Multiplierless Design of Folded DSP Blocks

LEVENT AKSOY, INESC-ID
PAULO FLORES and JOSE MONTEIRO, INESC-ID/Instituto Superior Técnico, Universidade de Lisboa

This article addresses the problem of minimizing the implementation cost of the time-multiplexed constant multiplication (TMCM) operation that realizes the multiplication of an input variable by a single constant selected from a set of multiple constants at a time. It presents an efficient algorithm, called ORPHEUS, that finds a multiplierless TMCM design by sharing logic operators, namely adders, subtractors, adders/subtractors, and multiplexors (MUXes). Moreover, this article introduces folded design architectures for the digital signal processing (DSP) blocks, such as finite impulse response (FIR) filters and linear DSP transforms, and describes how these folded DSP blocks can be efficiently realized using TMCM operations optimized by ORPHEUS. Experimental results indicate that ORPHEUS can find better solutions than existing TMCM algorithms, yielding TMCM designs requiring less area. They also show that the folded architectures lead to alternative designs with significantly less area, but incurring an increase in latency and energy consumption, compared to the parallel architecture.

Categories and Subject Descriptors: B.2.0 [**Arithmetic and Logic Structures**]: General; B.5.1 [**Register-Transfer Level Implementation**]: Design

General Terms: Design, Algorithms

Additional Key Words and Phrases: Time-multiplexed constant multiplication, folded architecture, multiplierless design, area optimization, finite impulse response filter, discrete cosine transforms, integer cosine transforms

ACM Reference Format:
Levent Aksoy, Paulo Flores, and Jose Monteiro. 2014. Multiplierless design of folded DSP blocks. ACM Trans. Des. Autom. Electron. Syst. 20, 1, Article 14 (November 2014), 24 pages.
DOI: http://dx.doi.org/10.1145/2663343

1. INTRODUCTION

Multiplication of constant(s) by input variable(s) is a ubiquitous operation in many DSP applications such as fast Fourier transforms (FFTs), FIR and infinite impulse response (IIR) filters, discrete cosine transforms (DCTs), and integer cosine transforms (ICTs). Since the constants are fixed and determined beforehand in these DSP blocks, the constant multiplications are generally realized under a shift-adds architecture using only adders, subtractors, and shifts [Nguyen and Chatterjee 2000]. Note that shifts by a constant value can be realized using only wires that represent no hardware cost. In the last two decades, efficient algorithms have been proposed for not only minimizing the number of operations, but also for optimizing the gate-level area, delay, throughput, and power dissipation of the shift-adds design of constant multiplications [Aksoy et al.

This work was supported by national funds through FCT, Fundação para a Ciência e a Tecnologia, under the project PEst-OE/EEI/LA0021/2013.
Authors' addresses: L. Aksoy (corresponding author), P. Flores, INESC-ID, Rua Alves Redol, 9, 1000-029, Lisboa, Portugal; email: levent@algas.inesc-id.pt; J. Monteiro, INESC-ID, Rua Alves Redol, 9, 1000-029, Lisboa, Portugal and Instituto Superior Tecnico, Universidade de Lisboa, Alameda da Universidade, 1649-004, Lisboa, Portugal.

DOI: http://dx.doi.org/10.1145/2663343

2008, 2010, 2012; Boullis and Tisserand 2005; Dempster and MacLeod 1995; Hartley 1996; Johansson et al. 2005, 2011; Kang and Park 2001; Kumm et al. 2012; Thong and Nicolici 2010; Park and Kang 2001; Voronenko and Püschel 2007].

Over the years, many design architectures, such as parallel, folded, bit-serial, and digit-serial, have also been introduced addressing the area, latency, and energy consumption of DSP applications [Parhi 1995, 1999]. The folded and digit-serial architectures are preferred over the parallel architecture in DSP applications with stringent constraints on circuit area. However, since these architectures need multiple clock cycles to obtain the result, they lead to designs with higher latency and energy consumption with respect to the parallel architecture [Parhi 1999].

In the design of folded DSP blocks, TMCM is a fundamental operation. For example, single-input single-output (SISO) and single-input multiple-output (SIMO) TMCM operations occur in folded design of filters, recursive DCT kernels, and filter banks [Demirsoy et al. 2003, 2004, 2007]. Consider an SISO TMCM example with a set of n constants. As illustrated in Figure 1, it can be realized in three different ways: (i) the *mux-mul* architecture includes a generic multiplier and an n-to-1 MUX, where the primary select input i with $0 \leq i \leq n-1$ determines which constant is multiplied by the input variable x (Figure 1(a)); (ii) the *mcm-mux* architecture uses an n-to-1 MUX and a multiple constant multiplication (MCM) block that implements the constant multiplications using adders and subtractors (Figure 1(b)); (iii) the *mux-add* architecture includes adders, subtractors, and adders/subtractors (determined by a select input) and MUXes (Figure 1(c)). The *mux-add* architecture increases the number of constants that a logic operator can generate and provides the possible sharing of logic operators, yielding less complex TMCM designs when compared to other architectures [Tummeltshammer et al. 2007].

In the last decade, many efficient algorithms were introduced for the minimization of the design complexity in TMCM operations targeting the *mux-add* architecture and the application-specific integrated circuit (ASIC) and field programmable gate arrays (FPGA) design platforms [Aksoy et al. 2013b, 2014; Chen and Chang 2009; Demirsoy et al. 2007; Sidahao et al. 2004, 2005; Tummeltshammer et al. 2007; Turner and Woods 2004]. However, the exact method of Sidahao et al. [2004] can only be applied to a small number of constants and the solution quality of the approximate methods [Aksoy et al. 2013b; Chen and Chang 2009; Demirsoy et al. 2007; Sidahao et al. 2005; Tummeltshammer et al. 2007; Turner and Woods 2004] depends heavily on the TMCM instance. Recently, we introduced an approximate algorithm ORPHEUS [Aksoy et al. 2014] that combines efficient heuristics from both MCM and TMCM techniques and yields better solutions than previously proposed algorithms. To the best of our knowledge, ORPHEUS is the only algorithm that can handle both SISO and SIMO TMCM instances, targeting an ASIC design platform. In this article, we describe ORPHEUS and explore its properties and performance in detail. Furthermore, we present the fully folded realizations of FIR filters and linear DSP transforms. In order to explore the trade-off between area and latency in the fully folded and unfolded (parallel) realizations, we also consider the partially folded design of these DSP blocks. We show how these fully and partially folded realizations can be efficiently obtained using the solutions of ORPHEUS. Experimental results include the comparison of ORPHEUS with prominent TMCM algorithms, the comparison of different TMCM architectures, the comparison of fully and partially folded designs of DSP blocks with their parallel designs, and the exploration of the impact of design parameters on the area, latency, and energy consumption of the DSP blocks.

The rest of the article proceeds as follows. Section 2 introduces the background concepts related to the proposed TMCM algorithm ORPHEUS. Section 3 describes ORPHEUS

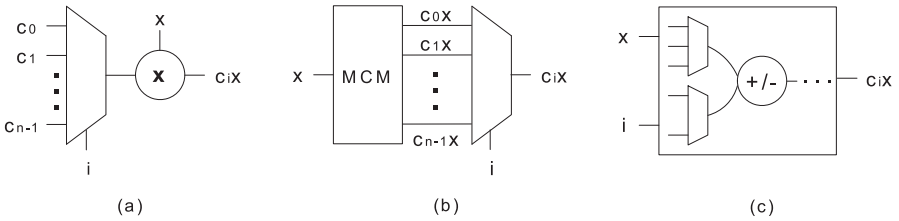

Fig. 1. TMCM architectures: (a) *mux-mul*; (b) *mcm-mux*; (c) *mux-add*.

and Section 4 presents the details on the folded design of FIR filters and linear DSP transforms. Section 5 shows the experimental results and, finally, Section 6 concludes the article.

2. BACKGROUND

In this section, we give the background concepts, introduce the TMCM problem, and present an overview on previously proposed TMCM algorithms. We note that, since the common input variable x is multiplied by multiple constants in MCM and TMCM, the implementation of constant multiplications is in fact equal to the implementation of constants. For example, $3x$ realized as $3x = x \ll 1+x$ can be rewritten as $3 = 1 \ll 1+1$ by removing the variable x from both sides. This terminology is used interchangeably throughout this article.

2.1. Number Representation

The *binary* representation decomposes a number in a set of additions of powers of two. The signed digit system makes the use of positive and negative digits. The *canonical signed digit* (CSD) representation [Hartley 1996] has two main properties: (i) two nonzero digits are not adjacent; and (ii) the number of nonzero digits is minimum. The *minimal signed digit* (MSD) representation [Park and Kang 2001] is obtained by dropping the first property of the CSD representation. Thus, a constant may have several representations under MSD, but all with a minimum number of nonzero digits.

Consider 23 defined in six bits. Its binary representation, 010111, includes 4 nonzero digits. It is represented as $10\bar{1}00\bar{1}$ in CSD and both $10\bar{1}00\bar{1}$ and $01100\bar{1}$ denote 23 in MSD using 3 nonzero digits, where $\bar{1}$ stands for -1.

2.2. 0-1 Integer Linear Programming (ILP)

The 0-1 ILP problem is the optimization of a linear objective function subject to a set of linear constraints and is generally defined as follows.[1]

$$\text{minimize} \quad \mathbf{w}^T \cdot \mathbf{x} \tag{1}$$

$$\text{subject to} \ \mathbf{A} \cdot \mathbf{x} \geq \mathbf{b}, \quad \mathbf{x} \in \{0,1\}^k \tag{2}$$

In the objective function of Eq. (1), w_i in $\mathbf{w}$ is a weight value associated with each variable x_i, where $1 \leq i \leq k$. In Eq. (2), $\mathbf{A} \cdot \mathbf{x} \geq \mathbf{b}$ denotes a set of j linear constraints, where $\mathbf{b} \in \mathbb{Z}^j$ and $\mathbf{A} \in \mathbb{Z}^j \times \mathbb{Z}^k$.

[1]A maximization objective can be easily converted to the minimization objective by negating the objective function. Less-than-or-equal and equality constraints are respectively accommodated by the equivalences $\mathbf{A} \cdot \mathbf{x} \leq \mathbf{b} \Leftrightarrow -\mathbf{A} \cdot \mathbf{x} \geq -\mathbf{b}$ and $\mathbf{A} \cdot \mathbf{x} = \mathbf{b} \Leftrightarrow (\mathbf{A} \cdot \mathbf{x} \geq \mathbf{b}) \wedge (\mathbf{A} \cdot \mathbf{x} \leq \mathbf{b})$.

2.3. Multiplierless Design of Constant Multiplications

The constant multiplications can be categorized in four classes.

(1) The *single constant multiplication* (SCM) operation realizes the multiplication of a single constant r by a single variable x, namely $y = rx$. It is used in compilers [Bernstein 1986] and in the design of FFTs [Qureshi and Gustafsson 2009] and fast DCTs [Thong and Nicolici 2010].
(2) The *multiple constant multiplication* (MCM) operation computes the multiplication of m constants in a set R by a single variable x, namely $y_j = r_j x$ with $1 \leq j \leq m$. It occurs in the transposed form FIR filters [Kang and Park 2001].
(3) The *constant array-vector multiplication* (CAVM) operation implements the multiplication of a $1 \times n$ constant array R by an $n \times 1$ input vector X, namely $y = \sum_k r_k x_k$ with $1 \leq k \leq n$. It appears in IIR and direct form FIR filters [Hartley 1996].
(4) The *constant matrix-vector multiplication* (CMVM) operation realizes the multiplication of an $m \times n$ constant matrix R by an $n \times 1$ input vector X, namely, $y_j = \sum_k r_{jk} x_k$ with $1 \leq j \leq m$ and $1 \leq k \leq n$. It is used in the design of linear DSP transforms [Boullis and Tisserand 2005].

