Arithmetic Optimization for Custom Instruction Set Synthesis

Ajay K. Verma*, Yi Zhu†, Philip Brisk*, Paolo Ienne*
*Ecole Polytechnique Fédérale de Lausanne (EPFL), School of Computer and Communication Science, CH-1015, Lausanne, Switzerland
E-mails: {ajaykumar.verma,philip.brisk,paolo.ienne}@epfl.ch
†Department of Computer Science and Engineering, University of California, San Diego, La Jolla, CA 92093
E-mails: y2zhu@ucsd.edu

Abstract—**One of the ways that custom instruction set extensions can improve over software execution is through the use of hardware structures that have been optimized at the arithmetic level. Arithmetic hardware, in many cases, can be partitioned into networks of full-adders, separated by other logic that is better expressed using other types of logic gates. In this paper we present a novel logic synthesis technique that optimizes networks of full adders and is intended for use in the context of custom instruction set synthesis. Unlike earlier work (e.g., Three Greedy Approach [1], [2]) our approach does not require any prior knowledge about the functionality of the circuit. The proposed technique automatically infers the use of carry-save arithmetic, when appropriate, and suppresses its use when unfavorable. Our approach reduces the critical path delay through networks of full adders, when compared to the Three Greedy Approach, and in some cases, reduces the cell area as well.**

I. Introduction

One important component of application-specific systems is the extensible processor which the user may augment with custom *instruction set extensions (ISEs)* prior to fabrication. The hardware unit corresponding to an ISE is called an *Application-speci¿c Functional Unit (AFU)*. This paper focuses on the process of AFU synthesis, i.e., generating a good hardware implementation of each AFU. This may seem unneeded, as logic synthesis is generally considered to be mature; however, this is not the case. Logic synthesis techniques work quite well for finite state machines and control dominated circuits, but are generally ineffective for arithmetic circuits, meaning that there is a large gap between arithmetic circuits designed by hand and the results of logic synthesis.

This paper presents an arithmetically-oriented logic synthesis technique for ISEs that focuses on networks of *full adders* (FA) and *half adders* (HA). An FA (HA) is a circuit having three (two) input bits, that counts the number of input bits set to 1 and outputs the result as an unsigned two-bit binary number. Many arithmetic circuits, including multi-input adders and the partial product reduction trees of parallel multipliers, employ some rudimentary form of counting, and are built from networks of FAs and HAs. The most common way of computing a multi-input addition is through *compressor tree* introduced by Wallace [3] and Dadda [4]. A compressor tree takes a set of n integers and reduces them to two output values *sum (S)* and *carry (C)* whose sum is the same as the sum of n input integers. An appropriate final adder [5] computes the sum $S + C$. Compressor trees were further optimized for delay by Oklobdzija *et al.* [1], [2]. They introduced the Three Greedy Approach (TGA), which algorithmically constructs delay-optimal compressor trees and can account for input bits with non-uniform arrival times.

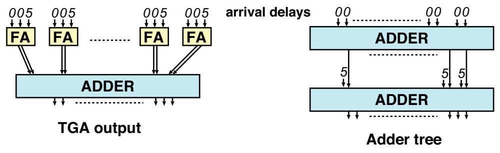

Fig. 1. In case of skewed input arrival delays an adder tree might be preferable over compressor tree. This is because the compressor tree must wait for the three input integers to arrive before it can perform any computation. Due to the late arrival time of the third integer, it may be profitable to add the early arriving integers first with a CPA, in parallel with the logic that delays the third integer. The final CPA, which adds the sum of the first two integer to the late-arriving third integer, can begin as soon as the third integer arrives, and does not have to wait for a compressor tree.

There are three main issues with the TGA. The first issue is that the TGA requires some foreknowledge that the circuit to optimize is a multi-input adder. The algorithm presented in this paper does not require this foreknowledge, and can also optimize other classes of circuits realized as networks of FAs and HAs. Note that, the ISE identification algorithms do not provide any functional information about the selected ISE.

A second weakness is that the TGA assumes the best implementation of a multi-input adder is a compressor tree. If the arrival time of the input bits is significantly skewed the best implementation may include one or more adders embedded in a compressor tree. An example of this phenomena is shown in Fig. 1. where TGA produces a sub-optimal result.

