Structural Reverse Engineering of Arithmetic Circuits

Alan Mishchenko Baruch Sterin Robert Brayton
Department of EECS, UC Berkeley
{alanmi, brayton}@berkeley.edu, baruchs@gmail.com

Abstract

This paper focuses on detecting arithmetic components in large gate-level circuits using cut enumeration and structural analysis. It is based on the assumption that adder trees, the key part of arithmetic components, are represented using half- and full-adders, whose boundaries can be traced down to individual nodes present in the gate-level circuit structure. The proposed method detects the elementary adders, then adder trees composed of these adders, and finally, arithmetic components. The detection leads to annotating nodes of the original gate-level circuit with exact boundaries of each component. The proposed method is useful in applications, such as design analysis, post-synthesis optimization, and formal verification.

1. Introduction

Gate-level implementations of hardware designs often contain arithmetic components, such as adders, subtractors, multipliers, dividers, etc. These components are present in the gate-level netlist as combinations of gates without explicit boundaries between them and the surrounding glue logic. *Reverse engineering* (RE) detects these boundaries whenever possible, and recognizes the functionality of arithmetic components contained within the boundaries.

The proposed method to detect such boundaries rests on the assumption that inputs, outputs, and internal cut-points between half- and full-adders constituting an arithmetic component (for example, a multiplier) are present in the gate-level netlist. The proposed structural RE relies only on the *existence* of the cut-points, without assuming anything about their locations, ranks, or polarities, which are discovered automatically by the algorithm.

The *strength* of the proposed method is in its applicability to a variety of arithmetic components, including unsigned and signed, as well as truncated ones (when some of the output bits are missing). For example, the method can handle Booth and non-Booth multipliers. It is fast (takes less than a second for circuits with 100K gates) and is scalable (does not incur major slow-downs when applied to circuits with millions of gates). It works well for most of the gate-level netlists derived from industrial designs.

The *weakness* of the proposed method is that it assumes that the adder trees are composed of elementary adders and that the inputs/outputs of these can be traced down to specific nodes in the gate-level design representation. This assumption has been shown to hold for a large number of public and industrial designs containing arithmetic components. However, it is not hold when an adder tree contains generalized adders [8] instead of elementary adders, or when heavy logic synthesis is applied to the gate-level representation and some or all cut-points are synthesized away. On the positive side, it was found experimentally that some types of technology-independent synthesis, for example, AIG rewriting [2], do not remove the cut-points between elementary adders. The exploration of the robustness of the proposed method in the presence of logic synthesis is beyond the scope of the present work.

The proposed *structural* approach to RE differs from the previous methods for *functional* approach to RE, such as [11][12]. It also differs from methods based on constructing algebraic polynomials, such as [3][10]. A related technique for detecting adder chains during synthesis and mapping for FPGAs has been used in [9]. Another similar approach based on different heuristics has been presented in [16].

RE of arithmetic circuits has numerous applications, including design analysis, post-synthesis optimization, and formal verification. A practical approach to RE developed in this paper enables progress in these fields. One motivating application deferred to future work, is combinational equivalence checking (CEC), a notoriously hard problem for designs with arithmetic components.

The rest of the paper is organized as follows. Section 2 contains relevant background. Section 3 outlines the RE framework. Section 4 describes RE for elementary adders. Section 5 describes RE for adder trees. Section 6 describes classifications of arithmetic components based on recognized adder trees. Section 7 contains experimental results and Section 8 concludes the paper.

2. Background

2.1 Boolean function

In this paper, *function* refers to a completely specified Boolean function $f(X)$: $B^n \rightarrow B$, $B = \{0,1\}$. The *support* of function f is the set of variables X, which can influence the output value of f. The *support size* is denoted by $|X|$.

The *truth table* of an n-input Boolean function is a bit-string containing 2^n bits representing values of the function under all possible 2^n input assignments ordered by their integer representation. For example, the truth table of a 3-input majority gate, denoted MAJ3 in the paper, contains 8 bits, 11101000, as shown in the last column of Figure 1.

2.2 Boolean network

A *Boolean network* (or *circuit*) is a directed acyclic graph (DAG) with nodes corresponding to Boolean functions and edges corresponding to wires connecting the nodes.