A straightforward way of realizing constant multiplications under a shift-adds architecture, called the digit-based recoding (DBR) technique [Ercegovac and Lang 2003], has two steps: (i) define the constants under a particular number representation, namely binary or CSD; and (ii) for the nonzero digits in the representation of constants, shift the input variables according to digit positions and add/subtract the shifted variables with respect to digit values.

For an MCM example, consider $23x$ and $49x$. Their decompositions under binary are

$$23x = (010111)_{bin}x = x \ll 4 + x \ll 2 + x \ll 1 + x$$
$$49x = (110001)_{bin}x = x \ll 5 + x \ll 4 + x$$

that lead to a multiplierless design with 5 operations as shown in Figure 2(a).

Further reductions in the number of operations can be found by sharing the common partial products among the constant multiplications. The proposed algorithms can be categorized in two classes: (i) the *common subexpression elimination* (CSE) methods; and (ii) the *graph-based* (GB) techniques. The CSE methods [Aksoy et al. 2008; Boullis and Tisserand 2005; Hartley 1996; Park and Kang 2001] define the constants under a number representation, such as binary, CSD, or MSD. Then, considering possible subexpressions that can be extracted from the nonzero digits in the representations of constants, the "best" subexpression, generally the most common, is chosen to be shared among the constant multiplications. Their main drawback is their dependency on a number representation. The GB techniques [Aksoy et al. 2010; Dempster and MacLeod 1995; Thong and Nicolici 2010; Voronenko and Püschel 2007] are not restricted to any particular number representation and aim to find intermediate subexpressions that enable to realize the constant multiplications with a minimum number of operations. They consider a larger number of realizations of a constant and obtain better solutions than the CSE methods, but require more computational resources due to a larger search space. Additionally, hybrid techniques combine methods from both CSE and GB algorithms [Aksoy et al. 2012] and increase the number of implementations of a constant obtained by a CSE method considering alternative realizations [Dempster and MacLeod 2004; Ho et al. 2008].

For our MCM example, the exact CSE algorithm [Aksoy et al. 2008] finds a minimum solution with 4 operations when constants are defined under binary representation (Figure 2(b)). The exact GB method [Aksoy et al. 2010] obtains a minimum solution

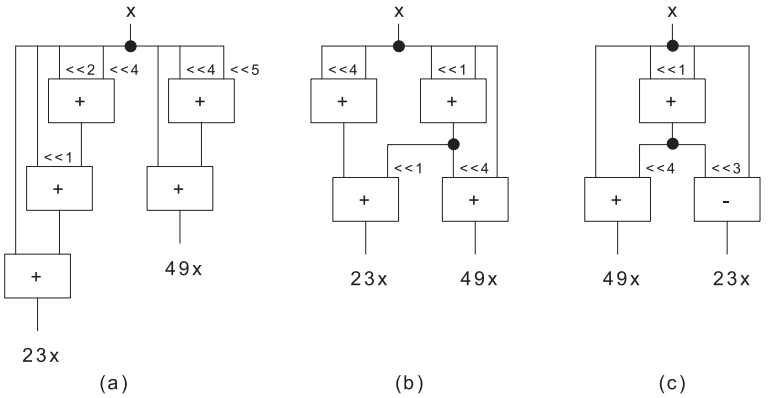

Fig. 2. Multiplierless designs of $23x$ and $49x$: (a) DBR method [Ercegovac and Lang 2003]; (b) exact CSE algorithm [Aksoy et al. 2008]; (c) exact GB algorithm [Aksoy et al. 2010].

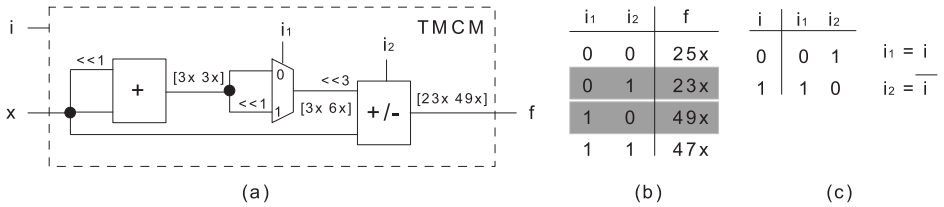

Fig. 3. (a) Solution of ORPHEUS on $C = [23\ 49]$; (b) outputs of the TMCM design; (c) mapping of select inputs.

with 3 operations (Figure 2(c)). In both solutions, the partial product $3x$ is shared among the constant multiplications.

2.4. Time-Multiplexed Constant Multiplication

Considering both SISO and SIMO TMCM operations, we represent the constants of the TMCM operation in an $m \times n$ matrix C, where m and n denote the number of outputs and time slots, respectively. As a simple example, consider an SISO TMCM instance with a 1×2 matrix $C = [23\ 49]$. This constant matrix indicates that $23x$ and $49x$ are required at the output of the TMCM design at time slots 1 and 2 (in other words, when the primary select input i is equal to 0 and 1), respectively. Note that, in SISO TMCM instances, m is 1 and n refers to the number of constants in an array.

Figure 3(a) illustrates the TMCM design obtained by ORPHEUS on $C = [23\ 49]$. All possible values at its output f under the select inputs of an MUX and an adder/subtractor are shown in Figure 3(b). The adder/subtractor is assumed to act as an adder and a subtractor when its select input is 0 and 1, respectively. Observe that the TMCM operation can also generate $25x$ and $47x$. In order to obtain the desired outputs, a combinational logic is required to map the primary select input i to the select inputs of the MUX and the adder/subtractor (Figure 3(c)).

Thus, the *TMCM problem* is defined as follows: Given the constant matrix C, find a set of logic operators, namely adders, subtractors, adders/subtractors, and MUXes, that realizes the TMCM operation and leads to a TMCM design occupying minimum area.

2.5. Related Work

The TMCM algorithms of Demirsoy et al. [2007], Sidahao et al. [2004, 2005], and Turner and Woods [2004] target the FPGA design platform. In Demirsoy et al. [2007]

and Turner and Woods [2004], the basic structure consists of an adder, a subtractor, or an adder/subtractor, that may include a 2-to-1 MUX at one of its inputs requiring no additional hardware in an FPGA. The algorithm of Turner and Woods [2004] generates all possible configurations of the basic structure under the given FPGA technology and maps the decompositions of constants to basic structures. The method of Demirsoy et al. [2007] can find the fewest basic structures by identifying the common partial products among the constant multiplications. In Sidahao et al. [2004, 2005], the basic structure is assumed to include a 4-to-1 MUX whose inputs are a, b, $a + b$, and 0, considering the Xilinx Virtex/Virtex-II slice architecture. The exact algorithm of Sidahao et al. [2004] formalizes the problem of minimizing the number of basic structures as a 0-1 ILP problem. Since the size of the 0-1 ILP problem grows exponentially as the number of constants in the TMCM operation increases, the algorithm of Sidahao et al. [2004] can only be applied to TMCM instances with a small number of constants. In the method of Sidahao et al. [2005], first, the MCM solution of each array of constants is found and then these graphs are merged, exploiting the common partial terms. Note that all these techniques can target both SISO and SIMO TMCM operations.

In Aksoy et al. [2013b], Chen and Chang [2009], and Tummeltshammer et al. [2007], the SISO TMCM operations are targeted under an ASIC design platform and the cost of a TMCM operation is computed based on the area cost of each logic operator in a standard cell library. Given an array of constants, the algorithm of Tummeltshammer et al. [2007], called DAGfusion, finds the SCM graph of each constant using the minimum number of addition and subtraction operations [Gustafsson et al. 2002]. Then, it obtains a TMCM solution by merging the SCM graphs using MUXes iteratively. In Chen and Chang [2009], first, the multiplierless realization of constant multiplications is found using an MCM algorithm and, second, a scheduling algorithm is applied to design the TMCM operation. The algorithm of Aksoy et al. [2013b] combines both the exploitation of the most common partial terms and merging of SCM graphs that is realized by DAGfusion. In the algorithm of Aksoy et al. [2013b], the problem of finding the basic structures, including the most common partial terms, was formulated as a 0-1 ILP problem and the selection of basic structures among alternative realizations was accomplished efficiently using a decision tree. To the best of our knowledge, only ORPHEUS [Aksoy et al. 2014], presented in the following section, can target both SISO and SIMO TMCM operations under an ASIC design platform. Note that any TMCM algorithm, which can target only SISO TMCM instances, can be applied to each output of an SIMO TMCM instance and its solutions can be combined to realize the SIMO TMCM instance. However, in this case, the common logic operators among different outputs may not be extracted and shared properly.

3. ORPHEUS: AN APPROXIMATE TMCM ALGORITHM

The main motivations behind the development of ORPHEUS are to combine efficient techniques proposed for both MCM and TMCM operations into a single TMCM algorithm, to obtain better solutions than existing TMCM algorithms using little computational resources, and to handle both SISO and SIMO TMCM instances under the ASIC design architecture. ORPHEUS is designed as an iterative method and includes two main parts: (i) optimal and (ii) heuristic. In its optimal part, the arrays of constants, that is, the rows of the constant matrix, which can be realized using a single logic operator with available arrays of constants, are synthesized. Note that the input variable x and its shifted versions are always available. If there still exist arrays of constants to be implemented, ORPHEUS switches to its heuristic part. In this part, an array of constants is chosen, its three alternative implementations are found, and the one having the smallest implementation cost is favored. ORPHEUS iterates its optimal and heuristic parts until all rows of the constant matrix are synthesized.

The steps of ORPHEUS, as described in detail in the following sections, are given next. In these steps, the set M, that initially consists of an input variable denoted by 1, will include constants whose multiplications by the input variable will be realized under a shift-adds architecture based on the solution of an efficient MCM algorithm [Voronenko and Püschel 2007]. Also, the matrix S, initially consisting of a single array of n 1's, will include the synthesized arrays of constants. The set I will include logic operators realizing the TMCM operation.

(1) Given the constant matrix C of the TMCM operation, in a preprocessing phase, determine the nonredundant target matrix T.
(2) For each row of T, T_i, search for an optimal realization. If such a realization exists, move T_i from T to S, update M if necessary, and add this realization to I.
(3) If T is empty, go to step 10.
(4) Select an array of constants from T, T_i.
(5) Find a realization of T_i using a single operation, one of whose inputs is a row of S or its shifted version and that has the smallest estimated cost $ecost_1$.
(6) Find a realization of T_i under the *mcm-mux* architecture and compute its estimated cost $ecost_2$.
(7) Find a realization of T_i using a single operation that has the minimum number of distinct partial terms at its inputs and the smallest estimated cost $ecost_3$.
(8) Among these three realizations, choose the one with the minimum cost value and update T, S, I, and M.
(9) If T is not empty, go to step 2.
(10) Synthesize the constants of M under a shift-adds architecture, update, and return I.

