Arithmetic Optimization using Carry-Save-Adders

Taewhan Kim[†] William Jao[‡] Steve Tjiang[†]

[†]Synopsys Inc.
700 E. Middlefield Rd.
Mountain View, CA 94043

[‡]Aimfast Corp.
846 Stewart Dr.
Sunnyvale, CA 94086

Abstract

Carry-save-adder(CSA) is the most often used type of operation in implementing a fast computation of arithmetics of register-transfer level design in industry. This paper establishes a relationship between the properties of arithmetic computations and several optimizing transformations using CSAs to derive consistently better qualities of results than those of manual implementations. In particular, we introduce two important concepts, *operation-duplication* and *operation-split*, which are the main driving techniques of our algorithm for achieving an extensive utilization of CSAs. Experimental results from a set of typical arithmetic computations found in industry designs indicate that automating CSA optimization with our algorithm produces designs with significantly faster timing and less area.

1 Introduction

Hardware designers have long applied many arithmetic optimization techniques for implementation of arithmetic functionality like addition, subtraction, and multiplication. Among them, carry-save-adder(CSA)[1] has proved a powerful mechanism to improve timing with little, if any, area penalty (even reduced area). Figure 1(a) shows the structure of an n-bit CSA.

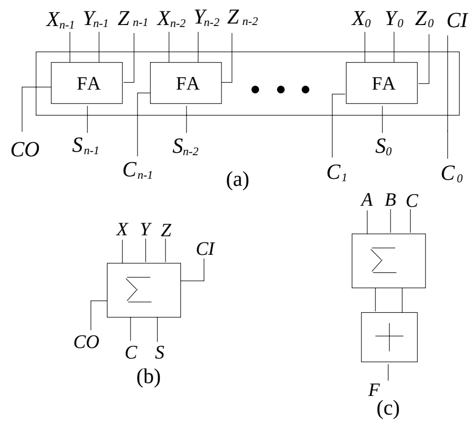


Figure 1: An n-bit CSA and an example of use

The n-bit CSA consists of n disjoint full adders(FAs). It consumes three n-bit input vectors and produces two outputs, i.e., n-bit sum vector S and n-bit carry vector C. We use the block symbol in Figure 1(b) to represent a CSA operation. Unlike the normal adders (e.g., ripple-carry adder(RCA) and carry-lookahead adder(CLA)), a CSA contains no carry propagation. Consequently, the CSA has the same propagation delay as only one FA delay (compared to RCA's n FA delay where n is the bit-width), and the delay is constant for any value of n. For sufficiently large n, the CSA implementation becomes much faster and also relatively smaller in size than the implementation of normal adders.[1]

There has been an extensive research work on the arithmetic optimizations in several areas[2, 3, 4]. They, however, focused on the transformation of operations using techniques such as simple algebraic manipulations and constant propagation; they did not address the problem of arithmetic optimization using CSAs. This paper introduces the concept of CSA transformation and presents an algorithm that effectively utilizes CSAs to derive consistently better quality of results than that of manual implementations.

We define a *CSA tree* to be a tree of CSA operators and one adder at the root of the tree. A CSA tree can be used to transform an arbitrary number of additions to produce two adding operands and the adder is used at the root of CSA tree to produce a final sum. In other words, an expression of N additions can be transformed to a CSA tree of depth $log_{1.5}(N)$ and the overall delay is $log_{1.5}(N)$ plus the delay of final adder (which is about $log_2(n)$ for a CLA). For example, expression $A+B+C$ can be transformed into one CSA and one adder as shown in Figure 1(c).

Based on several results of our experimentations, our transformation can be stated as follows: Given a non-cyclic dataflow graph of arithmetic operations, we wish to transform the computations using as many CSA operations as possible while preserving the functionality of the design.

[1]Note that a CSA performs the same functionality of the conventional adders in the sense that each reduces the number of adding operands by one, i.e., a CSA reduces from 3 to 2 and an adder from 2 to 1.

