A Timing-Driven Synthesis of Arithmetic Circuits using Carry-Save-Adders

Taewhan Kim and Junhyung Um

Department of Computer Science
and Advanced Information Technology Research Center(AITrc)
Korea Advanced Institute of Science & Technology
Taejon, 305-701 KOREA

Abstract— Carry-save-adder (CSA) is one of the most widely used types of operation in implementing a fast computation of arithmetics. An inherent limitation of the conventional CSA applications is that the applications are confined to the sections of arithmetic circuit that can be directly translated into addition expressions. To overcome this limitation, from the analysis of the structures of arithmetic circuits found in industry, we derive a set of simple, but effective CSA transformation techniques. Those are (1) *optimization across multiplexors*, **(2)** *optimization across design boundaries* **(restricted notion of [3]), and (3)** *optimization across multiplications*. **Based on the techniques, we develop a new timing-driven CSA transformation algorithm that is able to utilize CSAs extensively throughout the whole circuits. Experimental data for practical testcases are provided to show the effectiveness of our algorithm.**

I. Introduction

Timing of circuit is one of the most important design criteria to be optimized in several phases of synthesis process. The work presented in this paper belongs to RTL synthesis in that we optimize timing in operation level from the given latency of design and cycle time. However, unlike the previous approaches to tree-height reduction in which they focused on the transformation of operations using techniques such as simple algebraic manipulations and constant propagation, this work utilizes a new fast operation, carry-save-adder (CSA)[1], which leads to a completely different scheme for tree-height reduction.

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 and n-bit carry vector. Unlike the normal adders, a CSA contains no carry propagation. 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. Note that CSA transformation is not limited to addition only[2, 4]. We can replace a subtraction by adding the negation of the subtraction. A multiplication can be decomposed into two possible options:[2] (a) sum of products and (b) a partial multiplication and a final addition obtained by decomposing wallace tree synthesis model for multiplication[4].

Our algorithm is designed to overcome the limitations of the conventional CSA transformations[2] that have not been able to optimize operation trees across multiplexors, across design boundaries and/or across multiplications, which appear quite often in industrial designs.

II. CSA Optimizations

A. Optimizing Across Multiplexors (OA_{mux})

A conditional statement in design description or a resource sharing introduces multiplexors in the circuit. Fig. 1(a) and (b) show an example of a part of VHDL design description with a conditional statement and its corresponding translated circuit graph, respectively. The dotted arrow in Fig. 1(b) indicates a critical path of the circuit. We carry out the transformation in three steps:

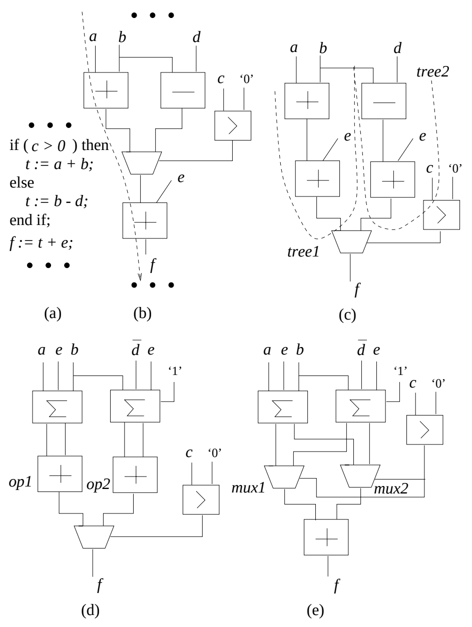
1. *Move-up:* The operation on the critical path whose input comes from the multiplexor moves up across multiplexor to form an operation tree as shown in Fig. 1(c). Because of the duplication of the operation, the implementation area will increase, but the timing is unchanged.

2. *Transformation:* The operation tree identified on the conditional branch of the critical path (i.e., *tree1*) is then transformed into a CSA tree as shown in Fig. 1(d).[1] In addition, when the right conditional branch contains an operation tree with the duplicated operation as root (i.e., *tree2*), it is also transformed into a CSA tree.

