Layout-Aware Synthesis of Arithmetic Circuits

Junhyung Um and Taewhan Kim
Department of Electrical Engineering & Computer Science
and Advanced Information Technology Research Center(AITrc)
Korea Advanced Institute of Science and Technology

ABSTRACT

In deep sub-micron (DSM) technology, wires are equally or more important than logic components since wire-related problems such as crosstalk, noise are much critical in system-on-chip (SoC) design. Recently, a method [12] for generating a partial product reduction tree (PPRT) with optimal-timing using bit-level adders to implement arithmetic circuits, which outperforms the current best designs, is proposed. However, in the conventional approaches including [12], interconnects are not primary components to be optimized in the synthesis of arithmetic circuits, mainly due to its high integration complexity or unpredictable wire effects, thereby resulting in unsatisfactory layout results with long and messed wire connections. To overcome the limitation, we propose a new module generation/synthesis algorithm for arithmetic circuits utilizing carry-save-adder (CSA) modules, which not only optimizes the circuit timing but also generates a much regular interconnect topology of the final circuits. Specifically, we propose a two-step algorithm: (*Phase 1*: CSA module generation) we propose *an optimal-timing CSA module generation algorithm* for an arithmetic expression under *a general CSA timing model*; (*Phase 2*: Bit-level interconnect refinements) we *optimally refine the interconnects between the CSA modules* while retaining the global CSA-tree structure produced by Phase 1. It is shown that the timing of the circuits produced by our approach is equal or almost close to that by [12] in most test-cases (even without including the interconnect delay), and at the same time, the interconnects in layout are significantly short and regular.

Categories and Subject Descriptions
B.2.4. [**Arithmetic and Logic Structures**]: High-Speed Arithmetic – *Algorithms,Cost/Performance*

General Terms: Algorithms, Design and Peformance

Keywords: Carry-save-adder, layout, high performance

1. INTRODUCTION

The optimization of arithmetic expression has been the subject of intensive investigation in the several phases of design process. A lot of techniques for synthesizing arithmetic circuits are proposed [1]-[12]. A carry-save-adder (CSA) module [20] is one of the most frequently used implementations in the operation level since it has an advantage of fast timing possibility with, if any, little area penalty.

DAC 2002, June 10-14, 2002, New Orleans, Louisiana, USA.

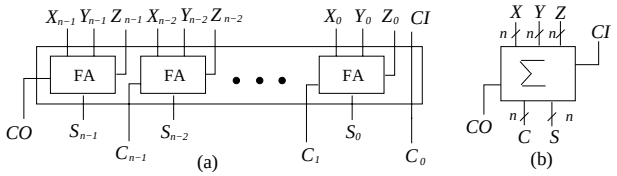


Figure 1: A structure of an *n*-bit CSA

Figure 1(a) shows the structure of *n*-bit CSA. (The block symbol in Figure 1(b) represents a CSA module.) An *n*-bit CSA consists of *n* disjoint full adders (FAs). Note that unlike normal adders (e.g., RCA (ripple-carry-adder) and CLA (carry-lookahead-adder)), there is no carry propagation between the FAs in the CSA. Thus, for a sufficiently large value of *n*, the CSA implementation becomes much faster and also relatively smaller in size than the implementations of normal adders.

There are several works related to efficient representation, allocation and transformation of CSAs in VLSI design automation. Kim *et al.* [10] performs the CSA transformation in three steps: (a) an identification of operation tree to be transformed, (b) a translation of the expression of the identified tree into an addition expression, and (c) a conversion of the addition expression into a CSA-tree. Given a non-cyclic dataflow graph of arithmetic computations, the algorithm iteratively performs the three steps until there is no candidate expression of tree to be transformed. A few extensions to the work in [10] were proposed. Mathur and Saluja [13] improved the implementation of the first step in [10] by extracting a characterization of safe grouping of operations to be transformed. Um and Kim [14] improved the procedure of the third step in [10] by providing an optimal-timing CSA-tree allocation algorithm using a constant CSA delay model. Further, Kim and Um [15] extended the scope of the CSA transformation to include operations across multiplexors, across design boundary, and across multiplications. On the other hand, Yu *et al.* [16] extended the scope of the CSA transformation to include operations across registers. Kim and Kim[17] proposed a design exploration algorithm by trading timing and area of the circuits using CSAs. In addition, Yu *et al*. [18] proposed a carry-save representation for DFG (dataflow graph) to make it easy to solve the CSA allocation and retiming together.

Contrary to the CSA module (word-level) based implementations mentioned above, there have been numerous research works on the design of fast arithmetic circuits, especially multipliers (e.g., [19]), using *bit-level FAs* as basic implementation components. (An extensive survey can be found in [20, 21].) The main issue of the bit-level arithmetic optimization is implementing fast Partial Product Reduction Trees (PPRTs) using FAs. Oklobdzija *et al.* [11] proposed an optimal algorithm, called *three-greedy* algorithm, for constructing a minimum-delay PPRT using FAs under a restricted FA timing model. Further, Stelling *et al.* [12] proposed an algo-

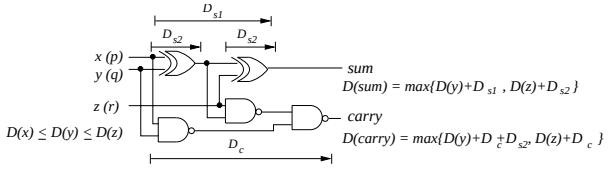


Figure 2: Timing model for FA

rithm called *two-greedy*, which is an improvement of the *three-greedy* algorithm, by developing a number of new techniques for analyzing and designing PPRTs.

