2009 19th IEEE International Symposium on Computer Arithmetic

Challenges in Automatic Optimization of Arithmetic Circuits

Ajay K. Verma
AjayKumar.Verma@epfl.ch

Philip Brisk
Philip.Brisk@epfl.ch

Paolo Ienne
Paolo.Ienne@epfl.ch

Ecole Polytechnique Fédérale de Lausanne (EPFL)
School of Computer and Communication Sciences
CH-1015 Lausanne, Switzerland

Abstract

Despite the impressive progress of logic synthesis in the past decade, finding the best architecture for a given circuit still remains an open and largely unsolved problem, especially for arithmetic circuits. In many cases, the outcome of even the most advanced synthesis techniques is highly dependent on the input description of the circuit, and the optimizations themselves barely modify the architecture of the circuit itself. Once the input description is converted to an appropriate architecture, logic synthesis performs local optimizations quite effectively; however, finding the best architecture up front is a nontrivial problem. This paper reviews recent results in arithmetic logic synthesis that the authors have published in recent years. Progress has clearly been made, but much further work is still needed to narrow the gap between the effectiveness of logic synthesis techniques for arithmetic and control-oriented circuits.

1. Introduction

The performance of an arithmetic circuit highly depends on its architectural description. For example there are many different types of adders, e.g., the *Ripple Carry Adder*, *Carry Look-ahead Adder*, and others. Some architectures are optimized for critical path delay, while others are optimized for area. Since different architectures effectively implement the same logic function, one can use one of many identities, e.g., $abc = (a \oplus b \oplus c)(ab \oplus bc \oplus ac)$, to convert one description into another. However, current logic synthesis techniques are unable to apply many identities due to the large set of relations and the limitations of algebraic factoring in discovering the appropriate ones automatically. When a circuit is implemented in one particular architecture, the synthesis tools only explore a small design space close to the input description, and rarely, if at all, cross the architectural boundaries.

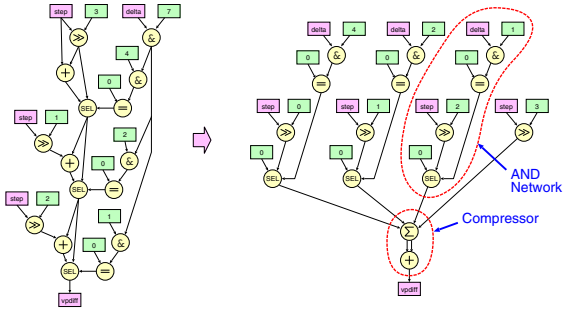

Figure 1. An example where swapping the arithmetic operations with logic operations creates the opportunity for a compressor tree.

This leaves it to the designer to figure out appropriate transformations to the underlying circuit in order to get the best performance. This is the case, even for commonly used circuits such as adders and multipliers. For example, consider Fig. 1, which shows two implementations of a kernel used by the adpcm-decoder. On the left hand side, several adders are separated by other nonarithmetic operations; on the right, the circuit is transformed so that the additions are clustered to form a 4-input adder. The 4-input adder is best implemented using a compressor tree optimized by the *Three Greedy Approach (TGA)* [6, 5]. An expert designer would recognize these identities and apply them manually to convert the first implementation into the latter. At the outset of the authors' foray into automatic arithmetic optimization, no commercially available synthesis tools or academic algorithms could perform this transformation automatically.

