Optimized Code Generation of Multiplication-free Linear Transforms

Mahesh Mehendale

Texas Instruments (India) Ltd.
Golf View Homes, Wind Tunnel Road,
Bangalore 560017, INDIA

G. Venkatesh, S.D. Sherlekar

Dept. of Computer Sc. and Engg.
Indian Institute of Technology, Powai,
Mumbai, 400076, INDIA

Abstract

We present code generation of multiplication-free linear transforms targeted to single-register DSP architectures such as TMS320C2x/C5x. We first present an algorithm to generate optimized code from a DAG representation. We then present techniques that transform a DAG so as to minimize the number of nodes and the accumulator-spills. We then introduce a concept of spill-free DAGs and present an algorithm for synthesizing such DAGs. The results for Walsh-Hadamard, Haar and Slant transforms show 25% to 40% reduction in the cycle count using our techniques.

1 Introduction

Many signal processing applications such as image transforms[1], error correction/detection involve matrix multiplication of the form $Y = A \star X$, where X and Y are the input and the output vectors and A is the transformation matrix whose elements are 1,-1 and 0. In this paper we present optimized code generation of these transforms targeted to programmable Digital Signal Processors. While some of the code optimization techniques presented in this paper are generic, our focus is primarily on single-register, accumulator-based DSP architectures such as TMS320C2x[2] and TMS320C5x[3].

Code optimization techniques discussed in the literature[4-7] address the problems of instruction selection and scheduling, register allocation and storage assignment to minimize code size and/or number of cycles. These techniques operate on a DAG representation of the code being optimized and can be applied to implement the multiplication-free linear transforms. However, the amount of optimization achieved using these techniques is limited by the initial DAG representation. Much higher gains are possible by optimizing the DAG itself.

One approach to optimizing the DAG is to minimize the number of additions by utilizing the redundancy in the computation of two or more outputs. In this paper we present an algorithm for minimizing number of additions, which is based on the algorithm presented in [8]. Our approach is different than that presented in [9] and is based on iterative elimination of two-element common subexpressions in the transformation matrix.

For single-register architectures, minimum number of additions in most cases does not translate into minimum number of execution cycles. The common subexpressions typically cause accumulator spills which result in 'load' and 'store' overhead. In this paper we present four DAG transformations to reduce the number of cycles by minimizing accumulator spills.

Another approach to DAG optimization is to minimize the number of additions under the constraint of zero accumulator spills. In this paper we present an algorithm for spill-free implementation of multiplication-free linear transforms in minimum number of execution cycles.

The paper is organized as follows. In section 2, we present the target architecture model which is a suitable abstraction of 'C2x/'C5x architectures. We then present an algorithm for integrated instruction scheduling and register+memory allocation. In section 3, we present the algorithm for minimizing number of additions. In section 4, we present DAG optimizing transformations to optimize code in terms of number of cycles. In section 5, we present the algorithm for spill-free implementation requiring minimum number of cycles. We present results for Walsh-Hadamard, Haar and Slant transforms in section 6 and conclude in section 7 with our future work.

2 Code generation from DAG representation

2.1 Target Architecture Model

The code generation techniques presented in this paper are targeted to the architecture model shown in figure 1. This model is a suitable abstraction of 'C2x and 'C5x and shows datapath of interest to multiplication-free linear transforms. As can be seen from the figure, the target architecture is a single-register, non-commutative machine in which the available operations are :

1. $< acc > \Leftarrow < acc > .op. < mem >, < shift >$
   (instructions ADD and SUB)
2. $< acc > \Leftarrow < mem >, < shift >$
   (instruction LAC)
3. $< mem > \Leftarrow < Acc >, < shift >$
   (instruction SAC)

33rd Design Automation Conference ®

DAC 96 - 06/96 Las Vegas, NV, USA 0-89791-833-9/96/0006..$3.50

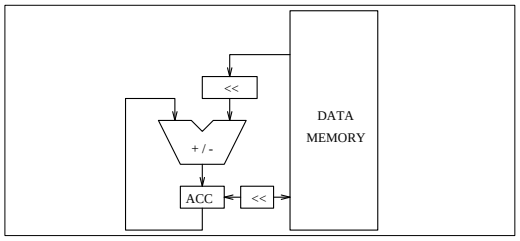

Figure 1: Target Architecture Model

4. $< acc > \Leftarrow - < Acc >$
(instruction NEG)

2.2 Code Generation Rules

One of the inputs to the code generator is a DAG representation of the desired computation. The output and the intermediate nodes represent either an ADD or a SUBTRACT operation and have fanin of 2.