Clearly, in terms of timing, the bit-level implementation using FAs is superior to the word-level implementation using CSAs. Conversely, in terms of layout efficiency for interconnects, the CSA implementation is superior to the FA implementation because the wires in the former are regularly grouped into buses. Note that in DSM technology, wires are equally or more important than logic components in layout since wire-related problems such as crosstalk, noise are much critical in system-on-chip (SoC) design. However, most of the previous works have focused on minimizing the circuit timing, and not much seriously taken into account the interconnect optimization for easy layout during the arithmetic circuit implementation.

In this paper, we propose a layout-aware synthesis algorithm for arithmetic circuits utilizing carry-save-adders (CSAs), which not only optimizes the circuit timing but also generates a much reliable interconnect topology of the final circuit. Specifically, we propose a two-step algorithm: (*Phase 1*: CSA module generation) we propose *an optimal-timing CSA module generation algorithm* for an arithmetic expression under *a general CSA timing model*; (*Phase 2*: Bit-level interconnect refinements) we *optimally refine the interconnects between the CSA modules* while retaining the global CSA-tree structure obtained in Phase 1. From experiments, it is shown that the timing of the circuits produced by our approach is equal or almost close to that of the optimal bit-level implementation by the work in [12] in most testcases (even without including the interconnect delay), and at the same time, the interconnects in layout are significantly short and regular.

2. PRELIMINARIES

2.1 CSA Timing Model

Let $D(x)$ denote the arrival time of x if x is a 1-bit signal, and $D(X) = max\{D(x_1), D(x_2), \cdots, D(x_n)\}$ if X is an n-bit vector signal $(x_1, x_2, \cdots, x_n)$.[1] Since n-bit CSA consists of n disjoint FAs, let us check the timing model of FA first.

Let p, q, r be the three input ports of an FA, and *carry* and *sum* be the carry and sum output ports of the FA, respectively. Let x, y and z ($D(x) \leq D(y) \leq D(z)$) be the bit addends, each assigned to p, q, and r. Based on a typical logic implementation for FA (See Figure 2.), the time needed by an FA to generate its *sum* output from its inputs amounts to the 2-level XOR delay. Thus, we use two delay parameters D_{s1} and D_{s2} to model the delay to *sum* output where D_{s1} indicates the delay from q to *sum* port and D_{s2} the delay from r to *sum* port. Similarly, we use D_c (2-level NAND delay) to indicate the *carry* delay of an FA. We refer the FA timing model mentioned above to as the *general timing model of FA*. Consequently, the arrival times for *sum* and *carry* signals from inputs are calculated by $D(sum) = max\{D(y)+D_{s1}, D(z)+D_{s2}\}$ and $D(carry) = max\{D(y)+D_c+D_{s2}, D(z)+D_c\}$. Note that the PPRT reduction algorithm in [12] used the general FA timing model and produces an optimal-timing FA-tree allocation.

We now define *a general CSA timing model* by extending the general timing model of FA as follows. According to the internal structure of CSA, in addition to the three (normal) input ports, a CSA has 1-bit carry input port. Let X, Y, Z ($D(X) \leq D(Y) \leq D(Z)$) be the n-bit vector addends, assigned to the three normal input ports of the CSA, and s be the signal assigned to carry input port. Then, the delays to *sum* and *carry* of the CSA are defined by $D(sum) = max\{D(Y)+D_{s1}, D(Z)+D_{s2}\}$ and $D(carry) = max\{D(y)+D_c+D_{s2}, D(z)+D_c\}$. Note the CSA-tree allocation algorithm in [14] produces an optimal-timing CSA-tree under the constraint of $D_{s1} = D_{s2}$, but is not optimal under the general CSA timing model. We propose a new CSA allocation algorithm (section. 3.1) which produces an optimal-timing CSA-tree under the general CSA timing model.

2.2 Motivating Example

Let us begin with an example of synthesizing an arithmetic circuit to see how our proposed approach is essentially different with the conventional methods. Suppose we want to add seven operands X_1, $\cdots$, X_7. Each operand is 3-bit wide, and its word-level and bit-level arrival times are shown in the table in Figure 3(a). Figure 3(b) shows the optimal-timing CSA-tree structure, which we call *circuit1*, produced by the algorithm in [14] under a restriction on the CSA timing model. On the other hand, Figure 3(c) shows the optimal-timing FA-tree structure, which we call *circuit2*, produced by the algorithm in [12] under the general FA timing model. From the comparisons between *circuit1* and *circuit2*, we can find that the interconnects in *circuit1* look more regular (i.e., grouped into buses) than those in *circuit2* since *circuit1* comes from a coarse-grained (word-level) module generation, but *circuit2* is faster than *circuit1* since *circuit2* comes from a fine-grained (bit-level) cell generation.

The motivation of our approach is to make use of both advantages of [14] (word-level) and [12] (bit-level). Figure 3(d) shows an improved optimal-timing CSA-tree structure, which we call *circuit3*, produced by our algorithm under the general CSA timing model. Consequently, the timing of *circuit3* (=11) is shorter than that of *circut1* (=13), and the interconnects in *circuit3* are still as regular as that in *circuit1*. We further improve the timing of *circuit3* by FA-level interconnect reordering. *circuit4* in Figure 3(e) shows the resultant CSA-tree structure optimized from *circuit3*, in which the FAs marked with dotted circle indicate the reordering of FA input connections. For example, in FA_9 its first and third connections are switched to reduce its output timings. Since the original CSA-level circuit structure is preserved during the (local) FA-level interconnect reordering, the degree of interconnect regularity in *circuit4* is retained in a global prospective. Nevertheless, the timing (=10) of critical paths in *circuit4* is reduced, and much close to the optimal (bit-level) timing (=9) in *circuit2*.

