Optimal Allocation of Carry-Save-Adders in Arithmetic Optimization

Junhyung Um Taewhan Kim
Dept. of Computer Science and Advanced Information Technology Research Center (AITrc)
Korea Adv. Institute of Science & Technology
Taejon, Korea

C. L. Liu
Dept. of Computer Science
National Tsing Hua Univ.
Hsinchu, Taiwan R.O.C

Abstract: Carry-save-adder(CSA) is one of the most widely used schemes for fast arithmetic in industry. This paper provides a solution to the problem of finding an optimal-timing allocation of CSAs. Specifically, we present a polynomial time algorithm which finds an *optimal-timing CSA allocation* for a given arithmetic expression. In addition, we extend our result for CSA allocation to the problem of optimizing arithmetic expressions *across the boundary of design hierarchy* by introducing a new concept, called *auxiliary ports*. Our algorithm can be used to carry out the CSA allocation step optimally and automatically, and this can be done within the context of a standard HDL synthesis environment.

1 Introduction

Timing is one of the most important design criteria to be optimized in several phases of the synthesis process. We study in this paper the timing optimization problem of arithmetic circuits using carry-save-adders(CSA)[1] as a key implementation of operations in RTL synthesis.

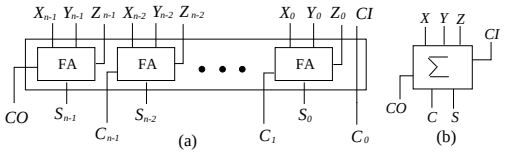

Figure 1: The structure of an n-bit CSA

Figure 1(a) shows the structure of an n-bit CSA. An n-bit CSA consists of n disjoint full adders(FAs). It has as input three n-bit input vectors X, Y, Z and produce two output vectors, an n-bit sum vector S and an n-bit carry vector C. We use the block symbol in Figure 1(b) to represent a CSA. The details on the CSA structure can be found in [2, 6]. Note that the CSA allocation scheme is not limited to addition only. We can convert other arithmetic operations like subtraction and multiplication into additions to be optimized by CSAs.[2]

In this paper, we provide a set of results on the optimality of the CSA allocation problem introduced by [2], from which we develop an efficient optimal algorithm for the CSA allocation. The optimality is then extended to solve the problem of a wide application of CSAs for arithmetic expressions across the design boundary.

2 An Algorithm for CSA Allocation

2.1 Delay Model for CSA

Because an n-bit CSA consists of n "disjoint" full adders, we can use a *constant delay model* for CSA: Let D_c denote the (longest) delay from the three input ports to the carry-out port, and D_s denote (longest) delay from the three input ports to the sum output port of a full adder.

2.2 Optimality of CSA Allocation

The CSA allocation problem can be stated as:

Given an arithmetic expression

$$f = x_1 + \cdots + x_m + s_1 + \cdots + s_n + c \quad (1)$$

where $x_i, \cdots x_m$ are multi-bit inputs, $s_1, \cdots s_n$ are single-bit inputs and c is a constant, and the arrival times of the inputs, for a constant delay model for CSA (defined in Sec. 2.1), determine an allocation of CSA structure for f such that the output arrival time is minimum.

Let us first consider the case that the constant c is 0. We divide the problem into two sub-problems: (i) $m - 1 \geq n$. In this case the number of multi-bit addends is large enough so that all single-bit addends can be used as carry inputs of the allocated CSAs[1]; (ii) $m - 1 < n$. In this case some of the single-bit addends shall be used as normal inputs to CSAs.

Algorithm 1 *CSA allocation for f in Eq.(1) when $c = 0$ and $m - 1 \geq n$:*

[1]The minimum number of CSAs required to complete the sum of m multi-bit addends is $m - 2$. This means that at least $m - 1$ carry-input ports (including the one on the final adder) are available to the single-bit addends.

0-7803-5832-X /99/$10.00 ©1999 IEEE.