A node *n* may have zero or more *fanins*, i.e. nodes driving *n*, and zero or more *fanouts*, i.e. nodes driven by *n*. The primary inputs (PIs) are nodes without fanins. The primary outputs (POs) are a subset of nodes of the network, connecting it to the environment. A *transitive fanin* (*fanout*) *cone* (TFI/TFO) of a node is a subset of nodes of the network, which are reachable through the fanin (fanout) edges of the node. The *TFO support* of a node is the set of POs reachable through the fanouts of the node.

2.3 And-Inverter Graph

And-Invertor Graph (AIG) is a combinational Boolean network composed of two-input AND gates and inverters. An AIG for a Boolean network can be derived by factoring the functions of the logic nodes found in the network. The AND/OR gates in the factored forms are converted into two-input ANDs and inverters using DeMorgan's rule and added to the AIG in a topological order.

A *cut C* of a node *n* is a set of nodes of the network, called *leaves* of the cut, such that each path from a PI to *n* passes through at least one leaf. Node *n* is called the *root* of cut *C*. The cut *size* is the number of its leaves. A *trivial cut* of a node is the cut composed of the node itself. A cut is *K-feasible* if the number of nodes in the cut does not exceed *K*. Only 3-feasible cuts are considered in this paper.

A *cut-point* is an AIG node representing the root of a cut or one of its leaves. A *polarity* of a cut-point, is the complemented attribute associated with the AIG node. The polarity is *positive* if the node represents a given Boolean function. The polarity is *negative* if the node represents the complement of the Boolean function.

2.4 Elementary adders

A *binary digit* is a zero or a one. An integer number *n* can be encoded as an ordered sequence of binary digits, d_i, such that $n = \Sigma_i(d_i*2^i)$. The value *i* is called the *rank* of digit d_i. The digit d_0 is the *least significant bit* (LSB). The digit d_k where $k = \log_2 n$ is the *most significant bit* (MSB). The MSB is typically written on the left. For example, $n = 6$ in binary encoding is represented as $110 = 1*2^2 + 1*2^1 + 0*2^0$.

Elementary adders add two or three binary digits of equal rank. An elementary adder can be a *full-adder* (FA) or a *half-adder* (HA). A FA adds three input digits of rank *i* and produces two output digits of rank *i* and *i*+1 called *sum* and *carry*, respectively. The truth table of the FA is shown in Figure 1. A HA is a FA with one input set to constant 0. A HA has two inputs and two outputs.

a	b	c	sum	carry
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Figure 1: The truth table of a full-adder.

The smallest AIG realizing a HA requires only 3 AIG nodes (two-input AND-gates) while a FA can be realized by two HAs and one two-input OR-gate, which together require only 7 AIG nodes. Many other AIG realizations of FAs and HAs are found in gate-level representations automatically generated by various tools.

The proposed approach to RE assumes that inputs and outputs of a component are present as cut-points in the original AIG. The cut-points are represented by AIG nodes, possibly complemented, such that the logic function of a component can be derived by traversing the AIG from the input cut-points to the output cut-points. In particular, a HA (or FA) in the AIG is specified by providing two (or three) input cut-points and two output cut-points.

2.5 Adder trees

Adder trees are Boolean networks containing elementary adders and no other gates. An example of an adder tree is a *ripple-carry adder* (RCA) taking two binary-encoded integers and producing the binary-encoded sum of these numbers, as shown in Figure 2. Assuming that the carry-in is 0 and the input integers have *n* bits each, a RCA takes *n* pairs of binary inputs whose ranks are 0 through *n*-1, and produces *n*+1 outputs whose ranks are 0 through *n*.

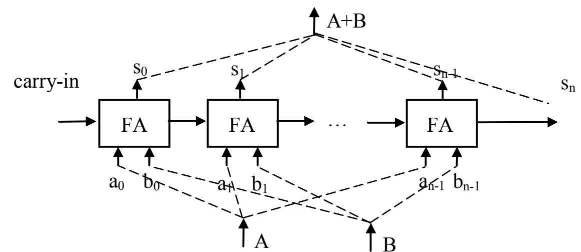

Figure 2: The structure of a ripple-carry adder.

A complete adder tree can have any number of inputs/outputs of each rank. The following invariant holds. For each input combination, the sum total of all input binary digits multiplied by power-2 of their ranks, is equal to the same expression for the output binary digits: $\Sigma_i(d_i*2^{r_i}) = \Sigma_o(d_o*2^{r_o})$, where indexes *i* and *o* iterate through inputs/outputs of an adder tree, respectively.

