Multiple Constant Multiplication By Time-Multiplexed Mapping of Addition Chains

Peter Tummeltshammer
Computer Science
University of Technology, Vienna, Austria
peter@bonusworld.at

James C. Hoe and Markus Püschel
Electrical and Computer Engineering
Carnegie Mellon University
{jhoe, pueschel}@ece.cmu.edu

ABSTRACT

An important primitive in the hardware implementations of linear DSP transforms is a circuit that can multiply an input value by one of several different preset constants. We propose a novel implementation of this circuit based on combining the *addition chains* of the constituent constants. We present an algorithm to automatically generate such a circuit for a given set of constants. The quality of the resulting circuits is evaluated after synthesis for a commercial 0.18μm standard cell library. We compare the area and latency efficiency of this addition chain based approach against a straightforward approach based on a constant table and a full multiplier.

Categories and Subject Descriptors

B.2.4 [**High-Speed Arithmetic**]: Cost/performance

General Terms

Algorithms, design

Keywords

Addition chains, multiplierless, directed acyclic graph, fusion

1. INTRODUCTION

This paper addresses the problem of creating optimal circuits for multiplying a fixed-point input value by one of N preset constants according to a $\lceil \log_2 N \rceil$-bit control input. These circuits are important primitives in the hardware implementations of linear DSP transforms (discrete Fourier transform, discrete cosine transforms, and others). A straightforward approach to implementing this multiplication circuit is to store the different constants in a lookup table and to use a control input to select one constant at a time from the table to feed one input of a full multiplier (shown later in Figure 2(a)). Intuitively, the full generality of a full multiplier is unnecessary for multiplying by a predetermined set of constants.

Our solution. This paper presents an alternative implementation that takes advantage of the redundancy and structure in the constituent constants to reduce hardware cost. Specifically, our proposed implementation leverages previous research on addition chains [3, 4]—deriving optimal multiplication circuits for one constant using only adders and wired shifts. We extend this work to create a reduced-area multiplication circuit for multiple constants by optimally "fusing" the individual constants' addition chains into a single network of adders, wired shifts, and multiplexers. The resulting fused addition chain circuit only requires as many adders as needed by the largest of the constituent addition chains. A similar idea was discussed in [2] but without an algorithm how to obtain or optimize these circuits.

Our results. We present an algorithm to generate the multiplication circuit based on fused additions chains. We evaluate our automatically generated multiplication circuits after synthesis for a commercial 0.18μm standard cell library. We compare our multiplication circuits against the straightforward implementations based on full multipliers and lookup tables. When multiplying by a small number of constants, our approach results in a considerably smaller synthesized circuit area at a latency penalty of no more than a factor of two. As the number of constants is increased, our approach incurs too much overhead to stay competitive. We report the break-even point between our approach and the straightforward implementation for different combinations of input port bit-width and constant bit-width. For example, for a 16-bit input port and 16-bit constants, our approach results in a smaller synthesized circuit area when supporting up to 15 constants.

Paper outline. Following this brief introduction, Section 2 discuses additional background and related work on the problem of multiplying a value by one or several constants. Section 3 presents, in detail, our approach to multiply by several constants using fused addition chains. Section 4 presents an evaluation of our approach. Section 5 offers conclusions.

2. MULTIPLICATION BY CONSTANTS

This section reviews the problem of multiplying an input by one or several fixed-point constants using only additions and shifts. Without loss of generality, we assume the multiplicative constants are fixed-point numbers between 0 and 1 (i.e., $0 \leq c < 1$). The number of fractional bits is denoted by w such that

$$c = 0.b_1 b_2 \cdots b_{w-1} b_w = \sum_{i=1}^{w} b_i 2^{-i}, \quad b_i \in \{0, 1\}. \qquad (1)$$

