Function Extraction from Arithmetic Bit-level Circuits

Maciej Ciesielski, Walter Brown, Duo Liu
University of Massachusetts, Amherst
ciesiel@ecs.umass.edu, webrown@umass.edu, duo@engin.umass.edu

André Rossi
Université de Bretagne-Sud - Lab STICC, France
andre.rossi@univ-ubs.fr

***Abstract*—**

The paper describes a method to derive a polynomial function computed by an arithmetic bit-level circuit. The circuit is modeled as a bit-level network composed of adders and logic gates and computation performed by the circuit is viewed as a flow of binary data through the network. The problem is cast as a Network Flow problem and solved using standard algebraic techniques. Extraction of the arithmetic function from the circuit is accomplished by transforming the expression at the primary outputs into an expression at the primary inputs. Experimental results show application of the method to certain classes of large arithmetic circuits.

I. Introduction

With the ever-increasing size and complexity of integrated circuits and systems on chip, hardware verification becomes a dominating factor of the overall design flow. Despite a considerable progress in verification of digital circuits for random and control logic, advances in formal verification of arithmetic designs have been lagging. This can be contributed to the difficulty in modeling wide arithmetic datapaths without resorting to computationally expensive Boolean methods that require bit-blasting, i.e., flattening the design to a bit level netlist.

This paper addresses arithmetic verification problem using algebraic approach recently proposed in [1]. It models the circuit as a network of standard arithmetic components, and computation performed by the circuit is viewed as a flow of binary data through the network, represented as a pseudo-Boolean expression. Functional correctness of an arithmetic circuit is then solved by transforming the symbolic expressions representing the flow at the circuit inputs (the *input signature*) into a polynomial expression at the primary outputs (*output signature*), and checking if the resulting expression matches the binary encoding at the primary outputs. This paper goes one step further: it shows how to extract arithmetic function computed by the circuit by deriving a unique input signature from the known output signature (output encoding). Such derived input signature then determines the arithmetic function implemented by the circuit. If the functional specification of the circuit is known and differs from the one derived by this process by some ΔF (a polynomial), assignment of input variables for which ΔF is nonzero will provide a bug trace.

A. Related Work

To the best of our knowledge the problem of arithmetic circuit extraction, as defined here, has not been attempted before. The literature related to this subject addresses extraction of arithmetic bit-level *structure* from gate-level implementations, [2],[3], without being concerned with the arithmetic function it implements. Automated techniques for extracting arithmetic bit level (ABL) information from gate level netlists have been proposed in the context of property and equivalence checking [2] but they do not address the issue of the function implemented by the network. In their work 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 [3]. However, no practical method for deriving such large polynomials and no systematic comparison against the specification have been proposed. Other papers that follow the general ABL approach, address the issue of debugging or functional verification in some form or another, but they all require knowledge of the circuit functionality [4]. Other Computer Algebra methods have been introduced to model arithmetic components as polynomials [5], [6] but they are mostly concerned with word-level view of computation and as such are not directly applicable to verifying arithmetic bit-level networks.

Traditional approach to check an arithmetic circuit against its functional specification is based on canonical, graph-based representations such as BDDs, BMDs, Taylor Expansion Diagrams (TED) [7], and others [8]. Application of BDDs to verification of arithmetic circuits is limited by a 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 [9] [10]. 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, including CryptoMiniSAT [11], which specifically targets XOR-rich circuits, but they are all based on a computationally expensive DPLL decision process. Several techniques combine linear arithmetic constraints with Boolean SAT in a unified algebraic domain [12] or use ILP to model the modulo semantics of the arithmetic operators [13] [14]. 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 [15].

SMT solvers integrate different theories (Boolean logic, linear integer arithmetic, etc.) into a DPLL-style SAT decision procedure [16]. However, in their current format, the SMT tools are not efficient at solving decision problems that appear in arithmetic circuits. Another approach to arithmetic verification, particularly popular in industry, is based on Theorem Provers, mostly for microprocessor verification [17] [9] [10]. These are large, 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 best known theorem proving systems are: HOL, PVS, and Boyer-Moore/ACL2. In general, these systems are highly interactive, requiring extensive user guidance and expertise for efficient use. 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 [18], are incomplete, as they rely on simple rewriting rules and use non-canonical representations.

