Reducing Hardware Complexity of Linear DSP Systems by Iteratively Eliminating Two-Term Common Subexpressions

Anup Hosangadi
ECE Department
University of California, Santa Barbara
Santa Barbara, CA 93106, USA
anup@ece.ucsb.edu

Farzan Fallah
Advanced CAD Research
Fujitsu Laboratories of America
Sunnyvale, CA 94085, USA
farzan@fla.fujitsu.com

Ryan Kastner
ECE Department
University of California, Santa Barbara
Santa Barbara, CA 93106, USA
kastner@ece.ucsb.edu

Abstract – **This paper presents a novel technique to reduce the number of operations in Multiplierless implementations of linear DSP transforms, by iteratively eliminating two-term common subexpressions. Our method uses a polynomial transformation of linear systems that enables us to eliminate common subexpressions consisting of multiple variables. Our algorithm is fast and produces the least number of additions/subtractions compared to all known techniques. The synthesized examples show significant reductions in the area and power consumption.**

1. INTRODUCTION

Most of the present generation portable embedded systems like mobile phones, video cameras etc. perform some kind of continuous signal (audio, video) processing (DSP) which is computationally very expensive. These computations are frequently implemented in custom hardware to meet the dual requirements of performance and power consumption. It is important to develop techniques that can do a good optimization at the behavioral level, since optimizations done at this level have the greatest impact on reducing hardware complexity and power consumption [1, 2]. Furthermore, it is important to develop techniques that are computationally efficient, so that they can be integrated into high level synthesis frameworks. Unfortunately current optimization techniques suffer from some drawbacks because of which they cannot take full advantage of the opportunities available at the behavioral level.

Many operations in DSP can be expressed in the form of a multiplication of a constant matrix with a vector of input samples X of the form shown in Equation I, where $c_{i,j}$ represents the (i,j)th element in an NxN constant matrix.

$$\mathbf{Y[i]} = \sum_{j=0}^{N-1} \mathbf{c_{i,j} * X[j]} \qquad \textbf{(I)}$$

Transforms of this type include Discrete Cosine Transform (DCT), Discrete Fourier Transform (DFT) etc, which are widely used in image, video and audio processing. From Equation I, it can be seen that most of these DSP computations consist of a number of multiplications with constants. However, since the values of the constants are known beforehand, the constant multiplications can be efficiently implemented in hardware as a series of hardwired shifts and additions [3-9]. Finding common subexpressions among these set of additions can reduce the hardware complexity and power consumption of these transform implementations. All the known techniques for finding common subexpressions [3-9] suffer from certain drawbacks because of which they cannot do a good optimization of matrix forms of linear systems (Equation I).

As a motivational example, consider the linear system shown in Figure 1, which is the integer transform used in the latest H.264 video coding standard [10, 11].

$$\begin{bmatrix} Y_0 \\ Y_1 \\ Y_2 \\ Y_3 \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 2 & 1 & -1 & -2 \\ 1 & -1 & -1 & 1 \\ 1 & -2 & 2 & -1 \end{bmatrix} \begin{bmatrix} X_0 \\ X_1 \\ X_2 \\ X_3 \end{bmatrix}$$

Figure 1. Integer transform used in H.264 video encoding

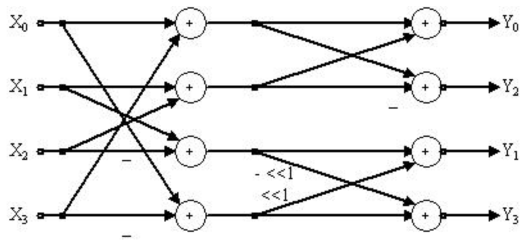

Figure 2. Implementing H.264 transform after using our technique

By decomposing the constant multiplications into shifts and additions and extracting common subexpressions using the technique we present in this paper, we have an implementation using only eight additions/subtractions as shown in Figure 2. It should be noted that this is the same implementation that is reported in [11], where the result is obtained manually. To the best of our knowledge there is no known technique that can produce an implementation with eight or fewer additions/subtractions.

