Optimizing High Speed Arithmetic Circuits Using Three-Term Extraction

Anup Hosangadi
University of California, Santa Barbara
anup@ece.ucsb.edu

Farzan Fallah
Fujitsu Labs of America, Inc.
farzan@fla.fujitsu.com

Ryan Kastner
University of California, Santa Barbara
kastner@ece.ucsb.edu

Abstract

Carry Save Adder (CSA) trees are commonly used for high speed implementation of multi-operand additions. We present a method to reduce the number of the adders in CSA trees by extracting common three-term subexpressions. Our method can optimize multiple CSA trees involving any number of variables. This optimization has a significant impact on the total area of the synthesized circuits, as we show in our experiments. To the best of our knowledge, this is the only known method for eliminating common subexpressions in CSA structures. Since extracting common subexpressions can potentially increase delay, we also present a delay aware extraction algorithm that takes into account the different arrival times of the signals.

1. Introduction

The increasing complexity of semiconductor devices has led to new directions in the research of Design Automation tools. The implementation of computationally intensive applications is an area that is receiving increasing attention. Modern embedded devices such as portable music and video players, cell phones and video cameras are computationally intensive. System designers often rely on precompiled libraries for implementing arithmetic functions. But such an approach may not be the best option with the available resources and for meeting the tight constraints on area, latency and power consumption.

Multi-operand adder structures are commonly used for the summation of partial products in multiplication, as in the Wallace and Dadda tree multipliers [1]. They are also used in the implementation of arithmetic expressions arising from the conversion of constant multiplications into shifts and additions [2-4]. Carry Save Adders (CSAs) are commonly used to implement these multi-operand adder structures. An n-bit CSA takes three n-bit operands as inputs and produces an n+1-bit carry output and an n-bit sum output. The CSA is made up of n Full Adder (FA) cells, where each FA takes in one bit from each of three inputs and produces a sum bit and a carry bit. This is illustrated in Figure 1. Since there is no carry propagation this addition is very fast.

A CSA tree is made up of a tree of CSAs, taking N inputs and then reducing it to two numbers, which can then be added using a fast adder such as a Carry Lookahead Adder (CLA). As an example of CSA trees, consider the set of expressions arising from a matrix multiplication, shown in Figure 2. The CSA trees for these expressions are shown in Figure 3.

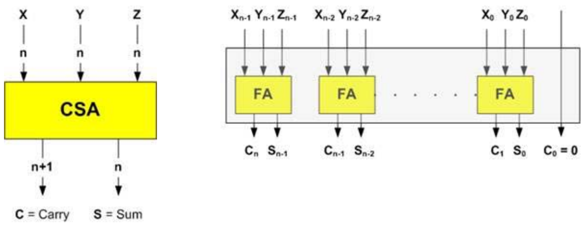

Fig 1. Carry Save Adder (CSA)

$$\begin{bmatrix} Y_1 \\ Y_2 \end{bmatrix} = \begin{bmatrix} 5 & 7 \\ 4 & 12 \end{bmatrix} \begin{bmatrix} X_1 \\ X_2 \end{bmatrix}$$

$$Y_1 = X_1 + X_1{<<}2 + X_2 + X_2{<<}1 + X_2{<<}2$$
$$Y_2 = X_1{<<}2 + X_2{<<}2 + X_2{<<}3$$

Fig 2. Example arithmetic expressions

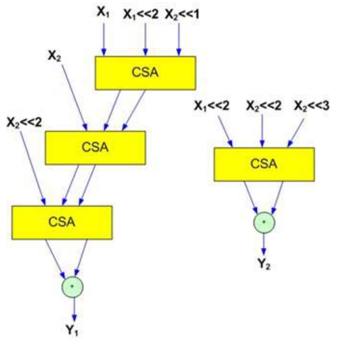

Fig 3. CSA trees for example expressions

Each CSA can be seen as a 3:2 compressor, since it takes three inputs and reduces it to two numbers. The CSA tree reduces N numbers to two numbers and can be considered as two separate trees of 3:2 compressors, each reducing $\left\lceil \frac{N}{2} \right\rceil$ numbers to one number. The height of the CSA tree can hence be described by the formula [1]

$$\text{Delay (N)} = \left\lceil \log_{1.5} \left\lceil \frac{\mathbf{N}}{2} \right\rceil \right\rceil \qquad \text{(I)}$$

If we observe the expressions carefully, we can see that there is a common three-term expression $D_1 = X_1 + X_2 + X_2{<<}1$. By extracting this subexpression, we can reduce the number of CSAs by one, as shown in Figure 4.

