Abstract—The multiplication of constants by a data input is an essential operation in digital signal processing (DSP) systems. For applications requiring a large number of constant multiplications under stringent hardware constraints, it is generally realized under a folded architecture, where a single constant selected from a set of multiple constants is multiplied by the data input at each time, called time-multiplexed constant multiplication (TMCM).

This paper addresses the problem of optimizing the complexity of a TMCM design and introduces an algorithm that finds the least complex TMCM design by sharing the logic operators, i.e., adders, subtractors, adders/subtractors, and multiplexors (MUXes). It includes efficient search methods, yielding better results than existing TMCM algorithms.

I. INTRODUCTION

The multiple constant multiplications (MCM) operation, realizing the multiplication of constants by an input variable, dominates the design complexity of many DSP systems. Since these constants are determined beforehand, the constant multiplications are generally replaced with addition, subtraction, and shift operations. In hardware, shifts can be realized using only wires without representing any cost. Over the years, efficient algorithms were proposed for the optimization of the number of adders and subtractors in the MCM operation [1]–[3].

However, as the number and size of the constants increase, the increase in complexity is inevitable if the constant multiplications are realized simultaneously in a parallel architecture. Hence, in DSP applications targeting low complexity, a folded architecture [4] is preferred, where the common resources are allowed to be shared in time, taking into account an increase in latency. In folded design of DSP systems, TMCM is a fundamental operation. Given a set of \( n \) constants, its straightforward realization, the \( \text{max-mul} \) architecture, includes a generic multiplier and an \( n \)-to-1 MUX, where \( i \) denotes the select input with \( 0 \leq i \leq n-1 \) (Fig. 1a). Another realization, the \( \text{mcm-mux} \) architecture, uses an \( n \)-to-1 MUX and an MCM block including adders and subtractors that implement the constant multiplications (Fig. 1b). An alternative solution, the \( \text{mux-add} \) architecture, includes adders, subtractors, and adders/subtractors (determined by a select input) with MUXes (Fig. 1c). It increases the number of constants that a logic operator can generate and provides the possible sharing of logic operators. Hence, it may yield less complex TMCM designs when compared to other architectures [5]–[10].

The single-input single-output (SISO) and single-input multiple-output (SIMO) TMCM blocks occur in folded design of filters and filter banks, respectively [8]. In this work, the constants of a TMCM operation are represented in an \( m \times n \) matrix \( C \), where \( m \) and \( n \) are the number of outputs and time slots (the number of constants in an array), respectively. Thus, given \( C \), the TMCM problem is defined as finding a set of logic operators, i.e., adders, subtractors, adders/subtractors, and MUXes, that realizes the TMCM operation and leads to a TMCM design with minimum complexity. Existing methods targeting the \( \text{mux-add} \) architecture consider different design platforms, i.e., application specific integrated circuits (ASIC) and field programmable gate arrays. They reduce the complexity of a TMCM operation by sharing the basic structures and merging the single/multiple constant multiplication graphs. The exact method [5] can only be applied to a small number of constants and the solution quality of the approximate methods [6]–[10] depends heavily on the TMCM instance.

This paper introduces our TMCM algorithm ORPHEUS that targets the \( \text{mux-add} \) architecture and the SIMO TMCM operations under an ASIC design platform, where the area values of logic operators are taken from a standard cell library. It includes two main parts: optimal and heuristic. In its optimal part, the arrays of constants, that can be realized using a single logic operator, are synthesized. In its heuristic part, an array of constants is chosen, its alternative implementations are found, and it is synthesized with the one that has the smallest cost. ORPHEUS iterates until all arrays of constants are synthesized.

II. ORPHEUS: A TMCM ALGORITHM

The steps of ORPHEUS, that will be described more briefly in the next subsections, are given below. 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 found by the MCM algorithm [3]. Also, the matrix \( S \), that initially consists of a single array of \( n \) 1s, will include the synthesized arrays of constants. Note that the input variable(s) is (are) always available. The set \( I \) will include logic operators realizing the TMCM operation.

1) Given the constant matrix of the TMCM operation \( C \), determine the non-redundant target matrix \( T \).
2) In an iterative manner, find an optimal realization of a row of \( T \), \( T_i \), if it exists, remove \( T_i \) 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 whose one of inputs is a row of \( S \) or its shifted version and that has the smallest estimated cost value \( \text{ecost}_1 \).
6) Find a realization of \( T_i \) under the \( \text{mcm-mux} \) architecture and compute its estimated cost value \( \text{ecost}_2 \).
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 value \( ecost_i \).