This paper summarizes an efficient and scalable set of algorithmic techniques, developed by the authors over the last few years. These techniques automatically explore the design space of arithmetic circuits. The proposed solu-

1063-6889/09 $25.00 © 2009 IEEE
DOI 10.1109/ARITH.2009.39

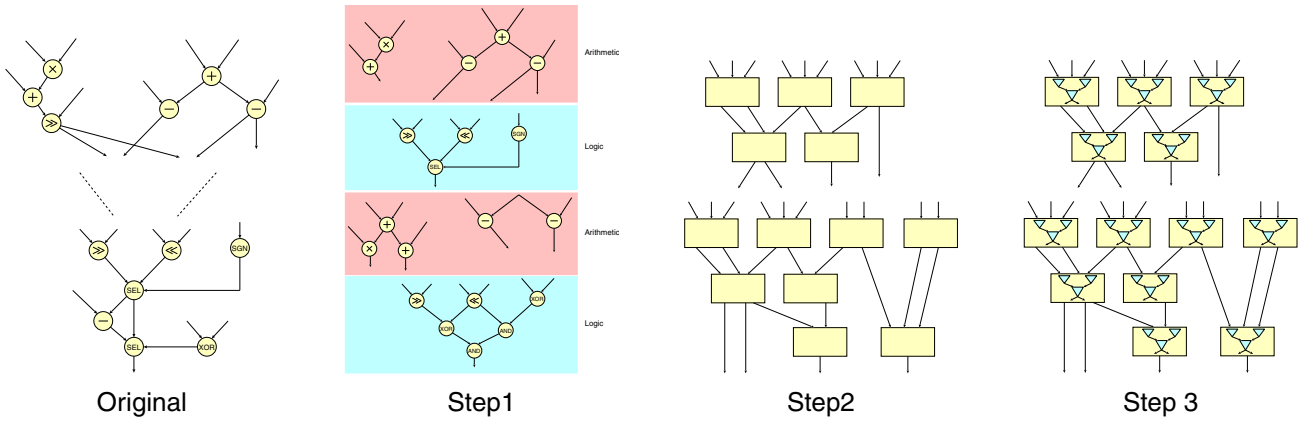

Figure 2. An illustration of how the three approaches can be used together to find an appropriate architectural implementation of an arithmetic circuit in reasonable time.

tion combines several previous contributions and involves three steps. First, at the word level, a collection of algebraic identities reorganizes the circuit into distinct arithmetic and nonarithmetic logic layers, which require different optimization strategies. As each layer can be optimized individually, the size of the space that must be searched during the optimization process is reduced significantly. Existing logic synthesis tools are sufficient for the nonarithmetic logic layers; this implies that the critical task is to optimize the arithmetic layers, which usually abound of XOR operations. The second and third steps optimize the arithmetic logic layers.

The second step optimizes the circuit by considering two or three contiguous layers at once, i.e., an arithmetic layer with a neighboring logic layer or two. This step uses the information flow in the underlying layers to determine a suitable architecture. It compresses the useful information regarding the circuit inputs into as few bits as possible, while attempting to increase parallelism. For example if a circuit corresponds to the Boolean expression $(ab + c)$, then the useful information about bits a and b is the value of ab, which can be coded into a single bit. This approach decomposes the arithmetic circuit into a set of small subcircuits, which are optimized locally by the third step. As the subcircuits are sufficiently small, the third step performs exhaustive enumeration, which is possible at this granularity but would be prohibitive if attempted in earlier stages.

The three steps increase in order of generality and computational complexity. The first step is fast, but will miss many potential optimizations. The third step is optimal and exhaustive, with a high-computational complexity—it would be impractical to use at a coarser granularity. A careful combination of the three steps can be used to find a near optimal implementation of an arithmetic circuit in a reasonable time. Fig. 2 shows the three-step optimization process.

The next three sections discuss the three steps in detail; conclusions and future work then close the paper.

2. Instruction Reordering to Minimize the Number of Layers

This section describes the first step of the proposed approach. The input is a dataflow graph, which represents computations at the arithmetic level on the granularity of words. The nodes of the graph correspond to primitive operations and the edges correspond to data dependencies. The implementation of each operation, e.g., a specific type of adder, is not specified. Transformations to reorder the operations in the data flow graph to minimize the number of distinct layers of arithmetic and nonarithmetic logic are systematically applied. The initial data flow graph is likely to have arithmetic and logic operations intermixed. Clustering arithmetic operations together facilitates optimizations, such as the use of carry-save arithmetic for multi-input addition. This type of optimization would be ineffective for a large collection of isolated adders that are separated by nonarithmetic logic.