In following sections, these steps are described through an SIMO example with a 4×2 matrix $C = [5\ 5; 2\ 1; 12\ 10; 92\ 196]$. This TMCM instance has 4 outputs and 2 time slots. The constant matrix C indicates that the first, second, third, and forth outputs compute $5x$, $2x$, $12x$, and $92x$ at time slot 1, respectively. Also, at time slot 2, they compute $5x$, x, $10x$, and $196x$, respectively We note that the final realization of this TMCM operation found by ORPHEUS is presented in Figure 6(a), where the select inputs of the MUX and adders/subtractors are not shown for the sake of clarity.

3.1. Step 1: Preprocessing Phase

The negative constants of C are converted to positive, because it is always assumed that the sign of a constant is handled where the constant multiplication is required using an adder/subtractor. For each row of C, C_i, its left shift ls_i is found by dividing C_i by 2 until at least one of its constants is odd. Then, $C_i \cdot 2^{-ls_i}$ is added to T without repetition. For our example, T is found as [5 5; 2 1; 6 5; 23 49].

3.2. Step 2: Searching for Optimal Realizations

First, we check each row of T, T_i, if it includes the same constant at its every time slot (column). If such an array of constants exists, it is moved from T to S and this constant is added to M without repetition. For our example, the row of T, namely [5 5], is moved from T to S and 5 is added to M.

Second, we check each T_i if its constants are 1, or its shifted versions, so that it can be realized using a single MUX. If there exists such an array of constants, it is moved from T to S and this MUX is added to I. For our example, the row of T, namely [2 1], can be realized using a single MUX (Figure 6(a)).

Third, we check each T_i if it can be realized using a single operation, that is, an adder, a subtractor, or an adder/subtractor, whose inputs are the rows of S or their shifted versions. While searching for such realizations, the arrays of constants Ds are

computed as

$$D \cdot 2^{ls_d} = T_i - \left(S_j \cdot 2^{ls_j}\right) \odot SA, \tag{3}$$

where $ls_d \geq 0$ denotes the left shift of D, ls_j stands for the left shift of S_j with $1 \leq j \leq |S|$ and $0 \leq ls_j \leq mbc(T_i) - mbc(S_j) + 1$ (where the $mbc(A)$ function determines the maximum bitwidth of constants in an array A), and SA is a $1 \times n$ sign array consisting of 1 and -1. Note that $\odot$ denotes the element-by-element product of arrays (e.g., in $C = A \odot B$, each entry in the array C is the product of the corresponding entries in arrays A and B). Thus, Ds are found by searching all possible S_j, ls_j, and sign arrays exhaustively. If D is equal to a row of S, then T_i can be realized using a single operation. If SA consists of 1's, then the operation is an adder, whereas if SA consists of -1's, then it is a subtractor. Otherwise, it is an adder/subtractor. If such a realization is found, it is added to I and T_i is moved from T to S. In our example, for the row of T, [6 5] as T_i, a D can be found as [1 1], which is a row of S, with $[1\ 1] \cdot 2^2 = [6\ 5] - ([2\ 1] \cdot 2^0) \odot [1\ 1]$, where [2 1] standing for S_j is another row of S. Thus, the realization of [6 5], namely $[1\ 1] \cdot 2^2 + ([2\ 1] \cdot 2^0) \odot [1\ 1]$, needs a single adder (Figure 6(a)).

This procedure iterates until there is no such row of T satisfying the preceding checks. For our example, at the end of this step, T, S, and M are [23 49], [1 1; 5 5; 2 1; 6 5], and $\{1, 5\}$, respectively.

3.3. Step 4: Selection of an Array of Constants

At this point, there exist row(s) of T that require(s) intermediate arrays of constants to be synthesized. ORPHEUS selects one row of T that has the minimum *tnzd* value, denoting the total number of nonzero digits of constants in an array under CSD. An array with a smaller *tnzd* value roughly indicates that it can be realized using a small number of logic operators. This metric is preferred for two reasons: (i) it may increase the sharing of logic operators, since ORPHEUS considers the synthesized arrays of constants of S in the realization of a row of T (steps 2, 5, and 7); (ii) it ensures the convergence of each realization to an array consisting of 1 or its shifted versions (step 5).

3.4. Step 5: First Alternative Realization

Note that every computation of D in Eq. (3) actually presents a possible realization of a row of T, T_i. To find the promising one, whenever a new D is determined, we first check whether its *tnzd* value is less than that of T_i to ensure its convergence. If so, next we determine the type of the operation realizing T_i and compute its cost in a given standard cell library, $cost_{op}$. Then we estimate the cost of D as if it will be realized under the *mcm-mux* architecture as shown in Figure 1(b). Initially, we determine the quantity of different constants of D, namely q. If $q > 1$, the cost of a q-to-1 MUX in a given standard cell library, namely $cost_{mux}$, is computed; otherwise, $cost_{mux}$ is set to 0. Then, we convert each constant of D to an odd constant, find the minimum number of operations required for its multiplication by an input variable using the solutions of the algorithm of Gustafsson et al. [2002], compute the cost values of these operations assuming them as adders, and add them to $cost_{add}$ that was initially set to 0. Note that the $cost_{add}$ value is computed without repeating the same odd constants. Thus, the implementation cost of an operation realizing T_i is found as $cost_{op} + cost_{mux} + cost_{add}$. Note that the array of constants S_j in Eq. (3) is already synthesized. After all possible realizations of T_i are considered, we choose the one with the minimum cost value which is assigned to $ecost_1$.

For the realization of the row of T, [23 49], $[3\ 6] \cdot 2^3 + ([1\ 1] \cdot 2^0) \odot [-1\ 1]$ is found to be the one with the minimum cost, requiring an adder/subtractor and the intermediate array of constants [3 6] that is estimated to need a 2-to-1 MUX and an adder.

$$23 = \begin{cases} 10\bar{1}00\bar{1} = \begin{cases} 100000 + 00\bar{1}00\bar{1} = 32 - 9 \\ 00\bar{1}000 + 10000\bar{1} = -8 + 31 \\ 00000\bar{1} + 10\bar{1}000 = -1 + 24 \end{cases} \\ 01100\bar{1} = \begin{cases} 010000 + 00100\bar{1} = 16 + 7 \\ 001000 + 01000\bar{1} = 8 + 15 \\ 00000\bar{1} + 011000 = -1 + 24 \end{cases} \end{cases} \quad 49 = \begin{cases} 10\bar{1}0001 = \begin{cases} 1000000 + 00\bar{1}0001 = 64 - 15 \\ 00\bar{1}0000 + 1000001 = -16 + 65 \\ 0000001 + 10\bar{1}0000 = 1 + 48 \end{cases} \\ 0110001 = \begin{cases} 0100000 + 0010001 = 32 + 17 \\ 0010000 + 0100001 = 16 + 33 \\ 0000001 + 0110000 = 1 + 48 \end{cases} \end{cases}$$

Fig. 4. Possible implementations of 23 and 49 under MSD.

3.5. Step 6: Second Alternative Realization

In this case, we consider the *mcm-mux* realization of T_i as shown in Figure 1(b). Its cost value is computed exactly as described for the estimation of the cost of D in step 5, except that the elements of M are taken into account in this case, meaning that if the odd version of a constant of T_i is an element of M, then it is not considered in computation of $cost_{add}$. Thus $ecost_2$ is found as $cost_{mux} + cost_{add}$. For our example, the estimated cost of [23 49] includes a 2-to-1 MUX and 4 adders.

3.6. Step 7: Third Alternative Realization

In this realization of T_i, we aim to find an implementation (an adder or a subtractor) for each constant of T_i such that these operations include the minimum number of distinct partial terms at their inputs. The reason behind this is that common partial terms reduce the sizes of MUXes and the number of elements in the intermediate array of constants [Aksoy et al. 2013b]. This problem is formulated as a 0-1 ILP problem that requires five steps as described in following.

First, we find all possible realizations of each constant of T_i by decomposing its nonzero digits in two partial terms when it is defined under MSD. Since a constant may have alternative representations under MSD, the number of possible realizations is increased. For the row of T, [23 49], the possible realizations of its constants are given in Figure 4. Removing the same realizations of 23 and 49, that is, $-1 + 24$ and $48 + 1$, respectively, there are 5 realizations for both constants.

Second, we represent the realizations of constants in a Boolean network that includes only AND and OR gates. For an adder, two AND gates, denoted as $AND_{p_1+p_2}$ and $AND_{p_2+p_1}$, are generated. For a subtractor, we assume that the first and second inputs are the partial terms with the positive and negative signs, respectively, and we generate an AND gate denoted as $AND_{p_1-p_2}$. For each constant of T_i, $T_i[h]$, where $1 \leq h \leq n$, an OR gate, $OR_{T_i[h]}$, is generated to combine all realizations of $T_i[h]$. Each partial term p_k at the first or second input of an operation is denoted as an optimization variable, $O_1|p_k|$ or $O_2|p_k|$, respectively. Figure 5 shows the network generated for the row of T, [23 49].

Third, the objective function of the 0-1 ILP problem is obtained as a linear combination of optimization variables whose weights are set to 1. Its constraints are obtained by finding the conjunctive normal form (CNF) formulas of each gate and expressing each clause of the CNF formulas as a linear inequality [Barth 1995]. For example, a two-input AND gate $c = a \wedge b$ is translated to CNF as $(a+\bar{c})(b+\bar{c})(\bar{a}+\bar{b}+c)$ and converted to linear constraints as $a - c \geq 0$, $b - c \geq 0$, $-a - b + c \geq -1$. The outputs of OR gates related to constants of T_i, namely $OR_{T_i[h]}$, are set to 1 since they need to be realized.

Fourth, a minimum solution is found using a 0-1 ILP solver. Based on the selected operations (the outputs of AND gates set to 1 in the solution), the realization of T_i is formed as

$$T_i = G_1 \cdot 2^{ls_1} + (G_2 \cdot 2^{ls_2}) \odot SA, \tag{4}$$

where G_1 (G_2) includes the first (second) input of the selected operations for each constant of T_i, ls_1 (ls_2) is the amount of left shift of G_1 (G_2), and SA denotes the sign

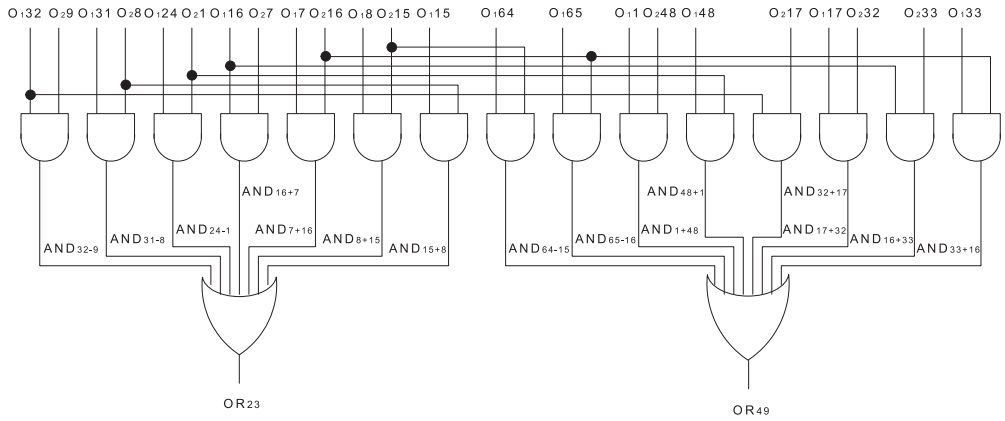

Fig. 5. The Boolean network generated for the array of constants [23 49] under MSD.

array and is determined based on the type of each operation. In other words, if $T_i[h]$ is realized using an adder, then $SA[h]$ is 1; otherwise, it is -1. For our example, the operations $7 + 16$ and $65 - 16$ are respectively found for constants 23 and 49, and the realization of [23 49] is obtained as $[7\ 65] \cdot 2^0 + ([1\ 1] \cdot 2^4) \odot [1\ -1]$ according to Eq. (4).