An original approach to functional arithmetic verification has been proposed in [19], where an arithmetic bit-level circuit is described by a system of linear equations. The resulting set of linear equations is then reduced to a single algebraic expression (the "signature") using Gaussian-like elimination and linear algebra techniques to reason about functional correctness of the design. The difficulty faced by this method is the case when not all internal signals can be eliminated from the signature, resulting in a "residual expression". 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. In [1] the same problem is solved by modeling the computation of an arithmetic function as a flow of binary data, and representing it as algebraic expression. The functional correctness is proved by showing equivalence between the input and output signatures. In this approach the issue of possible residual expression translates into a simpler problem that can be solved using algebraic methods.

B. Novelty and Contribution

In this work we extend the algebraic approach proposed in [1] by modeling the problem as a network flow problem. However, we solve a different problem, namely computing (extracting) a unique arithmetic function implemented by the circuit. The extracted function, in form of a unique polynomial in $Z/2^n$, can be compared to the known circuit functionality (if available) and used to provide a bug trace or counterexample, indicating for which set of primary inputs the output differs from the expected one. In contrast to theorem provers and traditional term rewriting techniques, which only address functional equivalence and property checking, but not extraction of functionality, 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 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.

II. Arithmetic Network Model

The circuit is represented as a network composed of two parts: a non-linear block, typically used for partial product generation and input recoding; and a linear network, which operates on such preconditioned signals using a summation tree. The result is encoded in an n-bit output. All arithmetic circuits have this kind of structure with the nonlinear part being relatively shallow. The linear part of the circuit is represented as a network of standard arithmetic components (adders and Boolean connectors), referred to as Arithmetic Boolean Level (ABL) circuits. Several techniques are available to convert a gate-level arithmetic circuit into such an ABL network [2], although a highly bit-optimized arithmetic circuits may contain a sizable number of logic gates that cannot be mapped onto (half) adders. In principle, those gates will be modeled using arithmetic operators, such as half adders, and described as linear equations, as described in Section II-B. Figure 2 shows a typical structure of such circuits.

A. Preliminaries

Arithmetic function computed by the circuit is expressed as a polynomial in terms of the primary inputs (PI). We refer to such a polynomial as *input signature*, denoted Sig_{PI}. Such a polynomial uniquely describes an arithmetic function computed by the circuit; it can be linear or nonlinear. For example, the input signature of an n-bit binary adder with inputs $\{a_0, \cdots, a_{n-1}, b_0, \cdots, b_{n-1}\}$, is $Sig_{PI} = \sum_{i=0}^{n-1} 2^i a_i + \sum_{i=0}^{n-1} 2^i b_i$, etc. 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: $Sig_{PI} = 4x_3 - 2x_2 - 2x_1 + x_0$. The integer coefficients, called *weights*, w_i, associated with the corresponding signals, are uniquely determined by the intended circuit function (specification). For example, in an adder, $w(a_i) = w(b_i) = 2^i$ for inputs $a_i,\ b_i$ at bit position i.

Similarly, the result computed by an arithmetic circuit can be expressed as polynomial in the *primary output* (PO) variables. We refer to such a polynomial as *output signature*, Sig_{PO}. In contrast to the input signature, Sig_{PO} is always linear as it represents a unique binary encoding of an integer number computed by the circuit. For example, the output signature of a 2-bit signed multiplier with outputs S_3, S_2, S_1, S_0 is $Sig_{PO} = -8S_3 + 4S_2 + 2S_1 + S_0$. In general, output signature of any arithmetic circuit with n output bits S_i is represented as $Sig_{PO} = \sum_{i=0}^{n-1} 2^i\ S_i$. The PO weights are also unique, defined by the known output encoding. Figure 2 shows an example of a 3-bit signed multiplier, with a nonlinear product generator block and a linear ABL structure. More complex circuits, such as Booth multipliers, have similar structure but are still relatively shallow compared to the summation network.