The third drawback of the TGA is that it assumes a single compressor tree is always the best implementation of a multi-input adder; however, situations exist where splitting one compressor tree can reduce the delay. An example is shown in Fig. 2 where the optimal implementation uses two smaller compressor trees, each with its own CPA. To address these shortcomings, we present an algorithm to optimize networks of FAs and HAs without foreknowledge of the function that it computes. The algorithm finds solutions that are least as good

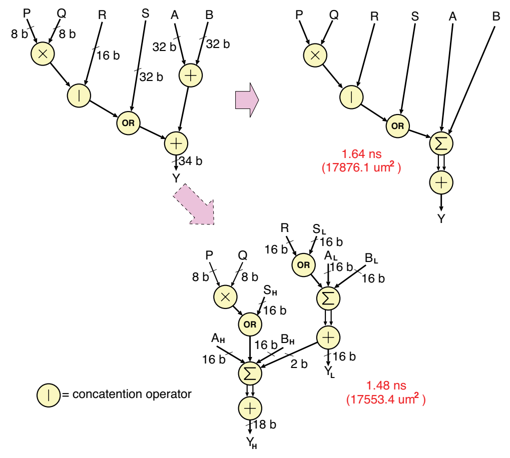

Fig. 2. An example of a multi-input addition, which shows that sometimes a hybrid of adder tree and compressor tree is faster than compressor tree alone.

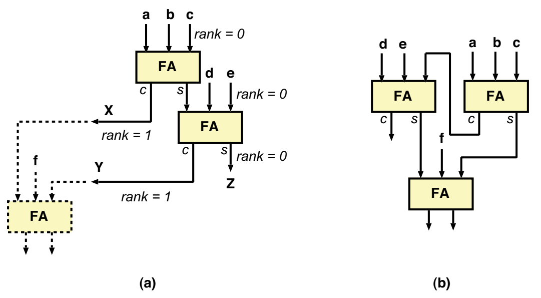

Fig. 3. (a) The inputs signals a, b, c, d, and e are symmetric if the outputs X and Y are used in a symmetric way, e.g., as the operands of a full adder etc. (b) A circuit which is not well-ranked, because for any assignment the three inputs of bottommost FA cannot have the same rank.

as the TGA in terms of delay, and often better.

II. State of the Art

ISE identification has matured during the last decade [6]. It has been observed that most of the chosen ISEs are arithmetic-oriented circuits. Multi-input addition is a common instance of arithmetic circuit found in ISEs [7].

Wallace [3] and Dadda [4] respectively introduced and optimized carry-save adders as a method for multi-input addition. Their techniques produced regular compressor trees which are suboptimal with respect to delay. Oklobdzija *et al.* [1], [2] introduced the TGA, which is optimal under the assumptions discussed in Section I. A similar approach at word level was presented by Um and Kim *et al.* [8].

All of these prior techniques assume that the circuit to synthesize is a multi-input adder or parallel multiplier and that the best implementation is a monolithic compressor tree. This paper presents a new method to optimize these circuits that is more general and is more effective at minimizing delay for a wider class of arithmetically-oriented circuits.

III. Main Idea and Algorithm

The input to our algorithm is a circuit built from FAs and HAs, which we intend to optimize. For circuits that are produced from high-level descriptions, we assume that each CPA is implented as a ripple-carry adder. Note that converting all adders to ripple-carry adder worsens the delay temporarily; however, this is done only to realize the circuit as a tree made of full and half adders. At the end of the algorithm, all carry chains are converted to faster CPAs.

The first step, described in Section III-A, is to test whether the input is a compressor tree-like circuit. If so, the circuit is optimized by rearranging the assignment of commutative input bits in order to reduce delay. Second, we determine whether instantiating additional HAs can reduce delay. Last, we identify ripple carry adders that occur within the compressor tree, and replace them with faster hybrid adders introduced by Stelling *et al.* [5].

A. Compressor Tree-like Circuits

This section presents an algorithm to partition the input signals into sets that are commutative, and then reassign them to reduce delay. A set of signals that are connected to the input of the same HA or FA are *symmetric* (symmetry, in this context, being a structurally-defined subset of commutativity). For example, in Fig. 3 (a) signal f and internal wires X and Y are symmetric, as they are inputs to the same FA. In principle, these three signals can be swapped in order to internally optimize the delay of an FA.

Our symmetry analysis uses the *rank* of a signal. We assign the ranks to signal wires in such a way that the following three properties are maintained:
(a) The ranks of all input signals of an FA (HA) are the same.
(b) The rank of the sum output of the FA (HA) is equal to the rank of the FA (HA) inputs.
(c) The rank of the carry output of the FA (HA) is equal to 1 plus the rank of the FA (HA) inputs.