In this paper, we first present a polynomial transformation of the linear computations. We then present an algorithm that iteratively selects and eliminates two-term common subexpressions. We show how this algorithm overcomes the drawbacks of other known methods, which we verify by experimental results. We present some related work on reducing complexity of linear systems in Section 2. In Section 3, we present our polynomial transformation of the linear systems and the algorithm to eliminate common subexpressions. We present experimental results in Section 4. We finally conclude and talk about future work in Section 5.

2. RELATED WORK

There have been a number of papers on the problem of constant multiplication optimization for linear systems, particularly for digital filters [3-9]. In this section we present a brief summary of some of these works and how they compare with our technique.

$F_1 = X_1 + X_2 + X_3$ $\quad$ $D_1 = X_1 + X_2$

$F_2 = X_1{<<}1 + X_2{<<}1 + X_3$ $\quad \Rightarrow \quad$ $F_1 = D_1 + X_3$

$F_2 = D_1{<<}1 + X_3$

(a) Searching for shifted forms of common subexpressions

$F_1 = X_1 - X_2 + X_3$ $\quad$ $D_1 = X_1 - X_2$

$F_2 = X_3 + X_2 - X_1$ $\quad \Rightarrow \quad$ $F_1 = D_1 + X_3$

$F_2 = X_3 - D_1$

(b) Considering common subexpressions with different signs .

Figure 3. Optimizations not performed by most other techniques

In [3], a scaling transformation of the constants followed by a bipartite matching algorithm was presented. This algorithm cannot find all possible digit patterns, since it does not consider common patterns which have different digit positions, which can be obtained by looking at different shifted forms of the constants. The authors also present a post processing step for the matrix form of linear systems where the expressions after the first optimization step (bipartite matching) are viewed as bit patterns, and the number of additions is further reduced by finding common bit patterns among these bit patterns. The main drawback of this method is that it does not consider the elimination of multiple variable common subexpressions together with single variable common subexpressions leading to inferior results. Furthermore, this technique cannot find common subexpressions that are shifted (Figure 3a) and those that have their signs reversed (Figure 3b). [4-6] present different techniques for finding common subexpressions in multiple constant multiplications by generating common digit patterns multiplying a single variable at a time. Since these techniques do not consider multiple variable common subexpressions, they do not generate good results.

In [9], a polynomial transformation of the constant multiplications is also used, which enables the detection of multi-variable common subexpressions using a rectangle covering of an intersection matrix. Each common subexpression in this formulation appears as a rectangle, where all the terms in the same column are required to have the same signs. This prevents common subexpressions with reversed signs to be detected. Signed digit representations such as Canonical Signed Digit (CSD) are very popular because they generally provide fewer additions/subtractions than two's complement representations [12]. The usage of signed digits implies that cases such as the one in Figure 3b are very common. Therefore, [9] would miss a number of such opportunities. Another disadvantage of [9] is that an exponential number of prime rectangles have to be investigated to find the best prime rectangle. Hence, a heuristic ping-pong algorithm has to be used to get a good prime rectangle, which could lead to inferior results in some cases.

In this paper we assume the representation for constants such as two's complement or CSD is given. We do not consider the effect of matrix splitting or scaling of constants [3, 4] since our technique deals with finding common subexpressions with a given constant matrix.

3. OUR TECHNIQUE

In this section, we present algorithms for the detection and elimination of common subexpressions in multiple variable linear computations. We based our algorithm on the methods originally developed for the concurrent decomposition and factorization of Boolean expression [13]. This technique finds a set of two cube (term) divisors and chooses the one that will save the most number of literals, in each iteration. We made the following changes to these methods to adapt them for the problem of optimizing linear DSP computations:

- The original methods [13, 14] work on a set of linear algebraic expressions. We use a polynomial transformation of the linear computations to handle the shift operations.
- The divisor extraction procedure in the original algorithm obtains divisors by dividing by the common literal (variable) support in every pair of cubes (terms) in a Boolean expression. Our algorithm divides by the common shift between every pair of two terms in the expression (explained in Section 3.2)
- The original methods work on algebraic expressions that do not have any subtractions. Since we encounter subtractions in linear DSP computations, we can do a further optimization

that can detect common subexpressions that have their signs reversed

3.1 Polynomial transformation

Using a given representation of the constant C, the multiplication with the variable X (assuming only fixed point representation) can be represented as