In the works that first presented this step [9, 8], a library of algebraic identities swaps the order of arithmetic operations (primarily adders) with logic operations. Fig. 3 is a general example of this type of transformation, where addition is advanced over a logic operation. For example if the logic operation is a multiplexer, denoted as SEL, then the transformation is based on the following identity:

$$c\,?\,\sum_{i=0}^{n-1} A_i : B = \sum_{i=0,i\neq j}^{n-1} (c\,?\,A_i : 0) + c\,?\,A_j : B.$$

Advancing adders over nonarithmetic operations tends to cluster the adders at the bottom of the data flow graph.

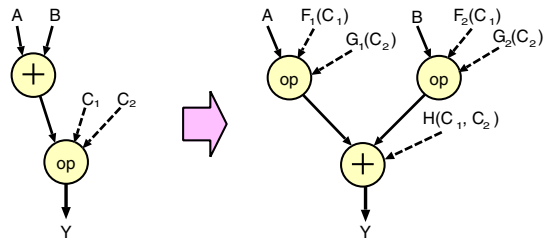

Figure 3. A general form of the sorting rules which advance addition over logic operations.

Chains and trees of adders are then replaced with multi-input addition operators that are best realized using carry-save addition. Additionally, each multiplier is replaced by a partial product generator followed by a multi-input adder, which can be merged with other adders as well.

Other identities swap the order of addition and multiplication with other logic operations such as left/right shift, bitwise NOT, etc. Ideally, we would like to swap any logic operation with any arithmetic operation, yielding two distinct layers of computation; however, this is generally impossible, as some logic operations, such as AND and OR, cannot be swapped with addition. The repeated application of these identities yields several clusters of arithmetic operations, which are effectively islands in a sea of nonarithmetic logic. Each of these islands can easily be identified and optimized independently.

The majority of arithmetic operations are adders, including subtractors, and also the partial product reduction trees of parallel multipliers, which are themselves a specific type of multi-input adder. The best implementation of a multi-input adder is called a compressor tree, and can be generated by existing methods such as the Three Greedy Approach [6, 5]. An example is shown in Fig. 1, where the original data flow graph has several adders separated by logic operations; the transformations form a multi-input addition cluster which can be replaced with a fast compressor tree.

The authors have formally proved that the system of these swapping transformations possesses properties such as persistency and local confluency [9, 8]. In other words, the use of one transformation does not render others inapplicable; moreover, the effect of applying a set of swapping transformations in any order does not change the final result. To summarize, the objective of the technique is to identify the transformations to apply, knowing that the order in which they are applied is immaterial.

In some cases, these transformations do increase the circuit area, often with marginal improvements in delay. To address this concern, the authors developed an algorithm to generate all Pareto-optimal implementations of a circuit with respect to delay and area. This approach improved the delay by up to 46%, and, in some favorable cases, reduced the area overhead by up to 10–20%.

3. Structuring an Arithmetic Circuit

The clusters of arithmetic operations identified by the techniques summarized in the preceding section are now ready for optimization; however, some parts of the nonarithmetic logic layers may be useful when optimizing the arithmetic logic, as they imply correlations between input bits to the arithmetic layers. For example, consider a floating point arithmetic unit. Depending on its implementation, a floating point adder or multiplier will contain a *Leading Zero Detector (LZD)* or a *Leading Zero Anticipator (LZA)*, which determines the length of the shift required for normalization. The LZD counts the number of zeroes of a single integer; in contrast, the LZA computes the number of leading zeroes in the sum of two integers, but without explicitly adding them. A naive implementation of the LZA circuit is therefore $\mathsf{LZA} = \mathsf{LZD}(A + B)$. Techniques exist that can transform the naive implementation into an optimized LZA; however, doing so requires the ability to optimize arithmetic and nonarithmetic logic at the same time.