- Set $\mathcal{M} = \{x_1, \cdots, x_m\}$ and $\mathcal{S} = \{s_1, \cdots, s_n, 0, \cdots, 0\}$ such that $|\mathcal{M}| - 1 = |\mathcal{S}|$;
- Sort the addends in $\mathcal{M}$ and $\mathcal{S}$ by their arrival times in non-decreasing order;

while ($|\mathcal{M}| \geq 3$**)**
- Select the first three addends from $\mathcal{M}$;
- Create a new CSA and connect them as inputs;
- Select the first addend from $\mathcal{S}$;
- Connect it (if not 0) as carry input;
- Insert the two outputs of the CSA to $\mathcal{M}$;

endwhile

/* Here, we make sure that $|\mathcal{M}| = 2$ and $|\mathcal{S}| = 1$ */
- Create a final adder;
- Connect the two addends in $\mathcal{M}$ as normal input;
- Connect the addend in $\mathcal{S}$ (if not 0) as carry input;

Our CSA allocation algorithm is similar to the Huffman's algorithm[3] for constructing a binary tree with minimum (weighted) path length. However, the CSA allocation problem is fundamentally different from minimum (weighted) path length problem and the more general problem with a class of combination functions (i.e., *quasi-linear function*)[4], for which Huffman's algorithm produces an optimal tree under the corresponding tree cost function: (1) In a strict sense, the CSA allocation algorithm constructs a "graph" of CSAs rather than a "tree" because a CSA node produces two outputs, and they may be used as inputs to different CSA nodes, creating a reconvergent path; (2) In addition to the class of multi-bit inputs, the allocation problem has another class of inputs, i.e., a set of single-bit addends with two different cost measures. Consequently, our approach to the proof of the optimality of the CSA allocation is completely different from that of Huffman's. (Due to the space limitation, we omit all details of the proofs here.)

Lemma 1 *Algorithm 1 yields a delay-optimal CSA allocation for subproblem (i).*

Proof We applied induction on the number of inputs m. We analyzed the arrival times of the addends generated with *Algorithm 1* and the ones without *Algorithm 1*, and extracted a set of inequality relations by which we claimed the theorem holds. □

Next, consider subproblem (ii). Because not all the single-bit addends can be used as a carry input to a CSA, some of them shall be used as a normal input to a CSA. Consequently, we should decide which and how many of the single-bit addends are to be used as normal inputs to some of the CSAs. Let k denote the number of single-bit addends to be used as normal inputs to the CSAs. We can easily show that $k \geq \lceil \frac{n-m+1}{2} \rceil$ from the fact that converting one single-bit addend to a multi-bit addend to be used as a normal input to a CSA creates one additional CSA and thus, results in one more carry input ports available to use, and at the same time, one less single-bit addends. Let us consider the problem of selecting k addends from the list of single-bit addends to be use as normal inputs to the CSAs.

Lemma 2 *To select k single-bit addends to be used as normal inputs to the CSAs, the k single-bit addends with the earliest arrival times will be selected. A subsequent application of Algorithm 1 generates a CSA structure whose timing is minimal among all CSA allocations with any possible selections of k single-bit addends (and a subsequent application of Algorithm 1).*

Moreover, we claim that a delay-optimal CSA allocation is obtained when the value of k is chosen to be $k = \lceil \frac{n-m+1}{2} \rceil$. That is,

Lemma 3 *The minimum number of single-bit addends, k, to be used as normal inputs to CSAs for delay-optimal CSA allocation is $\lceil \frac{n-m+1}{2} \rceil$.*

Consequently, from Lemma 3, we can summarize our CSA allocation algorithm for subproblem (ii) as follows:

Algorithm 2 *CSA allocation for f in Eq.(1) when $c = 0$ and $m - 1 < n$:*

- Set $\mathcal{M} = \{x_1, \cdots, x_m\}$ and $\mathcal{S} = \{s_1, \cdots, s_n\}$;
- Sort the addends in $\mathcal{S}$ by arrival times (non-decreasing);
- Move the first $\lceil \frac{n-m+1}{2} \rceil$ addends from $\mathcal{S}$ to $\mathcal{M}$;
- Apply *Algorithm 1*;

Finally, let us consider the case that $c \neq 0$. If it is a negative number, we need to do a sign-extension for the constant. This means that we shall treat it as a multi-bit addend. However, if it is a positive number, there are two choices: (1) breaking c into a form of $1 + 1 + ... + 1 (= c)$ to be used as carry inputs to CSAs, or (2) treating c as one multi-bit addend. (Obviously, using a strategy of mixing options (1) and (2) will produce inferior timings.) If $c > 0$ and $m - 1 \geq c + n$, we can use all single-bit and constant addends as carry inputs to CSAs (i.e., option (1)). This clearly produces a delay-optimal CSA allocation since the timing of CSA structure produced by *Algorithm 1* with option (1) and the timing of CSA structure with $c = 0$ are basically the same. However, if $c > 0$ and $m - 1 < c + n$, we claim that treating c as a multi-bit addend (i.e., option (2)) produces a delay-optimal allocation. In the following, we summarize our procedure for the optimal CSA allocation.