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) Design the MCM block for the constants of \( M \) using the MCM algorithm [3], update and return \( I \).

In following subsections, these steps are described through an example with a 4 × 2 matrix \( C=\{5 5; 2 1; 12 10; 92 196\} \).

**A. 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 and \( C_i \cdot 2^{-ls_i} \) is added to \( T \) without repetition. For our example, \( T \) is obtained as \( \{5 5; 2 1; 6 5; 23 49\} \).

**B. Step 2: Finding Optimal Realizations**

First, we check each row of \( T, T_i \), if it includes the same constant at its every time slot. If such an array of constants exists, it is removed from \( T \) to \( S \) and this constant is added to \( M \) without repetition. For our example, the row of \( T, \{5 5\} \), is removed 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 removed from \( T \) to \( S \) and this MUX is added to \( I \). For our example, the row of \( T, \{2 1\} \), can be realized using a single MUX.

Third, we check each \( T_i \) if it can be realized using a single operation, i.e., an adder, a subtractor, or an adder/subtractor, whose inputs are the rows of \( S \) or their shifted versions. While searching such realizations, the array of constants \( D \) is computed as follows for each \( S_j, ls_j \), and \( SM_k \):

\[
D \cdot 2^{-ls_j} = T_i - (S_j \cdot 2^{-ls_j}) \ast SM_k
\]  

(1)

where \( ls_j \geq 0 \) denotes the left shift of \( D, 1 \leq j \leq |S| \), \( ls_j \) stands for the left shift of \( S_j \), \( 0 \leq ls_j \leq mbc(T_j) - mbc(S_j) + 1 \) (where \( mbc(A) \) function determines the maximum bit-width of constants in an array \( A \)), \( SM \) is a \( 2^n \times n \) sign matrix including all possible signs, i.e., \( 1 \) and \( -1 \), for the \( 1 \times n \) array \( S_j \), and \( 1 \leq k \leq 2^n \). Note that \( \ast \) denotes the element-by-element product of arrays, e.g., in \( C = A \ast B \), each entry in the array \( C \) is the product of the corresponding entries in arrays \( A \) and \( B \).

Thus, if \( D \) is equal to a row of \( S \), then \( T_i \) can be realized using a single operation. If \( SM_k \) includes all \( 1s \), then the operation is an adder. If \( SM_k \) includes all \( -1s \), 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 removed from \( T \) to \( S \). In our example, for the row of \( T, \{6 5\} \), one possible \( D \) is found as \( \{1 1\} \cdot 2^{-2} = \{6 5\} - (\{2 1\} \cdot 2^0) \ast \{1 1\} \), which is also a row of \( S \). This realization of \( \{6 5\} \) needs a single adder.

This procedure iterates until there exists no such row of \( T \). For our example, \( T, S, \) and \( M \) are respectively found as \( \{23 49\}, \{1 1; 5 5; 2 1; 6 5\}, \) and \( \{1, 5\} \) at the end of this step.

**C. Step 4: Selection of an Array of Constants**

At this point, there exist array(s) of constants in \( T \) which require(s) partial arrays of constants. In each iteration of ORPHEUS, we select one of them that has the minimum \( tnzd \) value, denoting the total number of nonzero digits of constants in an array under the canonical signed digit (CSD) representation\(^1\) [1]. A smaller \( tnzd \) value roughly indicates that the array can be realized using less number of logic operators.

Hence, this metric is preferred to 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). It is also used to ensure the convergence of each realization to an array consisting of 1 or its shifted versions (Step 5).