In this work we assume that the boundary between the linear and nonlinear blocks is known (as in the multiplier

example above). However, it is possible to detect such a boundary automatically using the signature rewriting scheme adopted by the network-flow approach of [1]. The signals at the boundary between the linear and nonlinear block will be referred to as *intermediate inputs*, MI, and the corresponding signature as *intermediate signature*, Sig_{MI}. For example, in a 2-bit singed multiplier, mentioned earlier, $Sig_{MI} = 4x_3 - 2x_2 - 2x_1 + x_0$. Here, $x_3 = a_1b_1, x_2 = a_1b_0, x_1 = a_0b_1$, and $x_0 = a_0b_0$, are outputs of the nonlinear block with primary inputs a_0, a_1, b_0, b_1.

B. Algebraic Model

Each arithmetic or logic operator in the linear network is modeled with a set of linear equations. The 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:

$$HA : a + b = 2C + S; \quad FA : a + b + c_0 = 2C + S \quad (1)$$

Logic gates can be similarly derived from a half adder. An XOR(a, b) is obtained as the sum output S of HA(a, b), and the AND(a, b) as the carry-out output C of HA(a, b), as shown in Figure 1(a). If only one gate (say an AND) is needed, the other output (an XOR) is left unconnected. Such an unused signal is called *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 required by the flow. An OR gate R is modeled as: $OR :\ a+b = 2R-S$, where S represents an unused, floating signal, see Fig. 1(b). This model can often be simplified to $OR^* :\ a + b = R$ if $C = a \cdot b = 0$, see Fig. 1(c). 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. An inverter gate y =INV(x) is modeled by the equation: $x = 1 - y$.

Special attention must be given to fanouts; they are modeled as dummy modules, called $FBox$, that do not compute any arithmetic or logic function and simply replicate the signal as needed. Signal x_0 fanning out to $x_1, \ldots\ x_k$ is represented by an $FBox$ with inputs $x_0,\ x_s$ and outputs $x_1, \ldots\ x_k$, as shown in Fig. 1(d). Here x_s is a *fanout slack* variable added to compensate for the difference between $x_1 + \ldots x_k$ and x_0. Fanout slack variables play the same role as floating signals, except that they appear at the input to the fanout box.

By construction, each module of the linear network, described by a linear equation, satisfies the *Flow Conservation Law* (FCL). It simply states that weighted sum of the input bits to the module is equal to the weighted sum at its output. As such, the FCL also applies to the entire network [1].

III. Computing Input Signature

The method for computing input signature is based on the observation that in a functionally correct circuit the input and output signatures must be equivalent, i.e., they must evaluate to the same integer value for any integer input vector. For the linear portion of the network, the problem is modeled as a *Network Flow Problem*: the data is injected into input bits and flows through the network to be collected at the output bits. Such a network can be viewed as a transportation network, distributing data according to the edge capacities,

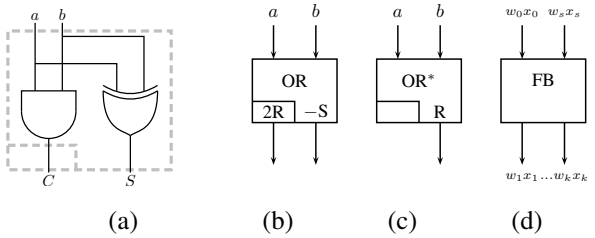


Fig. 1. Modeling logic gates: (a) C =AND(a, b), S =XOR(a, b), derived from half-adder: $a+b = 2C+S$; (b) generic model for OR: $a+b = 2R-S$; (c) simplified XOR* model: $a + b = R$; (d) model of fanout box.

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, encoded by the output signature. As described in [1], checking if a given circuit performs the given arithmetic function reduces to checking equivalence between the two signatures.

In this work, input signature Sig_{PI} is not known, and the transformation is performed from the known output encoding at the POs towards the PIs. Such a linear transformation can only be done in the linear network up to the intermediate inputs, MI,, at the boundary between the linear and nonlinear block. Transformation from the intermediate inputs to the primary inputs (PI) is then accomplished by a computing algebraic expressions for each signal in MI, in terms of the PIs. Each Boolean function in the logic cone of $x_i \in MI$ is recursively replaced by its equivalent algebraic expression: $a' = 1-a$, $a \wedge b = a \cdot b$, $a \vee b = a+b-a \cdot b$, $a \oplus b = a+b-2a \cdot b$, up to the PI set. The resulting algebraic expressions are then multiplied by the corresponding weight of the intermediate signature and added to create a final, non-linear input signature. In the case of the 2-bit signed multiplier, the intermediate linear signature, $Sig_{MI} = 4x_3 - 2x_2 - 2x_1 + x_0$ is transformed into a nonlinear input signature, $Sig_{PI} = (-2a_1 + a_0)(-2b_1 + b_0)$ at their primary inputs, by substituting expressions for each x_i in terms of the PIs: a_1, a_0, b_1, b_0. The resulting input signature then serves as the description of the arithmetic function implemented by the circuit. Figure 2, to be discussed later, illustrates this concept for a 3-bit signed multiplier.