Finally, we compute the cost of this realization as done in step 5. To do so, we define the type of operation based on SA and compute its cost, $cost_{op}$. For any of G_1 and G_2, if it is not included in S, then we estimate its cost in terms of $cost_{mux}$ and $cost_{add}$. For our example, the type of operation is adder/subtractor and only the cost of [7 65] is estimated that needs a 2×1 MUX and two adders.

This solution leads to an operation with common partial terms at the inputs. In order to include information about the design complexity into the 0-1 ILP problem, for each partial term p_k, we also take into account its bitwidth $bw(p_k)$ and number of nonzero digits under CSD $nzd(p_k)$. Hence, we modify the objective function of the previous 0-1 ILP problem, where the weight of each optimization variable, denoting a partial term p_k at the inputs of operations, is assigned to $bw(p_k) \cdot nzd(p_k)$. In this case, 23 and 49 are implemented as $8 + 15$ and $64 - 15$, respectively. The realization of [23 49] is found as $[1\ 8] \cdot 2^3 + ([15\ 15] \cdot 2^0) \odot [1\ -1]$, requiring an adder/subtractor. The intermediate arrays [1 8] and [15 15] are estimated to require a 2×1 MUX and an adder, respectively.

Among these two possible realizations, we determine the one with the minimum cost, which is assigned to $ecost_3$. For our example, the second one has the minimum cost.

3.7. Step 8: Selection of the Realization with Minimum Cost

Among the realizations found in steps 5, 6, and 7, if $ecost_1$ is the minimum, we add the required intermediate array of constants found in step 5 to T. If $ecost_2$ is the minimum, we add the odd versions of constants in T_i to M without repetition and move T_i from T to S. Otherwise, we add the required intermediate array(s) of constants found in step 7 to T. In case of equality of cost values, the realization found in step 6 is favored first, and then the one found in step 5. This is because that in the realizations of constants in M is larger than that among the realizations of the arrays of constants. For our example, $ecost_1$ is the minimum cost value, thus [3 6] is added to T.

3.8. Step 9: Iterations in ORPHEUS

ORPHEUS iterates until T is empty. For our example, in step 2 of the next iteration, [3 6] and [23 49] are respectively realized as $[5\ 5] \cdot 2^0 + ([2\ 1] \cdot 2^0) \odot [-1\ 1]$ and $[3\ 6] \cdot 2^3 + ([1\ 1] \cdot 2^0) \odot [-1\ 1]$.

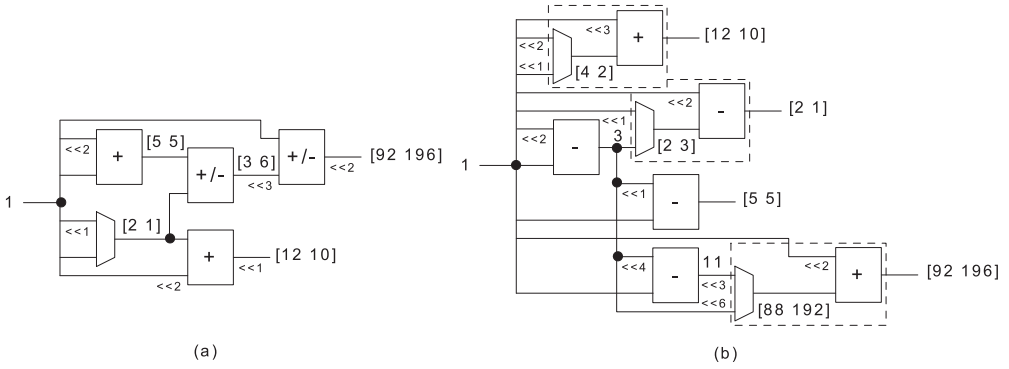

Fig. 6. Time-multiplexed realizations of $C = [5\ 5; 2\ 1; 12\ 10; 92\ 196]$: (a) ORPHEUS; (b) algorithm of Demirsoy et al. [2007].

3.9. Step 10: Realization of the Constant Multiplications

If M includes constants other than 1, the MCM algorithm of Voronenko and Püschel [2007] is applied to find the fewest operations realizing these constant multiplications and these operations are added to I. For each chosen *mcm-mux* realization, the necessary MUX is added to I.

The solution of ORPHEUS to our example C is presented in Figure 6(a). On the other hand, the solution of the algorithm of Demirsoy et al. [2007], that can target SIMO TMCM instances under the FPGA design platform, is illustrated in Figure 6(b). Recall that the basic structure in Demirsoy et al. [2007] consists of an adder, a subtractor, or an adder/subtractor with a 2-to-1 MUX, as shown in dashed blocks. Observe that such a restriction may disable the possibility of sharing common partial terms and may yield a design of an array of constants using a large number of logic operators as in the realization of [2 1], leading to a design with a large area under the ASIC design platform.

To support this work with synthesis results, we developed a computer-aided design (CAD) tool that can describe the TMCM operation obtained by ORPHEUS under the *mux-add* architecture in VHDL. It can also describe a TMCM design under the *mul-mux* and *mcm-mux* architectures.

4. FOLDED DESIGN ARCHITECTURES FOR FUNDAMENTAL DSP BLOCKS

This section describes the folded design of FIR filters focusing on the transposed form and of linear DSP transforms, emphasizing DCTs and ICTs. We note that the time-multiplexed design of filter banks was described in Demirsoy et al. [2007]. A TMCM architecture required for the design of FFTs can be found in Qureshi and Gustafsson [2009]. In Karkala et al. [2010], the sum of products in FFTs was considered in a folded architecture, and an optimization algorithm that exploits the common nonzero digits of constants in order to reduce the design complexity was introduced.

4.1. FIR Filters

Digital filtering is a ubiquitous operation in DSP applications and is realized using IIR or FIR filters. Although an FIR filter requires a larger number of coefficients than that of an equivalent IIR filter, it is preferred over the IIR filter due to its stability and phase linearity properties [Wanhammar 1999]. The output of an N-tap FIR filter, namely $y(n)$, is computed as $\sum_{k=0}^{N-1} r_k \cdot x(n-k)$, where N is the filter length, r_k is the k-th filter coefficient, and $x(n-k)$ is the k-th previous filter input with $0 \leq k \leq N-1$. Figure 7(a) presents the parallel design of the transposed form FIR filter that requires

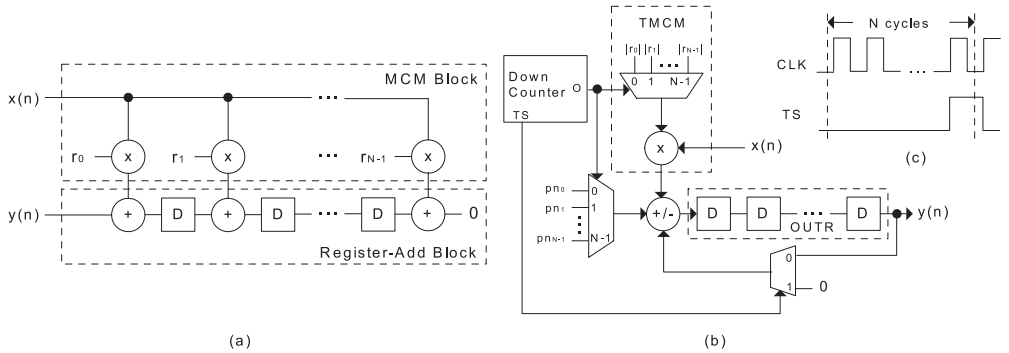

Fig. 7. (a) Parallel (unfolded) transposed form FIR filter design; (b) its fully folded design; (c) the timing signal *TS*.

N multipliers and $N-1$ registers and adders (the one having 0 as an input is not counted). The sizes of these logic operators depend on the bitwidths of the filter input and filter coefficients.

Figure 7(b) depicts the fully folded FIR filter of the transposed form of Figure 7(a) using the *mux-mul* architecture for the TMCM operation. Note that any architecture shown in Figure 1 can be used for the TMCM operation. In the folded design, the SISO TMCM operation realizes the multiplication of the filter input by the absolute value of the filter coefficient determined by the output of a counter. Note that pn_k denotes the sign of the filter coefficient r_k, namely positive (0) or negative (1). If the sign values of all coefficients are positive (negative), only an adder (a subtractor) is required. Otherwise, an adder/subtractor and an MUX as shown in Figure 7(b) are needed.[2] Moreover, the counter is $\lceil log_2(N-1) \rceil$-bit wide and counts from $N-1$ to 0. It also generates the timing signal TS shown in Figure 7(c) using additional hardware, where CLK denotes the clock signal that is fed to all registers in these circuits (not shown for the sake of clarity). $OUTR$ stands for cascaded $N-1$ registers and their sizes are equal to the bitwidth of the filter output bwo, computed as $bwi + \lceil log_2 \sum_{k=0}^{N-1} |r_k| \rceil$, where bwi denotes the bitwidth of the filter input. The size of the adder/subtractor is bwo and the bitwidth of the output of the TMCM operation is equal to $bwi + \lceil log_2 \max_k\{|r_k|\} \rceil$. Note that the filter input should not be altered in N clock cycles in order to compute the filter output accurately.

Although the complexity of the filter design under the fully folded architecture is reduced with respect to the parallel design, the filter output is obtained in N clock cycles under the fully folded architecture, increasing the latency and energy consumption. To explore this trade-off, we consider the hybrid form [Bougas et al. 2005] that is obtained by dividing the transposed form filter of Figure 7(a) into s subsections and moving the registers in-between subsections from the output-line (i.e., the signal path computing the filter output in Figure 7(a)) to input-line (the signal path carrying the filter input in Figure 7(a)). In this form, each subsection includes $\lceil N/s \rceil$ filter coefficients. If N is not multiple of s, extra $\lceil N/s \rceil s - N$ coefficients equal to 0 are added. Figure 8(a) shows the hybrid form FIR filter generated from a 4-tap transposed form filter when s is 2. Each subsection can be regarded as a transposed form filter including $\lceil N/s \rceil$ coefficients with an additional input and output, except the last subsection.

Thus, when each subsection of the hybrid form is realized under the fully folded architecture, as shown in Figure 8(b) for the hybrid filter in Figure 8(a), $\lceil N/s \rceil$ clock

[2]Similarly, in the parallel design of the transposed form of Figure 7(a), if a coefficient is negative, its absolute value is used in the MCM block and the related adder in the register-add block is converted to a subtractor.