Any gate-level description of a circuit containing arithmetic logic is likely to be XOR-dominated. XOR is non-monotonic and does not commute or associate with other operations such as AND and OR. Most commonly used logic synthesis tools use algebraic factorization, which is based on the monotonicity and distributivity of AND and OR operations. Consequently, these tools cannot effectively optimize arithmetic circuits.

To address this concern, the authors have proposed *Progressive Decomposition* [7], which finds an appropriate architectural implementation of an arithmetic circuit without any prior knowledge about its functionality. This is a significant improvement over methods such as TGA, which require prior knowledge that the desired circuit is a multi-input adder to be implemented as a compressor tree. Progressive Decomposition is not based on algebraic factoring, unlike most prior work; instead it exploits the ring structure of Boolean algebra under XOR and AND, and replaces Boolean factoring with the *Ideal Membership Problem*. This approach yields solutions whose quality is comparable to Boolean factoring, but with a computational complexity closer to algebraic factoring.

The authors have proved that Progressive Decomposition works for many arithmetic circuits, especially those that have effective online algorithms [3]. The key underlying ideas are also applicable to other types of arithmetic circuits as well; however, the general applicability is not supported by a formal proof.

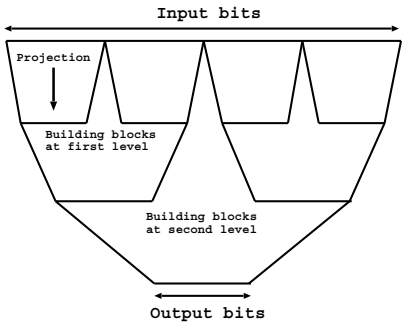

Figure 4. Illustration of the main idea to optimize the circuit by imposing hierarchy and structure.

```
progressiveDecomposition (List ℒ) {
// The function takes the list of input expressions
// builds the hierarchy to implement them.
  identities = ϕ;
  while (true) {
    G = findGroup(ℒ, k);
    (B, C) = findBasis(ℒ, G, identities);
    // This function returns a basis and coefficients
    // of its elements.
    (B, C) = minimizeBasisUsingLinearDependence(B, C);
    (B, C) = improveBasisUsingSizeReduction(B, C);
    identities = identities ∪ findIdentities(B);
    B = ReduceBasisUsingIdentities(B, identities);
    ℒ = rewriteExpr(ℒ, B);
    identities = rewriteExpr(identities, B);
    if(all elements in ℒ are literals) break; } }
```

Figure 5. The Progressive Decomposition algorithm.

Fig. 4 summarizes the rationale for Progressive Decomposition. Progressive Decomposition repeatedly selects a group of input bits to optimize, based on an appropriate metric. Given a group of input bits, it then calculates all of the information regarding these input bits that is required to evaluate the output bits of the circuit. The Boolean expressions corresponding to this information are called *Leader Expressions*, and the set of leader expressions is called a *Basis*. Most arithmetic circuits have fewer leader expressions than input bits, as the entropy of the output bits cannot exceed the entropy of the input bits; moreover usually the entropy of outputs is significantly smaller than that of input bits. The entire circuit is then rewritten in terms of leader expressions, and the circuit that computes them from the selected set of input bits is isolated as a separate subcircuit to be optimized in the third step of the flow. The leader expressions are now treated as inputs to the remaining circuit. The process of selecting input bits and computing leader expressions repeats until all leader expressions are the same as the selected input bits, or the rewritten circuit only has one input left.

The efficacy of Progressive Decomposition is based on two key intuitions. First, mutually independent computation of the information about different input bits improves the parallelism in the resulting implementation, and reduces the critical path delay. Second, the information about the input bits is reduced as much as possible at each step, which tends to reduce the overall circuit delay area.