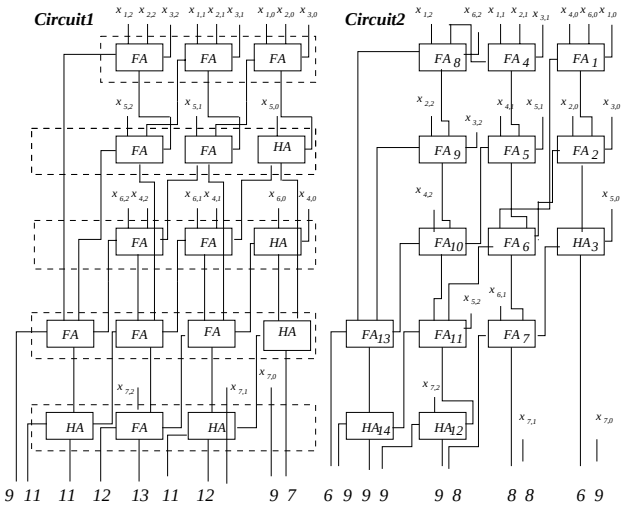
Finally, Figures 3(f) and (g) show the layouts (assumed a simple symmetrical FPGA-style mapping) with optimized routability for *circuit2* and *circuit4*, respectively, where the heavy line in Figure 3(f) indicates the longest net. Note that the layout in Figures 3(f) is optimal in the sense that we were not able to reproduce a layout for *circuit2* that has less number of the longest nets and/or connection blocks with maximum routing density than the layout in Figure 3(g). This means that, in addition to the gate delay, when we take into account the interconnect delay in measuring the performance of the final circuit, the overall timing of *circuit4* can be much shorter than that of *circuit2*. Furthermore, it is clear that this trend will be very likely as the circuit size increases.

In fact, *circuit3* and *circuit4* are produced by Phase 1 and Phase 2 of our proposed two-step approach. The comparison of the layouts clearly indicates that, among the four circuits in Figure 3, *circuit4* is the most efficient structure in terms of timing and layout in synthesizing arithmetic circuits. We now provide the details on the two

[1] $D(X) = 0$ if X is a constant number.

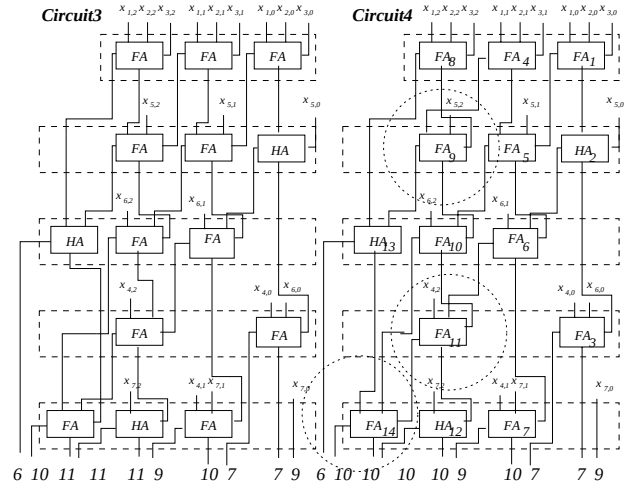
Word	Bit	Delay(Word)	Delay(Bit)
X_1	$x_{1,2}\ x_{1,1}\ x_{1,0}$	2	0, 0, 2
X_2	$x_{2,2}\ x_{2,1}\ x_{2,0}$	2	1, 0, 2
X_3	$x_{3,2}\ x_{3,1}\ x_{3,0}$	4	3, 0, 4
X_4	$x_{4,2}\ x_{4,1}\ x_{4,0}$	7	7, 1, 1
X_5	$x_{5,2}\ x_{5,1}\ x_{5,0}$	5	4, 3, 5
X_6	$x_{6,2}\ x_{6,1}\ x_{6,0}$	6	0, 6, 1
X_7	$x_{7,2}\ x_{7,1}\ x_{7,0}$	9	8, 8, 9

(a) Operands to be added



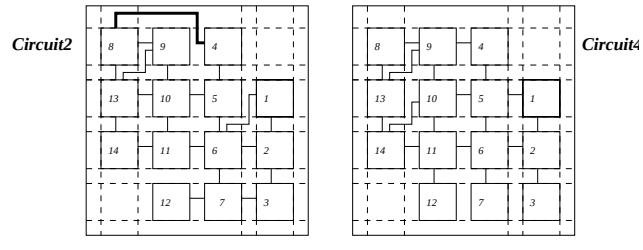
(b) Optimal CSA−tree by [14] with a resticted CSA timing model

(c) Optimal FA−tree by [12] using the general FA timing model



(d) Optimal CSA−tree by ours using thegeneral CSA timiing model (Phase 1)

(e) Optimal CSA−tree by ours using FA−level interconnect reordering (Phase 2)



(f) Layout for the result in (c)

(g) Layout of our result in (e)

Figure 3: An example showing the effectiveness of our approach, in terms of timing and layout

phases of our approach.

