Microprocessors and Microsystems 34 (2010) 151–162

Contents lists available at ScienceDirect

Microprocessors and Microsystems

journal homepage: www.elsevier.com/locate/micpro

Search algorithms for the multiple constant multiplications problem: Exact and approximate

Levent Aksoy [a,*], Ece Olcay Güneş [a], Paulo Flores [b]

[a] Istanbul Technical University, Department of Electronics and Communication Engineering, 34469, Maslak, Istanbul, Turkey
[b] INESC-ID / IST, TU Lisbon, Rua Alves Redol, 1000-029, Lisbon, Portugal

ARTICLE INFO

Article history:
Available online 14 October 2009

Keywords:
Multiple constant multiplications problem
Depth-first search
Breadth-first search
Graph-based algorithms
Finite Impulse Response filters

ABSTRACT

This article addresses the multiplication of one data sample with multiple constants using addition/subtraction and shift operations, i.e., the multiple constant multiplications (MCM) operation. In the last two decades, many efficient algorithms have been proposed to implement the MCM operation using the fewest number of addition and subtraction operations. However, due to the NP-hardness of the problem, almost all the existing algorithms have been heuristics. The main contribution of this article is the proposal of an exact depth-first search algorithm that, using lower and upper bound values of the search space for the MCM problem instance, finds the minimum solution consuming less computational resources than the previously proposed exact breadth-first search algorithm. We start by describing the exact breadth-first search algorithm that can be applied on real mid-size instances. We also present our recently proposed approximate algorithm that finds solutions close to the minimum and is able to compute better bounds for the MCM problem. The experimental results clearly indicate that the exact depth-first search algorithm can be efficiently applied to large size hard instances that the exact breadth-first search algorithm cannot handle and the heuristics can only find suboptimal solutions.

1. Introduction

In several computationally intensive Digital Signal Processing (DSP) operations, such as Finite Impulse Response (FIR) filters as illustrated in Fig. 1 and Fast Fourier Transforms (FFT), the same input is multiplied by a set of constant coefficients, an operation known as multiple constant multiplications (MCM). The MCM operation is a central operation and performance bottleneck in many applications such as, digital audio and video processing, wireless communication, and computer arithmetic. Hence, hardwired dedicated architectures are the best option for maximum performance and minimum power consumption.

In hardware, the constant multiplications are generally realized using addition/subtraction and shifting operations in the shifts-add architecture [1] due to two main reasons: (i) the design of a multiplication operation is expensive in terms of area, delay, and power consumption. Although the relative cost of an adder and a multiplier in hardware depends on the adder and multiplier architectures, note that a $k \times k$ array multiplier has k times the area and twice the latency of the slowest ripple carry adder; (ii) the constants to be multiplied in the MCM operation are determined by the DSP algorithms beforehand. Hence, the full-flexibility of a multiplier is not required in the implementation of the MCM operation. Thus, since shifts are free in terms of hardware, the MCM problem is defined as finding the minimum number of addition/subtraction operations that implement the constant multiplications and has been proven to be an NP-hard problem in [2].

The multiple constant multiplications also allow for a great reduction in the number of operations, consequently in area and power dissipation of the MCM design, when the common partial products are shared among different constant multiplications. As a small example, suppose the constant multiplications $11x$ and $13x$ as given in Fig. 2a. The shift-adds implementations of constant multiplications are presented in Fig. 2b–c. Observe that while the multiplierless implementation without partial product sharing requires four operations, Fig. 2b, the sharing of partial product $9x$ in both multiplications reduces the number of operations to 3, Fig. 2c.

The last two decades have seen much tremendous effort on the design of efficient algorithms proposed for the MCM problem. These methods focus on the maximization of the partial product sharing and can be categorized in two classes: Common Subexpression Elimination (CSE) algorithms [3–7] and graph-based techniques [8–12]. In CSE algorithms, initially, the constants are defined under a number representation namely, binary, Canonical Signed Digit (CSD) [4], or Minimal Signed Digit (MSD) [6]. Then, all possible subexpressions are extracted from the representations of the constants and the

* Corresponding author. Tel.: +90 212 2856733; fax: +90 212 2856535.
E-mail addresses: aksoyl@itu.edu.tr (L. Aksoy), ece.gunes@itu.edu.tr (E.O. Güneş), pff@inesc-id.pt (P. Flores).