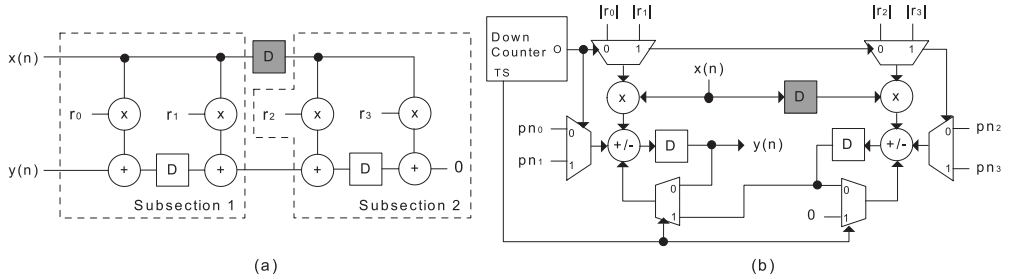

Fig. 8. (a) A hybrid form FIR filter obtained when N is 4 and s is 2; (b) corresponding partially folded filter.

cycles are required to obtain the filter output. In each of s subsections of the partially folded design, there exist one TMCM operation and $\lceil N/s \rceil - 1$ cascaded registers. Note that the sizes of multipliers and MUXes may be smaller than those in the fully folded design since a smaller number of coefficients is included in each subsection. Additionally, there is one counter counting from $\lceil N/s \rceil - 1$ to 0. Note that the registers between subsections, colored in gray, are triggered at the end of $\lceil N/s \rceil$ clock cycles, since the inputs of subsections fed from these registers should not be altered in this period.

The folded designs of direct form FIR filters can be found similarly. Our CAD tool was improved to design the partially and fully folded FIR filters under any TMCM architecture shown in Figure 1.

4.2. Linear DSP Transforms

Over the years, many fast and efficient algorithms for DCTs have been introduced [Chen et al. 1977; Hou 1987]. DCTs are widely used in image data compression and video coding [Banerjee et al. 2007]. The $n \times n$ DCT matrix B is an orthonormal matrix, that is, $BB^T = I$, where I is the identity matrix, and its floating-point entries b_{ij} with $0 \leq j \leq n-1$ are determined as follows.

$$b_{ij} = \begin{cases} \sqrt{1/n} & i = 0 \\ (\sqrt{2/n}) \cdot \cos(\pi(j + 0.5)i/n) & 1 \leq i \leq n-1 \end{cases} \tag{5}$$

The complexity of DCTs is dominated by the floating-point multipliers, hence ICTs were proposed to be alternative to DCTs [Cham 1989]. The $n \times n$ ICT matrix E is an orthogonal matrix, that is, $EE^T = K$, where K is a diagonal matrix and includes integer values. Due to the orthogonality and boundary constraints, as well as the similarity relations with the entries of the DCT matrix B that an ICT matrix must satisfy, there exist many ICTs [Cham 1989; Cheung et al. 1991; Aksoy et al. 2013a].

Considering the DCT and ICT matrices as an $n \times n$ constant matrix R, the computation of DCTs and ICTs can be regarded as a CMVM operation as given in Figure 9(a). The parallel design of this CMVM operation requires n^2 multipliers and $n^2 - n$ adders, where the sizes of the multipliers and adders depend on the bitwidth of the input variables and constants, and the order of summations. Figure 9(b) presents the fully folded realization of the CMVM operation under the assumption that each of its outputs is required to be computed simultaneously. Although the TMCM operation is depicted under the *mux-mul* architecture, it can be realized under the *mcm-mux* or *mux-add* architecture. In Figure 9(b), pn_{jk} with $0 \leq j, k \leq n-1$ denotes the sign of the entry r_{jk}, namely positive (0) or negative (1) of the constant matrix R. The SIMO TMCM operation realizes the multiplications of an input variable by all entries of each column of R determined by a counter. It requires n multipliers and n-to-1 MUXes.

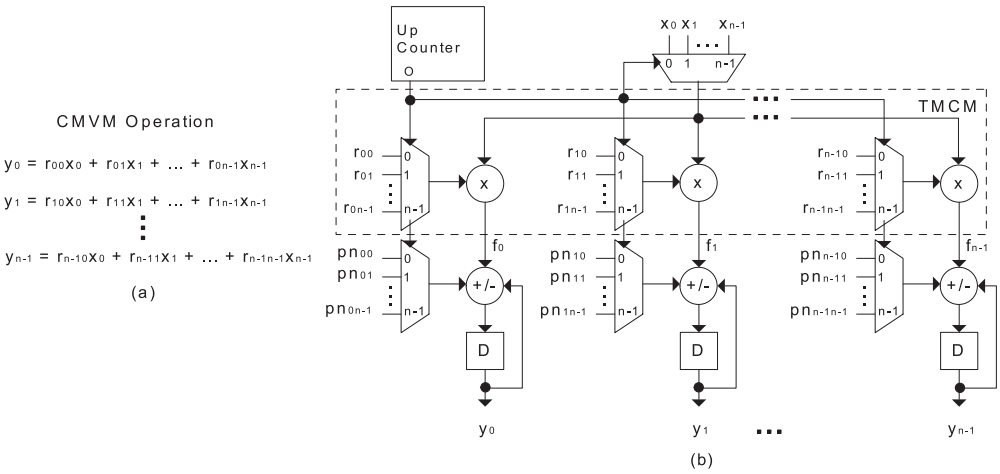

Fig. 9. (a) The outputs of a CMVM operation; (b) fully folded design of a CMVM operation.

The fully folded design additionally needs n adders/subtractors and accumulator registers, $n + 1$ n-to-1 MUXes, and one counter. Assuming that the bitwidths of input variables are equal to bwi, the size of the relevant output of the TMCM operation f_j is $bwi + \lceil log_2 max_k(r_{jk}) \rceil$ and the sizes of the relevant adder/subtractor and accumulator register required for the computation of the output y_j are $bwi + \lceil log_2 \sum_{k=0}^{n-1} |r_{jk}| \rceil$. Also, the counter is $\lceil log_2(n-1) \rceil$-bit wide and counts from 0 to $n-1$. The computation of each output of the CMVM operation y_j requires n clock cycles. Although it is not shown in this circuit, the counter also generates a signal similar to the *TS* signal in Figure 7(c) to reset the accumulator registers after each n clock cycles.

The partially folded design of the CMVM operation, that requires less number of clock cycles than the CMVM operation under the fully folded architecture, can be obtained in three steps. First, the n input variables are split into $g = \lceil n/v \rceil$ groups, each consisting of v variables. Second, the sums of products associated with these groups of input variables are implemented under the fully folded architecture. Third, the relevant outputs of these designs are summed to obtain the outputs of the CMVM operation. Thus, the number of clock cycles required to compute the outputs of the CMVM operation is reduced to v. Note that, if n is not multiple of v, then $vg - n$ input variables are added into the CMVM operation and the relevant entries in the constant matrix R are filled with 0. As an example, consider a 4×4 constant matrix and assume that v is 2. Also, suppose that one group consists of x_0 and x_1, and the other of x_2 and x_3. The computations of the outputs of the CMVM operation and their partially folded design are given in Figure 10.

The partially folded design includes g subcircuits, each including a single SIMO TMCM operation, n adders/subtractors and accumulator registers, and $n + 1$ v-to-1 MUXes. Each TMCM operation consists of n multipliers and v-to-1 MUXes. Also, a counter that counts from 0 to $v - 1$ is required to synchronize the computation and to reset the accumulator registers for the next computation. Finally, for each output of the CMVM operation, $g - 1$ adders are needed to obtain the output. Although the number of logic operators increases as g increases (v decreases), in this case their sizes may decrease since a small number of constants are considered in each group.

In this work, our CAD tool was enhanced to design the partially and fully folded CMVM operations under the *mux-mul*, *mcm-mux*, and *mux-add* architectures.

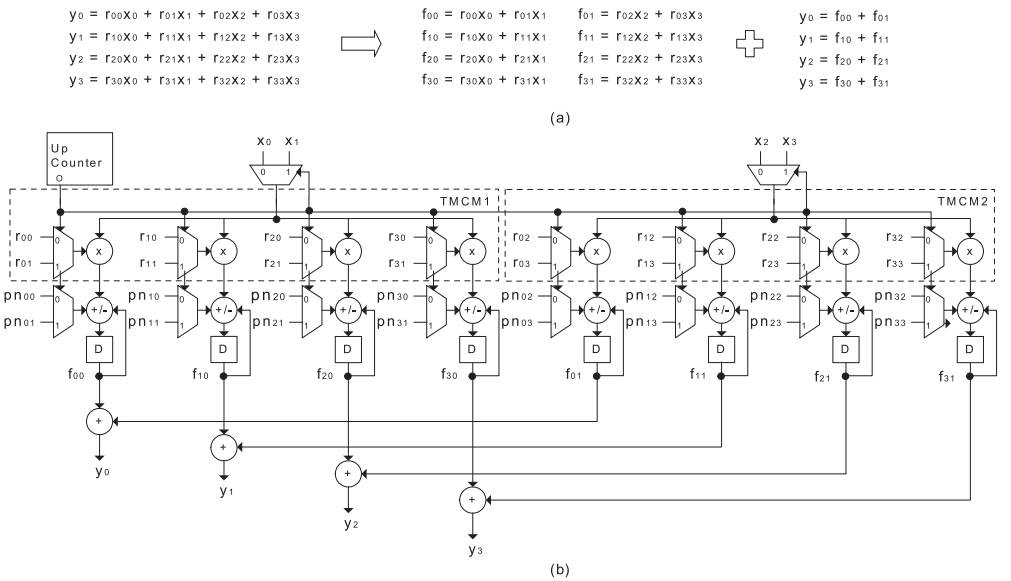

Fig. 10. (a) Partitioning of the outputs of a 4×4 CMVM operation when g is 2; (b) corresponding partially folded design.

Similarly, the given folded architectures can be applied to IIR filters, filter banks, and other linear DSP transforms such as the Reed-Muller transform and error correcting codes.

5. EXPERIMENTAL RESULTS

This section is divided into two sections. In the first, we present the results of ORPHEUS and compare them with those of prominent TMCM algorithms. In the second, we present the gate-level implementations of SIMO TMCM instances under the *mux-mul*, *mcm-mux*, and *mux-add* architectures and of FIR filters and linear DSP transforms under the unfolded, partially folded, and fully folded architectures. ORPHEUS was written in MATLAB and its results were obtained on a PC with Intel Xeon at 2.4GHz and 10GB memory. It uses the 0-1 ILP solver SCIP (http://scip.zib.de/). The Synopsys Design Compiler was used as the synthesis tool with the UMCLogic 0.18-μm Generic II library. In the synthesis script, relaxed timing constraints were used in order to provide more freedom to the synthesis tool to optimize area. The functionality of designs was verified on 10,000 randomly generated input signals in simulation, from which we obtained the switching activity information that was used by the synthesis tool to compute the power dissipation.