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

where L represents the left shift from the least significant digit and the i's represent the digit positions of the non-zero digits of the constant, 0 being the digit position of the least significant digit. Each term in the polynomial can be positive or negative depending on the sign of the non-zero digit. For example the constant multiplication $(6)_{decimal}$ *X = $(10\text{-}10)_{CSD}$\*X = $XL^3 - XL$. In the case of real constants represented in fixed point, the constant can be converted into an integer and the final result can be corrected by shifting right. For example, the constant multiplication $(0.101)_{binary}*X = (101)_{binary}*X*2^{-3} = (X + XL^2)*2^{-3}$. The linear system in Figure 1 can be transformed using Equation II as shown in Figure 4.

$$\begin{aligned}
Y_0 &= X_0 + X_1 + X_2 + X_3 \\
Y_1 &= X_0L + X_1 - X_2 - X_3L \\
Y_2 &= X_0 - X_1 - X_2 + X_3 \\
Y_3 &= X_0 - X_1L + X_2L - X_3
\end{aligned}$$

Figure 4. Polynomial representation of linear system shown in Figure 1.

3.2 Generating two-term divisors

Figure 5 shows the algorithm Divisors that is used to generate two-term divisors. A **two-term divisor** of a polynomial expression is the result obtained after diving any two terms of the expression by their least exponent of L. This is equivalent to factoring by the common shift between the two terms. Therefore, the divisor is guaranteed to have at least one term with a zero exponent of L. A **co-divisor** of a divisor is the exponent of L that is used to divide the terms to obtain the divisor. A co-divisor is useful in dividing the original expression if the divisor corresponding to it is selected as a common subexpression.

As an illustration of the divisor generating procedure, consider the expression Y_1 in Figure 4. Consider the terms X_0L and $-X_3L$. The minimum exponent of L in both these terms is L. Therefore, after dividing by L, we obtain the divisor $(X_0 - X_3)$ with co-divisor L. The other divisors generated for Y_1 are $(X_0L + X_1)$, $(X_0L - X_2)$, $(X_1 - X_2)$, $(X_1 - X_3L)$ and $(-X_2 - X_3L)$. All these divisors have co-divisors 1.

The importance of these two-term divisors is illustrated by the following theorem.

Theorem: There exists a multiple term common subexpression in a set of expressions *if and only if* there exists a non-overlapping intersection among the set of divisors of the expressions.

```
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 pair of terms (t_i, t_j) in P_i)
        {
            MinL = Minimum exponent of L in (t_i, t_j); // co-divisor
            t_i^1 = t_i/MinL;
            t_j^1 = t_j/MinL;
            d = (t_i^1 + t_j^1); // divisor;
            {D} = {D} ∪ (d, MinL);
        }
    }
    return {D};
```

Figure 5. Algorithm to generate two-term divisors

This theorem basically states that there is a common subexpression in the set of polynomial expressions representing the linear system, if and only if there are at least two non-overlapping divisors that intersect. Two divisors are said to be **intersecting** if their absolute values are equal. For example, $(X_1 - X_2L)$ intersects both $(-X_2L + X_1)$ and $(X_2L - X_1)$. Two divisors are considered to be **overlapping** if one of the terms from which they are obtained is common. For example consider the following constant multiplication $(10101)_{binary}$*X, which is transformed to ${}_{(1)}X + {}_{(2)}XL^2 + {}_{(3)}XL^4$ in our polynomial representation. The numbers in parenthesis represent the term numbers in this expression. Now according to the divisor generating algorithm, there are two instances of the divisor $(X + XL^2)$ involving the terms (1, 2) and (2, 3) respectively. Now these divisors are said to overlap since they contain the term 2 in common. Two divisors are said to intersect, if they are the same, with or without reversing the signs of the terms. For example the divisor $(X_1 - X_2L)$ intersects with both $(X_1 - X_2L)$ and $(-X_1 + X_2L)$.

Proof:

(If)

If there is an M-way non-overlapping intersection among the set of divisors of the expressions, by definition it implies that there are M non-overlapping instances of a two-term subexpression corresponding to the intersection.