The other input to the code generator is a sequence in which the nodes of the DAG need to be evaluated. Given the sequence and the DAG, following rules are used to generate the code :

Let 'current' node be the latest evaluated node and 'new' node be the new node for which the code is being generated.

1. If the 'current' node is not one of the fanin nodes of the 'new' node, save the 'current' node (SAC instruction), load the left fanin node of the 'new' node (LAC instruction) and ADD/SUBTRACT the right fanin node of the 'new' node.
2. If the 'current' node is a left fanin node of the 'new' node, ADD/SUBTRACT the right fanin node of the 'new' node.
3. If the 'current' node is a right fanin node of the 'new' node and the 'new' node function is SUBTRACT, negate the 'current' node (NEG) instruction and ADD the left fanin node of the 'new' node.
4. If the 'new' node is an output node or an intermediate node with fanout ≥ 2, store the new node (SAC instruction) before proceeding with the next node.

For a given DAG, the code size and consequently the number of cycles depend on the sequence in which the nodes are evaluated. The code optimization problem thus maps onto the problem of finding an optimum sequence of DAG node evaluations.

2.3 Computation Scheduling Algorithm

We now present an algorithm for scheduling the DAG computations for minimum number of cycles. The algorithm uses the following knowledge-base derived from the code generation rules presented in sub-section 2.2.

1. A node can be scheduled for computation only if both its fanin nodes are already computed or are input nodes.
2. The computation of output nodes and the intermediate nodes with fanout ≥ 2, always needs to be stored irrespective of the next computation node.
3. If the 'current' node is one of the fanin nodes of the 'new' node, it avoids accumulator-spill and hence reduces the 'store' and 'load' overhead.

These factors are used to assign weights to the candidate nodes at each iteration of the scheduling algorithm and the node with the highest weight is selected. Here is the overall algorithm :

scheduled-node-list = {}; current-node = $\emptyset$

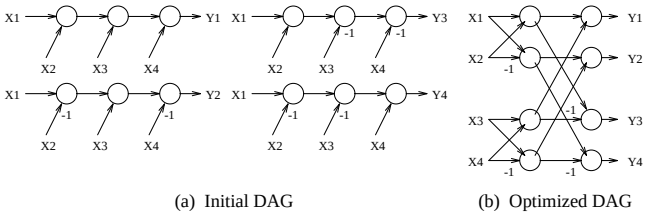

Figure 2: DAGs for 4x4 Walsh-Hadamard transform

```
while (no.of-scheduled-nodes < total-no.of
      intermediate+output nodes) {
    /* build candidate-node-list */
    candidate-node-list = {}
    for all (node_i ∉ scheduled-node-list) {
      if ((node_i.left-fanin + node_i.right-fanin) ∈
        (input+scheduled node-list))
        candidate-node-list += node_i
    /* assign weights to the candidate-nodes */
    for (each node_i ∈ candidate-node-list) {
      node_i.weight = 1
      if ((node_i ∈ output-node-list) .or.
        (node_i.fanout ≥ 2))   node_i.weight++
      if ((node_i.left-fanin = current-node) .or.
        ((node_i.right-fanin = current-node) .and.
        (node_i.op = ADD)))   node_i.weight += 2
      if (node_i.fanout-node.right-fanin ∈
        scheduled-node-list)   node_i.weight += 2
    }
    /* schedule the node with the highest weight */
    find (node_m ∈ candidate-node-list) such that
      node_m.weight is maximum
    scheduled-node-list += node_m
    current-node = node_m
}
```

3 Minimizing Number of Additions

The amount of optimization achievable using the computation scheduling algorithm is limited by the DAG representation. In this section we present an algorithm that minimizes the number of additions and effectively the number of nodes in the DAG to compute the multiplication-free linear transform.