We define the rank of an FA (HA) to be the same as the rank of its inputs. One such assignment if it exists, can be found easily by using an iterative algorithm. In each iteration a node is chosen whose rank is not assigned. The rank of this node is set to be zero and the ranks of all FA (HA) reachable from this node are computed according to the three rules mentioned above. If there is no such assignment possible, then the algorithm flags an error. Applying this algorithm on the circuit of Fig. 3(a) results in the shown assignments. A circuit is considered *well-ranked* if and only if it has a valid rank assignment, e.g., the circuit of Fig. 3(a) is well-ranked, while the circuit of Fig. 3(b) is not. By construction, compressor trees are always well-ranked. This is the first condition for our algorithm to be applicable. In a well ranked circuit, the symmetric signals are grouped based on the following theorem.

Theorem 1: The input signals of two FAs FA_1 and FA_2 are symmetric, if and only if the following two conditions are true:
(a) Either (1) the sum output of FA_1 and FA_2 are unused, or (2) there are paths from FA_1 and FA_2 to a common FA via sum edges only.
(b) The carry signals of all FAs in the above mentioned paths are symmetric.

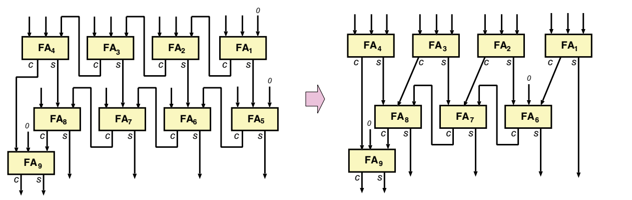

Fig. 4. An example showing how the symmetric analysis can be used to convert the adder tree into compressor tree.

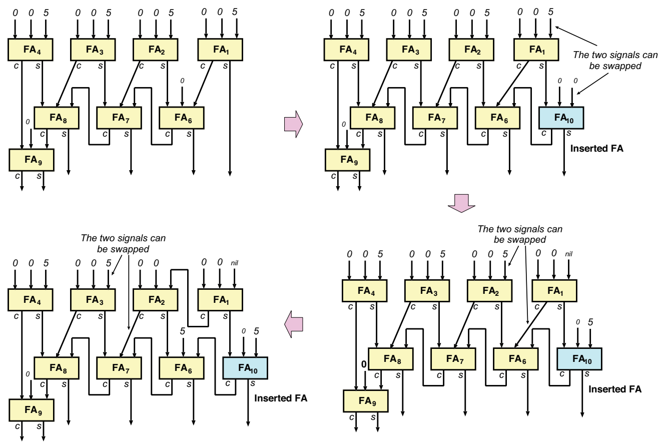

Fig. 5. In case of skewed arrival delays of inputs, the use of HAs instead of FAs can lead to better results. This figure illustrates how this can be achieved using our algorithm.

Checking the first property is trivial: it suffices to remove all carry edges from the graph; the two FAs in question are symmetric if and only if they remain connected. The second condition requires a reverse topological traversal of the FA network. We must determine which signals of rank $r+1$ are symmetric before we can determine the symmetry of signals of rank r. This is because the ranks of carry signals are one higher than those of the input signals. Hence, the symmetry information about the signals of rank $r+1$ can be used to check the symmetry of carries of two FAs of rank r. The symmetry of the highest ranked signals can be detected by checking if they are inputs of the same FA.

Referring back to Fig. 3(a), the sum signals of the first two FAs lead to a sum output, Z. This satisfies the first criterion for symmetry. Since the carry outputs of these two FAs, signals X and Y are inputs of a third FA, the second criterion is satisfied as well. Hence, the input signals a, b, c, d, and e are symmetric.

B. Exploiting Symmetry

Once we have the groups of symmetric signals, we swap them in order to reduce the critical path delay of the circuit. Next, we consider all signal pairs which can be swapped, and choose the one which maximizes the reduction in critical path delay. The process is continued until no further reduction in delay can be obtained by swapping any two symmetric signals.