Fig. 5 shows pseudocode for Progressive Decomposition [7]. The critical step is to compute the leader expressions for a given set of input bits. To compute the leader expressions, the original Boolean expression is rewritten in the *Reed-Muller Form* (XOR-of-product form). Reed-Muller form is canonical, and has the advantage that the Boolean expressions form a ring under the operations XOR and AND.

To compute the leader expression of a group of input bits B of a Boolean expression E, each product term in E is split into two terms, one containing the bits of B, and the other containing the remaining bits. Pairs of product terms are merged together if their first or second product terms are identical. Furthermore, the following Theorem proves that two product terms can be merged, even if they share no common product term between them, under specific cases.

Theorem 1 *An expression of the form $PQ \oplus RS$ can be factored as $(P \oplus R)T$, if $Q \oplus S \in N(P) \oplus N(R)$,*

where $N(P)$ denotes the null space of a Boolean expression P, i.e., the set of all Boolean expressions which cannot be true simultaneously with P. Checking the membership of $Q \oplus S$ in $N(P) \oplus N(R)$, is formulated as an instance of the ideal membership problem, which is based on the properties of ring algebra. After merging, Progressive Decomposition finds a minimal set of pairs, such that the first product term in all of them corresponds to the set of leader expressions. The process is explained in greater detail using examples in the authors' prior work [7].

The relations among leader expressions can be found by enumerating all Boolean circuits of bounded depth with inputs as leader expressions, and checking if the output of any of these circuits is 0 or 1. These identities can remove redundant leader expressions and can compute the null spaces of leader expressions as well.

Fig. 6, shows the result of applying Progressive Decomposition to a naive implementation of the *Leading Zero Detector* circuit. Progressive decomposition removes the high-fanin dependency, replacing it with several small fan-in circuits that have a regular structure. This optimized implementation has been proposed in the past by Oklobdzija [4]; however, to the best of our knowledge, no prior automatic synthesis technique has been able to convert the naive implementation into the optimized one shown here.

One limitation is that Progressive Decomposition cannot

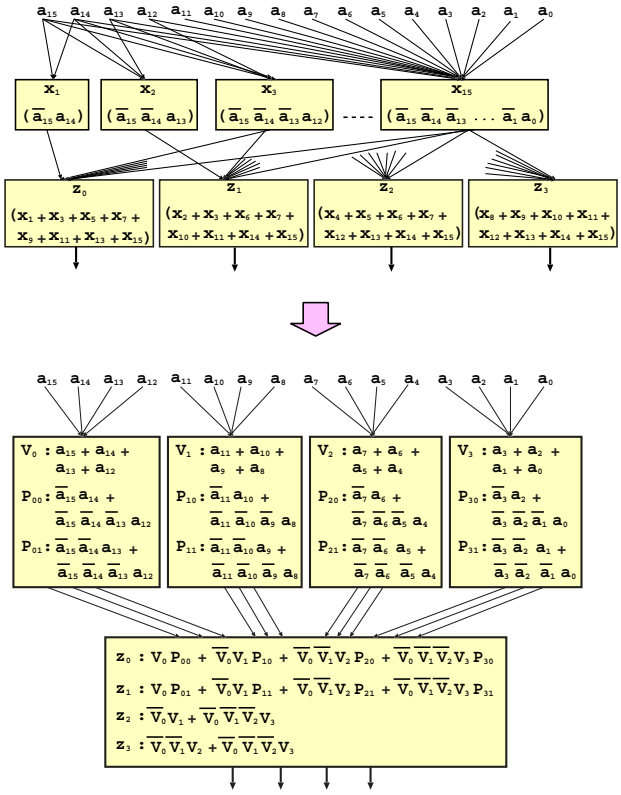

Figure 6. An example showing the efficacy of Progressive Decomposition, which can convert a naive specification into its manually designed implementation without any prior knowledge about the functionality of the circuit.

always compute leader expressions, especially when the entropy of the output bits is close to the entropy of the input bits. An example where this occurs is in the partial product generator of a parallel multiplier. When multiplying an m-bit unsigned integer with an n-bit unsigned integer, the partial product generates $m \times n$ partial product bits to sum with a compressor tree. Progressive Decomposition can optimize the compressor tree portion of the circuit, but not the complete multiplier because the partial product generator is a high-fanout component. To address this limitation, the authors intend to develop further techniques to compute bases with overlapping input bits that can handle multipliers and other circuits where Progressive Decomposition fails.