A. Weight Computation

Input signature is computed by computing weights (integer coefficients, w_i,) associated with all the network signals, including the floating signals and the fanout slacks. The weight computation procedure uses a modified version of weight propagation proposed in [1], adapted to the fact that input signature is not known, and all the constraints on the weights must be resolved using the output signature only. Weights must satisfying the following *compatibility rules*, determined by the algebraic models of the network components: all inputs to an adder $HA : a+b = 2C+S$ must have the same integer weight, k, equal to the weight of the S output, while the output C of the adder must have weight $2k$. The weight of the output of an $OR^* : a + b = R$ must be the same as its inputs, and the weights of the outputs of an $OR : a+b = 2R-S$ must be $2k$ and -$1k$, respectively. The weight of the inverter with output weight k must have the input weight -k, etc.

Every time a weight is assigned to one of the outputs of the network module, the weights of its input(s) and of the second output (if applicable) are assigned values determined by the respective compatibility rule. Computation of fanout weights, represented by Fbox, requires special treatment, since the fanout slack variable can take any value needed to satisfy the flow conservation law (FCL). To solve this problem, additional computation is invoked that relies on the fundamental FCL theory, described in Section III-B. Basically, such weights need to be computed using flow conservation law, expressed by Equation 3, but applied locally to the cone of influence of the signal under consideration. Specifically, computation of fanout weight can only be done after: i) all floating signals that have the fanout in their cone have their weights computed; and ii) all components that take the fanout as input have their weights resolved. The complexity of this process, which relies on symbolic substitution using TED, increases in the worst case exponentially, but only over a small part of the network defined by the cone of influence.

The weight assignment process is illustrated with an example of a 3-bit signed multiplier in Fig. 2. In the following, we will use notation kx_i to indicate that signal x_i is assigned weight k, as shown in the figure. Buffers, present in the figure, are the artifact of converting the name of a signal into the one needed at the input or output, and do not affect weight propagation.

The assignment starts from the known weights of POs, determined by the output encoding. The weights are processed from the least significant bit to the most significant bit, and are propagated upwards towards the MI inputs. In this case, $1x_{44}$ is connected through a buffer to signal x_{45}, determining its weight, $1x_{45}$. Next, signal $2x_{46}$ is propagated through a buffer to $2x_{11}$ and up to HA_0, imposing the assignment of weights $2x_9$, $2x_{10}$ and $4x_{12}$. Propagation of of $4x_{47}$ similarly assigns weights $4x_{21}$, $4x_3$ (further propagated to $-4x_4$) and to $4x_{16}$ and $8x_{20}$, etc. Propagation of $8x_{48}$ through the inverter INV7 assigns negative weights to all signals of HA_9, including the portion of $-8x_7$ which also fans out to HA_8, but with weight $+8x_7$ (sic). This apparent discrepancy is handled by computing weight of the fanout slack variable, $-8x_7S$, etc. The correctness of fanout assignments will be verified using the FCL rule, as explained in the next section.