Fig. 4 shows an example of how symmetry can be used to convert a 3-input adder tree, comprised of two CPA layers, into a carry-save adder followed by a CPA. Consider the inputs of FA_4 and FA_8, which also have the same rank. It is straightforward to verify that these signals are symmetric. The assignment of signals within the symmetric sets can be safely swapped and any input to FA_4 can connect to FA_8. In the example, we swap the input connected to FA_8 with the carry-output of FA_3 connected to FA_4. Proceeding similarly, this converts the first ripple-carry adder into a carry-save adder and eliminates FA_5 (two of its inputs are set to 0), and shortens the critical path by the delay of an FA.

C. Introduction of Half Adders for Delay Improvement

The use of HAs instead of FAs may be preferable at certain points in the circuit as it may help in moving an input with extremely large arrival delay from the compressor tree to the final adder, and thus making the circuit more balanced.

Our approach processes sets of symmetric signals in increasing order of rank. Let r be the current rank. We introduce a *dummy* FA FA_r with two inputs set to zero and the third connected to a circuit output bit of rank r. Next, we look for an FA, denoted FA_{r+1}, that has at least one input bit set to zero, i.e., technically an HA. If such an FA exists, the carry out signal of FA_r replaces the zero input of FA_{r+1}.

If no such FA exists, then we instantiate a new one whose inputs are: the constant value 0, the carry-out signal of FA_r, and the circuit output of rank $r+1$. The sum output of this new FA will have rank $r+1$ and its carry-out will have rank $r+2$. We repeat the process for the new FA until either the carry out signal is absorbed into an existing FA, or there is no signal of higher rank than the rank of inserted FA.

Next, the symmetric set analysis from the preceding section is performed again, followed by the permutation of commutative sets of wires; the circuit delay is computed. If the critical path delay improves, then the introduction of HAs at rank r is beneficial; if not, the HA is removed and the process restarts at rank $r+1$. The process is explained on an example in Fig. 5.

D. Delay Estimation Model

In this section we present an efficient delay estimation method. A circuit made of FAs contains carry chains which should be replaced by faster adders. Hence, the critical path delay calculation must consider this. In our approach we recognize instances of carry chains in the circuit, precompute the delays of faster adders of various bitwidths, and store the results in a table. The delay is then distributed uniformly along all of the FAs in the carry chain and critical path delay is computed.

IV. Experimental Results

We have implemented our algorithm in C++ and it takes as an input either integers to add and their arrival times or a circuit constructed as a network of FAs and HAs. Each benchmark is synthesized using three distinct approaches. First, the tool generates a VHDL implementation of the original circuit. Second, we generate VHDL code using the TGA method [1].

TABLE I
OPTIMIZATION RESULTS FOR ALL OUR BENCHMARKS.

Sequential Adder Tree		
Naive Implementation	$45947.6\mu m^2$	$3.92ns$
Three Greedy Approach	$47597.5\mu m^2$	$1.46ns$
Our Algorithm	$51512.6\mu m^2$	$1.42ns$
Parallel Adder Tree		
Naive Implementation	$46237.0\mu m^2$	$1.87ns$
Three Greedy Approach	$47597.5\mu m^2$	$1.46ns$
Our Algorithm	$46519.7\mu m^2$	$1.44ns$
Multi-Input Addition with Skewed Arrival Delays		
Naive Implementation	$70064.0\mu m^2$	$3.76ns$
Three Greedy Approach	$66970.4\mu m^2$	$3.75ns$
Our Algorithm	$70207.0\mu m^2$	$3.62ns$
16×16 **Shift-and-Add Multiplier**		
Naive Implementation	$62310.1\mu m^2$	$2.24ns$
Three Greedy Approach	$45182.5\mu m^2$	$1.77ns$
Our Algorithm	$56921.4\mu m^2$	$1.72ns$
Multiplication Accumulator		
Naive Implementation	$67795.4\mu m^2$	$2.43ns$
Three Greedy Approach	$48778.3\mu m^2$	$2.03ns$
Our Algorithm	$54479.8\mu m^2$	$1.74ns$
Multi-Input Addition with Non-Uniform Delays of Bits		
Naive Implementation	$18494.8\mu m^2$	$1.65ns$
Three Greedy Approach	$17876.1\mu m^2$	$1.64ns$
Our Algorithm	$17553.4\mu m^2$	$1.48ns$
Multi-Input Addition Using Wallace Tree		
Naive Implementation	$54080.6\mu m^2$	$1.57ns$
Three Greedy Approach	$57234.0\mu m^2$	$1.49ns$
Our Algorithm	$53066.1\mu m^2$	$1.49ns$

Lastly, we apply the circuit optimization method presented in this paper. The circuits are then synthesized for UMC $0.18\mu m$ CMOS.