Multiplication By One Constant. Eq. (1) shows that the product cx of an input x and a w-bit constant c is given by $\sum_{i=1}^{w} b_i 2^{-i} x$. This summation can be directly mapped into a series of shifts (scalings by powers of 2) and additions. More commonly, both software and hardware compilers generally use signed digit (SD) recoding to reduce the number of additions associated with multiplying by a constant [5, chap. 6]. An SD constant is $0.b_1 ... b_{w-1} b_w = \sum_{i=0}^{n} b_i 2^{-i}$ where $b_i \in \{\overline{1}, 0, 1\}$ and $\overline{1}$ stands for -1. The most salient aspect of SD recoding is in replacing the occurrences of sequences of k 1's, $1 \cdots 11$, by $10 \cdots 0\overline{1}$, which yields a saving of $k - 2$ additions. For example, SD recoding for 16-bit numbers re-

DAC'04, June 7–11, 2004, San Diego, California, USA.

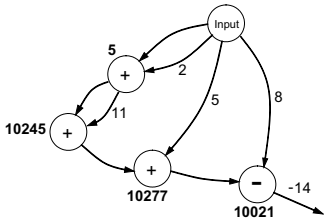

Figure 1: The DAG corresponding to multiplying by $c = 0.10011100100101$.

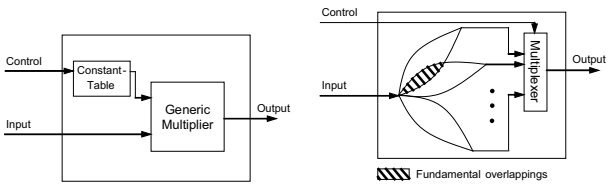

Figure 2: (a) A logic block that can multiply an input by several preset constants stored in a lookup table. (b) An alternate solution using addition chains.

sults in 27% less additions on average. (For brevity, we use in the remainder of the paper the term "addition" for both additions and subtractions since these operations are virtually identical in complexity and map to similar hardware structures.)

The *addition chain* method for multiplying by a constant can further reduce the number of additions by allowing the results of intermediate additions to be shifted and reused in arbitrary subsequent additions (a more precise name would be addition/shift chain; we use addition chain for brevity). Addition chains are best computer-represented and visualized as directed acyclic graphs (DAGs). An example DAG for multiplying x by $c = 0.10011100100101$ is given in Figure 1. The nodes of the graph represent additions and the edges represent the dataflow between additions. Each addition/subtraction node has an in-degree of 2, i.e., two operands. Each edge is labelled by a positive or negative integer k, which represents the shift, or scaling by 2^k, applied to the operand at this edge. In the trivial case $k = 0$, we omit the label. Each node can be labelled by the intermediate result f computed at this node. These numbers are called *fundamentals* of the DAG. In other words, if x is the DAG input and f the fundamental of a node, then the output of this node is fx. Every DAG can be transformed into an equivalent *normalized* DAG of equal cost where 1) for each addition node, one of the operands is not shifted and 2) all fundamentals are odd [3, theorem 2]. (The example in Figure 1 is normalized.) In this paper, we will only consider normalized DAGs.

In the example in Figure 1, only 4 additions are required to compute cx, compared to 5 additions required by the best SD method. The problem of finding an optimal addition chain for a constant is know to be NP-hard [1] and has been frequently studied in the literature, e.g., [3, 4]. Recently, [4] has developed an algorithm that finds optimal addition chains for constants up to a maximal bitwidth of 19, and showed that 5 additions are sufficient in all cases. In this paper, we use a re-implementation of this method.

Multiplying by Several Constants. In this paper, we are interested in developing an area efficient combinational logic block that can multiply a fixed-point input value by one of the N preset constants $c_1, \ldots, c_N$ according to a $\lceil \log_2 N \rceil$-bit control input. This is a different problem from multiplying an input by several constants simultaneously (e.g., in an FIR filter). The most straightforward implementation of the current problem is shown in Figure 2(a) where the preset constants are stored in a lookup table.