Consider the 4x4 Walsh-Hadamard transform[1] shown below. The DAG representation of this computation is shown in figure 2(a). It has 12 nodes (i.e. 12 additions+ subtractions).

$$\begin{bmatrix} Y1 \\ Y2 \\ Y3 \\ Y4 \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & -1 & 1 & -1 \\ 1 & 1 & -1 & -1 \\ 1 & -1 & -1 & 1 \end{bmatrix} \begin{bmatrix} X1 \\ X2 \\ X3 \\ X4 \end{bmatrix}$$

From the DAG it can be observed that there is some amount of redundancy in the computation. For example, the sub-computation (X1+X2) is used to compute both Y1 and Y3. Similarly, the sub-computation (X1-X2) is used to compute both Y2 and Y4. The total number of additions can be reduced by pre-computing such common sub-computations. The common sub-computations are of two types

1. CS++ in which the elements in 2 columns of the matrix are both 1 or both -1 for more than 1 row (e.g. X12+, columns 1,2 for rows 1 and 3).

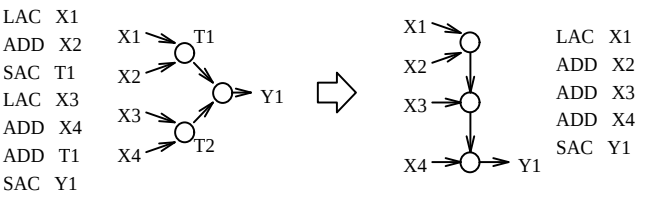

Figure 3: Tree to Chain Conversion

2. C+- in which the elements in 2 columns of the matrix are +1,-1 or -1,+1 for more that 1 row (e.g. X12-, columns 1,2 for rows 2 and 4)

The amount of reduction due to pre-computing a common sub-computation depends on the number of rows in which it appears. The DAG minimization algorithm uses a steepest descent approach[8] which during each iteration pre-computes a common sub-computation with the highest occurrence. This is done by constructing a common sub-computation graph with the nodes representing the columns in the transformation matrix. There are 2 arcs between every two nodes, which represent CS++ and CS+- type common sub-computations. The arcs are assigned weights indicating the number of occurrences of the sub-computations.

Once the sub-computation corresponding to the most weighted arc is identified, the transformation matrix is updated to reflect the pre-computation. This is done by adding a new column to the transformation matrix and suitably updating the matrix elements. The common sub-computation graph is also updated by adding a new node and re-computing arc weights. This procedure is repeated until no arc has a weight of two or more.

Figure 2(b) shows the DAG for 4x4 Walsh-Hadamard transform with minimum number of additions+subtractions. It has 8 nodes (8 additions+subtractions) compared to 12 nodes of the original DAG shown in figure 2(a).

4 DAG Optimizing Transformations

We used the scheduling algorithm discussed in section 2, to schedule the DAGs shown in figures 2(a) and 2(b). While the DAG shown in figure 2(a) requires 20 cycles, the DAG in figure 2(b) requires 22 cycles to compute the transform, eventhough it has 4 less nodes. Clearly, fewer number of nodes does not always translate into fewer number of cycles. The main reason for the DAG in figure 2(b) requiring more cycles, is that all its intermediate nodes have fanout ≥ 2. For a single-register or accumulator-based architectures, such intermediate nodes result in accumulator spilling, and consequently in 'store' and 'load' overhead.

In this section we present four DAG transformations that minimize the accumulator-spill and hence the number of execution cycles.

4.1 Tree to Chain conversion

This transform converts a 'tree' structure in a DAG to a 'chain' structure. This eliminates need to store the intermediate computations and hence reduce the number of cycles. Figure 3 shows an example of this transform. While the DAG with a 'tree' structure requires 7 cycles to compute the output, and the transformed 'chain' structure performs the computation in 5 cycles.