**D. Step 5: First Alternative Realization**

Possible realizations of a row of \( T, T_i \), are found based on the computation in Eqn. 1. To avoid the exponential number of sign values for the constants of \( S_j \), we traverse each constant, \( S_j[h] \) with \( 1 \leq h \leq n \), compute \( D[h] \) for both sign of \( S_j[h] \), i.e., 1 and \(-1 \), and choose the one that leads to \( D[h] \) with minimum number of nonzero digits under CSD. After \( D \) and the sign array, corresponding to \( SM_k \) in Eqn. 1, are determined, we check if the \( tnzd \) value of \( D \) is less than that of \( T_i \) to ensure the convergence of \( D \). If so, the type of operation is determined as done in Step 2 and its cost in a given standard cell library, \( cost_{op} \), is found [9]. We also estimate the cost of \( D \) as if it will be realized under the mcm-mux architecture. First, the different constants of \( D \) and its quantities, i.e., \( r \), are found. If \( r > 1 \), the cost of an \( r \)-to-1 MUX in a given standard cell library, \( cost_{max} \), is found [9]. Otherwise, it is set to 0. Second, we convert each constant of \( D \) to an odd constant, find the minimum number of operations determined by the algorithm [11] required for its multiplication by the input variable, compute the cost values of these operations assuming them as adders, and add them to \( cost_{add} \), which was initially set to 0. The \( cost_{add} \) value is computed without repeating the same odd constants. Thus, the implementation cost of \( T_i \) is found as \( cost_{op} + cost_{max} + cost_{add} \). After all possible realizations of \( T_i \) are found considering each \( S_j \) with all its possible \( ls_j \) values, we choose the one with the minimum cost value that is assigned to \( ecost_1 \).

For the row of \( T, \{23 49\} \), its realization with the minimum cost is found as \( [3 6] \cdot 2^3 + ([1 1] \cdot 2^0) \ast [-1 1] \), requiring an adder/subtractor and the partial array of constants \( [3 6] \) which is estimated to need a 2-to-1 MUX and an adder.

**E. Step 6: Second Alternative Realization**

The cost value of the mcm-max realization of \( T_i \) is computed as described for the estimation of the cost of \( D \) in Step 5, except that the elements of \( M \) are considered in this case. 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 \( cost_{max} + cost_{add} \). For our example, the estimated cost of \( \{23 49\} \) includes a 2-to-1 MUX and 4 adders.

**F. 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 partial array of constants. This problem is formalized as a 0-1 integer linear programming (ILP) problem. First, we find all realizations of each constant of \( T_i \) by decomposing its nonzero digits in two partial terms when it

---

\(^1\)An integer can be written in CSD using \( k \) digits as \( \sum_{j=0}^{k} d_j2^j \), where \( d_j \in \{0, 1, T\} \) and \( T \) denotes \(-1 \). The nonzero digits are not adjacent and a constant is represented with minimum number of nonzero digits under CSD.
Boolean network that includes only AND and OR gates. For an
is found as a linear combination of optimization variables
ations, all including minimum number of nonzero digits.
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 \( \text{AND}_{p_1+p_2} \) and \( \text{AND}_{p_3+p_1} \), are generated due to the commutative law of addition. For a subtractor, we assume that the first input is the partial term with the positive sign and the second input is the one with the negative sign and we generate an AND gate denoted as \( \text{AND}_{p_1-p_2} \). For each constant of \( T_i \), \( T_i[h] \), where \( 1 \leq h \leq n \), an OR gate, \( O_T[h] \), 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. The network generated for the row of \( T \), \([23,49]\), is shown in Fig. 2b.

Third, the objective function of the 0-1 ILP problem is found as a linear combination of optimization variables whose weights are 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 [12]. For example, a 2-input AND gate, \( c = a \land b \), is translated to CNF as \((a+\overline{c})(b+\overline{c})\overline{a}+b+c \) and converted to linear constraints as \( a-c \geq 0, b+c \geq 0, -a-b+c \geq -1 \). The outputs of OR gates associated with constants of \( T_i \), \( O_{T[i,h]} \), are set to 1, since they need to be implemented.

Forth, a minimum solution is found using a 0-1 ILP solver and based on the selected operations (the outputs of AND gates set to 1 in the solution), the realization of \( T_i \) is formed as:

\[
T_i = G_1 \cdot 2^{l_1} + (G_2 \cdot 2^{l_2}) \cdot \text{SA}
\]

where \( G_1 \) \((G_2)\) includes the first (second) inputs of the selected operations for each constant of \( T_i \), \( l_1 \) \((l_2)\) is the amount of left shift of \( G_1 \) \((G_2)\), and \( \text{SA} \) denotes the sign array and is determined based on the type of each operation, i.e., if \( T_i[h] \) is realized using an adder, then \( \text{SA[h]} \) 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 \([7,65] \cdot 2^0 + ([1,1] \cdot 2^4) + [1,-1] \) according to Eqn. 2.