Addition chains provide an appealing alternative, since, beyond their optimality for one constant, they allow for potential additional savings through "overlapping". A simple proposal for an addition chain-based approach is given in Figure 2(b). The leaf-shaped objects depict the DAGs for $c_1, \ldots, c_N$. A multiplexer before the output selects one of the output products according to the control input. The shaded region represent overlaps between two DAGs where the same fundamental nodes are used for both constants. In other words, the shaded regions represents implementation savings due to sharing common subexpressions between different DAGs. Whether (b) presents any savings compared to (a) depends on the number N of multiplicative constants included and the degrees of overlaps between the individual DAGs.

The simple proposal in Figure 2(b) instantiates one adder for each unique fundamental in the overlapping DAGs. This is, in general, suboptimal for our problem. Since only one product is visible through the output multiplexer at any moment, the results of some adders are unused. This opens the opportunity for the fundamentals from different DAGs, equal or not, to share the same adders by time-multiplexing, thus exploiting the topological similarities between the DAGs to a much larger degree. As the central contribution of this paper, we offer in the next section an algorithm to fuse addition chains in a way suitable for time-multiplexing.

3. DAG FUSION

This section describes an algorithm to generate a multiplierless hardware functional block for multiplying an input value by one of N given preset constants $c_1, \ldots, c_N$ according a $\lceil \log_2 N \rceil$-bit control input. The input to the algorithm is a set of N DAGs representing optimal additions chains for $c_1, \ldots, c_N$; the output is a composite (or fused) DAG that consists exclusively of additions, shifts, and multiplexers. The composite DAG has the same input-output behavior as Figure 2(a). The number of additions in the composite DAG is equal to the largest number of additions required by any of the input DAGs. In the discussion below, we first provide the details of the fusion algorithm for $N = 2$ and then extend to the general case of N DAGs.

3.1 Fusing Two DAGs

Let DAG_L and DAG_R be the addition chains for two constants with n and m addition nodes, respectively. We denote the respective node sets by $\text{Nodes}_L = \{N_{L,0}, N_{L,1}, ..., N_{L,n-1}\}$ and $\text{Nodes}_R = \{N_{R,0}, N_{R,1}, ..., N_{R,m-1}\}$. Without loss of generality, we assume $n \geq m$. Intuitively, the fusion algorithm tries to find and exploit similarities in the two DAGs' topology. These similar regions are fused and allow the additions in DAG_L and DAG_R to time-multiplex the same adder instantiations with little hardware overhead. In regions that are dissimilar, adders can still be time-multiplexed by the additions in DAG_L and DAG_R, but multiplexers must be inserted to connect the correct input sources to the shared adders or to correct for different shifts of the addition's operands.

A small example. Figure 3(c) gives the composite DAG after fusing the optimal DAGs for multiplying by 45 (a) and 19 (b). Each of the initial DAGs requires 2 additions, and thus the composite DAG shown in (c) also requires $\max\{2, 2\} = 2$ additions. With both multiplexers set to select their left inputs, the active datapaths in the composite DAG correspond to DAG (a), and with both multiplexers set to their right input it corresponds to DAG (b). Note that in the fusion process, we replaced 2 additions by 2 multiplexers. This saving would not be possible in the simple proposal in Figure 2 (b).

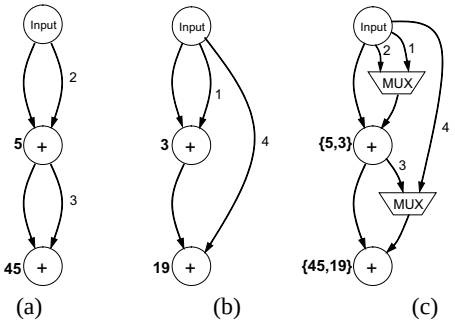

Figure 3: (a) DAG for 45, (b) DAG for 19, and (c) a composite DAG for both 45 and 19.