CSA_OPT *optimal CSA allocation for f in Eq.(1)*:

- Set $\mathcal{M} = \{x_1, \cdots, x_m\}$ and $\mathcal{S} = \{s_1, \cdots, s_n\}$;
- If $c < 0$, put it in $\mathcal{M}$ by sign-extension;
- If $c > 0$ and $m - 1 \geq c + n$, break c into $1 + 1 + ... + 1 (= c)$ and put them in $\mathcal{S}$;
- If $c > 0$ and $m - 1 < c + n$, put c in $\mathcal{M}$;
- Apply *Algorithm 2*;

Theorem 1 *CSA_OPT generates a delay-optimal CSA allocation for f in Eq.(1).*

Figure 2 shows an example of CSA allocation using *CSA_OPT*. Let us consider an addition expression which

consists of 2 multi-bit addends A and B and 4 single-bit addends c, d, e and f, and constant 2 as shown in Figure 2(a).[2] We note first the constant and some of single-bit addends must be used as normal inputs to CSAs. In particular, since $m - 1 = 2 - 1 < 2 + 4 = c + n$, we insert constant 2 to the multi-bit list $\mathcal{M}$ as shown at the top of Figure 2(b). According to *Algorithm 2*, lists $\mathcal{M}$ and $\mathcal{S}$ satisfy the inequality relation, $m - 1 = 3 - 1 < 4 = n$. Consequently, $\lceil \frac{n-m+1}{2} \rceil$ (= $(4-3+1)/2 = 1$) addend in $\mathcal{S}$ with the earliest arrival time shall move to list $\mathcal{M}$ as shown at the bottom of Figure 2(b). Finally, we apply the CSA allocation algorithm for the updated $\mathcal{M}$ and $\mathcal{S}$ according to *Algorithm 1*: The first three addends in $\mathcal{M}$ and the first addend in $\mathcal{S}$ are selected and connected to the input ports of a new CSA as shown in Figure 2(c). The subsequent iterations in *Algorithm 1* generate a CSA structure shown in Figure 2(d) in which single-bit e was selected and used as a carry-input to the second CSA because its arrival time is earlier than that of f. Note that the calculations of the arrival times of the two outputs of the CSA (in Figure 2(c)) are slightly different in that the arrival time of the carry-out vector of the CSA is determined by the arrival time of the addend used as carry input to the CSA as well as the times of the addends used as normal inputs to the CSA.

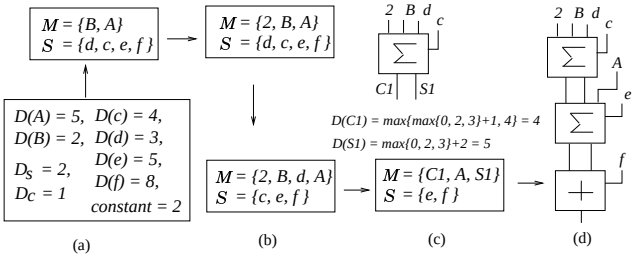

Figure 2: An example of illustrating the flow of *CSA_OPT* for CSA allocation

3 Extension: boundary optimization

For more complex arithmetic expressions, it is quite often and natural to divide an arithmetic expression into multiple designs. We illustrate our motivation of optimization over different designs by using the examples in Figure 3. Figure 3(a) consists of two designs, A and B. Figure 3(b) shows the CSA tree produced by disregarding the boundary of design A and applying the *CSA_OPT* algorithm to the whole design. Consequently, we obtain the best timing, *9.5+D(ADD)*, at the expense of destroying the design boundary. On the other hand, Figure 3(c) shows the CSA tree produced by applying our algorithm to designs A first, and then to B. As a result, the timing of the structure is worse than that of Figure 3(b) since *D(ADD)* $\geq$ *D(FA)* = $max\{D_c, D_s\}$ = *1.5*. This is due to the inability of merging the two operations *op2* and *op3* in different designs into CSA. Consequently, we need a mechanism to optimize the chain of non-CSA operations across the design boundary in order to increase the overall performance while preserving the boundary. To this end, we introduce a concept of allocating at least one additional port (called *auxiliary port*) to the design boundary. For example, in Figure 3(c), to merge the two operations *op2* and *op3* into CSA we move up the boundary of design A to the outputs of *csa2* as shown with the heavy line. This implies that two (output) ports on the boundary of A are required rather than one regular port r.