Figure 3 shows an adder tree of a 3-bit 3-argument multi-input adder, composed of 7 FAs arranged into 2 RCAs. The first RCA takes two 3-bit arguments and produces a 4-bit sum. The second RCA takes the remaining 3-bit argument and the 4-bit sum and produce the 5-bit result. Note that the input/output ranks are the same as the variable indexes.

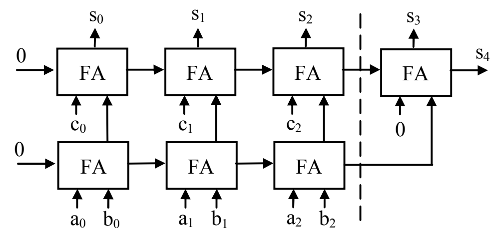

Figure 3: An adder tree of a 3-bit 3-argument adder.

The sum/carry outputs of a FA have Boolean functions of three-input XOR3/MAJ3 gates, respectively. Both XOR3 and MAJ3 satisfy the following property: complementing all of their inputs is equivalent to complementing their

output. As a result, simultaneous complementation of all inputs and outputs of an adder tree composed of FAs only, does not change the functions of the outputs of the tree computed in terms of their inputs. If HAs are present, they are converted into FAs with constant-0 inputs. If the constant inputs are present, these are complemented along with other inputs. As a result, the above property holds for an arbitrary adder tree composed of both FAs and HAs.

Another property is that constant propagation applied to an adder tree results in an adder tree. This is proved by observing that a FA becomes a HA, possibly with complemented inputs, if one of its inputs is a constant. Similarly, if two FA inputs are equal (represented by the same variable), a FA becomes a HA, and no additional gates are created. Thus, an adder tree has a "resilient" structure, which does not change when constants and equivalences propagate through it. An adder tree also does not change under some types of logic synthesis, such as [2].

This paper concentrates on the RE of adder trees because they have "resiliency" under some transforms, as shown above, and because they are a key component of many arithmetic circuits. For example, a typical multiplier has three components [8]: partial product generator, multi-input adder, and carry-propagate adder, as shown in Figure 4. The latter two combined together form an adder tree.

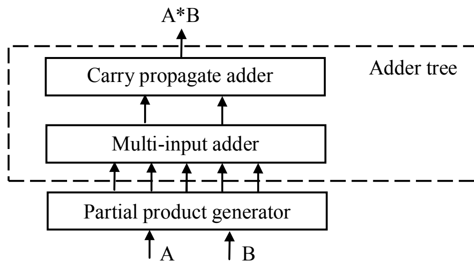

Figure 4: The structure of a typical multiplier.

3. Reverse-engineering framework

This section describes the proposed reverse-engineering framework. The *input* is an AIG derived from a gate-level netlist containing arithmetic components. The *output* is a reverse-engineered AIG, that is, an AIG annotated with boundaries of arithmetic components.

The *annotation* includes: (a) the type of a component, (b) a topologically ordered array of elementary adders creating the adder tree, (c) cut-points representing the component inputs/outputs and intermediate nodes between the elementary adders creating the adder tree, (d) polarity of these cut-points, (e) ranks of these cut-points.

Given this annotation, it is possible to extract the subset of AIG nodes representing a component as a separate AIG. It is also possible to insert an AIG representing a component into the original AIG, by stitching at the input/output boundaries. If the inserted AIG is functionally equivalent to the extracted AIG, the result is functionally equivalent to the original AIG. The extraction/insertion of a component can be used to normalize the structure of arithmetic components found in the original AIG, which is required in applications such as formal verification.

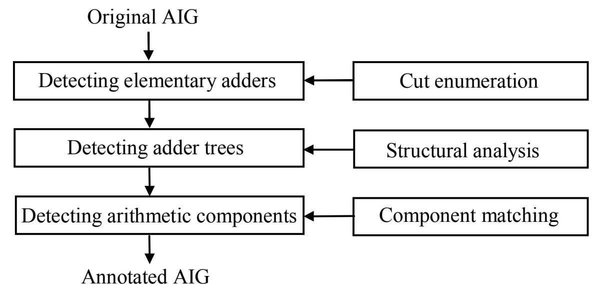

Figure 5. The proposed reverse-engineering framework.