3. A NEW SYNTHESIS APPROACH

The proposed layout-conscious synthesis approach is performed in two optimization phases. Given an arithmetic expression to be implemented, in *Phase 1*, we propose a timing-optimal CSA-tree construction algorithm, which we refer to as *partial-greedy-CSA* algorithm, for an arithmetic expression under the general CSA timing model. In *Phase 2*, we propose a layout-aware HA merging technique (called *HA-merge*) to refine the CSA-tree structure (*Phase 2.1*), and an optimal FA input reordering technique (called *FA-ref*) to further improve the timing of the circuit without deteriorating the CSA block-level interconnect structure (*Phase 2.2*).

3.1 Phase 1: Optimal-Timing CSA Allocation

The CSA allocation problem is stated as: *(Problem 1) Given an addition expression*[2] *of*

$$F = X_1 + X_2 + \cdots + X_m + s_1 + s_2 + \cdots + s_n + c \quad (1)$$

where $X_1, \cdots X_m$ are multi-bit addends, $s_1, \cdots s_n$ are single-bit addends and c is a constant, and the arrival times of the addends, for the general delay model for CSA (defined in section 2.1), determine a CSA-tree structure for F with a minimum delay.

Let us first consider a restricted case of Problem 1.

Subproblem 1: Problem 1 with $c = 0$ and $m - 1 \geq n$.

The inequality $m - 1 \geq n$ implies that the number of multi-bit addends is large enough for all single-bit addends to be used as carry inputs to CSAs.

Let R be the set of all possible CSA-trees for Subproblem 1, in which all single-bit addends are necessarily used as carry inputs to CSAs. Let us consider the following CSA-tree construction strategy, which we call *partial-greedy-CSA* algorithm: We allocate CSAs iteratively to sum up all the addends such that at each iteration, we allocate a new CSA with three multi-bit addends, in which two of them have the first and second earliest arrival times among the multi-bit addends, and one single-bit addend with the earliest arrival time assigned to the CSA inputs. Note that the CSA-trees produced by the *partial-greedy-CSA* algorithm will have different timings and structures depending on the way of selecting the remaining multi-bit addend at each iteration. (See Figure 4.) Let A be the set of all CSA-trees that can be produced by the *partial-greedy-CSA* algorithm.

We define a number of notations for the ease of our discussion: Let CSA_i denote the CSA created at the ith iteration of *partial-greedy-CSA* method. a_i, b_i and d_i ($D(a_i) \leq D(b_i) \leq D(d_i)$) denote the multi-bit addends connected to the three normal input ports (from left to right, X, Y, Z in Figure 1) of CSA_i, respectively, and s_i denote the single-bit addend to the *carry-in* port of CSA_i. In addition, $carry_i$ and sum_i denote *carry* and *sum* signals of CSA_i, respectively. Let T be a circuit generated by performing n iterations of CSA allocation in *partial-greedy-CSA* method. We use $D(x,T)$ to denote the arrival time of signal x in CSA-tree T, and $D(T)$ is set to $max\{D(x_1,T), \cdots D(x_n,T)), D(y_1,T), \cdots D(y_n,T)\}$ where x_i and y_i, $i = 1, \cdots n$, are *carry* and *sum* signals of CSA_i in T, respectively.[3]

Lemma 1 *There exists a CSA-tree $T' \in A$ such that $D(T') \leq D(T)$ for any $T \in R$. (We omit all the proofs due to the space of limitation.)*

Lemma 1 suggests that using *partial-greedy-CSA* method can reduce the search space for finding an optimal CSA-tree from R.

[2]The expression can contain subtraction and/or multiplication operations. As to converting it into an addition expression, refer to [10].

[3]We simply use $D(v)$ to indicate $D(v,T)$ if the meaning of T in $D(v,T)$ is implicitly clear in our discussion.

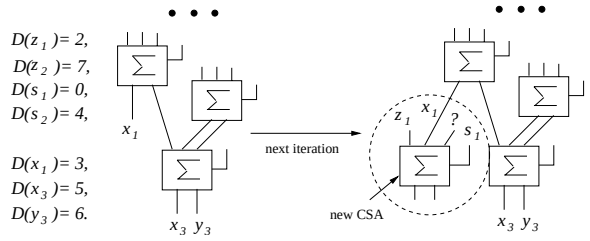


Figure 4: An illustration of *partial-greedy-CSA* algorithm

Note that the procedure of *partial-greedy-CSA* method is quite similar to the *two-greedy FA allocation* method proposed in [12]. The differences are (i) our CSA allocation solution considers the carry-out vector of a CSA as *any* CSA input during the next iterations of CSA allocation, whereas [12]'s FA allocation solution (in bit-weight 2^i) uses the carryout signal of an FA *only* as input to an FA allocated at the next bit-weight (i.e., 2^{i+1}); (ii) In addition to the multi-bit inputs, the CSA allocation problem has another input type, i.e., single-bit inputs, which have a freedom to be used either as normal inputs to CSAs or as carry inputs to CSAs, affecting the value of $D(carry)$ only. We found that some of the properties in [12] regarding the *two-greedy FA allocation* method also hold for our *partial-greedy-CSA* method. Now, let us consider an extended version of Subproblem 1.

Subproblem 2: Problem 1 with $c = 0$ and $m - 1 < n$.

The inequality $m - 1 < n$ implies that some of the single-bit addends must be used as normal inputs to CSAs. Thus, we should decide *which* and *how many* of the single-bit addends are to be used as normal inputs to CSAs. Let k be the number of single-bit addends to be used as normal inputs to CSAs. Then we can easily show $k \geq \lceil \frac{n-m+1}{2} \rceil$, from the fact that converting one single-bit addend to a multi-bit addend to be used as a normal input to a CSA requires one additional CSA and thus, resulting in one more carry input ports available, and at the same time, one less single-bit addends.