1-58113-049-x-98/0006/$3.50 DAC98 - 06/98 San Francisco, CA USA

2 Applicability of Transformation

CSA transformation is not limited to addition only. We can transform other arithmetic operations like subtraction and multiplication into additions to produce longer chains of additions. We replace a subtraction by adding the negation[2] of the subtraction. That is, (x-y) in expression is changed into ($x + \bar{y} + 1$). For multiplication, we can use two possible options:

- **sum of products:**

  A multiplication is decomposed into a set of shift-and-add operations, which is definitely beneficial when it has a constant as input.[3] With full decomposition we can effectively extend the addition tree by merging the decomposed additions with another additions. However, decomposition can increase the number of CSA operations drastically as the bit-width increases.

- **partial multiplication and addition:**

  Two types of implementation for multiplication operation are typically found in designs[6]: (i) wallace tree model for fast timing and (ii) carry-save array model for small area. The wallace tree model consists of two blocks called *partial-mult*[4] and *final-add*. Partial-mult uses the two inputs of the multiplication as input and produces two outputs, in which by adding them the final result of multiplication is obtained.[5] Consequently, by decomposing into a partial-mult and a final-add we can merge the final-add with a descendent operation to transform into CSAs. This option increases only one operation, but is less flexible than the case of full decomposition.

3 An Algorithm

Our transformation algorithm consists of three major steps: (1) identification of operation tree to be transformed, (2) translation of the expression of the identified tree into an addition expression, and (3) conversion of the addition expression into a CSA tree. Given a non-cycle dataflow graph of arithmetic computations, our algorithm iteratively performs the three steps until there is no candidate expression of tree to be transformed. The following subsections describe the details of the three steps.

3.1 Candidate Identification

As explained in Sec. 2, the CSA transformation can be applied to any type of operations that can be converted to additions. A *candidate cluster* is basically a tree which contains operations like additions/subtractions/multiplications. Our algorithm is a bottom-up (from the output boundary of design toward the input boundary) approach. From the outputs of design it finds an operation which has not been transformed yet in the previous iterations. We refer the operation to *root*. The root is then expanded toward the input boundary of design to construct a tree of operations. Note that the type of root must be one of addition, subtraction, and multiplication operations.

Suppose that operation A is a leaf of the operation tree expanded so far and B is an operation in which its output is used as an input of operation A. We expand the current operation tree by including B if the following four conditions are satisfied:

- Condition 1: A must be one of addition and subtraction operations.
- Condition 2: B must be one of addition, subtraction, and multiplication operations.
- Condition 3: Output of B must be single fanout. That is, B drives only an input of A.
- Condition 4: There should be no "leak" of data values through the connections from B to A up-to the output bit-width of root of the tree.

Condition 1 ensures that only addition and subtraction are used as non-leaf operations of the tree, and Condition 2 ensures that only addition, subtraction and multiplication are used as leaf operations. Condition 3 ensures that we do not allow transforming non-tree structure of operations. (We solve the multiple fanout case in Sec. 4.1.) Finally, Condition 4 is required for preserving the functionality of design before and after transformation. The examples shown in Figure 2 clarify the concept of leakage of data values. Suppose operation A is also the root of the current tree. Figure 2(a) shows a case of upper-bit truncation between B and A. Because up-to 8 bits of data values must be preserved, the truncation does not allow merging operation B with A. Similarly, Figure 2(b) shows a case of lower-bit truncation between the two operations. It also prevents merging the two operations. However, Figure 2(c) shows a case of preserved data values up-to 8 bits from B through A. Consequently, the two operation can be merged and transformed into CSA operations without changing the functionality of design.

3.2 Conversion to Additions

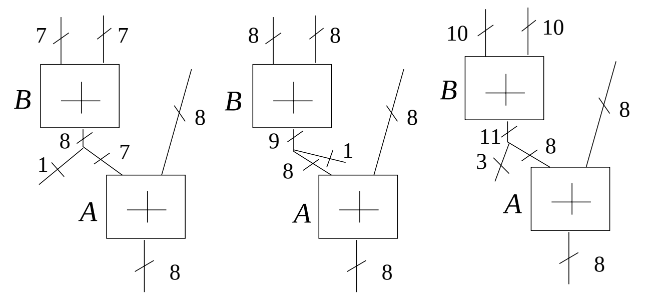
With the identified cluster of operation tree obtained from Step 1, this step converts the expression of operation tree consisting of additions, subtractions, and multiplications into a tree of additions. What we are interested in this step is to extract all the operands from the converted addition expression. Table 1 summarizes all possible conversion rules for basic operations. Note that there exist three different rules to convert a multiplication. Which rule to apply depends on constraints such as number of operations to allow, word size of operation, and optimization goals. For example, rule 6 (full decomposition) decreases timing, but increases the number of CSAs drastically. On the other hands, rule 5 (partial decomposition) is not much effective to decrease timing than that of rule 6, but maintains the number of CSAs in a certain

[2] We used two's complement.