5.1. Comparisons of TMCM Algorithms

For the comparison of ORPHEUS with DAGfusion [Tummeltshammer et al. 2007], we used randomly generated $1 \times n$ TMCM instances, where the maximum bitwidth of constants (*mbc*) ranges from 6 to 14 in steps of 2 and n ranges between 2 and 18 in steps of 4. For each group, 30 instances were generated. Table I presents the results of the algorithms, where #*BS* denotes the number of better solutions than ORPHEUS found against DAGfusion in terms of area, which is computed using the 0.18-μm standard cell library when the bitwidth of the input variable (*bwi*) was 16, as described in Tummeltshammer et al [2007]. In DAGfusion, the exhaustive search on all possible orderings of SCM graphs was limited to 10,000 orderings since it may take enormous CPU time. In this

Table I. Summary of Results of Algorithms on SISO TMCM Instances

mbc	*n*	2	6	10	14	18
6	#*BS*	27	19	21	24	29
	Min Gain (%)	−4.78	−19.85	−18.19	−16.91	−1.94
	Avg Gain (%)	9.16	4.20	6.75	9.94	17.03
	Max Gain (%)	30.30	31.90	28.98	28.21	32.75
	CPU DAGfusion (s)	0.76	2.90	8.08	11.29	19.34
	CPU ORPHEUS (s)	0.94	0.33	1.07	1.50	2.15
8	#*BS*	24	16	20	18	26
	Min Gain (%)	−9.46	−13.52	−19.95	−16.14	−7.98
	Avg Gain (%)	8.20	2.93	4.51	5.98	12.71
	Max Gain (%)	25.54	26.02	22.71	25.25	34.46
	CPU DAGfusion (s)	0.84	4.84	11.41	15.55	20.42
	CPU ORPHEUS (s)	1.39	1.75	2.24	2.83	4.19
10	#*BS*	22	15	16	19	24
	Min Gain (%)	−23.26	−14.75	−12.85	−12.91	−15.97
	Avg Gain (%)	6.99	1.27	0.75	7.59	7.81
	Max Gain (%)	28.48	27.66	22.96	28.51	27.75
	CPU DAGfusion (s)	0.83	7.18	20.97	113.34	99.28
	CPU ORPHEUS (s)	1.94	2.86	3.84	5.55	8.57
12	#*BS*	25	18	18	24	19
	Min Gain (%)	−12.06	−19.22	−14.69	−21.54	−20.33
	Avg Gain (%)	13.88	3.17	4.17	6.80	4.75
	Max Gain (%)	43.23	39.97	23.96	27.91	24.52
	CPU DAGfusion (s)	1.00	23.54	65.72	276.09	515.20
	CPU ORPHEUS (s)	2.85	4.89	7.12	8.68	13.42
14	#*BS*	23	21	19	19	24
	Min Gain (%)	−12.49	−24.13	−15.55	−25.02	−15.90
	Avg Gain (%)	10.10	4.64	4.12	6.37	8.93
	Max Gain (%)	37.23	24.26	27.72	28.21	25.27
	CPU DAGfusion (s)	1.15	12.12	39.97	53.53	48.93
	CPU ORPHEUS (s)	4.98	7.82	11.15	15.17	22.53

table, *Min Gain*, *Avg Gain*, and *Max Gain* are the minimum, average, and maximum gain in percentage found by ORPHEUS over DAGfusion, respectively. Average runtime of both methods is in seconds.

Observe from Table I that ORPHEUS obtains better solutions than DAGfusion (#*BS*) on more than half of the total number of TMCM instances, namely 30, in each group, except that including 10-bit 6 constants. Its maximum gain over DAGfusion is greater than 20% on every group of instances, reaching up to 43.23% on a TMCM instance with 12-bit 2 constants. Note that this value is higher than DAGfusion's maximum gain over ORPHEUS on every group of instances and its average gain reaches up to 17.03% on instances consisting of 6-bit 18 constants. Also, ORPHEUS requires less CPU time than DAGfusion on TMCM instances with $n \geq 6$. However, ORPHEUS cannot always find the best solution, primarily because ORPHEUS is an approximate algorithm developed as an iterative method, including efficient heuristics. Thus, in ORPHEUS, a decision made in an earlier iteration has a significant impact on the quality of the final solution. In turn, DAGfusion aims to consider all possible mergings of SCM graphs exhaustively. Secondarily, ORPHEUS was developed to handle both SISO and SIMO TMCM instances, whereas DAGfusion only targets SISO TMCM ones. However, taking into account an increase in its runtime, the solution quality of ORPHEUS can be increased by considering

Table II. Area Estimation for $C = [256\ 162\ 50\ 26\ 15\ 8\ 4\ 2\ 1]$

Method	Add	Sub	Add/Sub	MUX	Cost
[Tummeltshammer et al. 2007]	0	0	1 (12-bit) 1 (14-bit)	1 (14-bit) 3-to-1 1 (16-bit) 7-to-1	4704
[Demirsoy et al. 2007]	1 (10-bit)	1 (12-bit)	1 (10-bit) 2 (11-bit) 1 (12-bit) 1 (16-bit)	1 (8-bit) 2-to-1 1 (9-bit) 2-to-1 2 (10-bit) 2-to-1 1 (11-bit) 2-to-1 1 (12-bit) 2-to-1 1 (16-bit) 2-to-1	9578
[Chen and Chang 2009]	0	0	1 (12-bit) 1 (14-bit)	1 (14-bit) 3-to-1 1 (11-bit) 4-to-1 1 (16-bit) 4-to-1	4648
ORPHEUS	1 (14-bit)	0	1 (15-bit)	1 (14-bit) 4-to-1 1 (15-bit) 6-to-1	4452

Table III. Area Estimation for $C = [362\ 392\ 473]$

Method	Add	Sub	Add/Sub	MUX	Cost
[Tummeltshammer et al. 2007]	1 (19-bit)	1 (16-bit)	1 (12-bit)	2 (12-bit) 2-to-1 1 (19-bit) 2-to-1 1 (16-bit) 3-to-1 1 (20-bit) 3-to-1	6365
[Demirsoy et al. 2007]	1 (11-bit) 1 (12-bit) 1 (17-bit)	0	1 (11-bit)	3 (11-bit) 2-to-1 1 (14-bit) 2-to-1	5074
[Chen and Chang 2009]	1 (17-bit)	1 (11-bit) 1 (12-bit) 1 (14-bit)	0	1 (14-bit) 2-to-1 1 (16-bit) 2-to-1 1 (17-bit) 2-to-1 1 (18-bit) 3-to-1	5986
ORPHEUS	1 (10-bit)	1 (15-bit) 1 (17-bit)	0	1 (11-bit) 3-to-1 1 (12-bit) 3-to-1	4036

not only one realization of an array of constants in each iteration, but a few, and by including the unique merging technique of DAGfusion in ORPHEUS.

For the comparison of ORPHEUS with prominent TMCM algorithms, we also used two benchmark sets given in Chen and Chang [2009]. Tables II and III present the results of the algorithms, where *Cost*, denoting area, was computed using the 0.18-μm standard cell library when bwi was 8. The results of other TMCM algorithms were taken from Chen and Chang [2009]. Observe that ORPHEUS finds a TMCM design with the least complexity, where the gain over the second best solution in Tables II and III is 4.21% and 20.45%, respectively.

5.2. Comparisons of Design Architectures

This section presents the gate-level results of TMCM operations, FIR filters, DCTs, and ICTs.

5.2.1. TMCM. For this experiment, we used randomly generated $m \times n$ matrices, where m was 2, 4, 6, and 8, n was 4, 6, 8, 10, 12, 14, and 16. For each group, 30 instances were generated with 12-bit constants. Figure 11 presents the average gate-level area (in μm^2) of the TMCM designs under the *mux-mul*, *mcm-mux*, and *mux-add* architectures when bwi was 16 and 24. Note that the MCM designs in the *mcm-mux* architecture

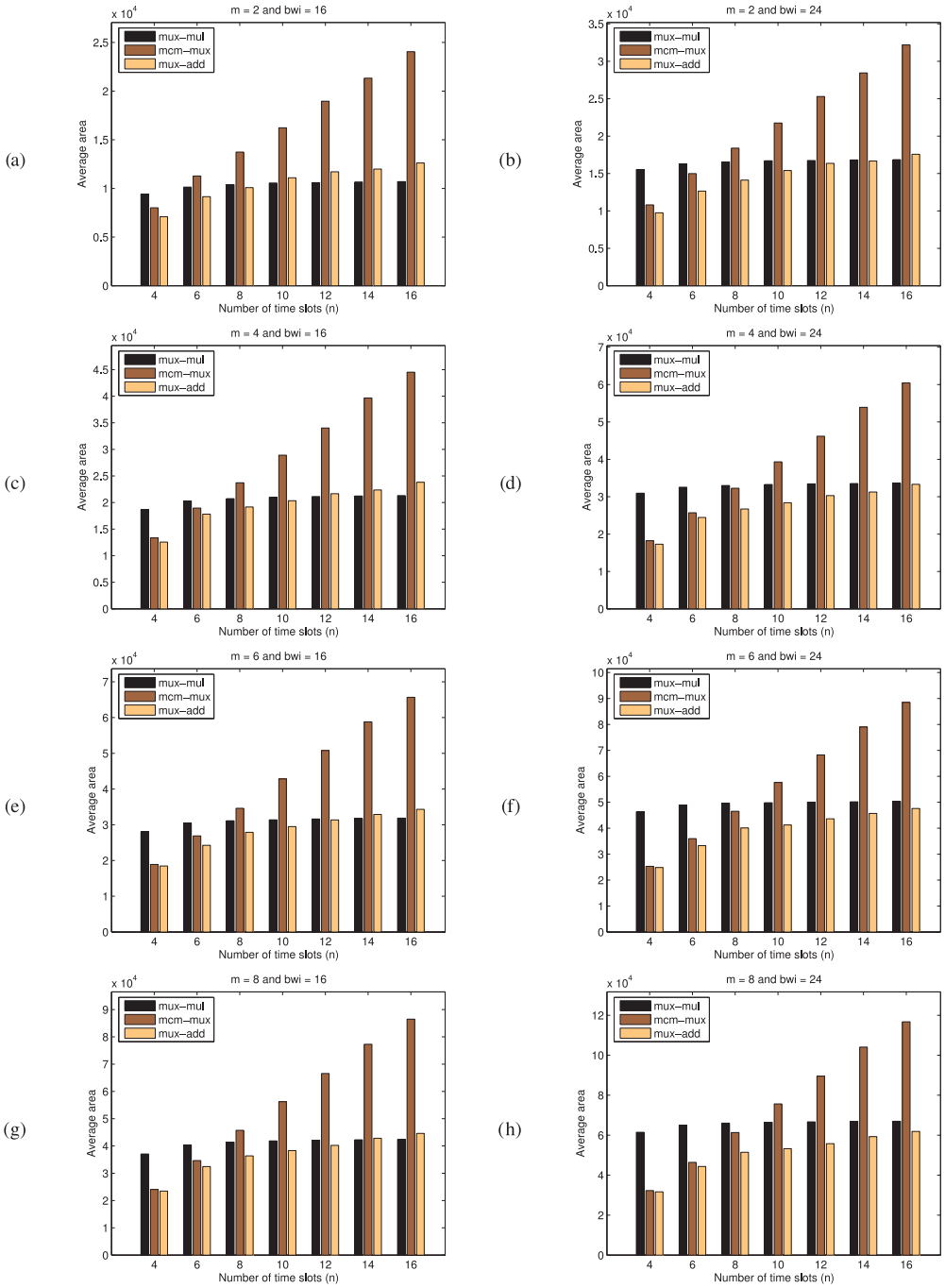

Fig. 11. Gate-level area results on SIMO TMCM operations: (a) $m = 2$ and $bwi = 16$; (b) $m = 2$ and $bwi = 24$; (c) $m = 4$ and $bwi = 16$; (d) $m = 4$ and $bwi = 24$; (e) $m = 6$ and $bwi = 16$; (f) $m = 6$ and $bwi = 24$; (g) $m = 8$ and $bwi = 16$; (h) $m = 8$ and $bwi = 24$.

Table IV. Specifications of FIR Filters

Filter	N	pass	stop	Q
1	15	0.1	0.2	12
2	51	0.07	0.12	16

were obtained by the exact MCM algorithm of Aksoy et al. [2010] and *mux-add* designs were found by ORPHEUS.