(Only if)

Suppose there is a multiple term common subexpression C, appearing N times in the set of expressions, where C has the terms $\{t_1, t_2, ...t_m\}$. Take any $e = \{t_i, t_j\} \in$ C. Consider two cases. In the first case, if e satisfies the definition of a divisor, then there will be at least N instances of e in the set of divisors, since there are N instances of C and our divisor extraction procedure extracts all 2-term divisors. In the second case where e does not satisfy the definition of a divisor (there are no terms in e with zero exponent of L), there exists $e^1 = \{t_i^1, t_j^1\}$ obtained (by dividing by the minimum exponent of L) which satisfies the definition of a divisor, for each instance of e. Since there are N instances of C, there are N instances of e, and hence there will be N instances of e^1 in the set of divisors. Therefore in both cases, an intersection among the set of divisors will detect the common subexpression.

3.3 Algorithm to eliminate common subexpressions

Figure 6 shows our iterative algorithm for detecting and eliminating two-term common subexpressions. In the first step, frequency statistics of all distinct divisors are computed and stored. 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.

In the second step of the algorithm, the best two-term divisor is selected and eliminated in each iteration. The best divisor is the one that has the most number of non-overlapping divisor intersections. The set of non-overlapping intersections is obtained from the set of all intersections by using an iterative algorithm in which the divisor instance that has the most number of overlaps with other instances in the set is removed in each iteration till there are no more overlaps. After finding the best divisor in {D}, the set of terms in all instances of the divisor intersections is obtained. From this set of terms, the set all divisors that are formed using these terms is obtained. These divisors are then deleted from {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 formed using 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}.