There is no known automated technique for performing such an optimization. In this paper, we present a technique for extracting three-term common subexpressions aimed at reducing the number of CSAs and thereby area. Our technique can be used for optimizing multiple expressions involving any number of variables. The rest of the paper is organized as follows. In Section 2, we present some related work on the optimization of CSA structures. In Section 3, we present our algorithm for extracting three-term common subexpressions. In Section 4, we present experimental results, where we observe the effect of our optimizations on the area and latency after logic synthesis and place and route. In Section 5, we present a delay aware optimization algorithm, where we consider the different arrival times of the signals. Finally we conclude the paper in Section 6.

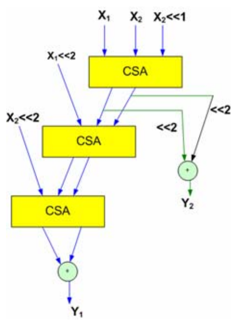

Fig 4. CSA trees after extracting a common subexpression

2. Related Work

The most recent work on using CSAs is in [2], where the authors have developed a technique to restructure dataflow graphs in programs to cluster as many arithmetic operators as possible and then implement them using CSA trees. The major contribution of the paper was in making most use of CSA structures for high speed, but there was no attempt in optimizing the CSA structures. [5] is one of the early works in this area, and presents a set of transformations to transform a dataflow graph to maximize the utilization of CSAs. In [6], an optimal algorithm for designing a CSA structure with shortest delay is presented, given the arrival times of all inputs.

In [7], algorithms for timing and low power driven synthesis of CSA structures are presented, where the optimizations are done at bit-level. For low power synthesis, the authors consider switching probabilities of the inputs, and allot the inputs to the CSAs to minimize the switching probabilities of various signals. In [4], some layout driven optimizations like converting half adders and merging them into full adders is suggested for layout improvement.

None of these works explore redundancy elimination as an optimization technique thought it can have a significant impact on the total area of circuits, as we show in our experimental results. Our techniques for redundancy elimination can be combined with the arithmetic clustering techniques in [2] and [5] for optimizations across design boundaries and non-arithmetic operators. There has been a lot of work on redundancy elimination for two input operators [8-10], but none of them has been applied to multi-input operators such as CSAs.

3. Three-Term Extraction Algorithm

We use a polynomial transformation of the arithmetic expressions that helps us to perform our optimizations. This polynomial transformation has been used for optimizing multiple-variable linear systems using two-input adders [9, 11]. We first generate a set of all potential three-term common subexpressions, which we call as **divisors**. We then use an iterative algorithm for eliminating common subexpressions.

3.1 Polynomial transformation of linear systems

Using the given representation of the constant C, the multiplication with the variable X (assuming only fixed point representation) can be represented as a summation of terms denoting the decomposition of the constant multiplication into additions and shifts as

$$C{*}X = \sum_i \pm \mathbf{XL^i} \quad \text{(II)}$$

The terms can be either positive or negative when the constants are represented using signed digit representations such as the Canonical Signed Digit (CSD) representation. The exponent of L represents the magnitude of the left shift and the i's represent the digit positions of the non-zero digits of the constants. For example the multiplication $7{*}X = (100\text{-}1)_{CSD}{*}X = X{<<}3 - X = XL^3 - X$, using the polynomial transformation. This polynomial transformation helps to find subexpressions involving any number of variables. Furthermore, it helps to detect common subexpressions that are shifted forms of each other. For example $(X_1 + X_2 + X_3)$ and $(X_1{<<}2 + X_2{<<}2 + X_3{<<}2)$ are equivalent except for the common shift by 2.