First, let us consider which of the single-bit addends are to be used as normal inputs to some of the CSAs when the number k is fixed. We claim the following:

Lemma 2 *To select k single-bit addends to be used as normal inputs to CSAs, the addends with the earliest arrival times will be selected for optimal-timing CSA-tree allocation.*

Moreover, the following lemma answers to the question of how many of the single-bit addends are to be used as normal inputs to some of the CSAs. we show that for optimal-timing CSA-tree allocation, the value of k should be chosen as $k = \lceil \frac{n-m+1}{2} \rceil$.

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

Finally, we complete the solution of Problem 1.

Subproblem 3: Problem 1 with $c \neq 0$.

If c 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)) because *partial-greedy-CSA* algorithm with the 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 better timing.

Theorem 1 *When we use* partial-greedy-CSA *method for CSA allocation, if either $c > 0$ and $m - 1 < c + n$ or $c < 0$ we should treat c as a multi-bit addend, and if $c > 0$ and $m - 1 \geq c + n$, we should break c into into a form of $1 + 1 + ... + 1 (= c)$ and use them as carry inputs to CSAs.*

We use a branch and bound search to find an optimal-timing CSA-tree structure by applying *partial-greedy-CSA* allocation strategy. Since the CSA allocation algorithm in [14] produces an optimal-timing CSA-tree using the CSA timing model constrained by $D_{s1} = D_{s2}$, we can use the timing of the CSA-tree, produced by [14] under the setting of the value of D_{s2} to the value of D_{s1} ($D_{s2} \leq D_{s1}$), as *an upper bound* of the optimal-timing in the branch and bound search. In addition, we devise a number of pruning techniques based on the following properties to speed up the execution of the branch and bound search. Let T be a CSA-tree produced by *partial-greedy-CSA* method, i.e., $T \in A$.

Property 1 *If there are CSA_i, $CSA_j \in T$ such that $i < j$,*[4] *$D(d_i) \leq D(b_j)$ and $D(d_i) > D(a_j)$, there is a CSA-tree $T' \in A$ such that $D(T') \leq D(T)$.*

Property 2 *If there are CSA_i, $CSA_j \in T$ such that $i < j$, $D(b_j) < D(d_i) \leq D(d_j)$ and $D(d_j) \geq D(d_i) + D_{s1} - D_{s2}$, there is a CSA-tree $T' \in A$ such that $D(T') \leq D(T)$.*

Property 3 *If there are CSA_i, $CSA_j \in T$ such that $i < j$ and $D(d_i) > D(d_j)$, there is a CSA-tree $T' \in A$ such that $D(T') \leq D(T)$.*

3.2 Phase 2: Optimal Refinement of CSA Timing

The objective of this phase is to reduce the timing of the CSA-tree further by refining the bit-level interconnect structures while maintaining the topology of the word-level CSA-tree structure obtained in Phase 1. Phase 2 is a two-step optimization: (Phase 2.1) Layout-aware *HA (half-adder) merging step* which merges a pair of HAs in a CSA-tree into an FA and (Phase 2.2) *FA refinement step*, which locally refines the FA's inputs.

Phase 2.1 (HA-merging): For the CSA allocation in Phase 1, if the bit-widths of three (normal) inputs to be added using a CSA are not totally equal,[5] or some input has a constant bit-value (e.g., logic-0 or 1) in a certain bit-position, some of the FAs in the CSA will be degenerated into HAs.

For each bit-column, from the rightmost bit column to the leftmost, we perform *HA-merge* from top to bottom, the leaf HAs first and the root HAs last. During the top-down HA-merge process, if the sum signal of an HA is fed to an input to another HA (i.e., tightly connected), we combine the two HAs into one FA, as shown in Case 1 of Figure 5(a). There are two possible ways of connecting the carry signal of the new FA, as described in the example in Figure 5(a) (i.e., trans1 and trans2). We have an important property for the HA-merge for Case 1.

Property 4 *If an HA-merge is performed for Case 1 in Figure 5(a), the timing of* trans2 *is always better than or equal to that of* trans1 *when the general timing model for HA (similar to that of FA) is used. Further, the timing of* trans2 *is always better than or equal to that of* case1_org.

Consequently, according to Property 4, when we encounter a pair of tightly connected HAs (i.e., Case 1) we apply HA-merge to the HAs and produce a merged FA in a form like trans2 in Figure 5(a).

On the other hand, if two HAs are not tightly connected as shown case2_org in Figure 5(b), *we do not directly merge the two HAs*

[4] Note that CSA_i is the CSA created at the *i*-th iteration of the *partial-greedy-CSA* method.

[5] This mostly happens when a multiplication is decomposed into a set of shift-add operations.

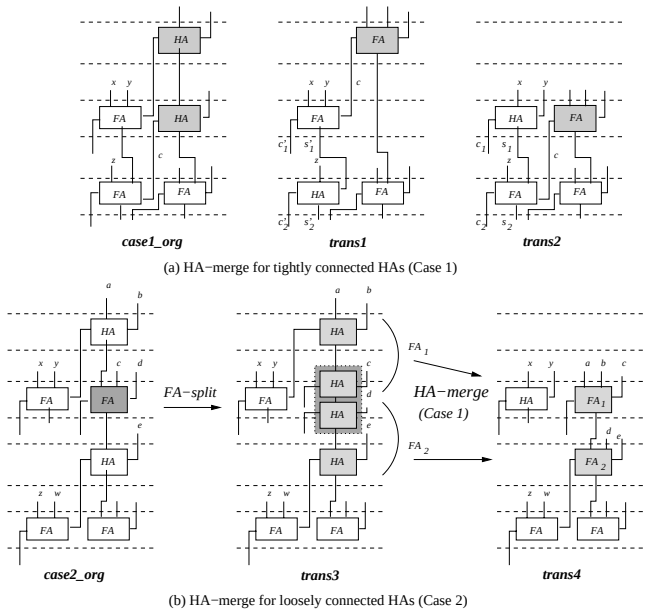


Figure 5: Examples for layout-aware HA-merge for (Case 1) tightly connected HAs and (Case 2) loosely connected HAs.