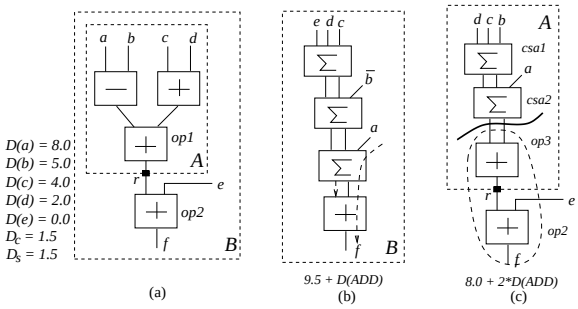

Figure 3: An example of merging operations in different designs into CSAs

The boundary optimization is viewed as partitioning one CSA structure into two part, one residing in design A and another residing in design B, and satisfying the following equality relation between the values assigned to ports r and its auxiliary ports $a_1, \cdots, a_t$ *before* the transformation and those *after* the transformation. (We denote $V_1(x)$ and $V_2(x)$ to the values assumed to port x before and after the transformation, respectively.)

$$V_1(r) = V_2(r) + V_2(a_1) + ... + V_2(a_t) \qquad (2)$$

Given an arithmetic expression across design boundaries, it is important to use the minimum number of auxiliary ports, t_{min}, (while producing optimal-timing) in order to minimize the run time overhead for computing Eq.(2) during simulation. We achieve this by iteratively applying *CSA_OPT* to the expression described as follows.

CSA_OPT_BOUND *optimal CSA allocation for f in Eq. (1) across design boundary*:

- Set t = # of addends in upper expression of f;

repeat

- t = t - 1; /* t: # of aux. ports currently available */
- Apply *CSA_OPT* to upper expr. of f with t aux. ports;
- Apply *CSA_OPT* to lower expr. of f together with the addends from the t ports;

until (there is a timing increase)

- t_{min} = t;

4 Experimentations

We tested our algorithm on a large number of arithmetic computations typically found in industrial designs. We

[2] We use $D(X)$ to denote the arrival time of X if X is an operand, and to denote the delay of the fastest implementation module of X if X is an operation, such as addition, multiplication. We assume that the arrival time of a constant is 0.

used Design Compiler package from Synopsys Inc. to perform the implementation selection, tree-height minimization and logic optimizations.[3] We used 8-bit operands for non-constant inputs of multiplication and used 16-bit operands for the rest. (One exception is iir_design, whose bit-widths are specified in Figure 4.) The multiplication was fully decomposed into additions. For the multiplication with a constant input we applied a signed-digit encoding scheme[5] to reduce the number of operations.

Optimizing expressions contained in single designs: We tested *CSA_OPT* on arithmetic expressions shown in the first column of Table 1. We obtained the minimum timings for all designs produced with and without using *CSA_OPT*. As the comparisons shown in Table 1 indicate, the improvements in both timing and area are significant. Moreover, the reductions are consistent. Since *CSA_OPT* is delay-optimal CSA allocation (also, allocating minimum number of CSAs), the comparisons are for reference only to show how much the timing/area reductions are normally expected when using our CSA techniques.

Expressions	RTL design time/area	*CSA_OPT* time/area	diff.
A_1+A_2+$\cdots$+A_9	3.69/ 6724	2.84/ 4077	-23% -39%
B_1-B_2-B_3-$\cdots$-B_9	3.56/ 5992	2.93/ 5628	-18% -6%
$A \cdot B - C \cdot D$ $+E \cdot F$	5.61/ 9383	4.39/ 7626	-22% -19%
$A \cdot B - C \cdot D - E$	5.40/ 7640	3.92/ 5650	-27% -26%
$A - B \cdot C + D \cdot E$ $+2$	5.65/ 6325	3.97/ 5413	-30% -14%
$A - B + C \cdot D$ $+E \cdot F + 3$	5.62/ 6643	4.03/ 5701	-28% -14%
$A \cdot B + C \cdot D$ $+E \cdot F + G \cdot H$	6.06/ 11749	4.53/ 9382	-25% -20%
$A \cdot B - C \cdot D + E$ $+F - G + H \cdot I + 10$	6.08/ 10073	4.27/ 8649	-30% -14%
$A \cdot 20 + B \cdot 37$ $+C \cdot D$	5.50/ 7880	3.71/ 4603	-27% -42%

