to functional, bit-level verification of arithmetic circuits. The circuit is modeled as a network of adders and basic Boolean gates, and the computation performed by the circuit is viewed as a flow of binary data through such a network. The verification problem is cast as a Network Flow problem and solved using symbolic term rewriting and simple algebraic techniques. Functional correctness is proved by showing that the symbolic flow computed at the primary inputs is equal to the flow computed at the primary outputs. Experimental results show a potential application of the method to certain classes of arithmetic circuits.

Keywords: Formal verification, Functional verification, Arithmetic verification, Bit-level arithmetic.

1 Introduction

One of the most challenging problems encountered in hardware design is functional verification of arithmetic circuits and datapaths. Boolean logic techniques, based on BDDs, so successfully used in logic synthesis, cannot solve large arithmetic problems as they require "bit-blasting", i.e., flattening of the design into bit-level netlists. Similarly, Boolean satisfiability (SAT) and Satisfiability Modulo Theories (SMT) solvers cannot handle complex arithmetic designs and require solving computationally expensive decision problems. On the other hand, theorem provers, popular in industry, require a significant human interaction and intimate knowledge of the design to guide the proof process. Typical approach in industry is to use a host of methods, including simulation-based and formal methods, which requires large teams of experts with high degree of expertise. While datapath verification has reached certain level of maturity [1,2], certain areas of arithmetic verification remain open for more research. According to Slobodova [3] "Multiplication function is beyond the capacity of BDDs and

V. Bertacco and A. Legay (Eds.): HVC 2013, LNCS 8244, pp. 327–343, 2013.
© Springer International Publishing Switzerland 2013

SAT solvers"; it requires decomposition into smaller entities, while there is "No automatic way of finding properties on the decomposition boundary".

The work described in this paper addresses some of those issues. It focuses on *functional verification*, i.e., proving correctness of arithmetic design w.r.t. its intended function, rather than targeting a specific property or checking equivalence between the implementation and specification. In this sense, functional verification can be viewed as a more general problem, as it has to overcome the issue of generating a complete set of properties that describe the intended functionality. Our approach is based on modeling an arithmetic circuit as a network of half adders and basic Boolean connectors and viewing the computation performed by the circuit as a flow of binary data through the network. The verification problem is cast as a special case of a Network Flow problem and solved using symbolic term rewriting and linear algebraic techniques.

1.1 Related Work

Several approaches have been proposed to check an arithmetic circuit against its functional specification. Different variants of canonical, graph-based representations have been proposed, including Binary Decision Diagrams (BDDs), Binary Moment Diagrams (BMDs), Taylor Expansion Diagrams (TED), and others [4]. Application of BDDs to verification of arithmetic circuits is somewhat limited due to prohibitively high memory requirement for complex arithmetic circuits, such as multipliers. BDDs are being used, along with many other methods, for local reasoning, but not as monolithic data structure [3,1,2]. BMDs and TEDs offer a linear space complexity but require word-level information of the design, which is often not available or is hard to extract from bit-level netlists. A number of SAT solvers have been developed to solve generic Boolean decision problems. The one potentially relevant to our work is CryptoMiniSAT, which targets XOR-rich bio-informatics circuits by replacing traditional CNF formula with XORs [5]. However, it is still based on a computationally expensive DPLL decision process and does not scale with the design size. Several techniques combine linear arithmetic constraints with Boolean SAT in a unified algebraic domain [6] or use ILP to model the modulo semantics of the arithmetic operators [7] [8]. In general, ILP models are computationally expensive and are not scalable. Some techniques combine a word-level version of automatic test pattern generation (ATPG) and modular arithmetic constraint-solving techniques for the purpose of test generation and assertion checking [9]. SMT solvers integrate different theories (Boolean logic, linear integer arithmetic, etc.) into a DPLL-style SAT decision procedure [10]. However, in their current format, the SMT tools are not efficient at solving decision problems that appear in arithmetic circuits.

A number of Computer Algebra methods have been introduced to model arithmetic components as polynomials [11,12]. Automated techniques for extracting arithmetic bit level (ABL) information from gate level netlists have been proposed in the context of property and equivalence checking [13]. ABL components are modeled by polynomials over unique ring, and the normal forms are computed w.r.t. Grobner basis over rings $Z/2^n$ using modern computer algebra

algorithms. In our view this model is unnecessarily complicated and not scalable to practical designs. A simplified version of this technique replaces the expensive Grobner base computation with a direct generation of polynomials representing circuit components [15]. However, no practical method for deriving such large polynomials and no systematic comparison against the specification have been proposed. Our work addresses this issue using a more efficient network flow model.

Industry also uses Theorem Provers, deductive systems for proving that an implementation satisfies a specification, using mathematical reasoning. The proof system is based on a large (and problem-specific) database of axioms and inference rules, such as simplification, rewriting, induction, etc. Some of the known theorem proving systems are: HOL, PVS, and Boyer-Moore/ACL2. The success of verification depends on the set of available axioms, rewrite rules, and on the *order* in which they are applied during the proof process, with no guarantee for a conclusive answer. Similarly, term rewriting techniques, such as [14], are incomplete, as they rely on simple rewriting rules (distributivity, commutativity, and associativity) and use non-canonical representations.

An entirely different approach to functional arithmetic verification has been proposed in [16]. In this approach the arithmetic circuit, composed of adders and connecting logic gates, is described by a system of linear equations. The resulting set of linear equations is then reduced to a single algebraic expression (the "signature" of the circuit) using Gaussian elimination and linear algebra techniques. If the resulting signature matches the input and output expressions (specified by input bit positions and binary output encoding) and does not contain any internal signals, then the circuit is considered functionally correct. The difficulty of this method lies in proving the case when not all signals can be eliminated and the signature contains a "residual expression"(RE), in those variables. In this case, for the circuit to be functionally correct, the residual expression must evaluate to zero. Proving this requires solving a separate and difficult Boolean problem. Furthermore, such an expression is not unique and the method does not offer means for choosing RE that would be easiest to solve.