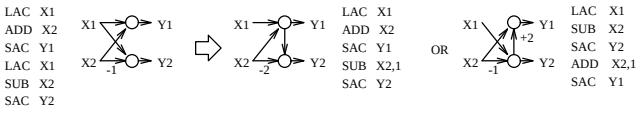

Figure 4: Serializing a Butterfly

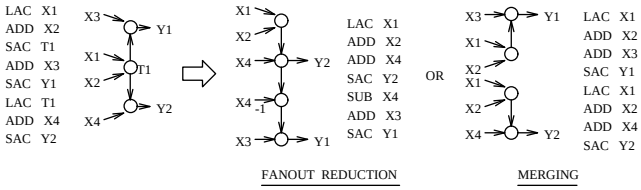

Figure 5: Fanout reduction and Merging

4.2 Serializing a butterfly

Many image transform DAGs have 'butterfly' structures that perform the computations of the type (Y1 = X1 + X2, Y2 = X1 - X2). Such butterfly structures can be serialized by computing one of the butterfly outputs in terms of the other output, and using a SHIFT operation which when performed along with ADD or SUBTRACT does not result in an additional cycle. Figure 4 shows the serialized DAGs which require 5 cycles compared to 6 cycles required for the butterfly computation.

As can be seen from the figure, there are two ways of serializing a butterfly depending on whether Y1 is computed in terms of Y2 (Y1 = Y2 + 2*X2) or Y2 is computed in terms of Y1 (Y2 = Y1 - 2*X2). The choice of the transform depends on the context in which the butterfly appears in the overall DAG.

4.3 Fanout reduction

Since the intermediate nodes with fanout ≥ 2 result in accumulator-spilling, this transformation reduces fanout of an intermediate node in a DAG. Unlike the first 2 transforms, this transform increases the number of nodes in the DAG by 1. Figure 5 shows an example of this transformation applied to a 4 input, 2 output DAG. It can be noted the fanout of the intermediate node T1 in the transformed DAG is 1 (i.e. 1 less than in the original DAG). While the original DAG has 3 nodes and requires 8 cycles, the transformed DAG has 4 nodes but requires 7 cycles.

4.4 Merging

Merging is another transform that reduces the fanout of intermediate nodes. Unlike the earlier transforms, this transform does not reduce the number of cycles. However it transforms the DAG so that other transformations can be applied to the modified DAG. Figure 5 also shows an example of the 'merging' transformation applied to the 4 input, 2 output DAG.

Figure 6 shows how these transformations can be applied to the DAG in figure 2(b). The resultant DAG requires 16 cycles (6 cycles less) to compute the 4x4 Walsh-Hadamard transform.

The amount of optimization possible using these transforms depends on the sequence in which the nodes are selected and the choice of transformations applied. One approach is to search the DAG for potential nodes for transformation and for a selected node, apply the transformation that results in most saving. This greedy approach does not often give the optimum solution. We

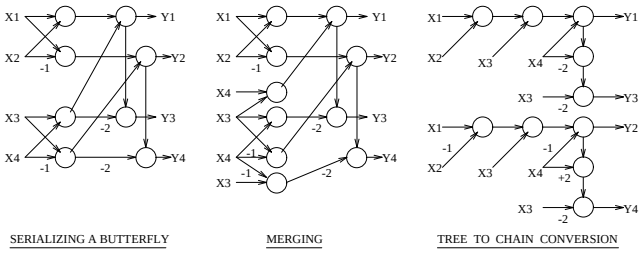

Figure 6: Optimizing DAG using transformations

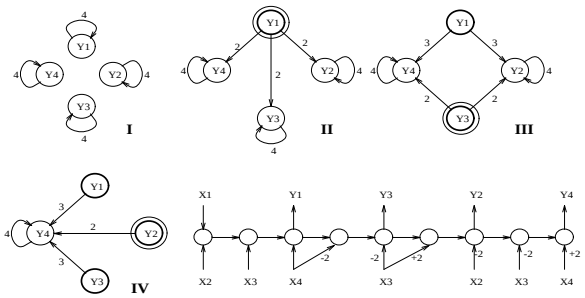

Figure 7: Spill-free DAG synthesis

are currently developing an algorithm that explores a wider search space to arrive at a DAG that requires fewest number of cycles.