Algorithm complexity: The algorithm spends most of its time in the first step where the frequency statistics for all the distinct divisors in the set of expressions is computed. The second step of the algorithm is very fast (linear in the number of divisors) due to the dynamic management of the set of divisors. The worst case complexity of the first step for an M x M constant matrix occurs when all the digits of each constant (assume N-digit representation) are non-zero. Each expression (the Y[i]'s in Equation I) will consist of MN terms. Since the number of divisors is quadratic in the number of terms, the total number of divisors generated for each expression would be of $O(M^2N^2)$. This represents the upper bound on the total number of distinct divisors in {D}. Assume that the data structure for {D} is such that it takes constant time to search for a divisor with given variables and exponents of L. Each time a set of divisors $\{D_{new}\}$ which has a maximum size of $O(M^2N^2)$ is generated in Step 1, it takes $O(M^2N^2)$ to compute the frequency statistics with the set {D}. Since this step is done M-1 times, the complexity of the first step is $\mathbf{O(M^3N^2)}$.

```
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;
       Divide affected expressions in {P_i} by d;
      {d^j} = set of intersecting instances of d;
      for each instance d^j in {d^j}
          Remove from {D} all instances of divisors formed
          using the terms in d^j;

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

Figure 6. Algorithm for extracting and eliminating common subexpressions

Applying our technique to the set of expressions in Figure 4 results in four common subexpressions (D_0-D_3) being detected. The final set of expressions is shown in Figure 7a, which can be implemented as shown in Figure 2. From Figure 7a, it can be seen that the common subexpressions $D_1 = (X_1 + X_2)$ and $D_2 = (X_1 - X_2)$ have instances that have their signs reversed (from Figure 7, D_1 is positive in Y_0 and negative in Y_2 and D_2 is positive in Y_1 and negative and shifted in Y_3). Since the rectangle covering method [9] cannot detect common subexpressions with signs reversed, it cannot detect D_1 and D_2. Hence, it results in an implementation with 10 additions/subtractions (two more than the method presented in this paper). The result of the rectangle covering algorithm is shown in Figure 7b.

$D_0 = X_0 + X_3$ $\quad$ $Y_0 = D_0 + D_1$
$D_1 = X_1 + X_2$ $\quad$ $Y_1 = D_2 + D_3L$
$D_2 = X_1 - X_2$ $\quad$ $Y_2 = D_0 - D_1$
$D_3 = X_0 - X_3$ $\quad$ $Y_3 = D_3 - D_2L$

(a) Using our method

$D_0 = X_0 + X_3$ $\quad$ $Y_0 = D_0 + X_1 + X_2$
$D_1 = X_0 - X_3$ $\quad$ $Y_1 = D_1L + X_1 - X_2$
$Y_2 = D_0 - X_1 - X_2$
$Y_3 = D_1 - X_1L + X_2L$

(b) Using rectangle covering

Figure 7. Optimizing H.264 transform

4. EXPERIMENTAL RESULTS

We compared the reduction in the number of additions/subtractions obtained by our method over known optimization techniques and observe its effect on the area, latency and power consumption of the synthesized hardware. We considered the transforms Discrete Cosine Transform (DCT), Inverse Discrete Cosine Transform (IDCT), Discrete Sine Transform (DST) and Discrete Hartley Transform (DHT). For DFT, the matrix was split into a real part (RealDFT) and an imaginary part (ImagDFT) and the optimizations were carried out separately for the two matrices. The experimental results were performed for the 8x8 constant matrices (8 point), where the constants are represented in fixed point using 16 digits of precision.

We compared our results with three known optimizations. We compared with [4] and [9], since they are related optimizations performed on the matrix forms of the linear systems. We also compared with [3], since it is the most widely referenced work on the multiple constant multiplication problem. In [3] a post processing step is suggested for matrix forms of linear systems where each expression after the first stage of optimization is viewed as a bit pattern and the number of additions are further reduced by finding common bit patterns. We also implemented this step to make a fair comparison with our technique. The number of additions/subtractions obtained by various methods is compared in Table 1. From Table 1, it can be seen that our method gives the least number of additions/subtractions compared to all known techniques. Our method produces a reduction of 10% over the best previous technique [9]. The average CPU time for our method is only 0.08s, which is much faster than the technique in [9], which takes 0.84 s.

We synthesized the designs using Synopsys Design Compiler™ and Synopsys Behavioral Compiler™ using the 1.0 µm CMOS power_sample.db technology library (since it was the only available library characterized for RTL power estimation) using a clock period of 50 ns.

Table 1. Comparing number of additions/subtractions produced by different methods

Example	Number of Additions/Subtractions				
	Original	[3]	[4]	[9]	Our Method
DCT	274	227	202	174	153
IDCT	242	222	183	162	143
RealDFT	253	208	193	165	144
ImagDFT	207	198	178	134	124
DST	320	252	238	200	187
DHT	284	211	209	175	158
Average	**263.3**	**219.7**	**200.5**	**168.3**	**151.5**

We used the Synopsys DesignWare™ library for our functional units. The designs were scheduled with minimum latency constraints. With these constraints, the tool schedules the design with the minimum possible latency, and then minimizes the area with the achieved latency. We then interfaced the Synopsys Power Compiler™ with the Verilog RTL simulator VCS™ to capture the total power consumption including switching, short circuit and leakage power. We simulated all the designs with the same set of randomly generated inputs.