1.2 Novelty and Contribution

In this work we follow the algebraic approach similar to [16], but solve the problem by modeling it as a computationally simpler *network flow* problem. Specifically, the computation performed by the circuit is modeled as a flow of binary data, represented as an algebraic, pseudo-Boolean expression. This representation provides important information about the circuit functionality and location of possible bugs. The verification proof reduces to showing the equivalence between the input and output expressions. Any possible discrepancy between the two expressions is captured in an algebraic expression, which, in contrast to "residual expression" in [16], is unique and related to fanouts and other signals that can be identified a priori. This feature greatly simplifies the final proof which can be solved using purely algebraic methods.

In contrast to theorem provers and traditional term rewriting techniques, the proposed method is complete. It is based on a complete set of algebraic expressions describing internal circuit modules, used as the rewriting rules. The result does not depend on the order in which the rules are applied; the order is fixed and unique. The method does not require expertise in formal verification, can be fully automated, and always terminates with a conclusive answer. Furthermore, no assumption is made about any structural similarity between the implementation and the specification, required by commercial verification tools.

2 Technical Approach

In this work we are concerned with a class of arithmetic circuits, i.e., combinational circuits with binary inputs that compute a (signed or unsigned) integer function; the result computed by the circuit is encoded in a finite number of binary outputs. The internal operators (circuit modules) are assumed to be binary adders (single-bit half adders and full adders) and basic Boolean logic gates. Such circuits are often referred to as Arithmetic Boolean Level (ABL) circuits [13]. Techniques exist that can convert a gate-level arithmetic circuit into such an ABL network, although a highly bit-optimized arithmetic circuits may contain a sizable number of logic gates that cannot be mapped onto (half) adders. Those gates will be modeled using arithmetic operators, such as half adders, and described as linear equations, as described in Section 3.1.

2.1 Basic Terminology

The arithmetic function computed by the circuit is expressed as a polynomial in terms of the primary inputs. We refer to such a polynomial as *input signature*, denoted by $Sig_{in}(N)$, for some circuit N. Such a polynomial is unique, as it uniquely describes an arithmetic function computed by the circuit; it can be linear or nonlinear. For example, the input signature of a 7-3 counter N_C, shown in Fig. 2 is simply $Sig_{in}(N_C) = x_1 + x_2 + x_3 + x_4 + x_5 + x_6 + x_7$. For a n-bit binary adder N_A with inputs $\{a_0, \cdots, a_{n-1}, b_0, \cdots, b_{n-1}\}$, the input signature is $Sig_{in}(N_A) = \sum_{i=0}^{n-1} 2^i a_i + \sum_{i=0}^{n-1} 2^i b_i$, etc. The integer coefficients, called *weights*, w_i), associated with the corresponding signals, are uniquely determined by the circuit structure and its specification. For the 7-3 counter the input weights are $w_i = 1$ for each signal x_i, while for an adder, $w(a_i) = w(b_i) = 2^i$ for inputs $a_i,\ b_i$ at bit position i.

Input signature for non-linear networks can be similarly obtained. For example, input signature of a 2-bit signed multiplier can be directly obtained from its high-level specification: $F = (-2a_1+a_0)(-2b_1+b_0) = 4a_1b_1-2a_0b_1-2a_1b_0+a_0b_0$. By substituting product terms by new variables, $x_3 = a_1b_1, x_2 = a_1b_0, x_1 = a_0b_1, x_0 = a_0b_0$, we obtain a linear input signature of the multiplier network in terms of these fresh variables: $Sig_{in}(M) = 4x_3 - 2x_2 - 2x_1 + x_0$. Again, the signal weights are uniquely defined by the specification.

The result computed by an arithmetic circuit can also be expressed as a polynomial in the *output* variables. This polynomial is always linear as it represents a unique binary encoding of an integer number computed by the circuit. We refer to such a polynomial as *output signature*. For example, the output signature of a 2-bit signed multiplier M with outputs Z_3, Z_2, Z_1, Z_0 is $Sig_{out}(M) = -8Z_3 + 4Z_2 + 2Z_1 + Z_0$. In general, output signature of any arithmetic circuit with n output bits S_i is represented as $Sig_{out}(N) = \sum_{i=0}^{n-1} 2^i \ S_i$. The output signal weights are also uniquely defined, in this case by the output bit position.

We also introduce the notion of a *cut* in the circuit, defined as a set of signals separating primary inputs from primary outputs. Each cut has its own algebraic signature, defined similarly to the input and output signatures. Specifically, a cut signature is a linear polynomial in the cut signals with coefficients specified by the integer signal weights. The computation of those weights is one of the basic steps of our verification procedure, to be described in detail in Section 3.4.

For nonlinear circuits, such as multipliers, the nonlinear part (contained between the primary inputs and the linear variables) is typically very shallow. This is the case not only for simple array multipliers mentioned above, but also for all signed and Booth-encoded multipliers and other circuits containing adder network structures (typical of all arithmetic circuits). Such a nonlinear block can be independently and easily verified using Boolean methods or word-level diagrams (BMD or TED). In this work we assume that the boundary between the linear and nonlinear blocks is known (as in the multiplier example above).