Table I summarizes the results of our experiments. The first two benchmarks add eight 32-bit integers. The first uses a sequential chain of adders, i.e., $((A_1+A_2)+A_3)+A_4+\ldots$; in the second case, the input integers are added using a balanced adder tree, i.e., $((A_1+A_2)+(A_3+A_4))+\ldots$. Although the two circuits have the same functionality, their delays of the naive implementations are significantly different. The TGA produces the same circuit in both cases, as the basic functionality is the same (multi-input addition). The TGA, of course, requires prior knowledge that the desired circuit is a multi-input adder; which may not be available to the synthesizer. Our algorithm, in contrast, accepted the circuit described as a network of full adders, i.e., the knowledge about the functionality of the circuit was absent. Note that the critical path delays of the two circuits generated by our algorithm are almost the same. This means that our approach is robust to the input representation of the circuit.

The third benchmark adds three 32-bit integers with skewed arrival times. This benchmark is similar to the example shown in Fig. 1. The naive implementation uses two ripple carry adders; the TGA replaces the first ripple carry adder with a carry-save adder instead. Our approach successfully reverts back to an adder tree (with fast adders) by the introduction of HAs as described in Section III-C.

The fourth benchmark is a 16×16 multiplier built from shifts and adds, which is typical of a software implementation of a multiplier. The naive implementation uses sequential addition. However, our approach is able to infer the use of compressor tree resulting in an implementation similar to the one produced by TGA.

The fifth benchmark is a multiply-accumulator circuit that computes $A \times B + C$, where A and B are 16-bit inputs and C is 32-bits. Both the naive implementation and TGA explicitly compute $A \times B$, including the final CPA, and then adds the result to C, although TGA employs an optimal implementation of the multiplier's partial product reduction tree. Our optimization strategy, in contrast, forms a single compressor tree that adds the partial product of A and B along with C, and one final CPA, rather than two.

The sixth benchmark is the example shown in Fig. 2, where the best implementation is to use two compressor trees, rather than one, in order to account for the variation in arrival time of the different bits. The naive implementation uses one adder tree, while the TGA builds one compressor tree; our approach, in contrast, finds the optimal implementation.

The last benchmark adds nine 32-bit integers. The naive implementation is a Wallace tree, while the TGA produces an optimal compressor tree. Our algorithm transforms the tree into a solution whose delay is also optimal.

V. Conclusions

To summarize, the arithmetic-oriented logic synthesis techniques proposed in this paper found delay-optimal circuits for all seven benchmarks. The TGA only found a delay optimal solution for the last benchmark, although it was near-optimal for several others. Moreover, the TGA requires prior knowledge that the circuit to optimize is a multi-input adder or a multiplier's partial product reduction tree, whereas, our method does not. Additionally, our method retains the benefit that it can automatically convert adder tree structures to compressor trees, when appropriate; at the same time, it can leave adder trees in place, or split a design into several compressor trees when disparities between the arrival times of the input bits favor such a structure.

References

[1] P. F. Stelling, C. U. Martel, V. G. Oklobdzija, and R. Ravi, "Optimal circuits for parallel multipliers," *IEEE Transactions on Computers*, vol. C-47, no. 3, pp. 273–85, Mar. 1998.

[2] 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*, vol. C-45, no. 3, pp. 294–306, Mar. 1996.

[3] C. S. Wallace, "A suggestion for a fast multiplier," *IEEE Transactions on Electronic Computers*, vol. C-13, no. 2, pp. 14–17, Feb. 1964.

[4] L. Dadda, "Some schemes for parallel multipliers," *Alta Frequenza*, vol. XXXIV, pp. 349–56, 1965.

[5] P. F. Stelling and V. G. Oklobdzija, "Design strategies for optimal hybrid final adders in a parallel multiplier," *Journal of VLSI Signal Processing*, vol. 14, pp. 321–31, Dec. 1996.

[6] P. Ienne and R. Leupers, Eds., *Customizable Embedded Processors—Design Technologies and Applications*, ser. Systems on Silicon Series. San Mateo, Calif.: Morgan Kaufmann, 2006.

[7] 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*, vol. CAD-27, no. 10, pp. 1761–74, Oct. 2008.

[8] J. Um and T. Kim, "An optimal allocation of carry-save-adders in arithmetic circuits," *IEEE Transactions on Computers*, vol. C-50, no. 3, pp. 215–33, Mar. 2001.