The algorithm. The fusion algorithm, called FusePairDags, is given below in pseudo code.

```
FusePairDags(DAG_L, DAG_R) {
  // assume DAG_L has equal or more additions than DAG_R
  best_cost = ∞;
  foreach (unique assignment of Nodes_R to Nodes_L) {
    current_dag = FusePair(DAG_L, DAG_R, current_assignment);
    current_cost = ComputeCostCoarse(current_dag);
    if (current_cost < best_cost) {
      best_cost = current_cost;
      best_dag = current_dag;
    }
  }
  return best_dag;
}
```

The algorithm enumerates all *admissible* assignments (injective mapping) of Nodes$_R$ to Nodes$_L$. Admissible means that the assignment respects the partial ordering of the nodes in both DAGs. For each such assignment, the function FusePair merges the edges of DAG$_L$ and DAG$_R$ in the best possible way w.r.t. the cost function ComputeCostCoarse. Since it is clear that the composite DAG has as many additions as DAG$_L$, ComputeCostCoarse counts the number of full bit-width multiplexers, e.g., 2 in Figure 3(c). Finally, the composite DAG with the lowest cost among all enumerated assignments is returned. Details on the node assignment and on the edge merging procedure are given next.

Node assignment. To fuse DAG$_L$ and DAG$_R$, one must assign each node $N_{R,i}$ in DAG$_R$ to a unique node $N_{L,j}$ in DAG$_L$, which means $N_{R,i}$ and $N_{L,j}$ will share the same addition in the composite DAG. Each assignment is an injective mapping ϕ : Nodes$_R$ $\rightarrow$ Nodes$_L$, i.e., no two nodes in DAG$_R$ are mapped to the same node in DAG$_L$. Further, to allow fusion, ϕ has to respect the respective orderings provided by the two DAGs. In the simplest and most frequent case, both DAG$_L$ and DAG$_R$ are totally ordered, which means there are $\binom{n}{m}$ possible assignments ϕ. In the general case, FusePairDags will enumerate all possible ϕ. In the worst case, DAG$_R$ can be embedded into $m!$ total orderings (when all nodes are parallel) and the number of unique ϕ is $n!/(n-m)!$. For the bit-widths considered in this paper, we have $n, m \leq 6$, and thus the number of ϕs does not impose a computational problem.

Merging Edges. Assume an assignment of nodes has been fixed. The function FusePair then creates a composite DAG starting from the input node. Consider the fusion of two addition nodes $N_{R,i}$ and $N_{L,j}$. Now one of three cases applies (note that addition nodes have exactly two operands):

- The two incoming edges of both nodes are shifted by the same values *and* belong to the same operand (predecessor node). Then the composite DAG does not need any multiplexers for this node.
- Exactly one incoming edge of each node is shifted by the same value and belongs to the same operand. In this case one multiplexer is needed in the composite DAG for this node to accommodate for the other respective edges. This case occurs for both nodes in Figure 3.
- In any other case, two multiplexers are needed for this node to accommodate for different input shifts and/or different operands.

Note that FusePair also tries to flip the incoming edges of a node (since addition is commutative) to improve the result. Thus, at most $2m$ fusions are tried for each call of FusePair. However, if an addition node and a subtraction node are fused, the result is a combined addition/subtraction node, and commutativity is lost.

3.2 Fusing Multiple DAGs

In this section we explain the algorithm for fusing N constant DAGs in the general case $N \geq 2$. We first provide a pseudo code description of the algorithm, which is essentially an iterative application of the fusion algorithm FusePairDags for two DAGs combined with a search over different orderings of fusing these DAGs. This section also analyzes the impact of reordering the DAGs and describes a more detailed cost function for selecting the best composite DAG among the re-ordered alternatives.