As stated earlier, Progressive Decomposition imposes the restriction that all bases must have disjoint sets of input bits; this renders the method inapplicable to many circuits. The authors' future work intends to extend Progressive Decomposition to remove these restrictions and make it more general. Another limitation of Progressive Decomposition is that some expressions can be exponentially large in Reed-Muller form, especially when several layers of nonarithmetic logic are included. Progressive Decomposition will fail on these circuits due to its high computational complexity. An additional avenue for future work is to extend Progressive Decomposition to any representation, e.g., *sum-of-products*, *product-of-sums*, or *factored form* [2]; this is left open for future work.

4. Complete Exploration of the Design Space

Progressive Decomposition imposes hierarchy on the circuit through its repeated computation of leader expressions; however, it does not provide an efficient circuit to compute the leader expressions. If they are not computed efficiently, the resulting circuit may have a poor implementation from the perspective of critical path delay and/or area. The leader expressions on the critical path should be optimized for delay, while noncritical leader expressions should be optimized for area.

To find the best implementation of the circuits that compute leader expressions, we enumerate all Pareto-optimal implementations and choose the smallest ones that do not increase the critical path. As the authors' approach entails exhaustive enumeration, its practical use is limited to small circuits. An efficient approach to compute all Pareto-optimal implementations of a circuit is presented in the authors' prior work [10].

Exhaustive enumeration saturates for circuits containing at most 5 input variables; however, through the use of novel pruning criteria, many of these implementations can be discarded. Specifically, consider a circuit that computes two leader expressions, z_1 and z_2, and the number of Pareto optimal implementations of z_1 and z_2 are m and n respectively. Intuitively, one would think that there are a total of $(m \times n)$ implementations of a circuit which computes both z_1 and z_2; however, it can be shown that at most $(m+n)$ of these implementations are Pareto-optimal, and the rest can be ignored. Furthermore, the circuits that compute z_1 and z_2 may share a common subcircuit, which can also be implemented in a Pareto-optimal fashion. These two properties show that the problem of computing all Pareto-optimal implementations a circuit can be solved efficiently using dynamic programming.

The enumeration algorithm computes the common subexpressions and their Pareto-optimal implementations in a bottom-up manner. At each step, all combinations of two input bits a and b are considered, along with $(a \oplus b)$ and ab as candidate subexpressions. Division based on Gröbner Bases [1] is used to find common subexpressions. Since any expression can be written using XOR and AND, the exploration computes all possible sets of common expres-

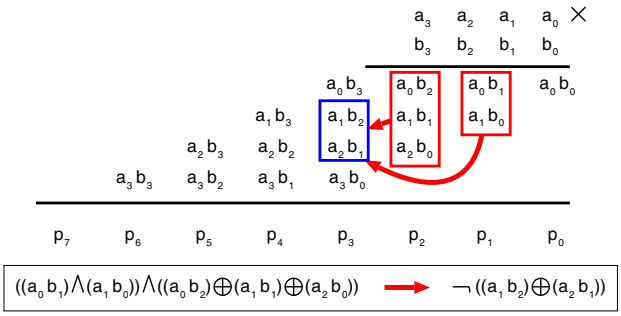

Figure 7. An illustration of complex dependencies among the partial product bits of a multiplier.

sions; however, the same common expression may be found on many branches on the exploration tree. To avoid duplication, a system of hash tables is used to prune exploration when common subexpressions found in the current branch are recognized as being found earlier. Additionally, branches are pruned if the estimated delay of the implementation found up until the current point exceeds some fixed bound.