The intermediate signature in terms of the MI signals, computed by this weight propagation procedure is:

$$16x_{35} - 8x_6 - 4x_2 + 4x_{13} - 4x_4 + 2x_{10} + 2x_9 - 8x_8 + x_{45} \quad (2)$$

B. Flow Conservation

It has been shown in [1] that a linear, arithmetic bit-level circuit satisfies the Flow Conservation Law expressed by the following condition:

$$Sig_{MI} + \Delta_{fn} = Sig_{PO} + \Sigma_{fl} \quad (3)$$

where Δ_{fn} and Σ_{fl} are the weighted sums of the slack fanouts and the floating signals, respectively, introduced in the network. This condition basically states that any additional flow into the network at the MI inputs, caused by fanouts, must be compensated by the flow of the floating signals, which do not reach outputs PO. The term Sig_{MI} represents signature of intermediate signals, MI, which are inputs to the linear portion of the network, and Sig_{PO} represents the signature at the primary outputs, PO. The above condition also implies that for Sig_{PO} to be equivalent to Sig_{MI}, one must satisfy the relation

$$\Delta_{fn} - \Sigma_{fl} = 0 \quad (4)$$

Only then Sig_{MI} represents the sought after linear input

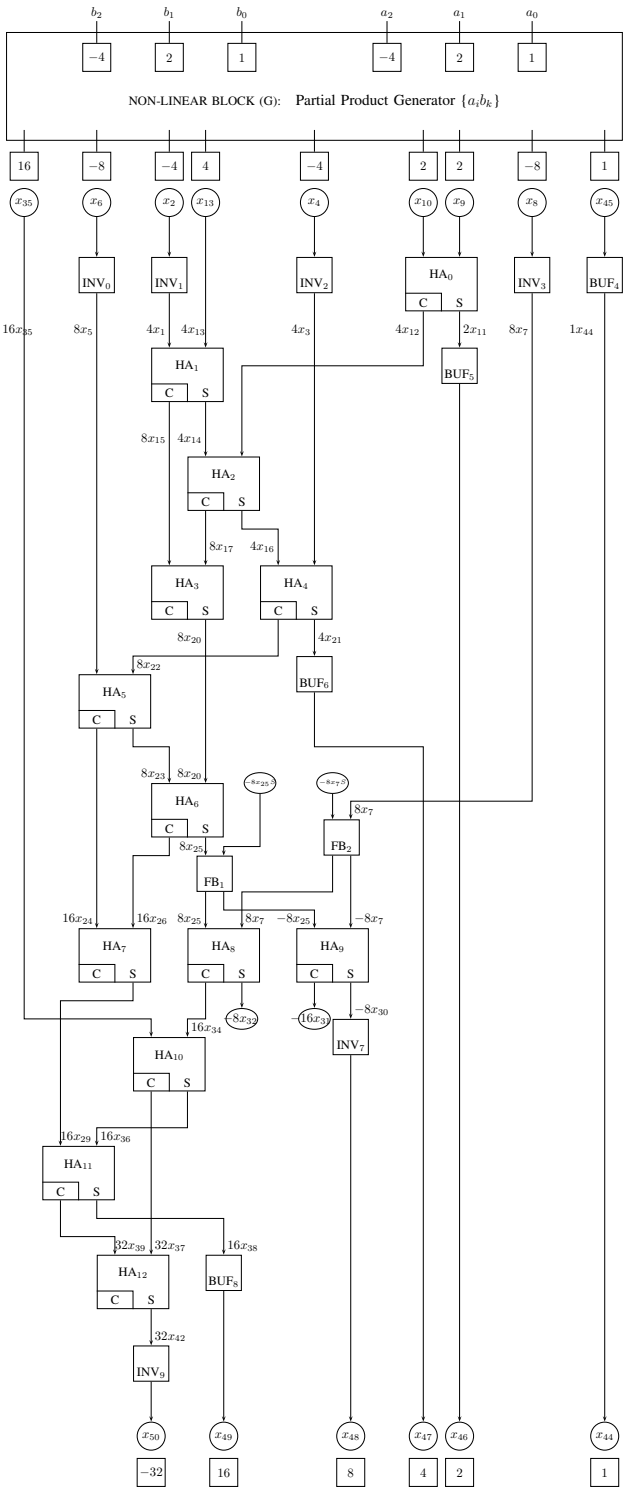
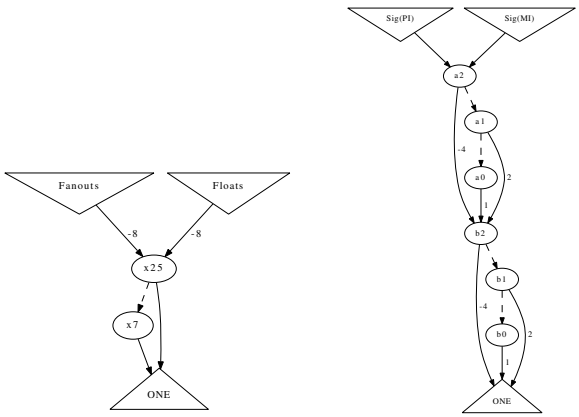


Fig. 2. Arithmetic bit-level network of a 3-bit signed multiplier.



