Arithmetic Transformations to Maximise the Use of Compressor Trees

Paolo Ienne and Ajay K. Verma
Federal Institute of Technology Lausanne
Processor Architecture Laboratory
IN-F Ecublens, 1015 Lausanne, Switzerland
{Paolo.Ienne, AjayKumar.Verma}@epfl.ch

Abstract

Complex arithmetic computations, especially if derived from bit-level software descriptions, can be very inefficient if implemented directly in hardware (e.g., by translation of the relevant C section in VHDL or Verilog). In this paper we show that known arithmetic optimisation techniques are in some cases insufficient to achieve the high-performance implementation that a designer could produce through an attentive study of the computation. We therefore introduce an algorithm to restructure dataflow graphs so that they can be synthesized in high-quality arithmetic circuits, especially when arithmetic operations are interspersed with logic operations. On typical software benchmarks, the new technique reduces the critical path by around 20–40% and generally achieves the quality of manual implementations. In many cases, our algorithm also manages to reduce the cell area by 10–20%.

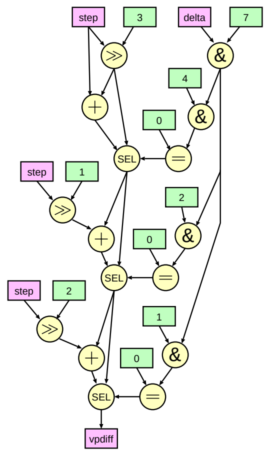

Figure 1. A typical dataflow graph obtained by parsing automatically the C-language bit-level description.

1. Introduction

With the decreasing cost of VLSI transistors and the increasing pressure toward energy-efficient high-performance systems-on-chip, complex arithmetic computations implemented in dedicated hardware will have a growing importance. Commercial libraries of arithmetic components have now reached virtually every designer; yet, implementations of arithmetic computations are often straightforward mappings of single operations to elements of a library with limited optimisation attempts.

This absence of optimisations is particularly negative in computations described at the bit-level in typical programming languages—as it is often the case of embedded applications in hardware/software codesign (e.g., limited precision fixed-point, ad-hoc limited precision floating-point, cryptography). Figure 1 shows a typical example of computational kernel; the figure represents the dataflow graph of the original description in C-language of the kernel of `adpcmdecode` [3]. The dataflow represents a slightly approximated 16×4 multiplier whose implementation should be practically as fast as a standard parallel multiplier of the same size. Yet, direct synthesis of this dataflow graph (appropriately rewritten in VHDL or Verilog) would result in a suboptimal implementation due to a sequence of carry-propagate additions interspersed with multiplexers (SEL nodes in the figure, resulting from an if-conversion pass on the code); a good implementation would instead replace the additions with a Wallace-like compressor tree followed by a single carry-propagate final adder [4]. Our goal is to achieve the speed of the latter implementation with automatic transformations of the dataflow graph.

2. Related Work

The typical focus in traditional high-level synthesis research has been on optimal scheduling of the dataflow graph

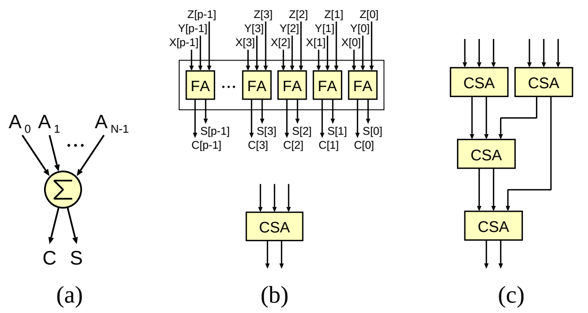

Figure 2. Compressor trees. (a) The symbol used here for a compressor tree. (b) A carry-save adder, which is the simplest 3-input compressor tree. (c) Carry-save adders used to build a compressor tree.

operations and on resource sharing and binding [1]. Some work has addressed the optimisation of important classes of computations (e.g., linear [6]) or, more recently, has used symbolic techniques to rewrite dataflow graphs to achieve minimal critical path with normal arithmetic operators [5]. Automatic improvements to the implementation of the arithmetic circuitry itself have seldom be addressed. On the other hand, once arithmetic operators (such as multipliers and adders) are created through libraries of generators (such as Synopsys's *DesignWare*), logic synthesis techniques have limited chances of improving the design.