5 Synthesis of Spill-free DAGs

A DAG that can be scheduled without any accumulator-spills provides certain advantages. Firstly, it simplifies code generation. Secondly, since there are no accumulator-spills, no intermediate storage is required, thus reducing the memory requirements to implement the transform. The DAG in figure 2(a) is an example of such a DAG. It however requires 20 cycles to compute the transform. It can be noted that the final optimized DAG in figure 6 is also a spill-free DAG. This DAG however requires just 16 cycles. The main reason for the reduced cycles is the fact that this DAG uses pre-computed outputs along with the inputs. For example, Y3 is computed in terms of Y1, X3 and X4, and Y4 is computed in terms of Y2, X3 and X4. Instead of generating a DAG with minimum number of additions and then applying transformations, this DAG can be directly generated from the transformation matrix, if the sequence of output computation is known.

We now present an algorithm that arrives at an optimum sequence of output computations such that the resultant spill-free DAG requires fewest number of cycles. Our algorithm operates on a graph whose nodes represent the outputs. The nodes are of three types - corresponding to
1. the most recently computed output
2. other outputs that are already computed
3. outputs that are yet to computed

Each node in the graph has an edge (self loop) that starts and ends in itself. These self-loops are assigned costs which are given by the number of cycles required to compute the output independently (i.e. without using any of the precomputed outputs). There are also edges between every 'already computed' output node to all the 'yet to be computed' nodes. Each edge is assigned a cost given by the number of cycles required to compute the 'yet to be computed' output in terms of the 'already computed output'. Our algorithm uses the steepest descent approach, which at every stage selects an output that results in minimum incremental cost. In case of more than one outputs having the same lowest incremental cost, one output is selected randomly. Once an output is selected, it is marked as the most recently computed output. All the edges between this node and already-computed nodes are deleted, and new edges are added between this node and the other 'yet to be computed' nodes. The newly added edges are then assigned appropriate costs. This process is repeated to cover all the outputs. The overall algorithm is given below :

```
already-computed-output-list = { }
most-recently-computed-output = ∅
/* construct initial graph and compute edge costs */
for (i=0,i<no-of-outputs;i++) {
  edge[i,i].cost = no. of non-zero entries in row-i + 1 }
repeat {
  find the edge E(M,N) with lowest cost.
  if (M == N) { /* self loop */
    generate DAG to compute output(N)
    in terms of only inputs
  } else {
    generate DAG to compute output(N)
    in terms of inputs and output(M)
  }
  /* update the graph */
  delete edge E(N,N)
  for each node (i ∈ already-computed-output-list) {
    delete edge E(i,N) }
  already-computed-output-list += N
  for each node (i ∈ yet-to-be-computed-output-list)
    { E(most-recently-computed-output,i).cost++ }
  most-recently-computed-output = N
  for each node i ∈ yet-to-be-computed-output-list {
    add edge E(N,i)
    E(N,i).cost = no. of mis-matches between rows
    N and i of the transformation matrix }
} until (yet-to-be-computed-output-list == {})
```

Figure 7 shows each iteration of the algorithm applied to the 4x4 Walsh-Hadamard transform matrix, and the resultant DAG. It can be noted that the resultant DAG is spill-free and requires just 14 cycles to compute the transform.

6 Results

The code generation algorithm presented in section 2, the algorithm for minimizing number of additions+subtractions presented in section 3 and the algorithm for synthesizing spill-free DAGs presented in section 5 have all been implemented in C under Unix.