in the same way as that of Case 1 because the merger causes a (global) distortion of the interconnection topology of the initial CSA-tree. Instead, we split the FA, one of whose inputs is directly fed from the upper HA, into two HAs in a chain and merge the upper HA with the decomposed HA that is directly connected to the upper HA into an FA. Thus, we perform HA-merge process for Case 2 in two steps: (Step 1) *FA-split* to convert Case 2 into Case 1, and (Step 2) *HA-merge* according to the procedure in Case 1. Figure 5(b) shows an example of the two steps. We have the following property, which justifies our HA-merge process.

Property 5 *If an HA-merge is performed for Case 2 in Figure 5(b), the timing of* trans4 *is always better than or equal to that of* case2_org *when the general timing model for HA is used.*

According to Properties 4 and 5, the proposed HA-merge can be performed iteratively, from top (leaf) CSAs to bottom of CSA-tree, which monotonically reduces the timing of the CSA-tree at each iteration.

Phase 2.2 (FA-refinement): In this phase, we perform a bit-level timing refinement while preserving the word-level global interconnect structure obtained in Phase 2.1. Note that Phase 1 used a word-level arrival times of input addends during the CSA-tree allocation. That is, if there is an n-bit addend $X = (x_{n-1}, x_{n-2} \cdots x_0)$, Phase 1 uses $max\{D(x_{n-1}), D(x_{n-2}) \cdots D(x_0)\}$ as the value of $D(X)$.

Since we try to preserve the word-level interconnect structure of the CSA-tree during this phase, we perform a sort of FA-level connection restructuring *inside of* the corresponding CSA block. Consequently, the values of the arrival times of Y_1, Y_2 and Y_3 determine the order of input connections to CSA_i, which in turn determines uniform relative orders of the inputs of all FAs in CSA_i, as shown in Figure 6(b). In Phase 2.2, for each FA we reorder its input connections to reduce the FA's output timings, as shown in Figure 6(c).

We perform the FA's input reordering for a CSA-tree topologically, the FAs in the leaf CSAs first and the FAs in the root CSAs last, which we call *FA-ref* method for FA's input reordering. We claim that *FA-ref* is timing-optimal.

Theorem 2 *When the bit-level rewiring for a CSA-tree is allowed only for the inputs to the same FA, the timing reduced by FA-ref method is optimal.*

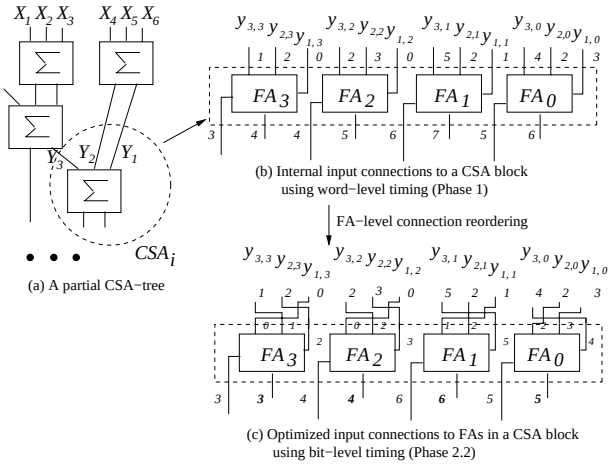


Figure 6: An example of illustrating FA-refinement.

4. EXPERIMENTAL RESULTS

We conducted a set of experiments to check the effectiveness of the proposed layout-conscious module generation/synthesis algorithm, consisting of *Partial-Greedy-CSA*, *HA-merge* and *FA-ref* techniques. Our algorithm was implemented in C++ and executed on a Pentium-3 500MHz Linux machine. We tested our algorithm on a set of random arithmetic expressions and benchmark examples [22], and then compared our results with that produced by the Stelling's bit-level delay-optimal[6] algorithm [12].

After the synthesis using ours and Stelling's for each testcase, we performed, for each synthesized design, placement[7] using Fiduccia and Mattheyses's partitioning algorithm [23], and routing[8] using Lee's maze routing algorithm [24]. We used the Synopsys Design Compiler [25] with $lcbg10pv(0.35u)$ technology library [26] to set the timing parameters for CSAs and FAs. We used a simple formula $\alpha \cdot L^2$ to approximate the delay of the wire connecting two FA cells where L is the number of blocks on the corresponding routing path. α is a parameter to reflect the relative delay difference between gates and wires, the value of which will depend on the layout style and target library used.

• **Synthesizing arbitrary arithmetic expressions:** We tested our algorithm using a set of arithmetic expressions, and the results are summarized in Table 1. The bit-level arrival times of the operands in the expressions are randomly generated in between 0.0 and 2.0ns. The table shows the critical path and longest net delays for the designed by ours and Stelling's [12]. Figure 7 show the layouts for the designs produced by Stelling's and ours for Exp.3 where we have 61% shorter critical path delay (highlighted in the figure). The comparisons indicate that the circuit structures produced by ours significantly reduce the critical path delay and the longest net delay compared to Stelling's. Note that as the α value increases (i.e., wire delay being more dominant than gate delay), the performance improvement is more significant.