(a) (b)

Fig. 3. (a) TED showing equivalence between the floating signals and fanouts; (b) Nonlinear signature of a 3-bit multiplier

signature. A naive way to prove this equality would be to express Δ_{fn} and Σ_{fl}, each, as a function of primary inputs and prove that the resulting expression is zero. However, instead of expressing Δ_{fn} and Σ_{fl} as a function of primary inputs it is sufficient to express Σ_{fl} in terms of the fanout variables only; and then prove that $\Sigma_{fl} = \Delta_{fn}$ in terms of the fanout signals. This verification is simpler and can be done easily using TED.

In our 3-bit signed multiplier design, $\Delta_{fn} = -8x_7 - 8x_{25}$ and $\Sigma_{fl} = 8x_{32} - 16x_{31}$ are indeed equivalent, indicating that the computed intermediate input signature, Sig_{MI} can be trusted. This equivalence is demonstrated by the TED in Figure 3(a). Both terms, called $Fanouts$ for Δ_{fn} and $Floats$ for Σ_{fl}, expressed in terms of fanout variables, point to the same canonical graph, clearly indicating that they are equivalent.

C. Computing Nonlinear Signature

Once the intermediate input signature Sig_{MI} of the linear portion of the network has been computed, we are faced with a task of transforming this signature into an input signature Sig_{PI} at the primary inputs of the entire design. Recall that Sig_{MI} is a linear pseudo Boolean function of the intermediate signals $x_i \in MI$, while Sig_{PI} is (in general) a nonlinear polynomial function expressed in the *primary inputs* of the design. The computed Sig_{PI} signature then describes the complete functionality of the circuit. As mentioned earlier, this task is achieved by transforming the computed signature Sig_{MI} into the signature at the primary inputs, PI, by recursively substituting variables in MI using respective algebraic formulas for logic gates. In general, this process can be computationally expensive, but the nonlinear blocks are usually shallow and such a computation is very fast. This operation is done readily using TED representation of polynomials [7].

We illustrate this process for the signed 3-bit multiplier in Figure 2, whose intermediate input signature Sig_{MI} has been already computed as (c.f. Eq.2).:

$$16x_{35} - 8x_6 - 4x_2 + 4x_{13} - 4x_4 + 2x_{10} + 2x_9 - 8x_8 + x_{45}$$

In this case, the nonlinear block contains only one level of logic, composed of AND gates, but in general such a block will have a more complex logic structure. TED representation will be able to represent such computed input signature in a canonical, *normal factored form* for arbitrarily complex logic structure. The result is shown in Figure 3(b). As we can see, the resulting function, encoded in factored form by the TED, is

$$Sig_{PI} = (-4a_2 + 2a_1 + a_0)(-4b_2 + 2b_1 + b_0)$$

One can determine that this is a 3-bit signed multiplier, provided that the condition $\Delta_{fn} - \Sigma_{fl} = 0$ is satisfied. This condition is easily verified using TED, as explained earlier, and is part of the entire process. In this case, $\Delta_{fn} = -8x_7 - 8x_{25}$ and $\Sigma_{fl} = 8x_{32} - 16x_{31}$ are equivalent, indicating that the computed primary input signature, Sig_{PI}, can be trusted.

IV. Results

The functional extraction technique described in this paper has been implemented in $Perl$ and integrated with the TDS (TED-based decomposition) system [7] and tested on a set of signed multipliers up to 56×56 bits. The current limitation is dictated by the word size of the TDS system, and not by the weight propagation and the actual verification algorithm and can be improved by manipulating the exponents of the weights instead of the integers (on-going work).