The example expressions in Figure 2 can be rewritten using our polynomial transformation as shown in Figure 5.

$$\boxed{\begin{aligned} Y_1 &= X_1 + X_1L^2 + X_2 + X_2L + X_2L^2 \\ Y_2 &= X_1L^2 + X_2L^2 + X_2L^3 \end{aligned}}$$

Fig 5. Polynomial transformation of example expressions

3.2 Three-term divisor extraction

We extract divisors for every expression by considering every combination of three terms and then dividing by the minimum exponent of L. For example, consider the expression Y_2 in Figure 5. The minimum exponent of L in this case is L^2. Dividing by L^2 gives us the divisor $d = X_1 + X_2 + X_2L$. The number of divisors in an expression with N terms is $\binom{N}{3}$. Figure 6 shows the algorithm for extracting three-term divisors. The importance of these three-term divisors is illustrated by the following theorem.

Theorem: There exists a three-term common subexpression iff there exists a non-overlapping intersection among the set of three-term divisors.

This theorem states that there is a three-term common subexpression if and only if there are at least two non-overlapping divisors that intersect. Two divisors are said to **intersect**, if their absolute values are equal. For example, $(X_1 + X_2 + X_2{<<}1)$ intersects with $(X_1 + X_2 + X_2{<<}1)$. Two divisors are said to be **overlapping** if at least one of the terms from which they are derived is common. For example, consider the expression $F = X_1 + X_1{<<}2 + X_1{<<}4 + X_1{<<}6 = X_1 + X_1L^2 + X_1L^4 + X_1L^6$. From the first three terms we obtain the divisor $d_1 = X_1 + X_1L^2 + X_1L^4$. From the last three terms we obtain the divisor $d_2 = (X_1 + X_1L^2 + X_1L^4)$, by dividing those three terms by L^2. Even though these divisors (d_1 and d_2) intersect, they are said to overlap since two of the terms from which they are derived are common.

```
Divisors({P_i})
{
    {P_i} = Set of expressions in polynomial form;
    {D} = Set of divisors and co-divisors = {Φ};

   for (every expression P_i in {P_i})
   {
     for (every combination of 3 terms ($t_i$, $t_j$, $t_k$) in P_i)
     {
       MinL = Minimum exponent of L in($t_i$, $t_j$, $t_k$); // co-
divisor
         $t_i^1$ = $t_i$/MinL;
         $t_j^1$ = $t_j$/MinL;
         $t_k^1$ = $t_k$/MinL;
         d = ($t_i^1 + t_j^1 + t_k^1$); // divisor;
         {D} = {D} ∪ (d, MinL);
       }
   }
  return {D};
}
```

Figure 6. Algorithm for generating three-term divisors

The proof of this theorem is quite straightforward.

Proof:

(If case): If M of the divisors in the set of all divisors intersect and are non-overlapping, then there are M instances of the same three-term expression in the set of expressions.

(Only If case): Suppose there are M non-overlapping instances of the same three-term expression $d_1 = (t_i + t_j + t_k)$ among the set of expressions. Now consider two different cases. In the first case, assume the minimum exponent of L among the terms in d_1 is 0. Then d_1 satisfies the definition of a divisor. Since our divisor generation algorithm extracts all possible three-term divisors, there will be M non-overlapping divisors representing d_1.

In the second case, assume that d_1 does not satisfy the definition of a divisor (i.e., there are no terms in d_1 with a zero exponent of L). Then we have $d_1' = (t_i' + t_j' + t_k')$ which is obtained by dividing each term in d_1 by the minimum exponent of L. Now d_1' satisfies the definition of a divisor, and reasoning as above, there will be M non-overlapping divisors representing d_1'.