[3] We used the signed-digit(SD) encoding scheme[5] for the multiplication with a constant input to reduce the number of decomposed operations.

[4] It is a functionality of wallace tree compressor.

[5] We use *S(partial_mult(A,B))* and *C(partial_mult(A,B))* to denote the two outputs of the partial-mult of operation $A * B$.



(a) upper-bit truncation (b) lower-bit trunction (c) preserved bit-flow upto 8 bits

Figure 2: Example of possible net-connections between two operations

	primitive exp.	new exp.
1	$x + y$	$x + y$
2	$x + y$ with c_{in}	$x + y + c_{in}$
3	x - y	x + (-y)
4	x - y with c_{in}	x + (-y) + c_{in}
5	x * y	$C(partial\_mult(x,y))$ + $S(partial\_mult(x,y))$
6	x * y	$\sum_{i=0}^{n-1} AND(x, y_i) << i$
7	x * $const$	summing from SD encoding[5]

Table 1: Conversion rules to additions for primitive expressions

amount. In addition, rule 7 can be used as a preprocessing step which guarantees that the number of operations is always within half of bit-size of the constant.

We apply the rules in Table 1 to the original expression of tree to obtain a set of operands, some of which might need to negated (e.g. -y in subtraction) or shifted (e.g. $AND(x, y_i) << i$ in multiplication). The operands may also include single-bit ones and constants. We then replace negated operands, e.g., -x by $\bar{x}$+1 and all constants are folded.

3.3 Conversion to CSAs

All the operands obtained from Step 2 are to be summed up by CSA tree. In this step, we build up the tree. Our algorithm constructs CSAs iteratively one by one. Each iteration selects three operands as input of a new CSA and two new operands are created from the CSA. At the next iteration, the three operands used in the previous iteration are removed from the operand set and the new two operands are added to the set. Therefore, whenever one CSA is created the size of operand set is reduced by one. CSA construction proceeds until only two operands in the set remain. Finally, a normal adder adds the two remaining operands to produce a final output of the expression.

Because the primary objective of our transformation is to reduce timing, each iteration selects, from all the possible triples of operands in the set, the triple with the earlist arriving time. We use an efficient linear-time heuristic: Initially we sort operands in the set in a non-decreasing order according to the delay to the operands from input boundary. If two operands have the same delay, we give a higher priority to the one of smaller bit-width. At each iteration, our algorithm picks the first three operands from the sorted list and creates a CSA. The delays to two output operands of CSA are then computed and the operands are inserted to the correct positions of the sorted list. Our experimentation indicates that our operand selection approach is very efficient and yet, does not degrade the quality of results when compared with that of the exhaustive search of selections. In addition, the algorithm can easily tune to optimize area of the resultant CSA tree by sorting the operands according to the size of bit-width of operands. (If two operands have the same bit-width we resolve it by giving higher priority to the one of shorter delay.)

We handle one-bit operands and a constant in different ways to reduce the number of CSAs created. This can be achieved by utilizing as many carry inputs of CSAs as possible. Each CSA created can consume an one-bit operand as carry input. Therefore, it is desirable to assign as many one-bit operands to the carry inputs of CSAs as possible because otherwise, each requires one additional CSA. On the other hand, for the constant operand it can be used as a multi-bit operand or be decomposed into logic-1 values to be used as carry input of CSAs. Using constant as a multi-bit operand requires only one additional CSA. However, when there are sufficiently large number of CSAs for the multi-bit (non-constant) operands, the constant can be assigned to the carry inputs of CSAs, avoiding unnecessary creation of an additional CSA. Furthermore, when there is a mixture of operands of (non-constant) single-bit and logic-1 values, because logic-1 operand takes zero delay, it is desirable to assign the logic-1 operands to CSAs which are far from the root of final CSA tree, and to assign the non-constant operands to CSAs which are close to the root of tree to reduce the timing of the critical path of the final CSA tree.