First, a structural Verilog code was generated for each multiplier using a multiplier generator software [20]. 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 process is fully automated, and the CPU time includes all phases of the process: computing signal weights; resolving fanout weights using TED; checking FCL condition $\Delta_{fn} - \Sigma_{fl}$; generating intermediate and final scripts for TED; and using TED to compute the final input signature in factored form. In all cases, the verification (which is an integral part of the system) confirmed that the input and output signatures were equivalent, so that the generated input signatures can be trusted. Using the *variable reorder* function of the TED, which produced the factored form of the signatures, we checked that the signatures indeed corresponded to the respective multipliers.

The results of our experiments are shown in Fig. 4. The experiments were run on an a PC with an Intel Core i7 processor @ 2.30 GHz and 7 GB memory. The complexity of the algorithm is almost quadratic in the number of logic gates. Specifically, asymptotic CPU runtime (computed using power test on the experimental data) is $O(n^{1.5})$ and memory usage is $O(n^2)$, where n is the number of logic gates. Since most of the research in this area has been done in the context of property checking rather than a complete functional verification, we could not find suitable data for comparison. Some new results are available in Galois field arithmetic [21] that enjoy certain properties that make the verification significantly easier than for integer arithmetic in $Z/2^n$.

V. Conclusions

The paper presents a novel idea of verifying an arithmetic circuits by extracting the arithmetic function that it implements. It can be particularly useful in reverse engineering of existing designs, to determine functions of arithmetic circuits with incomplete or lost documentation, and with research in

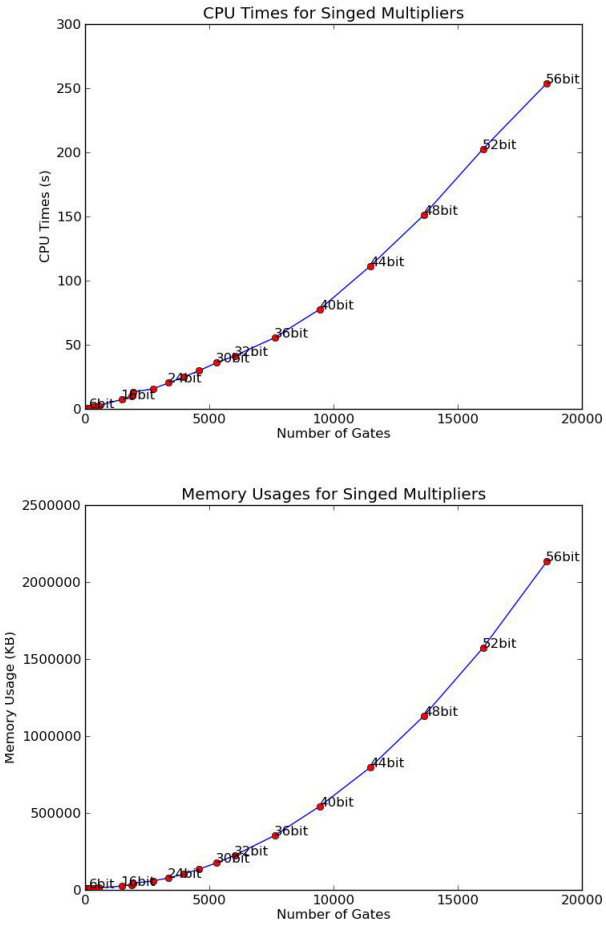


Fig. 4. Extracting arithmetic function: CPU time and memory usage for signed multipliers.

computer security. The method performs these tasks without resorting to expensive Boolean or bit-blasting methods. We are not aware of any other approach that addresses such defined functional extraction problem in arithmetic circuits. Another important application is identification and localization of bugs in the design. This can be accomplished by analyzing areas containing incompatible weights. Such errors may happen due to miss-wiring, crossing, or missing wires, which will result in violating the weight compatibility condition. It seems that the module which violates the weight assignment and the bit position that imposes a violating assignment may provide important information about the bug location.

An obvious limitation of this method is the need to generate an ABL network from an arbitrary gate-level arithmetic circuit, which is in general difficult. Also, different mappings onto ABL structure may result in a different set of floating and fanout signals, which may affect a complexity of proving relation (4). Nevertheless, we believe the method can find applications in verifying new arithmetic circuit architectures based on novel computer architecture algorithms, were the design is already specified in terms of adder trees and relatively few connecting gates. Future work will be devoted to extending this idea to sequential circuits and floating point arithmetic.