3.3 Iterative common subexpression elimination algorithm

Figure 7 shows our algorithm for three-term extraction. In the first step, **frequency statistics** of all distinct divisors is computed and stored. By frequency statistics we mean the number of instances of each distinct divisor. This is done by generating divisors $\{D_{new}\}$ for each expression and looking for intersections with the existing set $\{D\}$. For every intersection, the frequency statistic of the matching divisor d_1 in $\{D\}$ is updated and the matching divisor d_2 in $\{D_{new}\}$ is added to the list of intersecting instances of d_1. The unmatched divisors in $\{D_{new}\}$ are then added to $\{D\}$ as distinct divisors.

```
Optimize ({P_i})
{
    {P_i} = Set of expressions in polynomial form;
    {D} = Set of divisors = φ ;
  // Step 1. Creating divisors and their frequency statistics
   for each expression P_i in {P_i}
   {
       {D_new} = Divisors(P_i);
       Update frequency statistics of divisors in {D};
       {D} = {D} ∪ { D_new};
  }

  //Step 2. Iterative selection and elimination of best divisor
  while (1)
  {
      Find d = divisor in {D} with most number
               of non-overlapping intersections;
       if (d == NULL) break;
      Rewrite affected expressions in {P_i} using d;

      Remove divisors in {D} that have become invalid;

      Update frequency statistics of affected divisors;
      {D_new} = Set of new divisors from new terms added
                 by division;
       {D} = {D} ∪ {D_new};
      }
  }
```

Figure 7. Algorithm for three-term extraction

In the second step of the algorithm, the best three-term divisor is selected and eliminated at each iteration. The best divisor is the one that has the most number of non-overlapping divisor intersections. Those expressions that contain this best divisor are then rewritten. Since each CSA produces two outputs, a sum and a carry, each divisor also produces two numbers representing the two outputs. Figure 8 shows the rewriting of the expressions after the selection of the subexpression $D_1 = X_1 + X_2 + X_2<<1$, where D_1 is the extracted divisor, and D_1^S and D_1^C represent the sum and the carry outputs of D_1, respectively.

After selecting the best divisor, those divisors that overlap with it, no longer exist and have to be removed from the dynamic list {D}. As a result the frequency statistics of some divisors in {D} will be affected, and the new statistics for these divisors is computed and recorded. New divisors are generated for the new terms formed during division of the expressions. The frequency statistics of the new divisors are computed separately and added to the dynamic set of divisors {D}.

The algorithm terminates when there are no more useful divisors. For our example expressions, after rewriting the expressions as shown in Figure 8, the set of dynamic divisors {D} is updated. No more useful divisors are found after this, and the algorithm terminates. The optimized circuit is shown in Figure 4.

$$D_1 = X_1 + X_2 + X_2<<1$$
$$Y_1 = (D_1^S + D_1^C) + X_1<<2 + X_2<<2$$
$$Y_2 = (D_1^S + D_1^C)<<2$$

Figure 8. Expression rewriting

Algorithm complexity and quality: The algorithm spends most of its time in the first step where the frequency statistics of all distinct divisors are computed and stored. For an expression with N terms, the number of 3-term divisors is $\Theta(N^3)$. Therefore, the complexity of the first step, for the case of M expressions is $\Theta(MN^3)$. In the second step of the algorithm, each time a divisor is selected, the number of terms in the affected divisor is reduced by one. In the worst case, all expressions are reduced from N terms to two terms at the end of the algorithm. The number of steps to reduce from N terms to two terms is (N-2). Since there are M expressions, the complexity of this step is $\Theta(MN)$. The algorithm that we presented is a greedy heuristic. To the best of our knowledge, there has not been any work on finding the optimal solution to the common subexpression elimination problem. An optimal solution can be achieved by a brute force search of the entire space, where all possible subexpressions in different selection orderings are explored. Since this is impractical for even moderately sized examples, we do not compare our results with the optimal one.

4. Experimental Results

The goal of our experiments was to observe the impact of our optimizations on the total area and delay of the synthesized examples. We considered six examples, the H.264 Video transform [12], 8-point Discrete Cosine Transform (DCT), 8-point Inverse Discrete Cosine Transform (IDCT) and three Finite Impulse Response (FIR) filters (6-tap, 20-tap and 41-tap) [13]. All these examples consist of a number of multiplications with constants, which we decompose into additions and shift operations. We view these examples as a set of arithmetic expressions of the form shown in Figure 5 after using the polynomial transformation. We then optimize these expressions by doing three-term extraction using the algorithms described in Section 3.