Figure 5 outlines the RE framework. In particular, it shows that the computation involves three steps: detecting elementary adders using cut enumeration (Section 4), combining elementary adders into adder trees by dedicated structural analysis (Section 5), and finally, recognizing the type and number of arithmetic components by analyzing the type of adder trees (Section 6). For example, to detect a Booth multiplier, we need to detect the partial product generator implementing Booth re-coding and combine it with the adder tree, detected as shown in Section 4 and 5.

4. Recognizing elementary adders

This section presents a method to detect elementary adders in the original AIG.

The location of a FA in the AIG is given by five cut-points representing three inputs and two outputs of the FA. In the case of a HA, the third input is a constant 0. Since handling of a HA during RE is the same as handling of a FA with a constant-0 input, only FAs are discussed below.

Consider the output cut-points of a FA as Boolean functions in terms of the input cut-points. The function of the sum output is XOR3 ($sum = a \oplus b \oplus c$) while that of the carry output is MAJ3 ($carry = ab + ac + bc$).

Recognizing a FA requires that the Boolean functions of the FA outputs computed in terms of the inputs exactly match those of XOR3 and MAJ3. However, even if the inputs/outputs of a FA are correctly traced down to specific AIG nodes, the complemented attributes of the nodes may be missing after AIG construction. For example, a complemented attribute of the inputs of an XOR gate can be moved to the output when XOR gate normalization rules implemented in the AIG package are applied.

This is why, in this section, we relax the requirement of the FA outputs to match the functions of XOR3 and MAJ3 exactly. Instead, we require that the input/output cut-points are matched up to the complement. For example, instead of being an exact XOR3, the function of the sum output can be either XOR3 or its complement. Similarly, instead of being exactly MAJ3, the function of the carry output can be one of the 8 functions forming the NPN class of MAJ3 [4].

The proposed method to recognize FAs in the AIG involves the following computations:

- Computing all 3-input cuts of all AIG nodes.
- Computing truth tables of all cuts.
- Hashing the cuts by their ordered set of inputs.
- Finding pairs of 3-input cuts with identical inputs, belonging to different nodes, such that the Boolean

functions of the two cuts in terms of the shared inputs have NPN classes of XOR3 and MAJ3, respectively.

To compute the cuts, the 3-input cut enumeration is performed in a topological order as described in [7]. The truth tables of the cuts are obtained as a by-product of the cut enumeration. Thus, when two fanin cuts are merged during the cut computation and the resulting cut is 3-feasible, the truth tables of fanin cuts are permuted to match the fanin order of the resulting cut. These truth tables are then ANDed or XORed, depending on the node type, to get the resulting truth table. For the case of 3-input cuts, a dedicated pre-computation reduces the runtime of truth able computation to a small fraction of that of cut enumeration.

The next step in recognizing elementary adders, is hashing cuts by the ordered set of their leaves. As a result, two cuts having the same leaves but belonging to different nodes are found in the same bin. The bins of the hash table are examined and all cut pairs whose functions match NPN classes of XOR3 and MAJ3 are collected and stored. Each stored data set includes three inputs (shared cut leaves) and two outputs (different cut roots). Although individual cut leaves and cut roots can participate in more than one cut pair, each cut pair is characterized by a unique set of five cut-points, representing the location of one FA in the AIG.

5. Recognizing adder trees

The next step in the RE algorithm, is detecting adder trees. It is helpful to recall that an adder tree is a combinational Boolean network composed of elementary adders. Since each FA has the sum output (XOR3) and the carry output (MAJ3), an adder tree can be seen

- as a network of XOR3 gates, and
- as a network of MAJ3 gates.

The XOR3 and MAJ3 gates of these two networks may belong to several *independent subnetworks*, such that the gates in each subnetwork have fanin/fanout connections among themselves and do not have similar connection with gates from other subnetworks. To detect the independent subnetworks, we consider *top gates* defined as all XOR3 (MAJ3) gates whose outputs are not inputs of other XOR3 (MAJ3) gates. The independent subnetworks can be found by traversing the AIG from the top gates to their fanins.

The independent subnetworks of XOR3 and MAJ3 gates have different roles in adder trees. Indeed, the sum output of a FA has the same rank at the FA inputs, while the carry output has a rank that is larger by one, compared to the rank of the FA inputs. As a result, all fanins/fanouts of XOR3 gates in one XOR3 subnetwork have the same rank. Even if the XOR3 subnetwork has more than one top gate, all the inputs, outputs, and intermediate nodes belonging to this XOR3 subnetwork have the same rank.