Table 2. Synthesis results (Area and Latency)

Example	Area (Library Units)			Latency (Clock Cycles)		
	[4]	[9]	Our Method	[4]	[9]	Our Method
DCT	90667	73311	66759	10	11	10
IDCT	81868	66864	62883	10	11	10
RealDFT	90946	69827	64026	10	11	10
ImagDFT	75140	55940	54606	10	10	10
DST	108101	84715	81214	11	11	11
DHT	93939	71272	67775	11	11	10
Average	**90110**	**70322**	**66211**	**10.3**	**10.8**	**10.2**

Table 3. Simulation results (Power consumption)

Example	Power Consumption (µWatts)		
	[4]	[9]	Our Method
DCT	729	504	531
IDCT	662	547	569
RealDFT	707	544	554
ImagDFT	644	575	490
DST	607	718	595
DHT	598	545	527
Average	**657.8**	**572.2**	**544.3**

We compared the synthesis results (Area and Latency) and RTL power estimation of the designs optimized by our method with those obtained by [4, 9] since they were the best two implementations in terms of the number of additions/subtractions. The results are shown in Table 2 and Table 3. From Table 2, it can be seen that our optimized implementations have produced significant reductions in area over [4], with a maximum of 30% for the RealDFT example and an average of 26.5%. The area reductions over [9] are smaller, with a maximum of 9% reduction for the DCT example and an average of 5.8%. The latencies of all implementations are quite similar. The power consumption estimates from the RTL simulation also shows significant reductions over [4], with a maximum of 27% reduction for the DCT example and an average of 16.8%. Our method does not always give a lower power consumption compared to that produced by [9], but is lesser by 4% on an average. The total power consumption reported in Table 3 depends on the switching power due to the functional units, controllers and multiplexers in addition to leakage power. Therefore, a reduction in the number of operations does not necessarily imply a reduction in the total power consumption.

5. CONCLUSIONS

In this paper we presented a new technique for reducing the number of operations in linear DSP systems by iteratively eliminating two-term common subexpressions. Our technique overcomes some of the drawbacks of other known optimizations. Experimental results have shown that our technique produces the fewest number of additions/subtractions compared to all other methods. Furthermore our technique produces significant reductions in the area and power consumption of the synthesized implementations. The average CPU time for our algorithm is only 0.08s for the set of examples. Our technique can be integrated into high level synthesis frameworks for reducing linear computations involving multiple variables.

In future, we would like to extend our technique to optimize for different objective functions such as latency and power consumption with given constraints.

REFERENCES

[1] M.Puschel, B.Singer, J.Xiong, J.M.F.Moura, J.Johnson, D.Padua, M.Veloso, and R.W.Johnson, "SPIRAL: A Generator for Platform-Adapted Libraries of Signal Processing Algorithms," *Journal of High Performance Computing and Applications*, 2004.

[2] H. De Man, J. Rabaey, J. Vanhoof, G. Goossens, P. Six, and L. Claesen, "CATHEDRAL-II-a computer-aided synthesis system for digital signal processing VLSI systems," *Computer-Aided Engineering Journal*, vol. 5, pp. 55-66, 1988.

[3] 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.

[4] H.T.Nguyen and A.Chattejee, "Number-splitting with shift-and-add decomposition for power and hardware optimization in linear DSP synthesis," *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, vol. 8, pp. 419-424, 2000.

[5] R.Pasko, P.Schaumont, V.Derudder, V.Vernalde, and D.Durackova, "A new algorithm for elimination of common subexpressions," *IEEE Transactions on Computer Aided Design of Integrated Circuits and Systems*, 1999.

[6] H.Safiri, M.Ahmadi, G.A.Jullien, and W.C.Miller, "A New Algorithm for the Elimination of Common Subexpressions in Hardware Implementation of Digital Filters by Using Genetic Programming," *Proceedings of the IEEE International Conference on Application-Specific Systems, Architectures and Processors (ASAP)*, Boston, MA, 2000.

[7] M.Mehendale, S.D.Sherlekar, and G.Venkatesh, "Synthesis of multiplier-less FIR filters with minimum number of additions," *Proceedings of the ICCAD*, 1995.

[8] I.-C. Park and H.-J. Kang, "Digital filter synthesis based on an algorithm to generate all minimal signed digit representations," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 21, pp. 1525-1529, 2002.

[9] A.Hosangadi, F.Fallah, and R.Kastner, "Common Subexpression Elimination Involving Multiple Variables for Linear DSP Synthesis," *Proceedings of the IEEE International Conference on Application-Specific Architectures and Processors (to appear)*, Galveston, TX, 2004.

[10] T. Wiegand, G. J. Sullivan, G. Bjntegaard, and A. Luthra, "Overview of the H.264/AVC video coding standard," *Circuits and Systems for Video Technology, IEEE Transactions on*, vol. 13, pp. 560-576, 2003.

[11] H. S. Malvar, A. Hallapuro, M. Karczewicz, and L. Kerofsky, "Low-complexity transform and quantization in H.264/AVC," *Circuits and Systems for Video Technology, IEEE Transactions on*, vol. 13, pp. 598-603, 2003.

[12] K.Hwang, *Computer Arithmetic: Principle, Architecture and Design*: Wiley, 1979.

[13] J. Rajski and J. Vasudevamurthy, "The testability-preserving concurrent decomposition and factorization of Boolean expressions," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 11, pp. 778-793, 1992.

[14] J. Vasudevamurthy and J. Rajski, "A method for concurrent decomposition and factorization of Boolean expressions," *Proceedings of the Computer-Aided Design, 1990. ICCAD-90. Digest of Technical Papers., 1990 IEEE International Conference on*, 1990.