The algorithm. The fusion algorithm FuseNDags is given in pseudo code below. The input is an array of N DAGs, representing N constants $c_1, \ldots, c_N$, and an integer No_Iterations. The output is a composite DAG that multiplies by the c_i, $1 \leq i \leq N$, according to a $\lceil \log_2 N \rceil$-bit control input.

```
FuseNDags(DAG[N], No_Iterations) {
  best_cost = ∞;
  repeat for No_Iterations {
    randomly permute input DAG array;
    current_dag = FusePairDags(DAG[1], DAG[2]);
    for (i = 3 to N) do
      current_dag = FusePairDags(current_dag, DAG[i]);
    }
    current_cost = ComputeCostFine(current_dag);
    if (current_cost < best_cost) {
      best_cost = current_cost;
      best_dag = current_dag;
    }
  }
  return best_dag;
}
```

The algorithm enumerates, for No_Iterations, different orderings of the input DAG array. For a given ordering, the array is fused iteratively using the function FusePairDags. The lowest cost DAG among all different orderings is returned. The cost function ComputeCostFine used to distinguish between the DAGs computes a finer grain area estimate than ComputeCostCoarse in FusePairDags, which only counted the number of multiplexers. The subroutine FusePairDags needs to handle the case where DAG$_L$ is already composite, i.e., may contain multiplexers before each node. This generalization does not affect the node assignment procedure, but the edge merging subroutine now considers all multiplexed input edges to a node in DAG$_L$ as candidates for fusing with the input edges to the corresponding node in DAG$_R$.

Effect of ordering. The order in which DAGs are fused affects the final outcome. In one experiment, we generated ten different random sets of sixteen 16-bit constants, each requiring 5 additions (the maximum possible). In each case we fused the DAGs using 10,000 random orderings and evaluated the results using the area estimation function ComputeCostFine. The spread was about 10–15%. These differences arises because FusePairDags makes a local decision about where multiplexers are inserted, and these decisions can impact the options available to subsequent calls to FusePairDags for the remaining DAGs. For N DAGs, $N!$ dif-

Table 1: Area efficiency crossover points.

input bit-width	constant bit-width			
	8	12	16	20
8	12			
16	15	16	15	
32	20	12	11	10

ferent ordering must be considered to find the optimal ordering. Exhaustive enumeration is clearly impractical for large N. Instead, FuseNDags is parameterized to select the best result from No_Iterations many randomly chosen orderings. A small number of trials is sufficient in practice, and the worst case penalty is bound by the 10–15% margin discussed above.

Area estimation. To select among all fusion orders the composite DAG with the least area, FuseNDags uses the area estimation function ComputeCostFine. This function makes one pass through the composite DAG and recursively computes the bit-widths (bw's) needed for each of the occurring multiplexers, adders, subtractors, and adder/subtractors. Knowing the bit-width, say k, for any of these blocks, the area can be estimated in square micron as $a \cdot k$, where a is a constant depending on the ASIC technology and library used for mapping. The total DAG cost is then obtained by multiplying all bw's with the respective factors a above and summing them. We conducted several experiments comparing the area estimate computed by ComputeCostFine with the post-synthesis area. The average error was below 10% in all cases.

4. EXPERIMENTAL RESULTS

In this section we evaluate the serial constant multiplier design generated by the DAG fusion algorithm in Section 3 with the standard solution using a generic multiplier shown in Figure 2(a). For a fair comparison, the I/O for each design and the parameters for synthesis were the same in all cases. The designs are synthesized using the Synopsys design compiler v. 2002.05-SP2, a commercial 0.18μm standard cell library, and optimizing for area. All presented sizes are measured in square microns; all presented latencies are measured in nanoseconds.

We evaluated the design space comprised of the Cartesian product of $n \in \{8, 16, 32\}$, $w \in \{8, 12, 16, 32\}$, $N = 2, 3, \ldots, 20$, and $M = 10$, where

- n = bit-width of the input to the multiplication block;
- w = maximum bit-width of the constants considered;
- N = number of constants to be fused;
- M = number of random constant sets to be averaged over.