The use of the carry-save representation in building fast multiple-input adders is a traditional arithmetic technique. The single most important area of application is in the design of parallel multipliers [4]. A large body of literature exists on the best way to build the compressor tree of parallel multipliers and its analysis goes beyond the scope of this review. Among the latest contributions, [7] discusses various heuristic and optimal techniques to design arbitrary compressor trees (called *Partial Product Reduction Trees* by the authors—see Figure 2). We will use a simple heuristic similar to the *Three-Greedy Approach* [7] to obtain the results of Section 4.

To our knowledge, none of the classic works on compressor trees addressed the problem of exposing automatically additional opportunities for the carry-save representation. An exception is [2] which discusses some simple transformations of the topology of the circuit to maximise the use of cascaded *Carry-Save Adders* (CSAs, see Figure 2)—which in fact constitute compressor trees. In [2] and in [9], the main emphasis is in the optimal allocation of the addenda to the cascaded CSAs. Although somehow similar in the intents, our work is more radical in trying to exploit every possible occasion for proficient uses of compressor trees—the design of optimised compressor trees being left to algorithms such as those discussed in [7]. Synopsys's synthesizers also have some capabilities to infer the use of CSAs [8]; the opportunities seized by Synopsys's *Behavioural Optimisation of Arithmetic (BOA)* are probably similar to those described in [2]. We will show that, in addition, we address and successfully optimise some new cases of practical interest.

3. Arithmetic Optimisations

In essence, our goal is to transform acyclic dataflow graphs to maximise the opportunities to use compressor trees. We call compressor tree (see Figure 2) a circuit which takes $N \geq 3$ input words and produces 2 output words S and C:

$$(S, C) = \mathcal{F}\left(A_0, A_1, ..., A_{N-1}\right).$$

The function $\mathcal{F}$ can be any function such that

$$S + C = \sum_{i=0}^{N-1} A_i.$$

In general, the use of compressor trees will reduce the critical path of a hardware implementation of the dataflow graph—this is true in all but special cases and therefore, in our algorithm, we use compressor trees wherever possible.

3.1. Basic Idea

We assume to start from an acyclic dataflow graph representing the desired operation (e.g., Figure 1). This could be converted to a suitable hardware description language (e.g., VHDL or Verilog) and then a logic synthesizer and a library of standard arithmetic components can be used to implement the computation in hardware. Instead, to obtain faster and more efficient implementations, before writing out the graph for the synthesizer, we can apply three types of transformations to the graph. They address three different goals:

1. **Expose adders in other arithmetic operations.** It is a rather classic set of transformations, just rewriting operations such as subtractions and multiplications as logic operations and additions.

2. **Separate additions from logic operations.** This is the fundamental set of rules to achieve our goal: both software constructs (e.g., if-conversion) and other logic operations (e.g., shifts, inversions, bitwise operations, etc. needed in programs to describe high-level operations at the bit-level) are scattered among the sequences of adders exposed by the previous class of transformations. If such logic operations are not moved away, adders will not be merged and the opportunities for optimisation will be missed. These rules are the main novelty of our approach.

$$\begin{array}{rcl} -A & \Rightarrow & \overline{A}+1 \\ A-B & \Rightarrow & A+\overline{B}+1 \\ A*B & \Rightarrow & \mathsf{Sum}(\mathsf{Comp}(\mathsf{PP}(A,B))) \end{array}$$

Table 1. Rewriting transformations for non-primitive arithmetic operations.

$$\begin{array}{l} A+B+C+\ldots \\ \quad\Rightarrow\quad \mathsf{Sum}(\mathsf{Comp}(A,B,C,\ldots)) \\ \mathsf{Comp}(A+B,C,D,\ldots) \\ \quad\Rightarrow\quad \mathsf{Comp}(A,B,C,D,\ldots)) \\ \mathsf{Comp}(\mathsf{Comp}(A,B,\ldots),P,Q,\ldots) \\ \quad\Rightarrow\quad \mathsf{Comp}(A,B,\ldots,P,Q,\ldots)) \end{array}$$

Table 2. Transformations to collapse multiple additions into compressor trees.

3. **Combine adders in compressor trees and carry-propagate final adders.** This is also a set of rather simple transformations, roughly corresponding to those implemented already in commercial tools. Typical rules combine sequences of adders into compressor trees with a final adder, and absorb adders into larger compressor trees.