On the other hand, independent MAJ3 subnetworks are composed of signals whose ranks increase by one each time an output of a MAJ3 gate is traversed. Figures 6 and 7 show XOR3 and MAJ3 subnetworks for the 3-bit 3-argument multi-input adder shown in Figure 3.

To recognize an adder tree, as indicated in Section 3, the following information should be computed:

- a topologically ordered array of elementary adders,
- cut-points representing the component inputs/outputs and intermediate nodes between the elementary adders creating the adder tree,
- polarity of these cut-points,
- ranks of these cut-points.

To derive this information, we traverse the independent MAJ3 subnetworks belonging to the adder tree starting from the top gates. The traversal involves computing polarities and ranks of intermediate cut-points. The inputs of the adder tree are determined as the nodes where this traversal terminates. The outputs are the top gates of the XOR3 subnetworks having common internal cut-points with the MAJ3 subnetworks visited during the traversal.

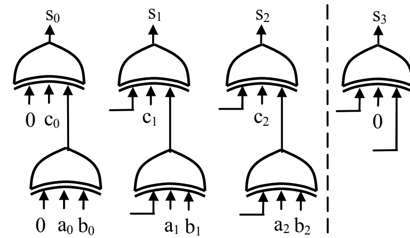

Figure 6. XOR3 subnetworks for adder tree in Figure 3.

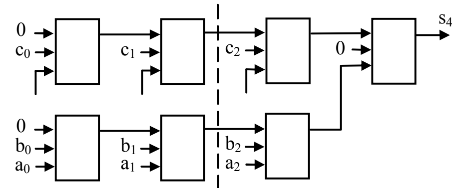

Figure 7. MAJ3 subnetworks for adder tree in Figure 3.

```
AdderTreeNormalize_rec( node N, int Rank, bool Polar ) {
    set rank of N to be Rank;
    set polarity of N to be Polar;
    G = MAJ3 gate driving N;
    if ( G == NULL ) { // there is no MAJ3 driver
        label N as input of the adder tree;
        return;
    }
    for each fanin A of G {
        if ( A is to be complemented for the cut to match MAJ3 )
            PolarFanin = NOT(Polar);
        else
            PolarFanin = Polar;
        AdderTreeNormalize_rec( A, Rank-1, PolarFanin );
    }
}
AdderTreeNormalize( array TopMajGates ) {
    int Rank = max length of a MAJ3 chain in the adder tree;
    for output N of each gate in the array TopMajGates {
        AdderTreeNormalize_rec( N, Rank, 1 );
    }
}
```

Figure 8: Pseudo-code of adder tree normalization.

Procedure **AdderTreeNormalize_rec**() in Figure 8 is called for each top MAJ3 gate. It determines the rank and polarity of each cut-point in the adder by propagating the complemented attribute from the output to the inputs of each MAJ3 gate in the tree. (This is possible since complementing the MAJ3 gate output is equivalent to

complementing all of its inputs.) The pseudo-code in Figure 8 is largely self-explanatory, except for the lines deciding when the complement is propagated to the fanins. For this, the procedure checks if, during cut computation, the input should be complemented to match the Boolean function of the cut with that of the MAJ3 gate. The two complemented attributes (the one coming from the upper level and the one determined by cut computation) are combined before the procedure recurs on the fanin.

To compute the rank of each internal cut-point, we start with a given rank at a root gate of the MAJ3 subnetwork and subtract one from it each time we recur on the inputs of the MAJ3 gate. At the end of the computation, we have assigned ranks and polarities to all of the internal cut-points as well as inputs of the adder tree.

It should be noted that if HAs are present in the adder tree, they are handled by converting them into FAs with a constant-0 input and handing these FAs as a regular FA in the above procedure. As a result, some constant-0 inputs may become constant-1 inputs after inverter propagation. This is expected and reflects the nature of some adder trees, which have constant-1 inputs.

A remarkable aspect of the proposed procedure, is that it works for any arithmetic component containing adder trees composed of elementary adders. In particular, an arithmetic component can be signed or unsigned; in both cases, the adder tree detection and handling is the same.

The proposed method also works for truncated adder trees and adder trees containing fast adders, as discussed below.

5.1 Handling truncated adder trees

An adder tree is *truncated*, if some of the outputs of the tree are missing in the AIG representation because they do not have fanouts in the design, and ended up being removed by a structural sweep.