• **Synthesizing filter designs:** We also tested our algorithm using high-level synthesis benchmark designs, and the results are summarized in Table 2. We assumed the arrival times of input operands are 0. The results indicate that our synthesis algorithm reduces the critical path delay 22% on the average compared to Stelling's. Furthermore, the reductions are consistent. In summary, the comparisons of the results strongly suggest that considering layout effects

[6] 'optimal' when considering gate delays only.

[7] The target layout we used is gate array (lattice) style for simplicity, but our approach can be applied to any layout style such as standard cell and FPGA.

[8] We set the channel capacity in the layout to 4.

Exp.	α	Stelling[12] crit. path delay/ long. net delay	Ours crit. path delay/ long. net delay	Improvement crit. path delay/ long. net delay
Exp. 1	0.04	11.27 / 3.24	8.87 / 1.96	21% / 40%
	0.02	6.78 / 1.62	5.59 / 0.98	18% / 40%
	0.01	4.54 / 0.81	3.95 / 0.49	13% / 40%
Exp. 2	0.04	14.69 / 6.76	8.14 / 3.24	45% / 52%
	0.02	7.89 / 3.38	4.86 / 1.62	38% / 52%
	0.01	4.49 / 1.69	3.22 / 0.81	28% / 52%
Exp. 3	0.04	46.43 / 17.64	18.22 / 9.00	61% / 49%
	0.02	23.79 / 8.82	9.50 / 4.50	60% / 49%
	0.01	12.48 / 4.42	5.14 / 2.25	59% / 49%
Exp. 4	0.04	21.92 / 9.00	14.43 / 6.76	34% / 25%
	0.02	12.16 / 4.50	8.49 / 3.38	30% / 25%
	0.01	7.28 / 2.25	5.52 / 1.69	24% / 25%
Exp. 5	0.04	18.42 / 9.00	13.41 / 6.76	27% / 25%
	0.02	10.42 / 4.50	7.96 / 3.38	24% / 25%
	0.01	6.42 / 2.25	5.24 / 1.69	18% / 25%
Exp. 6	0.04	19.27 / 9.00	11.88 / 3.24	38% / 64%
	0.02	10.87 / 4.50	7.48 / 1.62	31% / 64%
	0.01	6.67 / 2.25	5.28 / 0.81	21% / 64%
Avg.	0.04	29.06 / 9.11	12.49 / 5.16	**38**% / **43**%
	0.02	11.99 / 4.56	7.31 / 2.58	**34**% / **43**%
	0.01	6.98 / 2.28	4.73 / 1.29	**27**% / **43**%

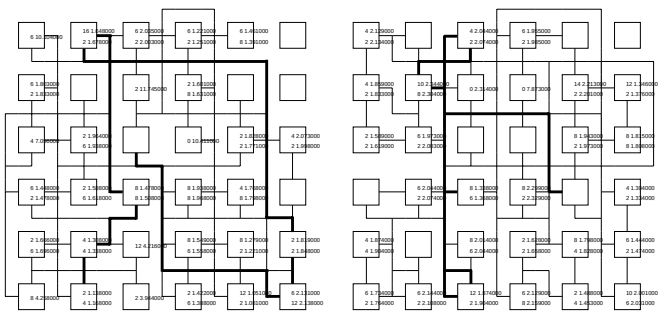
Table 1: Results for a set of arithmetic expressions Exp.1 : $X_1+\cdots+X_6$; Exp.2 : $X_1\cdot X_2+X_3+X_4$; Exp.3 : $X_1+X_2+X_3-X_4+X_5-X_6-X_7$; Exp.4 : $X_1+\cdots+X_8$; Exp.5 : $X_1\cdot X_2+X_3\cdot X_4+X_5\cdot X_6$; Exp.6 : $X_1\cdot X_2+X_3\cdot X_4-X_5\cdot X_6-X_7\cdot X_8$;

at the early (architectural-level) stage of the synthesis process can considerably improve the performance of the final circuits, and our proposed synthesis approach does the job quite well.

5. CONCLUSIONS

In this paper, we presented a new RT-level synthesis approach for arithmetic circuits, which considers two important design objectives: fast timing and easy layout. As the interaction between architectural/logical (synthesis) and physical (layout) domains becomes much important issue in DSM technology, considering layout effects at an early stage of the synthesis process is necessary. In the respect, we believe that the proposed new two-phase approach, which (Phase 1) *produces an optimal-timing arithmetic circuit using fast CSA (carry-save-adder) modules under general CSA timing model*, and (Phase 2) *refines the circuit structure obtain in Phase 1 by developing layout-aware optimal HA-merging and FA-input-reordering techniques*, can be used effectively in synthesizing (arithmetic) datapaths. From a set of benchmark filter designs, we confirm that the proposed approach produces designs with (gate) timing comparable to the optimal bit-level timing by [12] while greatly reducing the complexity of interconnects, resulting in much shorter critical path delay (22% reduction on average over [12]) and longest net delay (26% reduction) in the final circuits.