It should be noted that only the last class of transformations significantly affects the timing of the circuit which will implement the graph. The other transformations simply prepare the dataflow graph for the optimisation; only some of them have minor effects on the timing and we assume their impact negligible. Note that we do not implement any transformation which aims at optimising the logic operations—we fully rely for this on the capabilities of traditional logic synthesizers.

Table 1 lists the three rewriting transformations mentioned in bullet 1 above. The third transformation implements a classic parallel multiplier [4]: $\mathsf{PP}()$ produces a set of partial products, $\mathsf{Comp}()$ represents a compressor tree producing the sum of all partial products in carry-save representation, and finally $\mathsf{Sum}()$ is a normal carry-propagate addition of the sum and carry components produced by the compressor tree. Table 2 lists the optimising transformations mentioned in bullet 3; they are used to exploit whenever possible a carry-save representation. Note that none of the above transformations is applied when an intermediate result (such as $A + B$ in the second transformation) is also used elsewhere in the dataflow graph. Duplication of the parent node (the adder, in the second transformation) could make a further timing optimisation possible at the price of additional area; we do not implement such transformations but they could be easily included.

3.2. Sorting the Operation Order

The transformations mentioned in the second bullet of Section 3.1 are responsible to separate arithmetic operations from logic operations; the goal is to maximise the opportunities of using compressor trees. We formalise here the separation problem; we consider now the dataflow graph in its original form, before any other transformation is applied. A discussion of the ordering of all our transformations is deferred to the next section.

Let's call $G(V, E)$ the directed acyclic graph representing the dataflow of the required computation. Nodes V represent primitive operations and edges E represent data dependencies. Each graph G is associated to a graph $G^+(V \cup V^+, E \cup E^+)$ which contains additional nodes V^+ and edges E^+. The additional nodes V^+ represent input and output values of the circuit. The additional edges E^+ connect nodes V^+ to V, and nodes V to V^+. The nodes of G are ordered such that if G contains an edge (u, v) then u appears before v in the ordering. The function $\mathrm{Ord}(\cdot)$ returns the position of a node in the ordering.

The nodes V can be of several types; we assign them to two classes: arithmetic and logic. Arithmetic nodes are: addition, negation, subtraction, compressor tree, multiplication, and ordering comparisons (e.g., set greater). Logic nodes are: bitwise AND, bitwise OR, bitwise exclusive OR (XOR), bitwise negation, shift right, shift left, partial-product generation, selection (multiplexor), and equality comparisons (e.g., set equal). For brevity, we call A the former class and L the latter. The function $\mathrm{Class}(\cdot)$ returns the class of a node.

Finally, we call two graphs G and G' *semantically equivalent* if all outputs of two circuits implementing graph G and graph G' respectively are identical under any combination of input values.

A first formulation of our graph transformation problem is as follows:

Problem 1 *Given a graph G, transform it in a semantically equivalent graph $G'(V', E')$ where the following property holds: for all nodes u and v such that $\mathrm{Class}(u) = \mathsf{A}$ and $\mathrm{Class}(v) = \mathsf{L}$, either it is always $\mathrm{Ord}(u) < \mathrm{Ord}(v)$ or always $\mathrm{Ord}(u) > \mathrm{Ord}(v)$.*

We call such a graph G' *sorted*. For instance, the motivational example of Figure 1 is unsorted, whereas that of Figure 5 is sorted and thus apt for implementation. The fact that a graph is sorted is sufficient, although not necessary, to be able to produce an optimal implementation with the transformations of Table 2. If L operations are mixed within A operations, they can only prevent, in many cases, the creation of large and fast compressor trees.

Sorting would be a feasible task if it was always possible to exchange the order of class A and L nodes without changing the semantics of the dataflow graph. This is possible

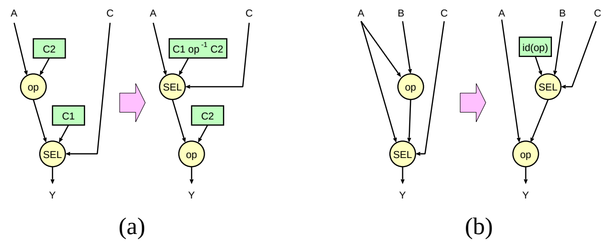

Figure 3. Special reordering transformations including a selector and an arithmetic node.

in some cases: For instance, if the graph contains an addition followed by a shift left operation by a positive quantity, the order of the two operations can be exchanged by shifting left the two operands by an equal amount and then performing the addition. Since shifting to the left is equivalent to multiplying by an appropriate power of two, this transformation corresponds to an elementary application of the distributive property. Unfortunately, reordering is not always possible: if in the previous example the left shift was replaced by a right shift (also by a positive amount), the operation would correspond to an *integer* division; in this case the distributive property cannot apply because the result would be wrong by exactly one unit if the sum of the two reminders exceeds the denominator.