We compared the code generated by our algorithm with that generated using optimizing C compiler for TMS320C5x. The DAGs for 4x4 Walsh-Hadamard transform shown in figures 2(a), 2(b) and 7 were converted to an equivalent C program and compiled with highest optimization level. The generated code which used indirect addressing, was converted to use direct addressing thus reducing number of cycles. Table I below shows the comparison in terms of number of cycles assuming that the program and data are available in on-chip memories.

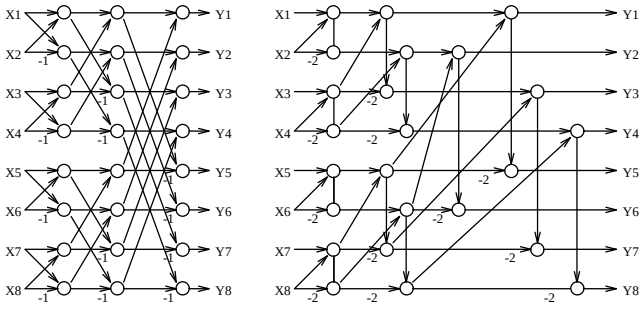

Figure 8: DAGs for 8x8 Walsh-Hadamard transform

DAG	'C5x C compiler no. of cycles	Code Generator no. of cycles
Fig.3	20	20
Fig.4	22	22
Fig.9	19	14

Table I: Code generator vs 'C5x C Compiler

The results show that our code generator generates as compact code as the 'C5x C compiler for the first 2 DAGs. It does better in case of the DAG in figure 7. The main reason for this is that the C compiler during its optimization phase modifies the DAG and in the process generates code with more number of cycles.

In sections 3,4 and 5 we have presented results for 4x4 Walsh-Hadamard transform in terms of minimizing number of additions, optimizing transformations and synthesis of spill-free DAGs. We now present results for 8x8 Walsh-Hadamard transform, 8x8 Haar transform and 4x4 Slant transforms.

The 8x8 Walsh-Hadamard transform[1] matrix is given by:

$$\begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & -1 & 1 & -1 & 1 & -1 & 1 & -1 \\ 1 & 1 & -1 & -1 & 1 & 1 & -1 & -1 \\ 1 & -1 & -1 & 1 & 1 & -1 & -1 & 1 \\ 1 & 1 & 1 & 1 & -1 & -1 & -1 & -1 \\ 1 & -1 & 1 & -1 & -1 & 1 & -1 & 1 \\ 1 & 1 & -1 & -1 & -1 & -1 & 1 & 1 \\ 1 & -1 & -1 & 1 & -1 & 1 & 1 & -1 \end{bmatrix}$$

The direct computation of this transform requires 56 additions+subtractions and the corresponding code executes in 72 cycles. The number of additions+subtractions can be minimized to 24 using the algorithm presented in section 3. The resultant DAG is shown in figure 8. The code corresponding to this DAG requires 64 cycles. We applied optimizing transformations to serialize all the butterflies in this DAG. The resultant DAG is also shown in figure 8. This DAG also has 24 nodes but the corresponding code requires 52 cycles.

We synthesized a spill-free DAG for the 8x8 Walsh-Hadamard transform using the algorithm presented in section 5. The resultant DAG is shown in figure 9. The DAG has 35 nodes and the corresponding code requires 44 cycles. The results so far indicate that for both 4x4 and 8x8 Walsh-Hadamard transforms, the spill-free DAGs result in most efficient code. We modified the DAG for 8x8 Walsh-Hadamard transform to extract a