Table 1 compares the number of CSAs for both the unoptimized and optimized expressions, calculated at the end of our algorithm. From these results it can be seen that the number of CSAs reduces on an average by **38.4%.** We built CSA trees for both the unoptimized and optimized expressions and generated Verilog code for them. We considered the implementation of these CSA trees both as a Standard Cell based design as well as on Field Programmable Gate Arrays (FPGAs). For the standard cell designs, we used a 0.25 µm technology library.

Table 1. Comparing number of CSAs

Example	Original # CSAs	Optimized # CSAs	Reduction %
H.264	105	78	25.7
DCT8	266	222	16.5
IDCT8	88	34	16.7
6-tap FIR	22	11	50.0
20-tap FIR	88	34	61.4
41-tap FIR	198	79	60.1
Average	**152.2**	**103.2**	**38.4**

The area and delay estimates after synthesis is compared in Table 2. Due to limitations of the tools available to us, we could not get the numbers after placement and routing. But from the post synthesis phase, we can observe significant reductions in the total area (average **32.7%).** The delay increases a little bit for all cases, but it is only 3.7% on the average.

Table 2. Area and delay for Standard cell designs

Example	Area (units)		Delay (ns)		% Reduction	
	Orig	Opt	Orig	Opt	Area	Delay
H.264	627	497	9.9	10.5	20.7	-6.0
DCT8	1650	1345	11.6	12.2	18.5	-5.8
IDCT8	1417	113	11.6	12.1	16.5	-4.0
6 tap FIR	188	107	10.3	11.0	43.1	-6.4
20 tap FIR	719	364	12.1	11.4	49.4	+5.6
41 tap FIR	1558	812	11.2	11.8	47.9	-5.4
Average	**1027**	**718**	**11.1**	**11.5**	**32.7**	**-3.7**

We also implemented these expressions using FPGAs. In addition to synthesis, we also performed place and route of the circuits. Table 3a shows the reduction in the number of occupied slices and Look Up Tables (LUTs), for the same set of six examples. The results show an average reduction of 14.1% in the total number of slices occupied, and about 12.9% reduction in the total number of LUTs used after optimizing the circuits with our extraction algorithm. In table 3b, we compare the critical path of the unoptimized and optimized designs.

Table 3a. Area comparison for FPGA

Example	# of Slices		# of LUTs		% Reduction	
	Orig	**Opt**	**Orig**	**Opt**	**Slices**	**LUTs**
H.264	1186	1090	2146	2007	8.1	6.8
DCT8	3202	2917	5929	5429	8.9	8.4
IDCT8	2786	2597	5133	4814	6.8	6.2
6 tap FIR	184	163	338	303	11.4	10.4
20 tap FIR	1533	867	2522	1548	43.4	38.6
41 tap FIR	1528	1440	2560	2379	5.8	7.1
Average	**1736**	**1512**	**3105**	**2747**	**14.1**	**12.9**

Table 3b. Latency comparison for FPGA

Example	Original (ns)	Optimized (ns)	% Reduction
H.264	11.6	13.1	-12.5
DCT8	26.8	28.4	-5.8
IDCT8	24.2	26.9	-11.5
6 tap FIR	13.1	13.1	0.0
20 tap FIR	15.9	16.7	-5.1
41 tap FIR	15.3	15.2	0.6
Average	**17.8**	**18.9**	**-5.7**

5. Delay Aware Extraction

The three-term extraction algorithm presented in Section 3 did not consider the impact of the optimizations on the total delay of the CSA tree. But performing extraction among the expressions can create certain dependencies among the signals that can cause the overall delay to increase. This delay can be reduced by reversing some of the optimizations using algorithms such as Tree Height Reduction (THR) [14], but these algorithms involve extensive backtracking and hence are very expensive. Instead, the delay can be controlled during the extraction algorithm.

5.1 Delay model

We use a simple delay model similar to the one used in [6]. We use a unit delay for both the sum and the carry outputs of a CSA, and use integer numbers for the arrival times of the various signals in the circuits. This model can be easily generalized to handle actual values for arrival times and delays of the CSAs. The authors in [6] present an optimal polynomial time algorithm for finding the fastest CSA tree for every expression. This algorithm is an iterative algorithm where in each step the terms of the expression are sorted according to non-decreasing availability times. The first three terms are then allotted to a CSA. This continues till only two terms remain.