Figure 3 illustrates the effects of different utilization of carry inputs of CSAs on the timing and area of final CSA tree. Suppose that A, B, C, D are mult-bit operands, E is single-bit, and the constant value is 2. We assume that the bit sizes of the multi-bit operands are the same. Also, we assume we have already computed delay to the operands as shown in Figure 3(a).[6] Figure 3 shows a set of possible transformations for the operands. Figure 3(a) shows the case that the single-bit operand E and constant 2 are not used as carry input of CSA. Consequently, four CSAs are created and timing of the final tree is worse than the other cases. On the other hand, Figures 3(b) and 3(c) show the cases that the single-bit operand and constant are used as carry input of CSA but not both, respectively. Each of them requires three CSAs and timing of the CSA trees is better than the previous one, but still there is a room to improve. Figures 3(d) and 3(e)

[6]We use notation $D(X)$ to denote the delay of operand X.

show the cases that the carry inputs of CSA tree are maximally utilized. However, according to the way of assigning single-bit operand and constant there is a big difference in timing. One guideline is that we need to assign the constant operands (i.e., logic-1) to earlier steps than the single-bit ones during the top down process of CSA tree construction. In fact, our algorithm produced the one in Figure 3(e) which is the best in terms of timing and area.

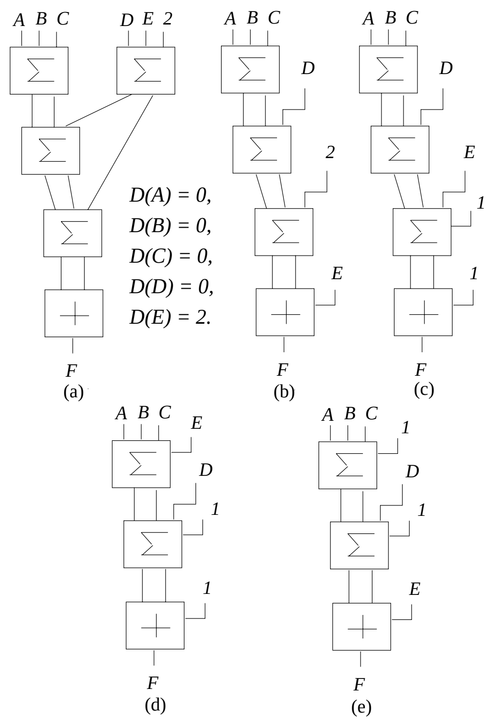


Figure 3: Effects of utilization of carry inputs on CSA tree

4 Extension to the Applicability of CSA Transformation

Many signal processings and data-intensive computations frequently contain an operation in which only the upper m bits among n bits of its output are fed input to a descendent operation. We call this partial use of an output the *lower-bit truncation* problem. Moreover, many designs often contain an operation whose output feeds several other operations. We call this the *multiple fanout* problem. In fact, the two problems correspond to the violations of Condition 4 (Figure 2(b)) and Condition 3 in Sec. 3.1, respectively. In the following, we provide solutions to the problems.

4.1 A Solution to the Multiple Fanout Problem

We begin with an example to demonstrate how our approach handles operations with multiple fanout. Suppose that Figure 4(a) is a partial structure of a design that we are going to transform into CSAs. Note that operation O_3 has multiple fanout. According to the algorithm described in Sec. 3, the first iteration will identify operation tree $tree_1$ and transform it into CSAs, and the second iteration will identify $tree_2$ and transform it into CSAs as show in Figure 4(b). Consequently, two CSAs and two adders (one for each CSA tree) are used. Suppose that a design contains a chain of operations in which n of them have multiple fanout. Then, our algorithm will identify n operation trees having each operation with multiple fanout as root. Consequently, n adders will be allocated on the chain of the transformed CSA trees. Here, our objective is that we want to reduce the number of adders from n to 1 and replace the remaining n-1 adders with CSAs. We accomplish this by introducing the concept of *CSA transformation without final adder*.