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



(a) Stelling's [12] (b) Ours

Figure 7: Comparison of layout results for Exp.3**. (The heavy lines indicate the critical paths.)**

Filter Design	α	Stelling's[12] crit. path delay/ long. net delay	Ours crit. path delay/ long. net delay	Improvement crit. path delay long. net delay
Lowpass	0.04	16.11 / 6.76	12.08 / 4.84	25% / 28%
	0.02	8.52 / 3.38	6.48 / 2.42	24% / 28%
	0.01	4.72 / 1.69	3.68 / 1.21	22% / 28%
Laplace	0.04	12.26 / 4.84	12.08 / 4.84	4% / 0%
	0.02	6.74 / 2.42	6.48 / 2.42	4% / 0%
	0.01	3.98 / 1.21	3.78 / 1.21	5% / 0%
Wavelet	0.04	46.43 / 17.64	18.22 / 9.00	61% / 49%
	0.02	23.79 / 8.82	9.59 / 4.50	60% / 49%
	0.01	12.48 / 11.86	5.14 / 2.25	59% / 49%
IIR	0.04	21.38 / 14.44	20.02 / 6.76	6% / 53%
	0.02	11.23 / 7.22	10.50 / 3.38	6% / 53%
	0.01	6.15 / 3.61	5.74 / 1.69	7% / 53%
Complex	0.04	18.68 / 11.56	16.53 / 11.56	12% / 0%
	0.02	9.99 / 5.78	8.69 / 5.78	6% / 0%
	0.01	5.61 / 2.89	4.77 / 2.89	7% / 0%
Avg.	0.04	22.97 / 11.05	15.79 / 7.40	**22**% / **26**%
	0.02	10.05 / 6.13	8.26/3.70	**21**% / **26**%
	0.01	5.49 / 4.60	5.49 / 4.60	**21**% / **26**%

Table 2: Results for a set of benchmark designs.

6. REFERENCES

[1] M. Potkonjak and J. Rabaey, "Optimizing Resource Utilization Using Transformations", *Proc. of ICCAD*, 1991.

[2] M. Potkonjak and J. Rabaey, "Maximally Fast and Arbitrarily Fast Implementation of Liner Computations", *Proc. of ICCAD*, 1992.

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

[4] A. Nicolau and R. Potasma, "Incremental Tree Height Reduction for High-level Synthesis", *Proc. of DAC*, 1991.

[5] R. Hartley and A. E. Casavant, "Tree-Height Minimization in Pipelined Architectures", *Proc. of ICCAD*, 1989.

[6] R. Hartley and A. E. Casavant, "Optimized Pipelined Networks of Associative and Commutative Operators", *IEEE Trans. on CAD* , Vol. 13, No. 11, 1994.

[7] K. K. Parhi, "High-Level Algorithm and Architecture Transformations for DSP Synthesis", *Journal of VLSI Signal Processing*, No. 9, 1995.

[8] M. Janssens, F. Catthoor, and H. De Man, "A Specification Invariant Technique for Operation Cost Minimization in Flow-graphs", *Proc. of 7th International Symposium on High-Level Synthesis*, 1994.

[9] M. Janssens, F. Catthoor, and H. De Man, "A Specification Invariant Technique for Regularity Improvement between Flow-Graph Clusters", *Proc. of EDAC*, 1996.

[10] T. Kim, W. Jao, and S. Tjiang, "Circuit Optimization using Carry-Save-Adder Cells", *IEEE Trans. on CAD*, Vol.17, No.10,1998.

[11] 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 Trans. on Computers*, Vol. 45, No. 3, 1996.

[12] P. Stelling, C.U. Martel, V.G. Oklobdzija, and R. Ravi, "Optimal Circuits for Parallel Multipliers", *IEEE Trans. on Computers*, Vol. 47, No. 3, 1998.

[13] A. Mathur and S. Saluja, "Improved Merging of Datapath Operators using Information Content and Required Precision Analysis", *Proc. of DAC*, 2001.

[14] J. Um and T. Kim, "An Optimal Allocation of Carry-Save-Adders in Arithmetic Circuits", *IEEE Trans. on Computers*, Vol. 50, No. 3, 2001.

[15] T. Kim and J. Um, "A Practical Approach to the Synthesis of Arithmetic Circuits using Carry-Save adders", *IEEE Trans. on CAD*, Vol. 19, No. 5, 2000.

[16] Z. Yu, M.-L. Yu, and A.N. Willson Jr., "Signal Representation Guided Synthesis using Carry-Save Adders for Synchronous Data-path Circuits", *Proc. of DAC*, 2001.

[17] Y. Kim and T. Kim, "Accurate Exploration of Timing and Area Trade-offs in Arithmetic Optimization using Carry-Save-Adders", *Journal of Circuits, Systems and Computers*, Vol. 10, Nos. 5&6, 2000.

[18] Z. Yu, K.-Y. Khoo, and A.N. Willson Jr., "The Use of Carry-Save Representation in Joint Module Selection and Ret iming", *Proc. of DAC*, 2000.

[19] C.S. Wallace, "A Suggestion for a Fast Multiplier", *IEEE Trans. on Computers*, Vol. 13, 1964.

[20] K. Hwang, *Computer Arithmetic: Principles, Architecture and Design*, Jhon Wiley and Sons, 1979.

[21] E. Swartzlander, *Computer Arithmetic*, Vols. 1&2, IEEE CS Press, 1990.

[22] N. D. Dutt, "High-Level Synthesis Design Repositories", *http://www.ics.uci.edu/ dutt*.

[23] C. Fiduccia, R. Matteyses, "A Linear-time Heuristic for Improving Network Partitions", *Proc. of DAC*, 1982.

[24] C. Lee, "An Algorithm for Path Connections and its Applications", *IRE Trans. on Electronic Computers*, VEC-10. pp. 346-365, 1961.

[25] Synopsys Inc., *Design Compiler User Guide*, 2000.

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