This approach can generate numerous implementations of a circuit. For an adder, it generates the ripple carry adder, carry lookahead adder, and a variety of hybrid adders, some of which have areas comparable to the former and delays comparable to fast prefix adders. For some circuits, the approach found implementations that are faster than the best manually known implementations; one such example is a multiplier. A careful inspection of the implementation that was found reveals that the exploration exploited correlations found among the partial product bits to generate a faster circuit implementation than the best known manual designs.

Fig. 7 shows the correlation among the partial product bits discovered by the exploration algorithm. Using these correlations many complex gates such as XOR's can be converted into simpler gates such as OR's. After discovering the partial product correlation, we decided to apply directly the method based on correlation to general multipliers [11]; this resulted in a multiplier which is 20% faster compared to one produced by applying the Three Greedy Approach on a partial product reduction tree. Although the exhaustive exploration method is limited to small circuits due to its computational complexity, it has discovered new implementations whose ideas can be generalized to larger circuits.

5. Conclusions and Future Work

This paper has described a three-step flow for optimizing arithmetically-oriented datapath circuits. The input to the flow is a data flow graph describing the computation, and the output is highly optimized structural synthesizable RTL code. The three steps optimize the circuit, starting from a fairly coarse granularity, and finishing with an optimal local optimization method that is applied to fine-granularity subcircuits. Although this approach has been successful, it relies on a first step that separates logic circuits into arithmetic and nonarithmetic regions, which are synthesized separately using distinct methods (those for the nonarithmetic logic being much more mature). The ideal solution, in contrast, would be able to optimize an entire circuit without resorting to partitioning with rule-based techniques to optimize different regions; ongoing work at EPFL intends to design and implement this type of solution.

References

[1] T. Becker and V. Weispfenning. *Gröbner Bases: A Computational Approach to Commutative Algebra*. Springer, New York, 1993.

[2] R. K. Brayton. Factoring logic functions. *IBM Journal of Research and Development*, 31(2):187–98, Mar. 1987.

[3] M. D. Ercegovac and T. Lang. *Digital Arithmetic*. Morgan Kaufmann, San Francisco, Calif., 2004.

[4] V. G. Oklobdzija. An algorithmic and novel design of a leading zero detector circuit: Comparison with logic synthesis. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, VLSI-2(1):124–28, Mar. 1994.

[5] V. G. Oklobdzija, D. Villeger, and S. S. Liu. A method for speed optimized partial product reduction and generation of fast parallel multipliers using an algorithmic approach. *IEEE Transactions on Computers*, C-45(3):294–306, Mar. 1996.

[6] P. F. Stelling, C. U. Martel, V. G. Oklobdzija, and R. Ravi. Optimal circuits for parallel multipliers. *IEEE Transactions on Computers*, C-47(3):273–85, Mar. 1998.

[7] A. K. Verma, P. Brisk, and P. Ienne. Progressive decomposition: A heuristic to structure arithmetic circuits. In *Proceedings of the 44th Design Automation Conference*, pages 404–9, San Diego, Calif., June 2007.

[8] A. K. Verma, P. Brisk, and P. Ienne. Data-flow transformations to maximize the use of carry-save representation in arithmetic circuits. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, CAD-27(10):1761–74, Oct. 2008.

[9] A. K. Verma and P. Ienne. Improved use of the carry-save representation for the synthesis of complex arithmetic circuits. In *Proceedings of the International Conference on Computer Aided Design*, pages 791–98, San Jose, Calif., Nov. 2004.

[10] A. K. Verma and P. Ienne. Towards the automatic exploration of arithmetic circuit architectures. In *Proceedings of the 43rd Design Automation Conference*, pages 445–50, San Francisco, Calif., July 2006.

[11] A. K. Verma and P. Ienne. Improving XOR-dominated arithmetic circuits by exploiting dependencies between operands. In *Proceedings of the Asia and South Pacific Design Automation Conference*, pages 601–8, Yokohama, Japan, Jan. 2007.