We use the algorithm in Sec. 3 for identifying expression tree. However, when an operation with multiple fanout is encountered (i.e., Condition 3), we mark the operation as a root of another operation tree and continue to expand the operation tree. Once we collect all the operation trees by crossing over operations with multiple fanout, we topologically sort the operation trees from input boundary of design to output boundary. We then transform the operation trees on the list one by one. For those trees with multiple fanout root, we do not allocate final adders in the resultant CSA trees. Instead, we generate final two outputs of each CSA tree. The two outputs are then used in two ways: (1) Both of them are used as input operand of the operations trees which depend on the operation tree corresponding to the transformed CSA tree without final adder and (2) A new adder is created and they are used as input of the adder. (we refer this process to as *operation-duplication*.) The output of adder is then used for the fanout of the root of operation tree corresponding to the CSA tree. Consequently, only one adder will be created at the end of the last operation tree of the sorted list of trees.

For example, in Figure 4(a) the first iteration will find two operation trees $tree_1$ and $tree_2$ in which there is a data flow from $tree_2$ to $tree_1$. $tree_2$ is then transformed into a CSA tree without final adder. It generates two outputs $T1$ and $T2$ and one new adder O_{dup} as shown in Figure 4(c). $tree_1$ is then transformed into a CSA tree with final adder by using $T1$, $T2$, D, and E as operands as shown in Figure 4(d). Note that the resultant transformation contains three CSAs and one adder in the two CSA trees and generates a faster timing than the one in Figure 4(b). However, the total area increases because one additional adder is created outside of the CSA trees.

4.2 A Solution to the Lower-bit Truncation Problem

We also begin with an example to demonstrate how our approach can solve the problem of lower-bit truncation. Figure 5(a) shows an example of lower-bit truncation. Note that only 8 upper-bits of the output of operation O_2 are used. Therefore, we cannot merge operation O_1 and O_2 into CSAs because there is a data leakage between the operations(i.e., Condition 4 in Sec. 3.1). We solve this problem by introducing the concept of *operation-split*: We split operation O_2 into two operations O_3 and O_4 as shown in Figure 5(b). This means that given the split operations our algorithm can identify an expression tree as

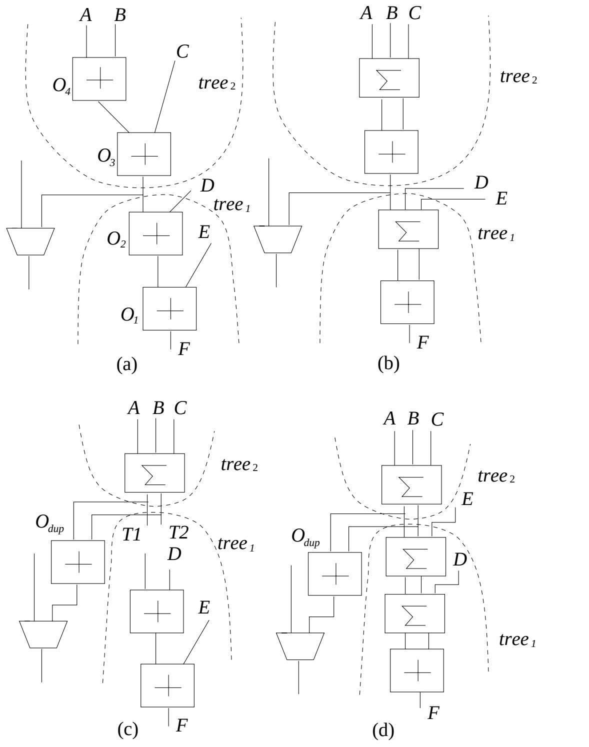


Figure 4: Examples of transformation of operation with multiple fanout

shown in Figure 5(c) where C, upper 8-bits of A and B, and carry-out of operation O_4 are the operands to be added. Figure 5(d) shows the transformed CSA tree from the operation tree in Figure 5(c).