3. *Move-down:* The final additions of the transformed tree on the conditional branch (i.e., *op*1, *op*2) move down across the multiplexor and are merged into an addition as shown in Fig. 1(e). Note that this step will duplicate the multiplexor, but does not increase timing.

B. Optimizing Across Design Boundaries (OA_{bound})

[1]We use Σ symbol to represent a CSA block.



```
if ( c > 0 ) then
   t := a + b;
else
   t := b - d;
end if;
f := t + e;
```

Fig. 1. An example of CSA transformation across multiplexor

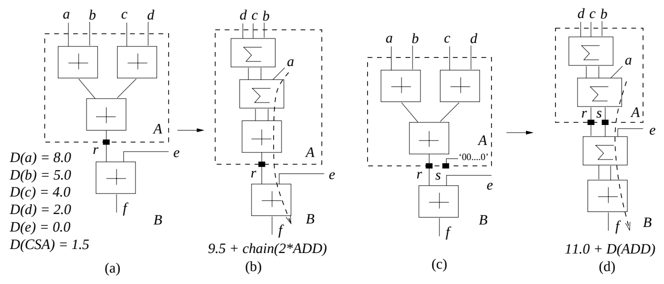


Fig. 2. Examples of transforming operations in different design hierarchy

Fig. 2(a) shows an arithmetic circuit that is contained in two sub-designs A and B.[2] Applying CSA transformation to the two designs one by one shall create two CSA trees as shown in Fig. 2(b). To merge operations in different design hierarchy into CSAs, we use the concept of allocating additional ports proposed in [3]. However, we restrict the number of additional ports to only one since it is simple enough and will not seriously degrade the simulation speed. Fig. 2(c) shows an example of the allocation of an additional port s associated with regular port r. Initially, it has been assigned with 0. Consequently, we are able to produce a CSA tree *without final adder* for the operation tree in design A and connect the two output addends of the tree, i.e., the sum and carry output vectors of the final CSA, to r and s of A to be merged into CSAs with other addends of the arithmetic expression in B. Fig. 2(d) shows the resultant CSA tree transformed from the expression in Fig. 2(c).

C. Optimizing Across Multiplications (OA_{mult})

An operation tree that is extracted to be transformed into CSAs may contain a multiplication operation. In that case, the multiplication is constrained to be a leaf of the tree.[2] To extend the applicability of CSA transformation across multiplication, we exploit the application of distribution rule to arithmetic expressions if the involved operations are on the critical path of the circuit.

III. The Complete Algorithm

A. Transformation Types

To maximize the effectiveness of the optimization techniques, it is necessary not only to consider the applications of the CSA techniques in Sec. II individually but also to consider the applications of mixtures of them together. Our algorithm employs 6 types of CSA transformation.

type1: *the conventional optimization*[2] - The operation tree to be transformed must be composed entirely of operations of types addition, subtraction, and/or multiplication where the multiplications are leaves of the tree.

type2: OA_{mux}; **type3**: OA_{bound}; **type4**: OA_{mult}

type5: $OA_{mult}+OA_{mux}$ - The preprocessing step of OA_{mult} (i.e., restructuring operation tree by distribution rule) is performed, and subsequently, OA_{mux} is applied to the restructured operation tree.

type6: $OA_{mult}+OA_{bound}$ - The preprocessing step of OA_{mult} is performed, and subsequently, OA_{bound} is applied to the restructured operation tree.