Observe that the number of time slots (n) has a significant impact on the area of the *mux-add* designs, since the number of logic operators in the solutions of ORPHEUS generally increases as n increases. However, it has a slight impact on the area of the *mux-mul* designs, since it affects only the size of those MUXes that occupy smaller area than the multipliers. As the number of outputs (m) and the bitwidth of the input variable (bwi) increase, the *mux-add* designs become less complex when compared to the *mux-mul* designs. This is due to the fact that, as m is increased, the number of multipliers in the *mux-mul* architecture is increased, and as bwi is increased, the size of the multipliers is increased. Observe that both m and n have a significant impact on the area of the *mcm-mux* designs, since the number of constants in the MCM block is increased as m and n are increased. However, this architecture may lead to TMCM designs occupying less area than those designed under the *mux-mul* architecture when n is decreased. Furthermore, as bwi increases, the complexity of a TMCM design also increases.

We also observed that the delay in TMCM operations under the *mux-add* architecture is generally higher than that designed under the *mux-mul* architecture, since the solutions of ORPHEUS include a large number of logic operators in series.

5.2.2. FIR Filters. To make a fair comparison between filter architectures, we used two low-pass FIR filters with asymmetric coefficients obtained using the *firgr* function of MATLAB with the *minphase* option. Table IV presents their specifications, where N is the filter length, *pass* and *stop* denote the normalized passband and stopband frequencies, respectively, and Q is the quantization value used to convert the floating-point coefficients to integers.

Table V presents the gate-level results of the transposed form parallel, partially folded, and fully folded filters whose constant multiplications are realized using Generic Multipliers (GMs) and using the solutions of optimization algorithms under the Shift-Adds (SA) architecture. For the partially and fully folded designs, GM and SA respectively indicate that the TMCM operations are realized under the *mux-mul* architecture and the *mux-add* architecture based on the solutions of ORPHEUS. For the parallel designs under the *SA* architecture, their multiplier blocks are realized using the solution of the exact MCM algorithm of Aksoy et al. [2010]. In this experiment, the bitwidth of the filter input bwi was 16 and the partially folded filters were generated with three different s values. In this table, *CA*, *NCA*, and *TA* stand, respectively, for combinational, noncombinational, and total area in mm^2. Also, D and L denote, respectively, the critical-path delay and latency (computed as $D \times cc$, where cc is the number of clock cycles required to obtain the filter output) both in ns. P and E are, respectively, the power dissipation in mW and energy consumption in pJ (computed as $P \times L$). The minimum values on total area, latency, and energy consumption are shown in bold.

First, consider the parallel, partially folded, and fully folded architectures. For filters using generic multipliers, their area is decreased using the folded architecture, but their latency and energy consumption are increased. For Filter 2, while the area ratio between the parallel and fully folded realizations is 3.9, the latency and energy consumption ratio between the fully folded and parallel realizations are 78.1 and 23.8, respectively. This is simply due to the 51 clock cycles required to obtain the filter output

Table V. Results of Transposed Form FIR Filters Using 16-Bit Filter Input

Filter	Architecture		CA	NCA	TA	D	L	P	E
1	Parallel	GM	31.4	6.3	37.7	7.2	7.2	4.0	28.8
		SA	16.7	6.3	23.0	7.1	**7.1**	2.5	**17.5**
	Partially Folded $s = 5$	GM	26.1	5.8	31.9	8.6	25.8	1.7	43.9
		SA	22.5	5.8	28.3	8.0	24.0	1.6	38.4
	Partially Folded $s = 4$	GM	23.0	6.5	29.5	8.4	33.6	1.7	57.1
		SA	22.1	6.5	28.6	9.3	37.2	1.8	67.0
	Partially Folded $s = 3$	GM	18.8	6.3	25.1	8.1	40.5	1.8	72.9
		SA	19.6	6.3	25.9	8.8	44.0	1.9	83.6
	Fully Folded	GM	7.3	7.0	**14.3**	8.5	127.5	1.5	191.3
		SA	8.6	7.0	15.6	9.4	141.0	1.6	225.6
2	Parallel	GM	139.3	26.0	165.3	8.1	**8.1**	17.7	143.4
		SA	69.2	26.0	95.2	9.1	9.1	9.8	**89.2**
	Partially Folded $s = 17$	GM	107.1	22.2	129.3	11.6	34.8	5.4	187.9
		SA	99.3	22.2	121.5	10.1	30.3	5.5	166.7
	Partially Folded $s = 9$	GM	67.9	25.8	93.7	10.4	62.4	6.6	411.8
		SA	70.5	25.8	96.3	11.3	67.8	6.8	461.0
	Partially Folded $s = 3$	GM	26.8	26.2	53.0	10.4	176.8	5.3	937.0
		SA	34.9	26.2	61.1	10.5	178.5	5.6	999.6
	Fully Folded	GM	14.3	28.2	**42.5**	12.4	632.4	5.4	3415.0
		SA	18.3	28.2	46.5	12.7	647.7	5.5	3562.2

in the fully folded design. Moreover, the partially folded filters have area, latency, and energy consumption values in-between those of the parallel and fully folded filters. For the filters under the shift-adds architecture, this is also true, except that the parallel realizations may have less complexity than the partially folded filters. For example, while the parallel realization of Filter 1 occupies less area than all its partially folded realizations in Table V, the parallel design of Filter 2 has less area than the partially folded designs when s is 17 and 9. This is due to more possible sharing of partial products in parallel designs than those in partially folded designs and the number of filter coefficients. Note that, as s increases, the number of TMCM operations increases while that of cascaded registers in the OUTR blocks decreases in the partially folded filters. Thus, as s increases, the *NCA* values increase and the *CA* values decrease, with the exception of Filter 1 when s is 4, because N is not a multiple of s and an extra coefficient is added.

Second, consider different realizations of constant multiplications in FIR filters, namely GM and SA. For parallel designs, filters designed using generic multipliers occupy larger area and consume more power than those designed under the shift-adds architecture. For the fully folded designs, they are better in terms of area, delay, and power dissipation than those realized under the shift-adds architecture. This is because, as the number and size of coefficients increase, the area of the TMCM design under the shift-adds architecture increases. Thus, for the partially folded designs, as s increases, since the number of coefficients in the TMCM operation is decreased, the area of filter designs under the shift-adds architecture decreases and becomes better than those using generic multipliers. We note that the delay and power dissipation of filters under the shift-adds architecture are strongly related to the gate-level area and the number of logic operators in series.

To explore the impact of bwi on the FIR filter designs, Table VI presents the results of Filter 2 when it is 24. Observe that, as bwi is increased, the values of the gate-level parameters also increase, since the sizes of logic operators in these designs increase.

Table VI. Results of the Transposed Form Filter 2 Using 24-Bit Filter Input

Architecture		CA	NCA	TA	D	L	P	E
Parallel	GM	195.3	32.8	228.1	9.9	**9.9**	25.1	249.5
	SA	90.8	32.8	123.6	10.6	10.6	14.1	**150.1**
Partially Folded $s = 17$	GM	149.9	29.0	178.9	12.4	37.2	7.2	266.3
	SA	133.4	29.0	162.4	11.9	35.7	7.2	257.8
Partially Folded $s = 9$	GM	95.0	33.0	128.0	12.2	72.9	8.6	628.8
	SA	94.3	33.0	127.3	12.7	76.5	8.9	677.0
Partially Folded $s = 3$	GM	36.8	33.0	69.8	12.2	207.0	6.8	1408.9
	SA	46.6	33.0	79.6	12.3	209.8	7.1	1494.8
Fully Folded	GM	18.7	35.0	**53.7**	14.5	738.3	6.9	5088.2
	SA	23.7	35.0	58.7	14.7	748.0	7.0	5257.5

5.2.3. Linear DSP Transforms. For this experiment, we used a 16 × 16 DCT whose floating-point constants are converted to integers with a quantization value 8 and a 16 × 16 ICT matrix, namely the $ICT16_3$ of Aksoy et al. [2013a] that includes 6-bit constants. The linear DSP transforms were designed under the parallel, partially folded, and fully folded architectures. For a fair comparison between the architectures, the parallel design of a linear DSP transform was described as a sequential circuit, where the registers were added to the outputs of the combinational circuit that compute the linear DSP transform. In designs using generic multipliers, the summations of constant multiplications are described as in a binary tree so that the delay of these designs is reduced. For their multiplierless designs, the state-of-art CMVM algorithm [Aksoy et al. 2012] was used to find the fewest operations under the minimum number of adder-steps, that is, the maximum number of operations in series. Note that the algorithm of Aksoy et al. [2012] finds very similar solutions to those of matrix decomposition techniques [Chen et al. 1977; Dong and Ngan 2006] in terms of the number of operations and the butterfly structure. In the partially folded designs, the groups of input variables were formed considering the symmetric property of the DCT and ICT matrices [Banerjee et al. 2007] and three different g values were used. Table VII shows the results on linear DSP transforms obtained when the bitwidth of input variables bwi was 16. The parameters given in Table VII have the same descriptions explained for Table V.

Observe from Table VII that the realization of linear DSP transforms under the shift-adds architecture leads to designs occupying less area and consuming less power with respect to those implemented using generic multipliers. Note that it was the opposite for all fully folded and some partially folded filter designs presented in Table V. This is because the linear DSP transforms have a larger number of outputs and require more multipliers than FIR filters. The fully folded designs have less complexity with respect to the parallel designs, but larger latency, since 16 clock cycles are needed to compute the linear DSP transforms. As g is increased, hence decreasing the number of variables in each group, the latency and energy consumption of the design decrease. However, the area of the design increases in this case. When g is greater than 2, all the partially folded linear DSP transforms under the shift-adds architecture occupy larger area than their relevant parallel designs. This is because the algorithm of Aksoy et al. [2012] exploits a larger number of common partial products than those achieved in the partially folded designs by ORPHEUS.

Table VIII presents the results of the 16 × 16 ICT designed under the parallel, partially folded, and fully folded architectures when bwi was 24. Observe that an increase in bwi leads to an increase in area, delay, latency, power, and energy consumptions, since it increases the sizes of logic operators.