2.2 Overview of the Method

Since the input and output signatures describe the same circuit, albeit in different sets of variables, in a functionally correct circuit the two signatures must be equivalent, in the sense that they must evaluate to the same integer value for any integer input vector. The proof goal of functional verification is then to show that one signature can be transformed into the other using expressions of the internal operators: adders and logic gates. This can be done by symbolically rewriting the input signature, using the properly linearized internal logic and arithmetic operators, and checking if the polynomial obtained by such transformation matches the output signature. This check can be easily done using canonical word level diagrams, such as BMD or TED. The transformation can also be done in the opposite direction, from outputs to inputs, and the resulting expression compared to the input signature. If the input signature is not known, it can be computed directly from the output signature by such a backward transformation.

The presumed equivalence between the input and output signatures suggests that the functional verification problem in an arithmetic circuit can be viewed as a *Network Flow Problem*: the data is injected into input bits and flows through the network to be collected at the output bits. The network modules act like nodes in a transportation network, distributing data according to the edge capacities, here represented as signal weights. In the functionally correct circuit,

the total flow into the inputs, described by the input signature, must be equal to the flow at the output of the circuit, described by the output signature.

While conceptually the equivalence between the input and output signatures can be determined by symbolic rewriting, it is actually accomplished by *computing the signal weights*. The concept of rewriting is presented here only to prove the correctness of our method. In an actual implementation, the proof will be accomplished by i) computing weights of the intermediate polynomials involved in the transformation; ii) checking if such computed weights are compatible with the input/output signatures; and iii) if the weights satisfy additional equivalence relations required for functional correctness. The details of this procedure are provided in Sections 3.4 and 3.5.

3 Arithmetic Network Model

For the presented network model to work, we have to make sure that each network node (represented by circuit module) satisfies *Flow Conservation Law (FCL)*. As we will see in the next section, this is automatically guaranteed by basic arithmetic operators, such as adders. Logic gates and fanouts are modeled in a similar fashion, to make sure that each satisfies FCL.

3.1 Algebraic Models

This section describes algebraic models of the circuit modules used in our method. They include: half-adders (HA), full-adders (FA), inverters (INV), buffers (BUF), and basic logic gates (AND, XOR, OR). Each of them is modeled with a single linear equation which satisfies FCL.

• A half-adder (HA) with binary inputs a, b, and a full adder (FA) with binary inputs a, b, c_0 and outputs S (sum) and C (carry out) are represented by the following equations:

$$HA : a + b = 2C + S; \quad FA : a + b + c_0 = 2C + S \tag{1}$$

• Logic gates, AND and XOR, can be obtained directly from the HA using a linear HA model: the XOR(a, b) is derived from the sum output S, and the AND(a, b) from the carry-out output C of HA(a, b), as shown in Fig. 1(a). If only one gate (say an AND) is used/needed, the other output (in this case corresponding to an XOR) is left unconnected. We refer to such an unused signal as a *floating signal*. The role of the floating signals in our model is to pick up the "slack" in the flow, so that the used output always assumes the correct binary values and the module satisfies the FCL.

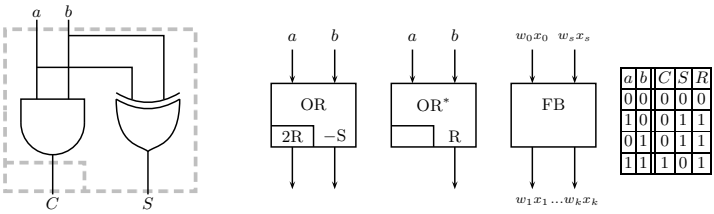
• The OR gate, $R =$OR(a, b), can be similarly derived from the HA using deMorgan's law, resulting in $\{a + b = 2C + S;\ C + S = R\}$. By combining the two equations we obtain a general OR model: $a + b = 2R - S$, see Fig. 1(b). Here, S represents an unused, floating signal. The set of equations for OR can often be simplified if $C = a \cdot b = 0$, i.e., when inputs a, b to the HA are never both 1. This happens often in arithmetic circuits whenever a, b come as reconvergent fanouts

from the C and S outputs of another HA, where they cannot be both 1. In this case the equation for the OR gate, denoted as OR*, simplifies to $a + b = R$, see Fig. 1(c). In summary, the OR gate is modeled as follows:

$$OR : a + b = 2R - S; \quad OR^* : a + b = R \tag{2}$$

• An inverter gate $y =$INV(x) is modeled by the equation: $x = 1 - y$. Similarly, a buffer with input x and output y can be modeled by the simple equation $x = y$.
• Special attention must be given to fanouts, which can be viewed as trivial modules. Such modules do not compute any arithmetic or logic function and simply replicate the signal as needed. In its original form a fanout node may not satisfy algebraic flow conservation law. For example, if signal x_1 fans out into two signals, x_2, x_3 then the equation $x_1 = x_2 + x_3$, with a constraint $x_1 = x_2 = x_3$, does not satisfy the algebraic flow conservation law. To fix this problem, we create a dummy fanout module, called $FBox$, with inputs x_0, x_s and outputs $x_1, \ldots\ x_k$ for a fanout with factor k, as shown in Fig. 1(d). Here x_s is a slack variable added to compensate for the difference between $x_1 + \ldots x_k$ and x_0. We refer to such a variable as *fanout slack*. The equation satisfying FCL for the FBox is: $w_0x_0 + w_sx_s = w_1x_1 + \ldots + w_kx_k$, where w_i is the weight associated with signal x_i.