We use this algorithm to find the minimum delay of given expressions, using our delay model. We then perform extraction, such that at each step, the delay of the expressions does not exceed this minimum delay.

5.2 Example

Consider the evaluation of the following arithmetic expressions,

$F_1 = a + b + c + d + e$

$F_2 = a + b + c + d + f$

Arrival times (a,b,c,d,e,f) = {2,0,0,0,0,0}

All signals are available at time t = 0, except for a, which is available at time t = 2. Using the optimal CSA allocation algorithm in [6], the minimum delay for both F_1 and F_2 is calculated as **3 + D(Add)**, where D(Add) is the delay of the final two input adder.

Figure 9a shows the evaluation of the two expressions after performing delay ignorant extraction. The arrival times of signals are shown along the edges of the circuit. In this example, the subexpression $D_1 = (a + b + c)$ is first extracted and then the subexpression $D_2 = D_1^S + D_1^C + d$ is extracted. This leads to an implementation with only four CSAs, but the delay of the circuit is now **5 + D(Add)**, which is two units more than the optimal delay.

Figure 9b shows the result of delay aware extraction. Here the subexpression (a + b + c) is not extracted because by doing so the delay increases. The divisor $D_1 = (b+c+d)$ does not increase the delay so it is extracted. After rewriting the expressions, the common subexpression $(D_1^S + D_1^C + a)$ is considered, but is not selected because it increases the delay. The delay aware extraction has one more CSA than the delay ignorant one, but it has the minimum delay.

$D_1 = a + b + c$
Delay(D_1) = 3
$F_1 = D_1^S + D_1^C + d + e$
$F_2 = D_1^S + D_1^C + d + f$

Delay(F_1,F_2) = 5 + D(Add)

$D_2 = D_1^S + D_1^C + d$
Delay(D_2) = 4
$F_1 = D_2^S + D_2^C + e$
$F_2 = D_2^S + D_2^C + f$

Delay(F_1,F_2) = 5 + D(Add)

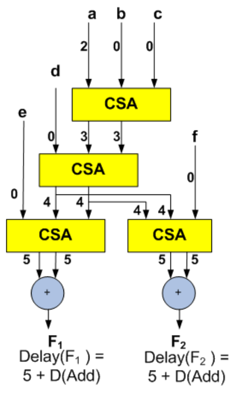

Figure 9a. Delay ignorant extraction

$D_1 = b + c + d$
Delay(D_1) = 1
$F_1 = D_1^S + D_1^C + e + a$
$F_2 = = D_1^S + D_1^C + f + a$

Delay(F_1,F_2) = 3 + D(Add)

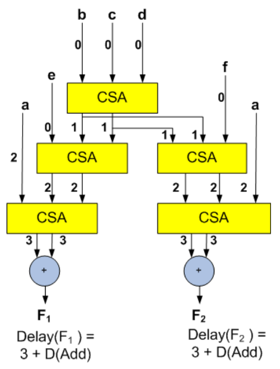

Figure 9b. Delay aware extraction

5.3 Algorithm

The delay aware extraction algorithm is a modification of the original algorithm that does not consider delay. In the algorithm shown in Figure 6, instead of finding the divisor that has the most number of non-overlapping instances, the divisor that has the most number of non-overlapping instances that do not increase the minimum delay is selected. This requires that the delay be calculated for every candidate divisor. The complexity of calculating the delay of an expression using the algorithm in [6] is quadratic in the number of terms in the expression.

5.4 Results

We experimented with the same set of examples that we presented in Section 4. Using the delay model described in 5.2, we compared the delay and the number of CSAs produced by the delay ignorant algorithm (I) and the delay aware algorithm (II). The results are presented in Table 4.