The possibilities to advance class L operations over class A operations are relatively limited:

- In many cases, no sorting is possible, at least in the general case. No transformations are available for basic bitwise logic operations, for effective shifts right, for partial product generation (i.e., a network of bitwise ANDs), for selectors, and for equality comparison (i.e., a bitwise XOR and an AND reduction operation).

- Bitwise NOT and effective shifts left can be advanced over sums and compressor trees. Exchange of bitwise NOT with addition is based on the relation $-A = \overline{A} + 1$. It can be easily shown that, when summing n addenda A_i, the following holds:

$$\overline{\left(\sum_{i=0}^{n-1} A_i\right)} = \sum_{i=0}^{n-1} \overline{A_i} + n - 1.$$

- Although no general sorting transformations can be identified for selectors following arithmetic operations, two special cases are represented in Figure 3. In the first case, the existence of two constants around the selector and the arithmetic operator can be exploited to exchange the order of the two nodes. Since the inverse operation needs to be used to compute a new constant (subtraction for addition, division for multiplication), numerical accuracy issues need to be taken care of (e.g., $C2$ must be a divisor of $C1$ if the arithmetic operation is a multiplication). In the second case, a particular but frequent sequence can be sorted when the arithmetic operation admits an identity element (e.g., zero for addition or one for multiplication).

- In some cases, it can be shown that the transformation for additions is sufficient to sort also graphs containing subtractions and multiplications, if these have previously been rewritten as indicated in Table 1. In other cases, the opposite applies: rewriting a complex operation makes further transformations impossible.

```
Optimise (G) {
  // sorting before rewriting
  advanceSelOverOpWithIdElem(G);
  advanceSelOverConstantMul(G);
  // rewriting
  rewriteSubtractions(G);
  rewriteNegations(G);
  rewriteMultiplications(G);
  do {
    // sorting involving only adders
    changes = advanceSelOverOpWithIdElem(G);
    changes |= advanceSelOverConstantAdd(G);
    changes |= advanceNotOverAddOrCompr(G);
    changes |= advanceShiftOverAddOrCompr(G);
    // collapsing adders
    changes |= mergeAddWithAdd(G);
    changes |= mergeAddWithCompressor(G);
    changes |= mergeCompressorWithCompressor(G);
    // cleaning-up
    changes |= propagateConstants(G);
    changes |= groupConstantsInCompressors(G);
    changes |= mergeShiftsByConstant(G);
    changes |= suppressUselessCompressors(G);
  } while (changes);
}
```

Figure 4. The optimisation algorithm.

3.3. Optimisation Algorithm

The fact that there are only limited sorting possibilities implies that our Problem 1 might not have a solution. We therefore relax the problem formulation to try to sort a graph "as much as possible". For this, it is useful to define the number of *layers* of a graph: it counts the maximum number of transitions A/L and L/A plus one, along the graph path which contains the largest number of transitions. Intuitively, for a given starting graph, transformations resulting into a smaller number of A layers will offer better possibilities of optimisation using the transformations of Table 2. A nontrivial sorted graph has two layers. Formally, we try to solve the following problem, instead of Problem 1:

Problem 2 *Given a graph G, transform it in a semantically equivalent graph G' (V', E') which has a minimal number of layers.*

Benchmark	Original		Synopsys BOA [8]		**Our Arithmetic Optimisations**		Manual Implementation	
	Delay (ns)	Area (μm^2)	Delay (ns)	Area (μm^2)	**Delay (ns)**	**Area (μm^2)**	Delay (ns)	Area (μm^2)
ADPCM Decoder [3]	1.92	15,071	1.92 (0%)	15,071	1.10 **(-43%)**	12,068 **(-20%)**	1.18	9,696
G.721 [3]	4.92	77,495	4.72 (-4%)	85,900	3.65 **(-26%)**	64,076 **(-17%)**	—	—
Shift-and-add 8-bit Multiplier	2.31	16,934	2.31 (0%)	16,934	1.49 **(-35%)**	14,888 **(-12%)**	1.63	9,204
Multiply Accumulate ($a \times b + c$)	2.02	14,662	1.47 (-27%)	14,217	1.55 **(-23%)**	19,539 **(+33%)**	—	—
Polynomial 4th degree (optimised)	5.22	130,644	5.64 (+8%)	144,053	5.63 **(+8%)**	170,142 **(+30%)**	—	—
Polynomial 4th degree (Hörner form)	6.67	96,299	6.43 (-4%)	103,540	6.32 **(-5%)**	121,995 **(+27%)**	—	—
Video Mixer [8]	4.88	206,492	4.46 (-9%)	143,098	4.65 **(-5%)**	195,755 **(-6%)**	—	—