Truncated adder trees are handled by the proposed method, except that, instead of a complete MAJ3 network, we consider several independent MAJ3 subnetworks.

A common situation is practice is when one or more MSBs of an adder tree are truncated. In this case, the XOR3 subnetworks corresponding to the truncated bits are removed. Among the remaining XOR3 subnetworks, the one with the largest rank has no corresponding MAJ3 gates because the outputs of these gates have rank exceeding the largest one, and therefore, these gates are also removed.

Consider the adder tree in Figure 3. This adder tree is not truncated, and therefore the MAJ3 network in Figure 6 is composed of only one subnetwork. Now consider the case when the two MSBs, s_4 and s_5, of the adder tree in Figure 3 are truncated. As a result, the top-right FA producing these MSBs is removed along with the MAJ3 gates feeding into it. Now, instead of one completely connected MAJ3 subnetwork, we have two subnetworks composed of two MAJ3 gates each. The vertical dotted lines in Figures 3, 6, and 7 show the truncation of the adder tree (Figure 3) and the XOR3 and MAJ3 subnetworks (Figures 6 and 7). The resulting truncated adder tree is shown in Figure 9.

It should be noted that the remaining sum MSB, s_2, is fed by the XOR3 subnetwork whose corresponding MAJ3 gates are removed after truncation. The proposed method detects the XOR3 and MAJ3 subnetworks and returns the truncated adder tree shown in Figure 9.

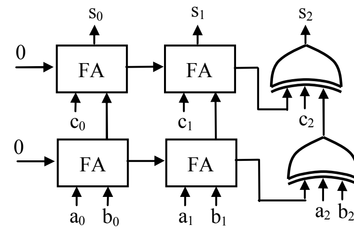

Figure 9: Truncated adder tree in Figure 3.

5.2 Handling fast adders

Adder trees in real designs often include *fast adders*, that is, adders whose carry-propagate chain is designed to reduce the delay of the resulting circuit. For example, a *carry look-ahead adder* is derived from an RCA by collapsing MAJ3 gates feeding into XOR3 gates [8].

The resulting fast-adder is not detected by the proposed structural method because only the XOR3 cuts, but not the MAJ3 cuts, can be identified in the original AIG. The lack of MAJ3 cuts does not allow the FAs to be detected by the cut-based method described in Section 4.

One way to handle fast adders is to infer the missing MAJ3 gates from the collapsed glue logic, and update the AIG by replacing the glue logic by the actual MAJ3 gates. This would make the FA detection work correctly. However, we found that the inference of MAJ3 gates is not needed in practice because fast adders are used only in the upper part of the adder tree. For example, in the structure of the typical multiplier shown in Figure 4, the fast-adder is used only in the last stage of addition. As a result, if the top-most fast adder is not represented as a set of elementary adders, the detected adder tree will be smaller. It will lack the chain of FAs on top, and it will have *pairs* of equal-rank outputs, instead of having *one* output for each rank.

The resulting adder tree is still useful in applications such as equivalence checking, which do not require adder trees in two copies of the design to match exactly. Any difference between the adders, including the missing part in one of them, can be detected and normalized away.

6. Component classification

Once adder trees in the gate-level netlist are recognized, completely or partially, the task it to identify arithmetic components containing these adder trees. For example, a multiplier is expected to have combinational logic of a partial-product generator (PPG) in the TFI of the adder tree. Depending on the multiplier design, the PPG can be a set of two-input gates (the array multiplier) or a set of dedicated 3- and 5-input functions (the radix-4 Booth multiplier).

Input rank	Input count
0	1
1	2
2	3
3	2
4	1

Figure 10. Distribution of input ranks in the adder tree of a 3-bit array multiplier.

The type of an arithmetic component is recognized by considering how many inputs/outputs of each rank are present in the corresponding adder tree. For example, a 3-bit array multiplier has an adder tree with $3^2 = 9$ inputs whose ranks are distributed as shown in Figure 10.

When a similar pattern of input ranks is observed in a tree, a multiplier is expected. Once a PPG is successfully indentified, a complete multiplier component is detected.

7. Experimental results

The methods described in this paper are prototyped in ABC [1] as command *&profile –a*. The implementation is being tested on several sets of benchmarks containing arithmetic components, including [13][14][15]. The results may be available in the final version of the paper.