Fig. 1 shows algebraic models for the basic modules, and the truth table to verify the logical correctness of the models. It is easy to verify that each such module satisfies the FCL.



a	b	C	S	R
0	0	0	0	0
1	0	0	1	1
0	1	0	1	1
1	1	1	0	1

(a) (b) (c) (d) (e)

Fig. 1. Modeling logic gates: (a) $C =$AND(a, b), $S =$XOR(a, b), derived from half-adder: $a + b = 2C + S$; (b) generic model for OR: $a + b = 2R - S$; (c) simplified XOR* model: $a + b = R$; (d) model of fanout box; (e) truth table for C, S, R.

3.2 Signature Rewriting

Before formalizing our verification model, we illustrate the verification approach with an example of a 7-3 counter, shown in Fig. 2. The circuit counts the number of 1s on the seven input bits and encodes the result in a 3-bit output word. Its structure is described by the following set of linear equations.

$$\begin{cases} FA_1 : x_1 + x_2 + x_3 = 2x_{11} + x_{12} \\ FA_2 : x_4 + x_5 + x_6 = 2x_{13} + x_{14} \\ FA_3 : x_{12} + x_{14} + x_7 = 2x_{15} + x_{10} \\ FA_4 : x_{11} + x_{13} + x_{15} = 2x_8 + x_9 \end{cases} \quad (3)$$

The input signature, $Sig_{in} = cut_0 = x_1+x_2+x_3+x_4+x_5+x_6+x_7$ is rewritten into an expression (cut signature) $cut_1 = (2x_{11} + x_{12}) + (2x_{13} + x_{14})$ using equations for FA$_1$: $(x_1+x_2+x_3 = 2x_{11}+x_{12})$ and FA$_2$: $(x_4+x_5+x_6 = 2x_{13}+x_{14})$. Similarly, expression for cut_1 is rewritten into cut_2 using equation for FA$_3$, and then into expression cut_3 using FA$_4$. The resulting expression $cut_3 = 4x_8 + 2x_9 + x_{10}$ matches exactly the output signature, $Sig_{out} = 4S_2 + 2S_1 + S_0$, which indicates that the circuit is correct (i.e., performs its intended function). Notice the weights

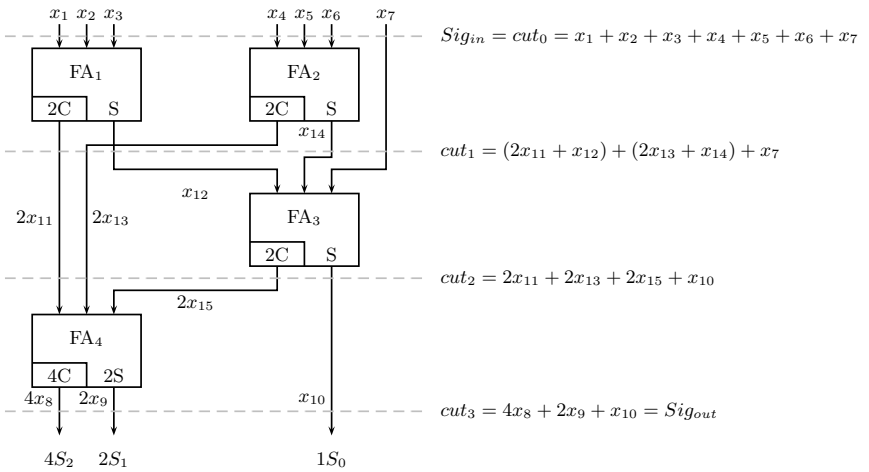


Fig. 2. Arithmetic network model of a 7-3 counter

associated with individual signals in this network. The weight of each input signal in this circuit is 1. The signature rewriting process gradually increases weights of some of the signals, eventually producing higher weights at the output bits. For example, one unit of x_1, x_2, x_3 each, when applied to FA$_1$, will produce one unit of x_{12} generated at output S of the adder, and two units of x_{11} (denoted in the figure as $2x_{11}$), generated at output C. This is a direct consequence of equation (1) of the adder. Then, signals x_{11}, x_{13}, x_{15}, each with weight 2, will produce outputs x_8 and x_9 with weights 4 and 2, respectively. This is simply the result of replacing the subexpression $2x_{11} + 2x_{13} + 2x_{15} = 2(x_{11} + x_{13} + x_{15})$ in cut_2 by $2(2x8 + x9)$, or, equivalently, of multiplying the equation (3) for FA$_4$ by constant 2.

In summary, the weights, which represent the amount of flow carried by the signals play an important role in computing the flow in the network. The next two

sections describe the process of computing the weights by propagating them from the primary outputs to primary inputs, without actually performing signature rewriting.

3.3 Weight Compatibility Constraints

As discussed in the preceding section, linear models of the arithmetic modules used in the network naturally impose constraints on signal weights. We refer to those rules as *Weight Compatibility* constraints. The weights which satisfy the compatibility condition are *unique*, and are determined solely by the output encoding and the network structure. These rules are simply a consequence of linear equations modeling the internal modules (adders, gates, inverters, and fanout boxes). Let w_x denote the weight of signal x. Then, the FA equation $a + b + c_0 = 2C + S$ imposes the following condition:

$$w_a = w_b = w_{c0} = w_S; \quad \text{and} \quad w_C = 2w_S$$

For the HA, the first constraint simply reduces to $w_a = w_b = w_S$. Note that for the AND and XOR gates, which use the FA/HA model, these rules will determine weights of the *floating signals*, i.e., the S signal for the AND gate and the C signal for the XOR gate.

Similar relation can be derived for the generic OR gate, modeled by $a + b = 2R - S$, namely:

$$w_a = w_b = -w_S, \quad \text{and} \quad w_R = -2w_S$$

The simplified OR* gate, governed by the equation $a + b = R$, has only one constraint, namely $w_a = w_b = w_R$.