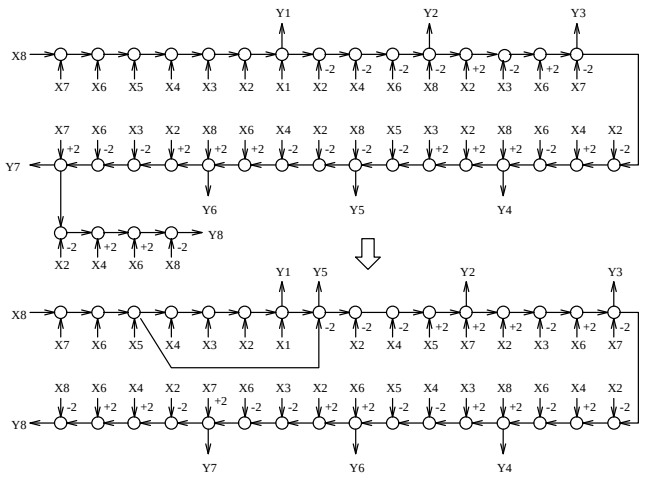

Figure 9: DAGs for 8x8 Walsh-Hadamard transform

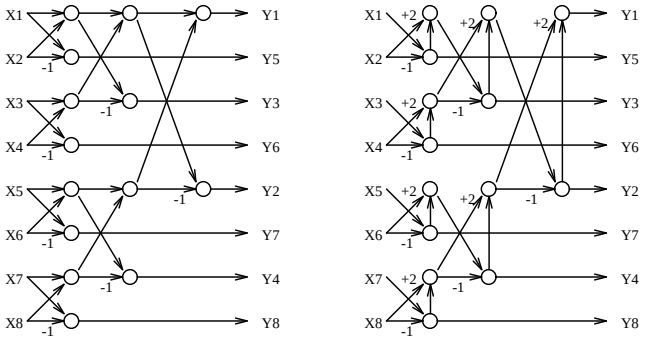

Figure 10: DAGs for 8x8 Haar transform

common sub-computation (X5 + X6 + X7 + X8). The resultant DAG is also shown in figure 9. This DAG has 32 nodes and it does result in one accumulator spill. The code corresponding to this DAG requires 42 cycles (2 less than the spill-free DAG).

The 8x8 Haar transform[1] matrix is given by :

$$\begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & -1 & -1 & -1 & -1 \\ 1 & 1 & -1 & -1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & -1 & -1 \\ 1 & -1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & -1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & -1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & -1 \end{bmatrix}$$

The direct computation of this transform requires 24 additions+subtractions and the corresponding code executes in 40 cycles. The number of additions+subtractions can be minimized to 14 using the algorithm presented in section 3. The resultant DAG is shown in figure 10. The code corresponding to this DAG requires 39 cycles. We applied optimizing transformations to serialize all the butterflies in this DAG. The resultant DAG is also shown in figure 10. This DAG also has 14 nodes but the corresponding code requires 30 cycles.

We synthesized a spill-free DAG for the 8x8 Haar transform using the algorithm presented in section 5.

The resultant DAG has 20 nodes and the corresponding code requires 32 cycles.

The 4x4 Slant transform[1] can be transformed into a 4x8 multiplication-free transform as shown below :

$$\begin{bmatrix} Y1 \\ Y2 \\ Y3 \\ Y4 \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 3 & 1 & -1 & -3 \\ 1 & -1 & -1 & 1 \\ 1 & -3 & 3 & -1 \end{bmatrix} \begin{bmatrix} X1/2 \\ X2/2\sqrt{5} \\ X3/2 \\ X4/2\sqrt{5} \end{bmatrix} =$$

$$\begin{bmatrix} 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & -1 & -1 & 1 & 0 & 0 & -1 \\ 1 & -1 & -1 & 1 & 0 & 0 & 0 & 0 \\ 1 & -1 & 1 & -1 & 0 & -1 & 1 & 0 \end{bmatrix} \begin{bmatrix} X1/2 \\ X2/2\sqrt{5} \\ X3/2 \\ X4/2\sqrt{5} \\ X1 \\ X2/\sqrt{5} \\ X3 \\ X4/\sqrt{5} \end{bmatrix}$$

The direct computation of the 4x8 transform requires 16 additions+ subtractions and the corresponding code executes in 24 cycles. The number of additions+subtractions can be minimized to 12 using the algorithm presented in section 3. The code corresponding to the resultant DAG requires 26 cycles.