8. Conclusions

The paper describes an efficient structural method to detect arithmetic components in a gate-level circuit represented as an AIG. The underlying assumption is that the key component of arithmetic logic, an adder tree, is composed of elementary half-adders and full-adders whose inputs and outputs can be traced down to specific circuit nodes, possibly complemented. This assumption holds for a large subset of practical test-cases, for which the proposed method identifies elementary adders, adder trees, and finally, arithmetic components, using structural analysis and 3-input function manipulation.

Future work will proceed in the following directions:

- applying the proposed method in combinational equivalence checking,
- making the method recover from the failure to detect a small number of adder cut-points,
- investigating the impact of logic synthesis on the success of the reserve-engineering procedure,
- extending the method to dividers and square-rooters.

9. Acknowledgements

This work was supported in part by SRC contract 2710.001 “SAT-based methods for scalable synthesis and verification” and NSF/NSA grant “Enhanced equivalence checking in cryptoanalytic applications”.

Special thanks are due to Yu-Liang (David) Wu, Xing (Sean) Wei, and Yi (Robin) Diao from Easy-Logic Technology Ltd for their hospitality and stimulating discussions during the first author’s visit in Hong Kong where much of this paper was written.

10. References

[1] Berkeley Logic Synthesis and Verification Group. *ABC: A System for Sequential Synthesis and Verification*. http://www-cad.eecs.berkeley.edu/~alanmi/abc

[2] P. Bjesse and A. Boralv, "DAG-aware circuit compression for formal verification", *Proc. ICCAD ’04*, pp. 42-49.

[3] M. Ciesielski, C. Yu, D. Liu, W. Brown, and A. Rossi, “Verification of gate-level arithmetic circuits by function extraction”, *Proc. DAC’15*.

[4] Z. Huang, L. Wang, Y. Nasikovskiy, and A. Mishchenko, "Fast Boolean matching based on NPN classification", *Proc. ICFPT'13*.

[5] A. Kuehlmann, V. Paruthi, F. Krohm, and M. K. Ganai, “Robust Boolean reasoning for equivalence checking and functional property verification”, *IEEE Trans. CAD*, Vol. 21(12), December 2002, pp. 1377-1394.

[6] A. Mishchenko, S. Chatterjee, R. Brayton, and N. Een, "Improvements to combinational equivalence checking", *Proc. ICCAD '06*, pp. 836-843.

[7] P. Pan and C.-C. Lin, “A new retiming-based technology mapping algorithm for LUT-based FPGAs,” *Proc. FPGA ’98*, pp. 35-42.

[8] B. Parhami, *Computer Arithmetic: Algorithms and Hardware Designs*, 2nd ed., Oxford University Press, New York, 2010.

[9] A. Petkovska, G. Zgheib, D. Novo, M. Owaida, A. Mishchenko, and P. Ienne, "Improved carry-chain mapping for the VTR flow", *Proc. FPT'15*, pp. 80-87.

[10] N. Shekhar, P. Kalla, and F. Enescu. “Equivalence verification of polynomial datapaths using ideal membership testing”. *IEEE TCAD’07*, vol. 26 (7), pp. 1320-1330.

[11] M. Soeken, B. Sterin, R. Drechsler, and R. Brayton, “Simulation graphs for reverse engineering”, *Proc. FMCAD’15*.

[12] P. Subramanyan, N. Tsiskaridze, W. Li, A. Gascón, W. Y. Tan, A. Tiwari, N. Shankar, S. A. Seshia, and S. Malik, “Reverse engineering digital circuits using structural and functional analyses,” *IEEE Trans. Emerg. Topics Computing*, 2014, vol. 2(1), pp. 63–80.

[13] J. Wang, 2012 ICCAD CAD competition. Problem 1. http://cad-contest.cs.nctu.edu.tw/CAD-contest-at-ICCAD2012/problems/p1/p1.html

[14] J. Wang, 2013 ICCAD CAD competition. Problem A. http://cad-contest.cs.nctu.edu.tw/CAD-contest-at-ICCAD2013/problem_a/

[15] C.-J. Hsu, 2015 ICCAD CAD competition. Problem B. http://cad-contest.el.cycu.edu.tw/problem_B/default.htm

[16] X. Wei, Y. Diao, T.-K. Lam, and Y.-L. Wu, “A universal macro block mapping scheme for arithmetic circuits”. *Proc. DATE’15*, pp. 1629-1634.