Unlike the case of solution of multiple fanout problem, the operation-split does not increase area. Furthermore, when the operation with lower-bit truncation has multiple fanout, the carry-out of lower one of two split operations can be extracted directly from the adder created from the solution of the multiple fanout problem.

5 Experimental Results

We tested our algorithm on a large number of arithmetic computations which are typically used in industrial designs. We used Design Compiler package from Synopsys Inc. to perform the implementation selection, tree-height minimization and logic optimizations for the designs transformed by our algorithm and the designs without using our algorithm[7], and compared their results.

- **Designs with additions, subtractions, and multiplications:**

  We tested our algorithm on designs with a mixture of additions, subtractions, and multiplications as shown in Table 2. We used 8-bit operands for non-constant inputs of multiplication and used 16-bit operands for the rest. We also assume that the arrival times of all input operands are 0. For multiplication operation, our algorithm fully decomposed it into additions. The results in Table 2 show a strong indication that our algorithm can extensively apply to the designs with additions, subtractions and multiplications.

[7]The tree-height minimization was performed on non-CSA operations. We used *lcbg10pv* (0.35μ) technology[7] for all test cases.

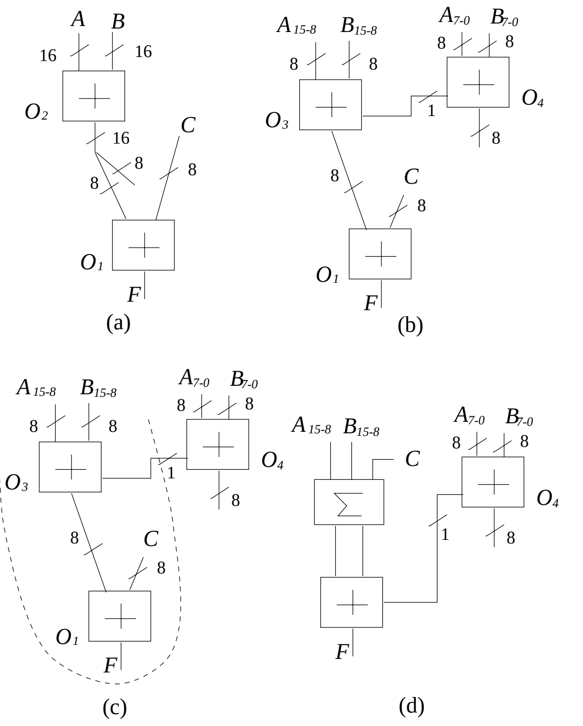


Figure 5: An example of transformation of operation with lower-bit truncation

- **Designs with multiple fanout:**

  We conducted our experimentation on three designs shown in Figure 6. (In all designs in the following, we assume that the arrival times of all input operands are 0.) Each design has operations with multiple fanout. We used two CSA transformation techniques: (i) without operation-duplication and (ii) with operation-duplication, and compared the results with that produced without CSA transformation. The results are summarized in Table 3. Note that operation-duplication can reduce timing of design much further, but it increases area. Consequently, the transformation with operation-duplication can be applied in a local way to those operations on the critical path of design to reduce timing with a minimal increase of area.

- **Designs with lower-bit truncation:**

  We also conducted our experimentation on the three designs shown in Figure 7. Note that the designs have operations with lower-bit truncation.

We used two CSA transformation techniques: (i) without operation-split and (ii) with operation-split, and compared the results with that produced without CSA transformation. The results shown in Table 4 reflect that the operation-split is a very powerful technique to overcome the limitation of lower-bit truncation problem and extend the applicability of CSA transformation to produce much faster timing and less area.