Finally, we compute the cost of this realization as done in Step 5. To do so, we define the type of operation based on

\[\text{cost} = \sum p_k \cdot \text{bw}(p_k) \cdot \text{nzd}(p_k)\]

for any of \( G_1 \) and \( G_2 \), if it is not included in \( S \), then we estimate its cost in terms of \text{cost}_{\text{max}} \) and \text{cost}_{\text{adj}} \). For our example, the type of operation is adder/subtractor and only the cost of \([7,65]\) is estimated.

This solution leads to an operation with common partial terms at the inputs. In order to include an information about the design complexity into the 0-1 ILP problem, for each partial term \( p_k \), we also take into account its bitwidth \( \text{bw}(p_k) \) 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 \( \text{bw}(p_k) \cdot \text{nzd}(p_k) \). In this case, 23 and 49 are implemented as \( 8+15 \) and \( 64-15 \), respectively and the realization of \([23,49]\) is determined as \([18] \cdot 2^3 + ([15,15] \cdot 2^0) \cdot [1-1] \), requiring an adder/subtractor and the partial arrays \([1,8]\) and \([15,15]\).

Among these two possible realizations, we determine the one with the minimum cost which is assigned to \( \text{ecost} \). For our example, the second one has the minimum cost.

\[\text{G. Step 8: Selection of the Realization with Minimum Cost}\]

Among the realizations found in Steps 5, 6, and 7, if \( \text{ecost} \) is the minimum, we add the required partial array of constants (found in Step 5) to \( T \). If \( \text{ecost} \) is the minimum, we remove \( T_i \) from \( S \) and add the odd versions of constants in \( T_i \) to \( M \) without repetition. Otherwise, we add the required partial 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. For our example, \( \text{ecost} \) is the minimum cost value, and thus, \([3,6]\) is added to \( T \).

\[\text{H. Step 9: Iterations in ORPHEUS}\]

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

\[\text{I. Step 10: Realization of the Necessary MCM Block}\]

If \( M \) includes constants other than 1, the MCM algorithm of \([3]\) is applied to find the fewest number of operations realizing these constant multiplications and these operations

\[\text{Fig. 2. (a) Possible implementations of 23 and 49 under MSD; (b) Boolean network generated for [23,49].}\]

\[\text{Fig. 3. Time-multiplexed realization of C = [5,5;2;12;10;92,196].}\]
are added to \( I \). Also, for each chosen \( mcm-mux \) realization, the necessary MUX is added to \( I \). The solution on our example \( C \) is given in Fig. 3, where the select inputs of the MUX and adders/subtractors are not shown for the sake of clarity.

### III. Experimental Results

This section presents the results of ORPHEUS, comparing them to those of prominent TMC algorithms. ORPHEUS was written in MATLAB and was run on a PC with Intel Xeon at 2.4GHz and 10GB memory. Note that DAGfusion [9] was obtained from www.spiral.net and its exhaustive search on all possible orderings of SCM graphs was limited to 10,000 orderings, since it may take enormous CPU time.

As the first experiment set, we used randomly generated \( 1 \times n \) TMC instances, where the maximum bitwidth of constants (mbc) varies in between 6 and 12 in increment of 2 and \( n \) ranges between 2 and 18 in increment of 4. For each group, there were 30 instances, a total of 600 instances. Table I presents the results of algorithms, where \#BS denotes the number of better solutions that ORPHEUS found against DAGfusion [9] in terms of complexity, computed using the 0.18-\( \mu \)m standard cell library when the bitwidth of the input variable (bwi) was 16. Min Gain, Avg Gain and Max Gain denote the minimum, average, and maximum gain in percentage obtained by ORPHEUS over DAGfusion, respectively. Average runtime of both methods is given in seconds.

Observe from Table I that ORPHEUS obtains better solutions than DAGfusion on more than half of the total number of TMC instances, \( i.e. \), 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 TMC instance with 12-bit 2 constants. Note also that its maximum gain 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. Furthermore, it requires less CPU time than DAGfusion on instances with \( n \geq 6 \).

As the second experiment set, we used two benchmark sets given in [10]. Tables II-III present the results of algorithms, where Cost, denoting the design complexity, was computed using the 0.18-\( \mu \)m standard cell library when bwi was 8. The results of other TMC algorithms were taken from [10].