The first constraint in each group simply means that the input weights must be the same. Changing any of the weights in a manner inconsistent with this constraint, would correspond to multiplying individual signals by different constants, which would invalidate the algebraic model (equations 1 and 2). The same is true for the buffer: one must not multiply each side of the equation by a different constant as this will change the relation between the two signals. On the other hand, multiplying the entire equation for any given module by a constant, will not change the relationship between the signals and will only increase the flow carried by those signals. This happens during the process of weight propagation, as shown in the 7-3 counter circuit.

Similarly, the compatibility constraints for an $FBox$ are derived directly from the $FBox$ equation: $w_0x_0 + w_sx_s = w_1x_1 + \ldots + w_kx_k$.
For a known set of signal weights $w_0, w_1, \ldots, w_k$ this will automatically determine weight of the *slack signal* x_s. The weight propagation procedure, described in the next section, guarantees that such weights can always be computed and have unique value.

In addition to the weight compatibility constraints, a *connectivity* rule needs to be imposed on the connections between the modules to correctly propagate the weights along the network wires. Such a rule is intuitively obvious: the weights of the signals on the two ends of a wire (buffer) must be equal. This, too, can

be justified by the mathematical model of the buffer, described by the equation $x_i = x_j$. This trivially imposes the constraint that $w_i = w_j$.

3.4 Weight Propagation

Computation of signal weights is an important first step in our verification procedure. The weights are computed by traversing the network from primary outputs (where they are determined by the binary encoding) to primary inputs, starting with the least significant bit, S_0. The assignment of weights must satisfy the compatibility conditions derived earlier. The weight assignment process is illustrated with an example of a parallel prefix adder, with input signature $Sig_{in} = 8(a_3 + b_3) + 4(a_2 + b_2) + 2(a_1 + b_1) + a_0 + b_0 + c_0$ and output signature $\{16C_{out} + 8S_3 + 4S_2 + 2S_1 + 1S_0\}$ imposed by the output encoding.

Fig. 3 shows the original gate-level design and Fig. 4(a) shows the network flow model of the circuit, obtained from gate level netlist using ABL extraction technique. In this design, each OR is represented by a simple OR* model ($R = a + b$), because it satisfies the simplifying conditions discussed earlier. The signals S_6, S_7, C_{10}, S_{11}, shown at the bottom of the circuit, are the *floating signals* coming from the output of HAs, which do not propagate any further. The signals d_9, d_8, d_7, d_{16}, shown at the top of the circuit, are the *fanout slack variables*, added as inputs to the FBoxes. In contrast to the input and output signatures, the weights of the floating and fanout slack signals are not known a priori and are computed during the weight propagation procedure.

The procedure starts with the least significant bit of the output, S_0. The weight 1 of signal S_0, connected to the S output of FA_0, matches the weight of that signal generated by FA_0. At the same time weight 2 is imposed on signal d_1 at the C output of that adder (to be denoted by $2d_1$). This assignment of weights at FA_0 is compatible with the weight (1) of its inputs. If the input weights were not known, this would also impose weights =1 on the inputs a_0, b_0, c_0. Propagation of $2S_1$ upwards similarly satisfies the weight compatibility at FA_1 (whose all inputs have weight 2) and imposes weight 4 on signal d_{16}. Propagation of $4S_2$ through HA_7 generates weights $(4d_7, 4d_{16})$ at the input to HA_7 and weight $8d_{18}$ at the C output of HA_7, see Fig. 4(a), etc. The procedure continues as long as the weights satisfy the weight compatibility conditions.