Interestingly the spill-free DAG can be synthesized directly from the 4x4 matrix with elements 1,-1,3 and -3. The four outputs can be computed as

Y1 = X1 + X2 + X3 + X4
Y2 = Y1 + X1≪1 - X3≪1 - X4≪2
Y3 = Y2 - X1≪1 - X2≪1 + X4≪2
Y4 = Y3 - X2≪1 + X3≪2 - X4≪1

The DAG for the above computation has 12 nodes and requires 17 cycles. The results presented so far are summerized in the table below (Ns - number of nodes, Cs - number of cycles):

Transform	Initial		Min. adds		Serial b'flies		spill-free	
	Ns	Cs	Ns	Cs	Ns	Cs	Ns	Cs
4x4 Walsh	12	20	8	22	8	19	9	14
8x8 Walsh	56	72	24	64	24	52	35	44
8x8 Haar	24	40	14	39	14	30	20	32
4x4 Slant	16	24	12	26	12	23	12	17

7 Conclusion and Future Work

In this paper we have presented techniques for optimized code generation of multiplication-free linear transforms. The code generation is targeted to single-register, accumulator-based DSP architectures such as TMS320C2x and TMS320C5x. We have presented a code generation algorithm that performs integrated scheduling and register allocation so as to minimize accumulator spills and generate code that executes in minimum number of cycles. Since the quality of the generated code is limited by the initial DAG representations, we have presented techniques for optimizing DAG representations of multiplication-free linear transforms. We have presented an algorithm which is based on iterative elimination of common-subcomputations so as to minimize the number of additions. We have shown that the resultant DAG though optimized in terms of number of nodes does not result in the most optimized code on a single-register machine. We have presented four optimizing transformations that optimize code by minimizing accumulator-spills. These transformations utilize the 'shift' operations which can be performed on the data being added/subtracted to/from the accumulator, without any clock cycle overhead. Finally, we have presented a new approach to DAG optimization that is based on synthesizing spill-free DAGs. This technique has been found to give promising results for most of the multiplication-free linear transforms that we have experimented with.

We have presented results for 4x4 Walsh-Hadamard, 8x8 Walsh-Hadamard, 8x8 Haar transform and 4x4 Slant transforms. The code generated using these optimization techniques requires 25% to 40% fewer number of cycles. We are currently evaluating the effectiveness of these techniques for error correcting/detecting codes and are also looking at optimized code generation of FIR filters on architectures that do not support a hardware multiplier.

As part of our future work, we are developing an algorithm to automate the DAG optimization by applying various transformations. As a first step we have developed an algorithm that searches the DAG for all butterfly patterns and serializes them by appropriately selecting one of the two possible options.

Our algorithm for synthesizing spill-free DAGs currently uses only one already-computed-output to compute the new output. Higher gains are possible if more than one already-computed-outputs are used. We are in the process of enhancing our synthesis process to comprehend such possibilities.

We are also looking at code optimization of multiplication-free linear transforms on multiple-register architectures. We believe that the algorithm for minimizing number of additions+subtractions can result in significant code optimization for register-rich architectures.

References

[1] Anil K. Jain, "Fundamentals of Digital Image Processing", Prentice Hall Inc. 1989

[2] TMS320C2x User's Guide, Texas Instruments, 1993

[3] TMS320C5x User's Guide, Texas Instruments, 1993

[4] A. Aho, R. Sethi and J. Ullman, "Compilers Principles, Techniques and Tools", Addison-Wesley, 1986

[5] Stan Liao et al., "Code Optimization Techniques for Embedded DSP Microprocessors", DAC 1995

[6] Stan Liao et al.,"Instruction Selection Using Binate Covering for Code Size Optimization", ICCAD-95

[7] Stan Liao et al.,"Storage Assignment to Decrease Code Size", ACM conference on Programming Language Design and Implementation, 1995

[8] Mahesh Mehendale, G. Venkatesh and S.D. Sherlekar, "Synthesis of Multiplier-less FIR Filters with Minimum Number of Additions", ICCAD-95

[9] M. Potkonjak et al., "Efficient Substitution of Multiple Constant Multiplications by Shifts and Additions using Iterative Pairwise Matching", DAC 1994, pp 189-194