B. Transformation Procedure

Given the operations on the critical path of the circuit, for each transformation type i, (i=1,2,$\cdots$,6), we extract all candidates of operation tree to be transformed by that type. Suppose there are total m_i candidates for type i. We define a cost function which will be used to select the best among the m_i candidates. For each of the m_i candidates, we apply the transformation of the corresponding type and compute the ratio

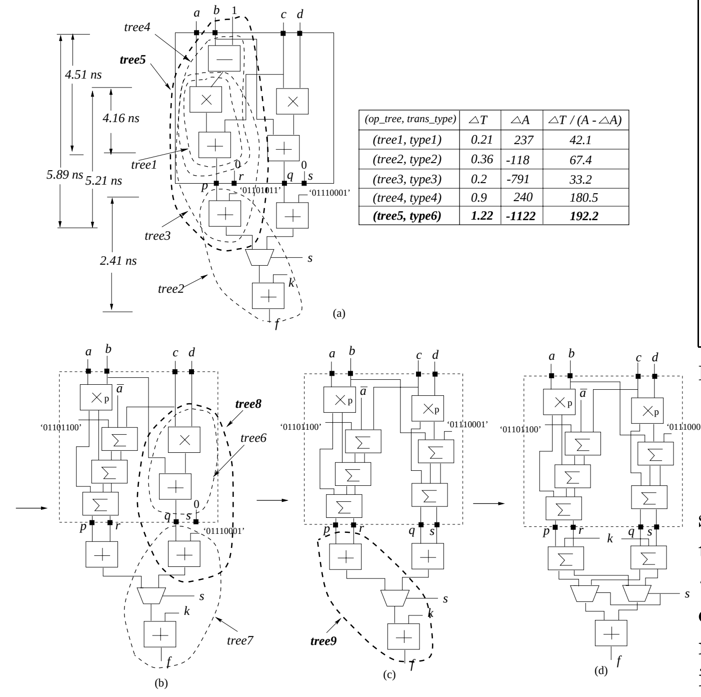
$$\Delta T/(A - \Delta A) \tag{1}$$

where ΔT and ΔA are the amounts of the decrease in timing and area of the resultant circuit, respectively, and A is the area of the circuit prior to the application of the transformation of the type.

The ratio indicates a measure of the effectiveness in decreasing the timing of the critical path by increasing the circuit area. Of the m_i candidates, we shall choose the candidate of operation tree which has the largest ratio of Eq. (1). Again, among the 6 candidates selected, one for each transformation type, we shall choose the candidate of operation tree with the largest ratio, and the operation tree is transformed by the corresponding transformation

[2]We use notation $D(x)$ to denote the delay to x (i.e., arrival time) if x is an operand, and denote the internal delay of the fastest implementation modules of x if x is an operation, such as CSA, addition, multiplication.

type. The process then repeats until the timing of the circuit is less than the specified cycle time or there is no more candidate applicable on the critical path of the circuit.



(op_tree, trans_type)	$\triangle T$	$\triangle A$	$\triangle T / (A - \triangle A)$
(tree1, type1)	0.21	237	42.1
(tree2, type2)	0.36	-118	67.4
(tree3, type3)	0.2	-791	33.2
(tree4, type4)	0.9	240	180.5
(tree5, type6)	**1.22**	**-1122**	**192.2**

Fig. 3. An example of showing the flow of our algorithm

Example: To illustrate the flow of our transformation procedure, we use the circuit graph in Fig. 3(a) where a, b, c, d, k and s are input signals of the design and f is an output signal. The circuit contains one sub-design as a module unit.

From the operations on the critical path, our algorithm extracts the best candidate of operation tree for each of transformation types according to Eq. (1). The entries of (*op_tree*, *trans_type*) in the table of Fig. 3(a) show all such pairs of operation tree and transformation type, and the dotted circles in the circuit graph of Fig. 3(a) show the corresponding operation trees. For example, *(tree5, type6)* in the table indicates that expression $a \cdot (b-1)$ in *tree5* is restructured into $a \cdot b - a$ by the distribution rule of OA_{mult}, which is then transformed by OA_{bound}. The last three columns of the table in Fig. 3(a) summarizes the values of the changes of timing and area, and the values of Eq. (1), respectively.