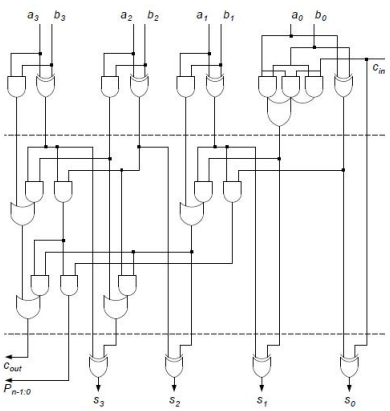


Fig. 3. Gate-level parallel prefix adder

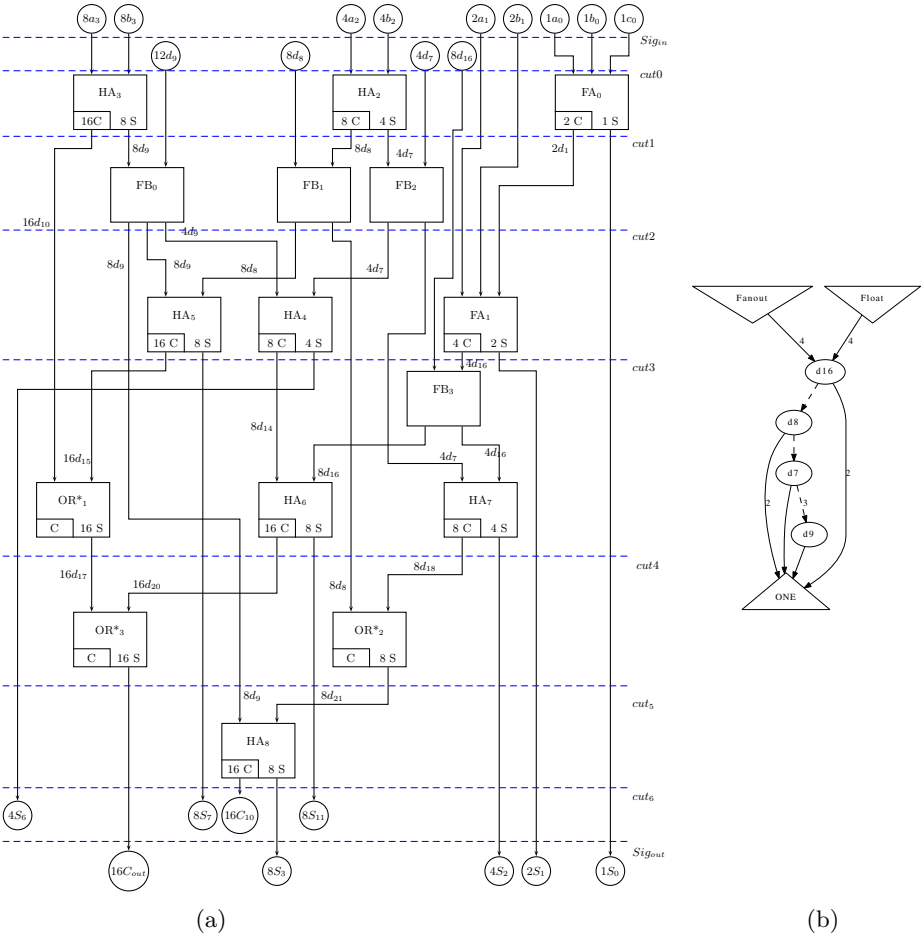


Fig. 4. (a) Network flow for parallel prefix adder; (b) TED showing equivalence between fanouts and floating signals

The floating and slack signals must be computed from the weights of already computed signals. Consider, for example, $Fbox_3$ associated with signal d_{16}. The weight of the right output signal, $4d_{16}$ has been already determined by back-propagating $4S_2$, but the other output from this Fbox will not be known until both inputs to HA$_6$ have been determined. This is made possible by the computation of weights originating at $16C_{out}$, resulting in the following sequence of weights:

$$16C_{out} \rightarrow 16d_{20} \rightarrow \{8d_{14}|8d_{16}\}$$

which fixes the left output of $Fbox_3$ to $8d_{16}$. The slack variable for this fanout box is then computed as the difference between the outgoing and incoming flow associated with this signal, i.e., $8d_{16} + 4d_{16} - 4d_{16} = 8d_{16}$. Other slack variables

at the input to Fboxes and floating signals at the outputs of the adders are resolved similarly, resulting in the weights shown in Fig. 4. In general, because the graph representing the arithmetic network is acyclic (DAG), there always exist an order which guarantees the resolution of the weights.

If at any point during the procedure the weights are incompatible, the circuit cannot produce weights which are compatible with the input weights, i.e., it does not compute the function specified by the input signature. An example in Section 3.6 illustrates this case. If the weights satisfy the compatibility conditions, the computation eventually reaches the primary inputs, where the input weights are compared with those in the input signature. If the weights at the primary inputs match those in the input signature, the circuit is considered functionally correct. Otherwise the circuit is faulty (either the network structure is wrong or the specification, given as input signature, is incorrect). Hence we have the following **necessary condition** for the circuit to implement the desired function:

For the circuit to compute the required function, the computed weights must satisfy the compatibility condition and must match the weights of the inputs.

This equivalence check can be done readily using a canonical word-level diagram, BMD or TED. The weight assignment for the parallel prefix adder example is shown in Fig. 4(a). The computed weights match those of the primary inputs, hence satisfying this necessary condition.

3.5 Proof by Flow Conservation

The final condition for functional correctness of the circuit is based on checking if its model satisfies the Flow Conservation Law (FCL). In the arithmetic network in which each module satisfies FCL, the flow into the input bits must be equal to the flow at the output bits. However, by construction of our model, the total input flow in addition to the flow into the primary inputs (expressed as input signature) also contains slack variables of the fanout boxes, denoted by Δ_{fn}. Similarly, the total output flow in addition to the flow out of the primary outputs (expressed as output signature) also contains floating signals associated with the unused variables, denoted Σ_{fl}. That is, in an arithmetic circuit which, by construction, satisfies flow conservation law, we have:

$$Sig_{in} + \Delta_{fn} = Sig_{out} + \Sigma_{fl} \tag{4}$$

where Δ_{fn} and Σ_{fl} are the weighted sums of the slack fanouts and the floating signals introduced in the network, respectively. In our example, $\Delta_{fn} = 8d_{16} + 12d_9 + 8d_8 + 4d_7$; and $\Sigma_{fl} = 16C_{10} + 8S_{11} + 8S_7 + 4S_6$. Intuitively, for the input signature to be equal to the output signature, the flow added by the fanouts must be compensated by the flow removed by the floating signals. As a result, if the input and output signatures match, the proof of functional correctness of the network reduces to proving that

$$\Delta_{fn} - \Sigma_{fl} = 0 \tag{5}$$

In summary: *The circuit is functionally correct if and only if: (i) there exists a compatible assignment of weights consistent with the input signature Sig_{in};*

and (ii) the amount of the flow introduced by fanouts Δ_{fn} is equal to the flow consumed by floating signals Σ_{fl}.

The first condition guarantees that the input signature can be rewritten into an output cut whose weights match those of the output signature, while the second condition satisfies the flow conservation law in this pseudo-Boolean network.

A naive way to solve this problem would be to express each of those terms as a function of primary inputs and prove that the resulting expression is zero. However, we only need to express Σ_{fl} in terms of the fanout variables. We then need to prove that $\Sigma_{fl} = \Delta_{fn}$ in terms of the fanout signals only. We perform this verification using TED. Figure 4(b) shows the TED for Δ_{fn} and Σ_{fl}, both expressed in terms of fanout variables only, clearly indicating that they are identical.