Table 3. Results of our arithmetic optimisations on some benchmarks.

We call such a graph G' *maximally sorted*. Clearly, if Problem 1 admits a solution, this is also a solution of Problem 2.

Figure 4 shows a heuristic algorithm to perform the task. It begins by applying everywhere in the graph the sorting rules which are effective on operations other than additions. Immediately thereafter, the rewriting rules of Table 1 are applied wherever possible. These two steps happen only once. Then a series of transformations is applied repeatedly until there is no further change. They belong to three classes: (1) sorting rules of involving only additions; (2) transformations to collapse all additions in compressor-trees as shown in Table 2; (3) some clean-up transformations to remove some "leftovers" of the preceding transformations. Clean-up transformations perform the following tasks: (a) implement a classic constant propagation pass; (b) look for several constants entering a compressor tree and replace them by a single one; (c) combine multiple successive shifts by constants; (d) check for degenerate compressors with two or less inputs and remove them.

Note that the sequence of application of the various transformations reflects the natural order expressed in the bullets of Section 3.1. The only exceptions are the sorting transformations that need to be applied before decomposing the arithmetic operator into primitive components. Also, normal sorting transformations and addition merges are repeatedly applied because each iteration may expose further opportunities. The algorithm of Figure 4 is not guaranteed to minimise the number of layers as required by Problem 2. Yet, by repeatedly looking for sorting opportunities it helps approaching the optimal solution.

After optimisation, a final pass analyses the required number of bits for all variables carried by the edges E' of G'. This is used to remove some unnecessary edges and to simplify some arithmetic components, such as partial product generators.

4. Experimental Results

We have written an experimental optimiser which takes dataflow graphs (possibly extracted automatically from application code written in C) and returns an optimised graph. Both the original and the optimised graphs can be converted to VHDL and compiled with a standard logic synthesizer—we used Synopsys's *Design Compiler*. Standard arithmetic components are generated by the synthesizer—in our case by the *DesignWare* library. In the case of the optimised graph, we implemented two special netlist generators for partial-product generators (essentially AND networks) and compressor trees. The latter are built with a technique similar to the already mentioned Three-Greedy Approach [7].

We have run several benchmarks and synthesized the result for minimal delay on an ASIC library of a common $0.18\mu m$ CMOS technology. The results are shown in Table 3. The "Original" columns refer to the direct synthesis of the dataflow graph. "Synopsys BOA" uses the optimisations available in some versions of Synopsys's *Design Compiler* and in *Behavioral Compiler* and mentioned in Section 2. The results of the columns "Our Arithmetic Optimisations" implement the algorithm of Section 3. The last columns show a reference manual implementation of the designs, when available.

The first three circuits in Table 3 are typical arithmetic dataflow graph coming from software description. The first is the example of Figure 1. The second (G.721) is a kernel of a standard audio encoder [3]: it represents an ad-hoc limited-precision floating-point multiplier including a small quantization table. The last is a classic shift-and-add multiplier as one could write it for a processor missing a native multiplication instruction; functionally, it has some similarity to the ADPCM example but is, in fact, coded in C in a profoundly different manner: it does not use `if-then` constructs and hence the dataflow graph does not contain SEL nodes. The bottom four benchmarks are simple, purely arithmetic, circuits: the first is a multiplication followed by an addition, the second and the third are fourth-degree polynomials written in two different forms, and the last one is a video mixer application appropriate for demonstrating classic arithmetic optimisations [8].