Equations	RTL Design timing/area	CSA design timing/area	diff.
A * B + 1	6.18 ns 3430 units	5.88 ns 3410 units	-5% -1%
A * 3E3E (0011111000111110)	6.86 ns 2920 units	6.55 ns 2068 units	-5% -29%
A*B + C*D + E*F	7.21 ns 11225 units	6.84 ns 10141 units	-37% -10%
A*B - C + D	7.19 ns 5185 units	6.37 ns 4880 units	-10% -6%
A + B - C - D - E - F	7.94 ns 5328 units	6.03 ns 3697 units	-24% -31%

Table 2: Comparison of results for expressions with addition, subtraction and multiplication

Designs	RTL Design timing/area	CSA design w/o duplicate timing/area	CSA Design w/ duplicate timing/area
fanout_1	7.11 ns 4010 units	6.83 ns 3600 units	6.24 ns 4659 units
fanout_2	7.96 ns 6328 units	7.13 ns 6072 units	7.04 ns 7037 units
fanout_3	7.86 ns 9344 units	7.07 ns 8992 units	6.87 ns 9772 units

Table 3: Comparison of results for designs in Figure 6

Designs	RTL Design timing/area	CSA design w/o split timing/area	CSA Design w/ split timing/area
truncate_1	6.01 ns 2315 units	5.94 ns 2138 units	4.02 ns 1721 units
truncate_2	7.75 ns 4610 units	7.25 ns 4560 units	5.81 ns 4133 units
truncate_3	7.98 ns 8374 units	7.25 ns 8161 units	6.45 ns 7405 units

Table 4: Comparison of results for designs in Figure 7

6 Conclusions

We presented a new technique to optimize arithmetic circuits. The technique automatically expands circuits consisting of adders, subtractors, and multipliers into their carry-save-adder (CSA) representation which are then optimized. The representation obviates the need for implementation selection, the automatic selection of the best implementation among several adder/subtractor/multiplier implementations, a time-consuming part of arithmetic optimization.

Acknowledgment- We wish to thank Ron Miller, Hazem Almusa, Reiner Genevriere, Sean Huang, and Tai Ly for making this work successful.

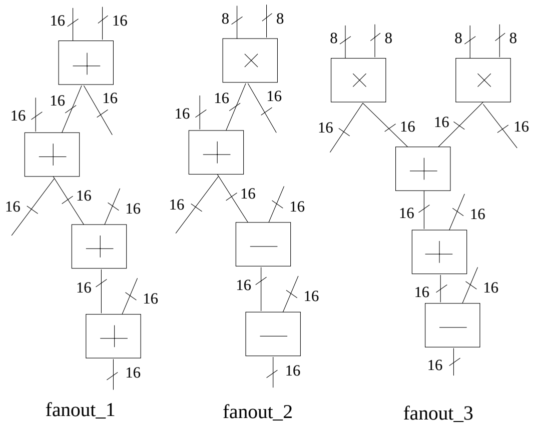


Figure 6: Designs with multiple fanout

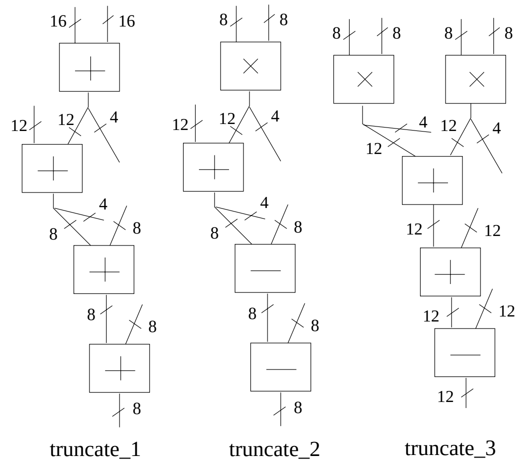


Figure 7: Designs with lower-bit truncation

References

[1] N. Weste and K. Eshraghian, *Principles of CMOS VLSI Design - A Systems Perspective*, Addition-Wesley Publishers, 1985.

[2] M. Potkonjak and J. Rabaey, "Optimizing Resource Utilization Using Transformations", *Proc. of ICCAD*, pp. 88-91, 1991.

[3] D. Lobo and B. Pangrle, "Redundant Operation Creation: A Scheduling Optimization Technique", *Proc. of DAC*, pp. 775-778, 1991.

[4] A. Aho and J. Ullman, *Principles of Compiler Design*, Addition-Wesley Publishers, 1977.

[5] K. Hwang, *Computer Arithmetic: Principles, architecture, and Design*, New York, 1979.

[6] Synopsys Inc., *DesignWare Components Databook*, 1996.

[7] LSI Logic Inc., *G10-p Cell-Based ASIC Products Databook*, 1996.