### TABLE I. SUMMARY OF ALGORITHMS ON SISO TMC INSTANCES.

<table>
<thead>
<tr>
<th>mbc</th>
<th>n</th>
<th>2</th>
<th>6</th>
<th>10</th>
<th>14</th>
<th>18</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>#BS</td>
<td>24</td>
<td>16</td>
<td>20</td>
<td>18</td>
<td>26</td>
</tr>
<tr>
<td></td>
<td>BS</td>
<td>6</td>
<td>8.68</td>
<td>12.91</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Avg Gain</td>
<td>0.76</td>
<td>8.08</td>
<td>11.29</td>
<td>19.34</td>
<td>21.5</td>
</tr>
<tr>
<td></td>
<td>Max Gain</td>
<td>30.30</td>
<td>31.90</td>
<td>28.98</td>
<td>28.21</td>
<td>32.75</td>
</tr>
<tr>
<td></td>
<td>CPU DAGfusion</td>
<td>0.7</td>
<td>2.9</td>
<td>8.08</td>
<td>11.29</td>
<td>19.34</td>
</tr>
<tr>
<td></td>
<td>CPU ORPHEUS</td>
<td>0.04</td>
<td>0.33</td>
<td>1.07</td>
<td>1.50</td>
<td>2.15</td>
</tr>
<tr>
<td>8</td>
<td>#BS</td>
<td>22</td>
<td>15</td>
<td>16</td>
<td>19</td>
<td>24</td>
</tr>
<tr>
<td></td>
<td>BS</td>
<td>6.99</td>
<td>1.27</td>
<td>0.75</td>
<td>7.58</td>
<td>7.51</td>
</tr>
<tr>
<td></td>
<td>Avg Gain</td>
<td>6.99</td>
<td>1.27</td>
<td>0.75</td>
<td>7.58</td>
<td>7.51</td>
</tr>
<tr>
<td></td>
<td>Max Gain</td>
<td>28.48</td>
<td>27.66</td>
<td>22.96</td>
<td>28.51</td>
<td>27.75</td>
</tr>
<tr>
<td></td>
<td>CPU DAGfusion</td>
<td>0.84</td>
<td>4.84</td>
<td>11.41</td>
<td>15.55</td>
<td>20.42</td>
</tr>
<tr>
<td></td>
<td>CPU ORPHEUS</td>
<td>1.39</td>
<td>1.75</td>
<td>2.24</td>
<td>2.83</td>
<td>4.19</td>
</tr>
<tr>
<td>10</td>
<td>#BS</td>
<td>25</td>
<td>18</td>
<td>24</td>
<td>19</td>
<td></td>
</tr>
<tr>
<td></td>
<td>BS</td>
<td>12.06</td>
<td>-19.22</td>
<td>-14.69</td>
<td>-21.54</td>
<td>-20.33</td>
</tr>
<tr>
<td></td>
<td>Avg Gain</td>
<td>13.88</td>
<td>3.17</td>
<td>4.17</td>
<td>6.80</td>
<td>4.75</td>
</tr>
<tr>
<td></td>
<td>Max Gain</td>
<td>43.23</td>
<td>39.97</td>
<td>23.96</td>
<td>27.91</td>
<td>24.52</td>
</tr>
<tr>
<td></td>
<td>CPU DAGfusion</td>
<td>1.00</td>
<td>23.54</td>
<td>65.72</td>
<td>276.95</td>
<td>515.20</td>
</tr>
<tr>
<td></td>
<td>CPU ORPHEUS</td>
<td>2.85</td>
<td>4.89</td>
<td>7.12</td>
<td>8.68</td>
<td>13.42</td>
</tr>
</tbody>
</table>

### TABLE II. AREA ESTIMATION FOR \( C = \{ 256 162 50 26 15 8 4 2 \} \).

<table>
<thead>
<tr>
<th>Method</th>
<th>Add</th>
<th>Sub</th>
<th>Add/Sub</th>
<th>MUX</th>
<th>Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>[9]</td>
<td>362</td>
<td>392</td>
<td>473</td>
<td>6365</td>
<td></td>
</tr>
</tbody>
</table>

Observe that ORPHEUS finds a TMC 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.

### IV. Conclusions

This paper introduced the TMC algorithm ORPHEUS that is equipped with efficient methods to maximize the sharing of logic operators. It was shown that it generally finds better solutions than previously proposed algorithms.

### REFERENCES