As shown in the table of Fig. 3(a), 5 pairs of (*op_tree*, *trans_type*) are extracted. Among them we select the pair *(tree5, type6)* with the largest value of Eq. (1). Operation tree *tree5* is then transformed by *type6* as shown in Fig. 3(b). Now, the critical path of the transformed circuit in Fig. 3(b) becomes the one passing through the right side of the circuit. Consequently, 3 (*op_tree*, *trans_type*) pairs are extracted as shown in the circles of the figure. Among them, *tree8* with *type3* is selected and transformed. Fig. 3(c) shows the resultant circuit. Finally, *tree9* with *type2* is selected and transformed into the circuit shown in Fig. 3(d).

```
The Complete Algorithm
/* $A$: area of current circuit */
/* $T$: timing of (critical path of) current circuit */
/* $\Delta A$: area decrease resulting from a transformation */
/* $\Delta T$: timing decrease resulting from a transformation */
• Calculate area/timing of the current circuit
    (using module implementation and logic optimization);
while ($T$ > cycle_time) {
    • Extract all candidates of (op_tree, trans_type);
    • Apply, for each candidate, trans_type to op_tree, and
      calculate changes of area and timing, $\Delta A$ and $\Delta T$
      (using implementation selection only); (statement_a)
    • Select and transform the candidate with the largest
      ratio of $\Delta T/(A-\Delta A)$;
    • Update area/timing of the circuit (statement_b)
      (using implementation sel. and logic opt.);
}
```

Fig. 4. The complete algorithm

Fig. 4 summarizes the flow of the algorithm. To speedup the timing-consuming process of the implementation selection and logic optimization we employ a *local measuring scheme* (statement_a). That is, we consider only the modules of the CSA tree transformed at the current iteration. Since the structure of CSA implementation is very regular, which is a set of disjoint fuller adders, it is relatively easy and also fast to calculate an accurate estimation of timing and area of the transformed tree. In addition, we attempt to remove any possible timing/area discrepancies induced by the local measuring scheme by performing again the steps of implementation selection and logic optimization on the entire circuit at the end of every iteration (statement_b).

As mentioned before, measuring the cost of applying *trans_type* to *op_tree* is based on the area and timing calculation that are done locally on the transformed CSA tree. Our algorithm uses a library of timing and area of the *logic-optimized* possible implementations such as shown in Table I where it contains, for each module type, the values of area and timing for each of area-efficient, timing-efficient, and area/timing-efficient implementations produced by the application of implementation selection and logic-optimization to the module.

IV. Experimental Results

We tested our algorithm on a large number of arithmetic computations that are typically found in industry. 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, those by [2] and those without using CSAs, and compared their results.[3] We used 8-bit operands for the inputs of the multiplication and 16-bit

[3]We used *lcbg10pv* (.35μ) technology[5] for the experimentation.

functions	bit-width	impl. selection + logic opt.		
		area-eff. time, area	moderate time, area	time-eff. time, area
$A + B$	16×16	3.47, 339	2.38, 686	1.37, 1172
A - B	16×16	3.55, 387	2.42, 735	1.47, 1387
$A + B$	8×8	1.71, 163	1.59, 274	0.94, 577
A - B	8×8	1.79, 187	1.60, 284	0.90, 697
$A \times B$	8×8	5.92, 1610	4.49, 2234	3.59, 2820
$A \times_p B$	8×8	-	2.22, 1393	-
$Mux(A,B)$	$n \times n$	-	0.32, 24·n	-
$Inverter(A)$	n	-	0.06, 3·n	-
$CSA(A,B,C)$	n×n×n	-	0.36, 21·n	-

TABLE I
An implementation library for modules to measure the changes of area and timing by local transformation

for the others and assumed that the arrival times of all input operands are 0.