From the results, one can see that the delay ignorant algorithm produces the least number of CSAs, but the delay is increased in all examples. The delay aware algorithm produces the optimal delay according to our delay model and [6], but due to the selective extraction of common subexpressions, the number of CSAs is increased by an average of **15.5%** over the delay ignorant algorithm. The delay aware algorithm still reduces the number of CSAs by **31.1%** over the original unoptimized expressions, for the same delay. The average CPU time for the delay aware algorithm is 11.4s, compared to 2.05s for the original algorithm

Table 4. Comparing delay ignorant (I) and delay aware (II) extraction

Example	# CSAs		Delay		CPU time (s)	
	(I)	(II)	(I)	(II)	(I)	(II)
H.264	78	79	9	8	0.2	1.95
DCT8	222	232	14	13	8.5	44.9
IDCT8	34	201	14	13	3.3	20.9
6 tap FIR	11	15	5	4	0.01	0.03
20 tap FIR	34	45	6	5	0.04	0.16
41 tap FIR	79	91	6	5	0.26	0.7
Average	**103.2**	**110.5**	**9**	**8**	**2.05**	**11.4**

6. Conclusions

We presented optimization techniques for high speed arithmetic circuits aimed at reducing the total area. Our optimization was achieved by extracting common three-term subexpressions and thereby reducing the number of Carry Save Adders. This is the only known method for performing such an optimization. We observed significant reductions in the total area after synthesizing the benchmark examples. We also modified our main algorithm to control the delay while performing the three-term extraction.

Acknowledgement

We would like to thank Tom Sidle at Fujitsu Laboratories of America for his assistance in performing this research

References

[1] B. Parhami, *Computer Arithmetic: Algorithms and Hardware Designs*: Oxford University Press, 2000.

[2] A. K. Verma and P. Ienne, "Improved use of the carry-save representation for the synthesis of complex arithmetic circuits," presented at International Conference on Computer Aided Design (ICCAD), 2004.

[3] T. Kim and J. Um, "A timing-driven synthesis of arithmetic circuits using carry-save-adders," presented at Design Automation Conference, 2000. Proceedings of the ASP-DAC 2000. Asia and South Pacific, 2000.

[4] J. Um and T. Kim, "Layout-aware synthesis of arithmetic circuits," presented at Design Automation Conference (DAC) , 2002. Proceedings. 39th, 2002.

[5] T. Kim, W. Jao, and S. Tjiang, "Arithmetic optimization using carry-save-adders," presented at Design Automation Conference (DAC), 1998. Proceedings, 1998.

[6] J. Um, T. Kim, and C. L. Liu, "Optimal allocation of carry-save-adders in arithmetic optimization," presented at (ICCAD) Computer-Aided Design, 1999. Digest of Technical Papers. 1999 IEEE/ACM International Conference on, 1999.

[7] J. Um, T. Kim, and C. L. Liu, "A fine-grained arithmetic optimization technique for high-performance low-power data path synthesis," presented at Design Automation Conference (DAC), 2000. Proceedings 2000. 37th, 2000.

[8] A.Hosangadi, F.Fallah, and R.Kastner, "Common Subexpression Involving Multiple Variables for Linear DSP Synthesis," presented at IEEE International conference on Application Specific Architectures and Processors (ASAP), Galveston, TX, 2004.

[9] A.Hosangadi, F.Fallah, and R.Kastner, "Reducing Hardware Complexity of Linear DSP Systems by Iteratively Eliminating Two Term Common Subexpressions," presented at IEEE/ACM Asia South Pacific Design Automation Conference (ASP-DAC), Shanghai, China, 2005.

[10] M.Potkonjak, M.B.Srivastava, and A.P.Chandrakasan, "Multiple Constant Multiplications: Efficient and Versatile Framework and Algorithms for Exploring Common Subexpression Elimination," *IEEE Transactions on Computer Aided Design of Integrated Circuits and Systems*, 1996.

[11] A.Hosangadi, F.Fallah, and R.Kastner, "Reducing Hardware complexity by iteratively eliminating two term common subexpressions," presented at Asia South Pacific Design Automation Conference (ASP-DAC), 2005.

[12] I. E. G. Richardson, *H.264 and MPEG-4 Video Compression*: John Wiley and Sons, 2003.

[13] S.K.Mitra, *Digital Signal Processing: A computer based approach*, second ed: McGraw-Hill, 2001.

[14] A. Nicolau and R. Potasman, "Incremental tree height reduction for high level synthesis," presented at Design Automation Conference, 1991. 28th ACM/IEEE, 1991.