Table VII. Results of Linear DSP Transforms Using 16-Bit Inputs

Transform	Architecture		CA	NCA	TA	D	L	P	E
DCT	Parallel	GM	437.0	7.3	444.3	7.3	**7.3**	40.0	290.2
		SA	94.2	7.3	101.5	8.0	8.0	9.7	**77.9**
	Partially Folded $g = 8$	GM	359.4	46.4	405.8	7.9	15.7	23.8	374.5
		SA	269.2	46.4	315.7	7.8	15.7	19.2	300.5
	Partially Folded $g = 4$	GM	287.9	27.0	314.9	8.7	35.0	19.2	671.8
		SA	156.6	27.0	183.6	8.6	34.6	12.9	446.1
	Partially Folded $g = 2$	GM	205.6	14.2	219.8	9.6	76.9	12.4	953.5
		SA	81.9	14.2	96.1	9.2	73.5	6.7	492.4
	Fully Folded	GM	151.2	7.5	158.7	10.3	165.3	9.3	1537.0
		SA	41.8	7.5	**49.3**	9.5	151.7	4.4	667.5
ICT	Parallel	GM	191.6	6.4	197.9	6.9	**6.9**	18.8	129.8
		SA	68.1	6.4	74.5	7.0	7.0	8.3	**57.9**
	Partially Folded $g = 8$	GM	186.5	24.7	211.2	6.7	13.4	11.2	150.3
		SA	170.5	24.7	195.2	7.0	14.0	10.3	144.0
	Partially Folded $g = 4$	GM	167.7	21.6	189.4	8.0	32.1	12.2	391.5
		SA	121.0	21.6	142.7	7.9	31.6	9.8	309.2
	Partially Folded $g = 2$	GM	120.9	12.3	133.2	8.6	68.6	7.9	542.3
		SA	64.1	12.3	76.3	9.4	75.4	5.3	399.7
	Fully Folded	GM	92.4	6.5	98.8	9.3	148.1	6.3	932.7
		SA	32.9	6.5	**39.4**	8.9	141.8	3.6	510.3

Table VIII. Results of the 16 × 16 ICT Using 24-Bit Inputs

Architecture		CA	NCA	TA	D	L	P	E
Parallel	GM	273.8	8.5	282.3	8.8	8.8	28.7	252.5
	SA	96.3	8.5	104.8	8.6	**8.6**	12.0	**102.8**
Partially Folded $g = 8$	GM	263.9	34.5	298.4	8.9	17.8	16.0	285.6
	SA	239.6	34.5	274.1	8.8	17.6	14.6	257.4
Partially Folded $g = 4$	GM	236.8	29.8	266.6	9.5	38.1	17.1	651.3
	SA	168.5	29.8	198.3	9.2	36.8	13.7	504.3
Partially Folded $g = 2$	GM	174.7	16.6	191.3	10.3	82.7	11.4	943.0
	SA	88.4	16.6	105.0	10.5	84.1	7.5	630.6
Fully Folded	GM	134.2	8.6	142.9	11.5	183.3	9.0	1650.1
	SA	45.3	8.6	**54.0**	10.6	170.3	5.0	851.6

6. CONCLUSIONS

This article presented an efficient approximate algorithm, ORPHEUS, developed for the optimization of design complexity in TMCM operations and described the multiplierless design of folded DSP blocks, including the transposed form FIR filters, DCTs, and ICTs. Experimental results on TMCM algorithms showed that ORPHEUS generally finds better solutions than existing algorithms, although it cannot always guarantee the best solution. Experimental results on TMCM architectures indicated that the use of the *mux-add* architecture and the solutions of ORPHEUS become more efficient with respect to the *mux-mul* architecture on $m \times n$ TMCM instances with a large m and a small n value. However, there still exist TMCM instances on which the *mux-mul* architecture leads to TMCM designs with the least area. Experimental results on FIR filters and linear DSP transforms showed that the fully and partially folded architectures present alternative circuits so that a designer can choose the one that fits best in an application. It was indicated that the folded design of DSP blocks including a small number of outputs

and a small number of multipliers may occupy less area when realized using generic multipliers, rather than its design under the shift-adds architecture.

REFERENCES

L. Aksoy, E. Costa, P. Flores, and J. Monteiro. 2008. Exact and approximate algorithms for the optimization of area and delay in multiple constant multiplications. *IEEE Trans. Comput.-Aided Des. Integr. Circ. Syst.* 27, 6, 1013–1026.

L. Aksoy, E. Costa, P. Flores, and J. Monteiro. 2012. Multiplierless design of linear dsp transforms. In *VLSI-SoC: Advanced Research for Systems on Chip*, S. Mir, C.-Y. Tsui, R. Reis, and O. Choy, Eds., Springer, 73–93.

L. Aksoy, E. Costa, P. Flores, and J. Monteiro. 2013a. Exploration of tradeoffs in the design of integer cosine transforms for image compression. In *Proceedings of the IEEE European Conference on Circuit Theory and Design (ECCTD'13)*. 1–4.

L. Aksoy, P. Flores, and J. Monteiro. 2013b. Towards the least complex time-multiplexed constant multiplication. In *Proceedings of the IFIP/IEEE International Conference on Very Large Scale Integration (VLSI-SoC'13)*. 328–331.

L. Aksoy, P. Flores, and J. Monteiro. 2014. Optimization of design complexity in time-multiplexed constant multiplications. In *Proceedings of the Design, Automation, and Test in Europe Conference (DATE'14)*. 1–4.

L. Aksoy, E. Gunes, and P. Flores. 2010. Search algorithms for the multiple constant multiplications problem: Exact and approximate. *J. Microprocess. Microsyst.* 34, 5, 151–162.

N. Banerjee, G. Karakonstantis, and K. Roy. 2007. Process variation tolerant low power DCT architecture. In *Proceedings of the Design, Automation, and Test in Europe Conference (DATE'07)*. 630–635.

P. Barth. 1995. A davis-putnam based enumeration algorithm for linear pseudo-boolean optimization. https://www.princeton.edu/∼chaff/papers/barth95davisputnam.pdf.

R. Bernstein. 1986. Multiplication by integer constants. *Softw. Pract. Exper.* 16, 7, 641–652.

P. Bougas, P. Kalivas, A. Tsirikos, and K. Pekmestzi. 2005. Pipelined array-based FIR filter folding. *IEEE Trans. Circ. Syst.* 52, 1, 108–118.

N. Boullis and A. Tisserand. 2005. Some optimizations of hardware multiplication by constant matrices. *IEEE Trans. Comput.* 54, 10, 1271–1282.

W.-K. Cham. 1989. Development of integer cosine transforms by the principle of dyadic symmetry. *IEE Proc. I Comm. Speech Vis.* 136, 4, 276–282.

J. Chen and C.-C. Chang. 2009. High-level synthesis algorithm for the design of reconfigurable constant multiplier. *IEEE Trans. Comput.-Aided Des. Integr. Circ. Syst.* 28, 12, 1844–1856.

W. Chen, C. Smith, and S. Fralick. 1977. A fast computational algorithm for the discrete cosine transform. *IEEE Trans. Comm.* 25, 9, 1004–1009.

K.-M. Cheung, F. Pollara, and M. Shahshahani. 1991. Integer cosine transform for image compression. http://ntrs.nasa.gov/archive/nasa/casi.ntrs.nasa.gov/19940025116.pdf.

S. Demirsoy, R. Beck, A. Dempster, and I. Kale. 2003. Reconfigurable implementation of recursive DCT kernels for reduced quantization noise. In *Proceedings of the IEEE International Symposium on Circuits and Systems (ISCAS'03)*. 289–292.

S. Demirsoy, A. Dempster, and I. Kale. 2004. Efficient implementation of digital filters using novel reconfigurable multiplier blocks. In *Proceedings of the Asilomar Conference on Signals, Systems and Computers (ACSSC'04)*. 461–464.

S. Demirsoy, I. Kale, and A. Dempster. 2007. Reconfigurable multiplier constant blocks: Structures, algorithm and applications. *Circ. Syst. Signal Process.* 26, 6, 793–827.

A. Dempster and M. Macleod. 1995. Use of minimum-adder multiplier blocks in FIR digital filters. *IEEE Trans. Circ. Syst. II* 42, 9, 569–577.

A. Dempster and M. Macleod. 2004. Digital filter design using subexpression elimination and all signed-digit representations. In *Proceedings of the IEEE International Symposium on Circuits and Systems (ISCAS'04)*. 169–172.

J. Dong and K. N. Ngan. 2006. 16×16 integer cosine transform for HD video coding. In *Proceedings of the 7^{th} Pacific Rim Conference on Advances in Multimedia Information Processing (PCM'06)*. 114–121.

M. Ercegovac and T. Lang. 2003. *Digital Arithmetic*. Morgan Kaufmann, San Fransisco.

O. Gustafsson, A. Dempster, and L. Wanhammar. 2002. Extended results for minimum-adder constant integer multipliers. In *Proceedings of the IEEE International Symposium on Circuits and Systems (ISCAS'02)*. 73–76.

R. Hartley. 1996. Subexpression sharing in filters using canonic signed digit multipliers. *IEEE Trans. Circ. Syst. II* 43, 10, 677–688.

Y.-H. Ho, C.-U. Lei, H.-K. Kwan, and N. Wong. 2008. Global optimization of common subexpressions for multiplierless synthesis of multiple constant multiplications. In *Proceedings of the Asia and South Pacific Design Automation Conference (ASP-DAC'08)*. 119–124.

H. Hou. 1987. A fast recursive algorithm for computing the discrete cosine transform. *IEEE Trans. Acoust. Speech Signal Process.* 35, 10, 1455–1461.

K. Johansson, O. Gustafsson, L. Debrunner, and L. Wanhammar. 2011. Minimum adder depth multiple constant multiplication algorithm for low power fir filters. In *Proceedings of the IEEE International Symposium on Circuits and Systems (ISCAS'11)*. 1439–1442.

K. Johansson, O. Gustafsson, and L. Wanhammar. 2005. A detailed complexity model for multiple constant multiplication and an algorithm to minimize the complexity. In *Proceedings of the IEEE European Conference on Circuit Theory and Design (ECCTD'05)*. 465–468.

H.-J. Kang and I.-C. Park. 2001. FIR filter synthesis algorithms for minimizing the delay and the number of adders. *IEEE Trans. Circ. Syst. II: Analog Digital Signal Process.* 48, 8, 770–777.

V. Karkala, J. Wanstrath, T. Lacour, and S. P. Khatri. 2010. Efficient arithmetic sum-of-product (SOP) based multiple constant multiplication for fft. In *Proceedings of the International Conference on Computer-Aided Design (ICCAD'10)*. 735–738.

M. Kumm, P. Zipf, M. Faust, and C.-H. Chang. 2012. Pipelined adder graph optimization for high speed multiple constant multiplication. In *Proceedings of the IEEE International Symposium on Circuits and Systems (ISCAS'12)*. 49–52.

H. Nguyen and A. Chatterjee. 2000. Number-splitting with shift-and-add decomposition for power and hardware optimization in linear DSP synthesis. *IEEE Trans. VLSI Syst.* 8, 4, 419–424.

K. Parhi. 1995. High-level algorithm and architecture transformations for DSP synthesis. *J. VLSI Signal Process.* 9, 1, 121–143.

K. Parhi. 1999. *VLSI Digital Signal Processing Systems: Design and Implementation*. John Wiley and Sons.

I.-C. Park and H.-J. Kang. 2001. Digital filter synthesis based on minimal signed digit representation. In *Proceedings of the Design Automation Conference (DAC'01)*. 468–473.

F. Qureshi and O. Gustafsson. 2009. Low-complexity reconfigurable complex constant multiplication for FFTs. In *Proceedings of the IEEE International Symposium on Circuits and Systems (ISCAS'09)*. 24–27.

N. Sidahao, G. Constantinides, and P. Cheung. 2004. Multiple restricted multiplication. In *Proceedings of the International Conference on Field-Programmable Logic and Applications (FPL'04)*. 374–383.

N. Sidahao, G. Constantinides, and P. Cheung. 2005. A heuristic approach for multiple restricted multiplication. In *Proceedings of the IEEE International Symposium on Circuits and Systems (ISCAS'05)*. 692–695.

J. Thong and N. Nicolici. 2010. A novel optimal single constant multiplication algorithm. In *Proceedings of the Design Automation Conference (DAC'10)*. 613–616.

P. Tummeltshammer, J. Hoe, and M. Puschel. 2007. Time-multiplexed multiple-constant multiplication. *IEEE Trans. Comput.-Aided Des. Integr. Circ. Syst.* 26, 9, 1551–1563.

R. Turner and R. Woods. 2004. Highly efficient, limited range multipliers for lut-based FPGA architectures. *IEEE Trans. VLSI Syst.* 12, 10, 1113–1117.

Y. Voronenko and M. Puschel. 2007. Multiplierless multiple constant multiplication. *ACM Trans. Algor.* 3, 2.

L. Wanhammar. 1999. *DSP Integrated Circuits*. Academic Press.

Received May 2014; revised August 2014; accepted August 2014