Table 1: Results for expressions in single designs

Optimizing expressions contained in multiple designs: We tested *CSA_OPT_BOUND* (with restricted number of auxiliary ports) on arithmetic computations in multiple designs in Figure 4, and summarize the results in Table 2. The second, third, and fourth columns of the table show the results of the original designs, the CSA optimized designs by [2] and the ones by *CSA_OPT_BOUND*, respectively. Each design is tested twice, one for minimizing timing (i.e., labeled with *t-opt*), another for minimizing area under a timing constraint (i.e., labeled with *a-opt*). The results are very impressive, reducing both timing and area significantly.

[3] The tree-height minimization is performed on non-CSA operations. In addition, we used *lcbg10pv* (0.35μ) technology[7] for all the test cases.

Expr.	RTL design time/area	CSA[2] time/area	Ours time/area	diff. over (RTL, [2])
hier_1 (t-opt.)	3.03/ 2919	3.06/ 2261	2.57/ 2017	(-25%, -12%) (-31%, -11%)
hier_1 (a-opt.)	3.03/ 2919	3.06/ 2472	2.88/ 1365	(-14%, -6%) (-53%, -40%)
hier_2 (t-opt.)	4.18/ 3993	3.18/ 3907	2.45/ 3632	(-41%, -23%) (-9%, -7%)
hier_2 (a-opt.)	4.18/ 3993	3.58/ 3255	3.60/ 2390	(-14%, same) (-40%, -27%)
iir_design (t-opt.)	6.57/ 13362	5.93/ 12073	4.94/ 11444	(-25%, -17%) (-14%, -5%)
iir_design (a-opt.)	6.57/ 13362	6.49/ 9172	6.17/ 9202	(-8%, -5%) (-30%, +1%)

Table 2: Results for expressions in Figure 4

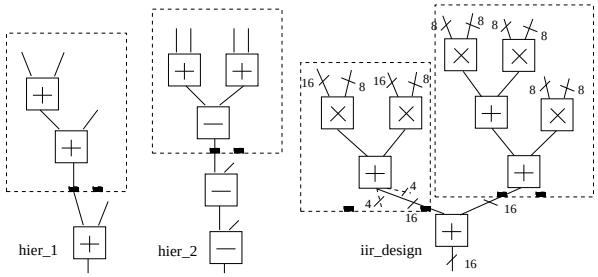

Figure 4: Expressions in multiple designs

5 Conclusions

We presented an *optimal-timing CSA allocation* algorithm for arithmetic expressions contained in single designs. The algorithm was then extended to solve the CSA optimization problem for expressions contained in multiple designs by introducing a new concept, *auxiliary ports*. Experimental results indicate that our work can be used effectively as a solution to the arithmetic optimization problem using CSAs to overcome the cycle time limit of the circuits.

Acknowledgment

This work was supported by the Korea Science and Engineering Foundation (KOSEF) through the Advanced Information Technology Research Center (AITrc).

References

[1] N. Weste and K. Eshraghian, *Principles of CMOS VLSI Design - A Systems Perspective*, Addition-Wesley Publishers, 1985.
[2] T. Kim, W. Jao, and S. Tjiang, "Arithmetic Optimization using Carry-Save-Adders", *DAC*, pp.433-438, 1998.
[3] D. Huffman, "A method for the construction of minimum redundancy codes", *Proc. of the IRE*, Vol.40, pp.1098-1101, 1952.
[4] D. Parker Jr., "Conditions for optimality of the Huffman algorithm", *SIAM Journal of Computing*, Vol.9, No.3 pp.470-489, 1980.
[5] K. Hwang, *Computer Arithmetic: Principles, architecture, and Design*, New York, 1979.
[6] Synopsys Inc., *DesignWare Components Databook*, 1998.
[7] LSI Logic Inc., *G10-p Cell-Based ASIC Products Databook*, 1996.