3.6 Debugging Faulty Circuits

The described method for functional verification can also help identify and localize bugs in a faulty circuit. Consider again the circuit in Fig. 2 but with wires x_{12} and x_{13} swapped. The question is whether this circuit will still work as a 7-3 counter; and if not, what causes the malfunction and how can the bug be identified. In this faulty configuration the equations for the affected adders, FA_3 and FA_4, are:

$$\begin{cases} FA_3 : x_{13} + x_{14} + x_7 = 2x_{15} + x_{10} \\ FA_4 : x_{11} + x_{12} + x_{15} = 2x_8 + x_9 \end{cases} \tag{6}$$

With this, the following cuts are generated during the rewriting process, starting with cut_0:

$$\begin{cases} cut_1 : (2x_{11} + x_{12}) + (2x_{13} + x_{14}) + x_7 \quad \text{(same as before)} \\ cut_2 : (2x_{11} + x_{12}) + x_{13} + (2x_{15} + x_{10}) \quad \text{(different)} \\ cut_3 : x_{13} - x_{12} + 4x_8 + 2x_9 + x_{10} \quad \text{(different)} \end{cases} \tag{7}$$

In this arrangement, the weight of x_{12} does not match the weights of other signals, $2x_{11}, 2x_{15}$, at the input to FA_4. Similarly, the weight of signal $2x_{13}$ does not match the weights of x_{14} and x_7, at the input to FA_3. This violates the weight compatibility discussed earlier. While the resulting expression of cut_3 contains the output signature $4x_8 + 2x_9 + x_{10}$, it also contains a "residual expression" $(x_{13} - x_{12})$. This indicates that the circuit computes a function that differs from the intended one by $(x_{13} - x_{12})$, hence it is incorrect. (It can be easily shown that $x_{13} - x_{12} \neq 0$). The identification of such a residual expression is useful in determining the source of the bug: it must be related to signals x_{13}, x_{12}.

4 Results

We tested our verification method on a number of signed multipliers up to 62×62 bits. First, a structural verilog code was generated for each multiplier using a generic multiplier generator software (courtesy of the University of Kaiserslautern). The verilog code was parsed to transform the multiplier circuit into a

network of HA, FA and basic logic gates from which a set of equations was generated in the required format. The structure of those designs made it possible to easily extract input signature required in our method. In general, however, transformation of an arbitrary gate-level circuit into an ABL network is a known difficult problem that can be computationally expensive; it may also result in different configurations since such a mapping is not unique. This, however, does not affect our approach; the different structures will only affect the effectiveness of the method but not the result. Each mapping will have its own, unique set of transformations and any of those will lead to the same conclusive answer regarding the circuit functionality.

The results of our experiments are shown in Fig. 5. The CPU time includes all phases of the process: preprocessing (which takes a negligible fraction of the entire process, taking only 3 sec for the 62-bit multiplier); computing signal weights; checking weight compatibility with input signature; creating symbolic equation for $\Delta_{fn} - \Sigma_{fl}$; generating script for TED; and using TED to check the equivalence condition (5). The experiments were run on a PC with an Intel i7 CPU @ 2.30GHz and 7.7 GB memory. Since most of the research in this

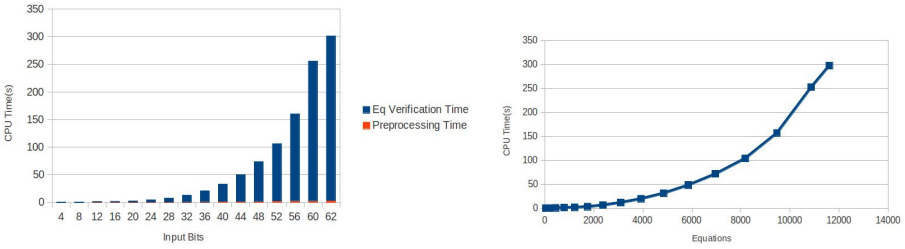


Fig. 5. CPU time for multipliers (a) in the number of bits (b) in number of equations

field has been done in the context of property checking rather than functional verification, we could not find suitable data for comparison. the CPU runtimes [1,2]. The runtime complexity of the procedure to compute algebraic signature of the network is quadratic in the number of equations (or, equivalently in the number of gates), c.f. Figure 5.

Comparison with SMT Solvers: In principle, the network can be described by a system of linear equations $Ax = b$ derived directly from the equations describing the network modules. The test for functional correctness can be obtained by checking if the network $Ax = b$ is compatible with the expected input and output signatures. This can be modeled as satisfiability (SAT) problem as follows. Let $Sig_{in}(N)$ and $Sig_{out}(N)$ be the primary input and output signature as defined in Section 3.1. Then, we need to show that: $(Ax = b) \wedge (Sig_{in}(N) \neq Sig_{out}(N)$ is unsatisfiable (unSAT). We performed this test on a number of multipliers using three SMT solvers that support Linear Integer Arithmetic: MathSAT, Yices, and Z3. The results, reported in Table 1 show that the SMT solvers were not able to solve the problem for multipliers larger than 8 bits,

while our method can verify the functional correctness of multipliers up to 62 bits in several minutes. Z3 ran out of memory (3 GB), while Yices was unable to complete the computation in 30 minutes. In some cases, MathSAT was "unable to perform computation" and is not reported here. We also attempted to solve the problem using BDDs, but (as expected) we were unable to build BDD for multipliers larger than 14 bits, due to the memory explosion.