We tested our algorithm on designs with multiplexor as shown in Fig. 5(a) and the results are summarized in the upper row of Table II. We also conducted our experimentation on four design expressions of the form $(a+x) \cdot c - d$ where x is variable b, 39, 7, or 1 and the results are summarized in the middle of Table II. The comparisons indicate that our CSA transformation technique with the operation-distribution can reduce the timing of design further. However, it increases area. Consequently, it is desirable to use the transformation technique selectively only when it is the last choice for reducing the timing of circuit to meet the cycle time. We also conducted our experimentation on the three designs shown in Fig. 5(b). The results shown in the low part of Table II reflect that combining our CSA techniques OA_{bound}, OA_{mux} and OA_{mult} all together becomes a very powerful vehicle to overcome the limitation of the conventional CSA transformations and enables us to extend the applicability of CSA transformation to the entire circuit to produce faster timing and smaller area.

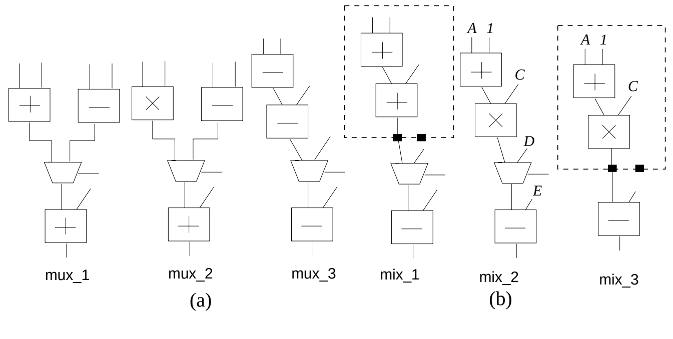


Fig. 5. (a) Designs containing multiplexors; (b) Designs containing multiplexor, sub-design, and/or multiplication of the form $(a + X) \cdot c$

Designs	RTL time/ area	[2] time/ area	Ours time/ area	Impr. over RTL	Impr. over [2]
mux_1	3.26 4742	3.26 4742	2.09 3565	36% 25%	36% 25%
mux_2	4.78 5036	4.78 5036	4.20 4265	12% 15%	12% 15%
mux_3	2.89 5756	2.81 5004	2.55 4115	12% 29%	9% 18%
$(a+b) \cdot c - d$	5.14 5066	4.08 3813	3.75 6052	27% -19%	8% -59%
$(a+39) \cdot c - d$	4.80 4290	3.93 3277	3.66 4360	24% -2%	7% -32%
$(a+7) \cdot c - d$	4.66 4308	3.87 3369	3.58 4033	23% 6%	7% -20%
$(a+1) \cdot c - d$	4.12 4892	3.43 3430	3.35 3217	19% 34%	2% 6%
mix_1	3.41 4456	3.20 4399	2.49 4051	27% 11%	22% 8%
mix_2	4.49 5296	4.27 4676	3.34 5129	26% 3%	22% -10%
mix_3	4.66 4329	4.36 4147	3.33 3607	29% 17%	24% 13%

TABLE II
Comparison of results for arithmetic circuits

V. Conclusions

This paper presented a new algorithm for optimizing arithmetic circuits using CSAs. We overcome some of the limitations of the conventional CSA transformations by proposing a set of effective CSA techniques for *optimizing across multiplexors*, *optimizing across design boundaries* (restricted form of [3]), and *optimizing across multipliers*. Those techniques allow an extensive utilization of CSAs throughout the whole circuit. We then integrated the techniques into our CSA optimization framework to fully exploit their effects in a global way. In addition, we devised a fast, but relatively accurate (bit-level) timing/area estimation scheme in the CSA transformations which in turn solidifies the effectiveness of our algorithm.

Acknowledgments

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, "Circuit Optimization using Carry-Save-Adder Cells", *IEEE TCAD*, October 1998.
[3] J. Um, T. Kim, C. L. Liu, "Optimal Allocation of Carry-Save-Adders in Arithmetic Optimization", *Proc. ICCAD*, 1999.
[4] Synopsys Inc., *DesignWare Components Databook*, 1996.
[5] LSI Logic Inc., *G10-p Cell-Based ASIC Products Databook*, 1996.