Figures 4 and 5 report two exemplary set of results for the cases $n = 8, w = 8$ and $n = 32, w = 8$, respectively. The x-axis is the number of constants N fused for $N = 2, 3, \ldots, 20$. In both figures, latency and area are shown. The area cross-over point of $N = 12$ constants in Figure 4 was among the lowest in the experiments, the cross-over point of $N = 20$ in Figure 5 was the largest. The area reduction of our method in both Figures decays approximately linearly starting from 40% and 50% for $N = 2$ to 0% at the crossover, respectively. The increase in latency is about constant around 40% and 120% percent, respectively. The not reported configurations of n and w exhibited similar behaviors; the area cross-over points are given in Table 1.

5. CONCLUSIONS

We proposed a new multiplication logic to support multiplication by one of several preset constants according to a control input. Our design is based on fusing the addition chains of the given constants

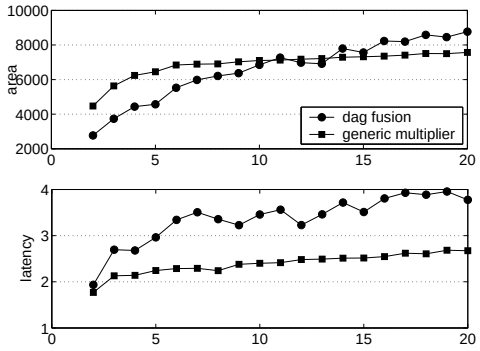

Figure 4: Area and latency comparison between fused DAGs and a generic multiplier solution; input bit-width = 8, constant bit-width = 8.

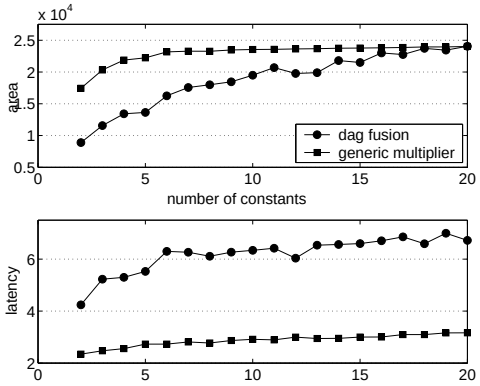

Figure 5: As Figure 4, but with input bit-width = 32 and constant bit-width = 8.

to time-multiplex a minimum number of adders. We presented an algorithm for generating such multiplication logic. Using the generated multiplication circuit, we evaluated the practicality of this new approach against a standard approach based on a constant table and a full multiplier. Our result showed that, for an interesting and relevant space of problems, our approach can offer considerable saving in circuit area albeit for the penalty of increased latency.

6. ACKNOWLEDGEMENTS

This paper describes work conducted at Carnegie Mellon University. Tummeltshammer was supported through the University of Technology, Vienna, Austria by a fellowship for short-term scientific research abroad. Hoe and Püschel were funded in part by NSF through awards 0325687 and 0310941.

7. REFERENCES

[1] D. Bull and D. Horrocks. Primitive operator digital filters. *IEE Proceedings G*, 138(3):401–412, 1991.

[2] S. S.Demirsoy, A. G. Dempster, and I. Kale. Design Guidelines for Reconfigurable Multiplier Blocks. In *IEEE Intl. Symp. on Circ. and Sys.*, vol. 4, pp. IV-293–IV-296, 2003.

[3] A. G. Dempster and M. D. MacLeod. Constant integer multiplication using minimum adders. *IEE Proceedings G*, 141(5):407–413, 1994.

[4] O. Gustafsson, A. Dempster, and L. Wanhammar. Extended results for minimum-adder constant integer multipliers. In *IEEE Intl. Symp. on Circ. and Sys.*, vol. 1, pp. I-73–I-76,2002.

[5] I. Koren. *Computer Arithmetic Algorithms*. A. K. Peters, 2nd edition, 2001.