Reading the results from the bottom group, one can see that on purely arithmetic benchmarks our optimisation tech-

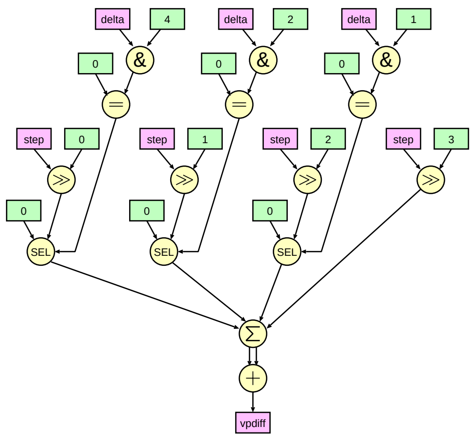

Figure 5. The motivational example of Figure 1 after optimisation.

niques achieve practically the same results of existing optimisers. The only noticeable difference concerns the area performance, but our algorithm addresses only delay and we have done no effort to minimise area. The first and best form of the polynomial is the only case where the use of such optimisations is counterproductive; it is one of the rare situations where compressor trees are not beneficial.

In the benchmarks of the first group, where logic operations are scattered among arithmetic computations, BOA misses completely the potentials for optimisation; our algorithm, on the other hand, is effective in separating logic and arithmetic operations and, in some cases, manages to sort completely the dataflow graph, making an optimal implementation possible. Once we have optimised the architecture of the arithmetic part, we then rely on the synthesizer for an effective simplification of the logic network. Timingwise, our results have critical paths reduced by 20–40% and are even marginally better than the reference manual implementations—probably due to a better quality of the generated compressor trees compared to the *DesignWare* components. On these circuits we also have small area advantages, in the range 10–20%.

As an example, Figure 5 shows the graph of Figure 1 after transformation: it is now fully-sorted. Once passed to a synthesizer, each complex branch—corresponding to a partial product generation network—gets simplified as expected into a simple set of AND gates.

5. Conclusions

We have presented an algorithm to optimise complex arithmetic circuits. Apart from a few pieces of previous work, we believe that too small work has been done in the area of arithmetic optimisation: on one side, logic synthesis is unable to attain, alone, the potentially achievable results; on the other side, the importance of implementing automatically efficient arithmetic circuits is growing due to the proliferation of digital signal-processing in ASIC and FPGA designs, and to the need of implementing software-derived computational kernels in hardware accelerators, coprocessors, or special functional units.

Our algorithm integrates well into a traditional synthesis flow and simply prepares the dataflow graph for the logic synthesizer to make the best out of it. It is based on extensive use of the carry-save representation and, compared to existing work in the domain, exposes more optimisation opportunities that previously available. On practical benchmarks, it improves the critical path of 20–40% and reduces the area by 10–20%.

The presented algorithm is heuristic and cannot guarantee to achieve optimal results—although it does in many cases. We are investigating possibilities to improve it, possibly to the point of achieving optimality.

References

[1] G. De Micheli. *Synthesis and Optimization of Digital Circuits*. McGraw-Hill, New York, 1994.

[2] T. Kim, W. Jao, and S. Tjiang. Circuit optimization using carry-save-adder cells. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, CAD-17(10):974–84, Oct. 1998.

[3] C. Lee, M. Potkonjak, and W. H. Mangione-Smith. MediaBench: A tool for evaluating and synthesizing multimedia and communicatons systems. In *Proceedings of the 30th Annual International Symposium on Microarchitecture*, pages 330–35, Research Triangle Park, N.C., Dec. 1997.

[4] A. R. Omondi. *Computer Arithmetic Systems*. Prentice Hall, New York, 1994.

[5] A. Peymandoust and G. De Micheli. Symbolic algebra and timing driven data-flow synthesis. In *Proceedings of the International Conference on Computer Aided Design*, pages 300–5, San Jose, Calif., Nov. 2001.

[6] M. Potkonjak and J. Rabaey. Maximally fast and arbitrarily fast implementation of linear computations. In *Proceedings of the International Conference on Computer Aided Design*, pages 304–8, Santa Clara, Calif., Nov. 1992.

[7] P. F. Stelling, C. U. Martel, V. G. Oklobdzija, and R. Ravi. Optimal circuits for parallel multipliers. *IEEE Transactions on Computers*, C-47(3):273–285, Mar. 1998.

[8] Synopsys. *Creating High-Speed Data-Path Components—Application Note*, version 2001.08 edition, Aug. 2001.

[9] J. Um and T. Kim. An optimal allocation of carry-save-adders in arithmetic circuits. *IEEE Transactions on Computers*, C-50(3):215–233, Mar. 2001.