Table 1. Comparison with SMT solvers (MO=memory out with 3 GB, TO=timeout after 1800 sec)

Design	Z3 (*sec*)	Yices (*sec*)	Our method (*sec*)
mult 3×3	0.23	0.02	0.21
mult 4×4	466.36	0.05	0.28
mult 8×8	MO	TO	0.57
mult 16×16	MO	TO	1.52
mult 24×24	MO	TO	3.63
mult 32×32	MO	TO	12.22
mult 40×40	MO	TO	31.57
mult 48×48	MO	TO	71.93
mult 56×56	MO	TO	157.24
mult 62×62	MO	TO	297.59
$PrefixAdder(4b)$	160.31	0.05	0.25

5 Conclusions

The goal of this paper was to present a novel idea of modeling the functional verification of arithmetic circuits without resorting to expensive Boolean or bit-blasting methods. As such, this approach has a potential application in formal verification and could be used in conjunction with existing methods for functional verification. Currently the method is applicable to designs with well defined input signature, expressed as a multivariate (possibly nonlinear) polynomial in the input variables. Typically such a signature is given as part of the specification; otherwise it can be extracted from the design by transforming the known output signature (binary encoding) backwards towards the inputs. In this sense, the method is directly applicable to extract circuit functionality from its hybrid arithmetic/gate-level structure.

An important application where this method can be particularly useful is the identification and localization of bugs in the design. This can be accomplished by analyzing areas containing incompatible weights, as illustrated in Section 3.6. Typically this will happen due to miss-wiring, crossing, or missing wires, which will result in incompatible weights. It seems that the module which violates the weight assignment and the bit position that imposes a violating assignment should provide important information about the bug location. We are not aware of any other approach that can so efficiently address this debugging issue.

The major limitation of this method is in generating ABL networks from an arbitrary gate-level arithmetic circuit, which in general is a difficult problem. Nevertheless, the method can be useful in verifying new arithmetic circuit architectures based on novel computer architecture algorithms, were the design is already specified in terms of adders and some connecting gates. The method can be readily extended to sequential circuits by converting them to bounded models, which is a part of the ongoing research effort. The extension to floating point arithmetic will need to be investigated.

Acknowledgment. This work has been supported by a grant from the National Science Foundation under award No. CCF-1319496.

References

1. Kaivola, R., Ghughal, R., Narasimhan, N., Telfer, A., Whittemore, J., Pandav, S., Slobodová, A., Taylor, C., Frolov, V., Reeber, E., Naik, A.: Replacing Testing with Formal Verification in Intel® Core™ i7 Processor Execution Engine Validation. In: Bouajjani, A., Maler, O. (eds.) CAV 2009. LNCS, vol. 5643, pp. 414–429. Springer, Heidelberg (2009)
2. Seger, C.-J.H., Jones, R.B., OLeary, J.W., Melham, T., Aagaard, M.D., Barrett, C., Syme, D.: An Industrially Effective Environment for Formal Hardware Verification. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 24(9), 1381–1405 (2005)
3. Slobodova, A.: A Flexible Formal Verification Framework. In: MEMCODE 2011 (2011)
4. Pradhan, D.K., Harris, I.G. (eds.): Practical Design Verification. Cambridge University Press (2009)
5. Soos, M.: Enhanced Gaussian Elimination in DPLL-based SAT Solvers. In: Pragmatics of SAT (2010)
6. Fallah, F., Devadas, S., Keutzer, K.: Functional Vector Generation for HDL Models using Linear Programming and 3-Satisfiability. In: Proc. Design Automation Conference, pp. 528–533 (1998)
7. Brinkmann, R., Drechsler, R.: RTL-Datapath Verification using Integer Linear Programming. In: Proc. ASPDAC, pp. 741–746 (2002)
8. Zeng, Z., Talupuru, K., Ciesielski, M.: Functional Test Generation based on Word-level SAT. J. Systems Architecture 5, 488–511 (2005)
9. Huang, C.-Y., Cheng, K.-T.: Using Word-level ATPG and Modular Arithmetic Constraint-Solving Techniques for Assertion Property Checking. IEEE Trans. on CAD 20(3), 381–391 (2001)
10. Biere, A., Heule, M., Maaren, H.V., Walsch, T.: Satisfiability Modulo Theories in Handbook of Satisfiability, ch. 12. IOS Press (2008)
11. Raudvere, T., Singh, A.K., Sander, I., Jantsch, A.: System Level Verification of Digital Signal Processing application based on the Polynomial Abstraction Technique. In: Proc. ICCAD, pp. 285–290 (2005)
12. Shekhar, N., Kalla, P., Enescu, F.: Equivalence Verification of Polynomial Data-Paths Using Ideal Membership Testing. IEEE Trans. on Computer-Aided Design 26, 1320–1330 (2007)

13. Wienand, O., Wedler, M., Stoffel, D., Kunz, W., Greuel, G.: An Algebraic Approach for Proving Data Correctness in Arithmetic Data Paths. In: Proc. ICCAD, pp. 473–486 (July 2008)
14. Vasudevan, S., Viswanath, V., Sumners, R.W., Abraham, J.A.: Automatic Verification of Arithmetic Circuits in RTL using Stepwise Refinement of Term Rewriting Systems. IEEE Trans. on Computers 56, 1401–1414 (2007)
15. Pavlenko, E., Wedler, M., Stoffel, D., Kunz, W.: STABLE: A new QF-BV SMT Solver for hard Verification Problems combining Boolean Reasoning with Computer Algebra. In: Proc. Design Automation and Test in Europe, pp. 155–160 (2011)
16. Basith, M.A., Ahmad, T., Rossi, A., Ciesielski, M.: Algebraic approach to arithmetic design verification. In: Formal Methods in CAD, pp. 67–71 (2011)