ACKNOWLEDGMENT:
This work has been supported by a grant from the National Science Foundation under award CCF-1319496.

REFERENCES

[1] M. Ciesielski, W. Brownn, and A. Rossi, "Arithmetic Bit-level Verification using Network Flow Model", *Haifa Verification Conference*, HVC'13, Nov. 2013, pp.327-343, Springer, LNCS 8244.

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

[3] 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," *Proc. Design Automation and Test in Europe*, 2011, pp. 155-160.

[4] O. Sarbishei, M. Tabandeh, B. Alizadeh, and M. Fyjita, "A Formal Approach to for Debugging Arithmetic Circuits" *IEEE Trans. on CAD* (TCAD), pp. 742-754, May 2009.

[5] T. Raudvere, A. K. Singh, I. Sander, and A. Jantsch, "System Level Verification of Digital Signal Processing application based on the Polynomial Abstraction Technique," *Proc. ICCAD*, 2005, pp. 285–290.

[6] N. Shekhar, P. Kalla, and F. Enescu, "Equivalence Verification of Polynomial Data-Paths Using Ideal Membership Testing," in *IEEE Trans. on Computer-Aided Design*, July 2007, vol.26, pp. 1320–1330.

[7] M. Ciesielski, D. Gomez-Prado, Q. Ren, J. Guillot, and E. Boutillon, "Optimization of Dataflow Computation using Canonical TED Representation," *IEEE Transactions on Computer Aided Design of Integrated Circuits and Systems*, vol. 28, no. 9, pp. 1321–1333, Sept. 2009.

[8] D.K. Pradhan, and I.G. Harris, ed., "Practical Design Verification", Cambridge University Press, 2009.

[9] 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 Core i7 Processor Execution Engine Validation" CAV 2009, LNCS 5643, pp. 414âĂŞ429, 2009.

[10] C-J.H. Seger, R.B. Jones, J.W. OâĂŹLeary, T. Melham, M.D. Aagaard, C. Barrett, and D. Syme, "An Industrially Effective Environment for Formal Hardware Verification", IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 24, no. 9 (September 2005), pp. 1381âĂŞ1405.

[11] M. Soos, "Enhanced Gaussian Elimination in DPLL-based SAT Solvers", *Pragmatics of SAT*, 2010.

[12] F. Fallah, S. Devadas, and K. Keutzer, "Functional Vector Generation for HDL Models using Linear Programming and 3-Satisfiability," *Proc. Design Automation Conference*, 1998, pp. 528–533.

[13] R. Brinkmann and R. Drechsler, "RTL-Datapath Verification using Integer Linear Programming," *Proc. ASPDAC*, 2002, pp. 741–746.

[14] Z. Zeng, K. Talupuru, and M. Ciesielski, "Functional Test Generation based on Word-level SAT," *J. Systems Architecture*, Elsevier. Aug. 2005, vol. 5, pp. 488–511.

[15] C.-Y. Huang and K.-T. Cheng, "Using Word-level ATPG and Modular Arithmetic Constraint-Solving Techniques for Assertion Property Checking," *IEEE Trans. on CAD*, vol. 20, no. 3, pp. 381–391, March 2001.

[16] A. Biere, M. Heule, H. V. Maaren, and T. Walsch, *Satisfiability Modulo Theories in Handbook of Satisfiability*, IOS Press, 2008, Chapter 12.

[17] A. Slobodova, "A Flexible Formal Verification Framework", MEMCODE 2011.

[18] 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," in *IEEE Trans. on Computers*, 2007, vol. 56, pp. 1401–1414.

[19] M. A. Basith, T. Ahmad, A. Rossi, and M. Ciesielski, "Algebraic Approach to Arithmetic Design Verification," *Formal Methods in CAD*, 2011, pp. 67–71.

[20] M. Wedler, Univ. Kaiserslautern, $MultGen$ software, 2010.

[21] J. Lv, P. Kalla, and F. Enescu, "Efficient Grobner Basis Reductions for Formal Verification of Galois Field Arithmatic Circuits," *IEEE Trans. on CAD*, vol. 32, no. 9, pp. 1409–1420, September 2013.