0141-9331/$ - see front matter © 2009 Elsevier B.V. All rights reserved.
doi:10.1016/j.micpro.2009.10.001

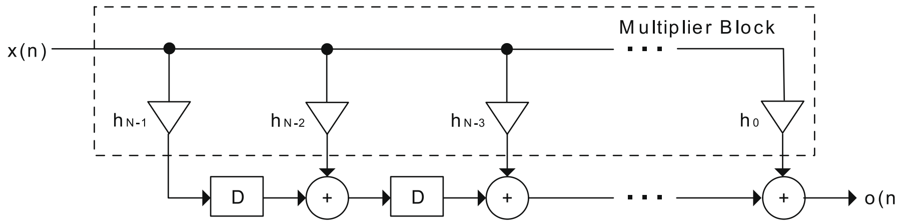


Fig. 1. Transposed form of a hardwired FIR filter implementation.

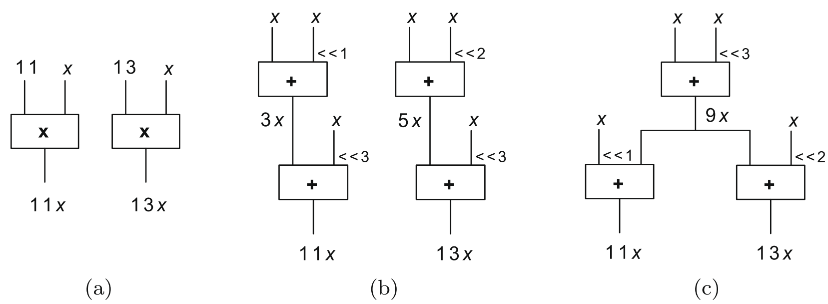


Fig. 2. (a) Multiple constant multiplications; shift-adds implementations: (b) without partial product sharing; (c) with partial product sharing.

"best" subexpression, generally the most common, is chosen to be shared in constant multiplications. For the example given in Fig. 2, the sharing of partial product 9x is possible, when constants in multiplications 11x and 13x are defined in binary, i.e., $11x = (1011)_{bin}x$ and $13x = (1101)_{bin}x$ respectively, and the most common partial product, i.e., $(1001)_{bin}x = x + (x \ll 3) = 9x$, is identified in both multiplications. The exact CSE algorithms of [13,14] formalize the MCM problem as a 0–1 integer linear programming problem and find the minimum number of operations solution by maximizing the partial product sharing. However, the search space of a CSE algorithm is restricted with the possible subexpressions that can be extracted from the representations of constants.

Furthermore, to increase the number of possible implementations of a constant, consequently the partial product sharing, the algorithm of [15] applies the CSE technique of [4] to all signed-digit representations of a constant taking into account up to m additional signed digits to the CSD representation, i.e., for a constant including n signed digits in CSD, the constant is represented with up to $n + m$ signed digits. This approach is applied to multiple constants using exhaustive searches in [16]. Also, the heuristic of [17] obtains much better solutions than the CSE heuristic of [6] by extending the possible implementations of constants based on MSD representation.

On the other hand, graph-based algorithms are not restricted to a particular number representation and synthesize the constants iteratively by building a graph. Although the graph-based algorithms find better results than CSE algorithms as shown in [12], all the previously proposed graph-based algorithms are based on heuristics and provide no indication on how far from the minimum their solutions are.

A large amount of work that considers the MCM problem has also addressed efficient implementations of the MCM operation in hardware. These techniques include the use of different architectures, implementation styles, and optimization methods, e.g., [18,19].

In this paper, we introduce exact and approximate algorithms designed for the MCM problem. Initially, we present an exact graph-based algorithm [20] that searches the minimum number of operations solution of the MCM problem in a breadth-first manner. Then, we describe an approximate graph-based algorithm [21] that finds similar results with the minimum solutions and can be applied on more complex instances that the exact algorithm cannot handle. The main contribution of this paper is the introduction of an exact depth-first search algorithm that uses the solution of the approximate algorithm, in the determination of the lower and upper bounds of the search space, and finds the minimum solution using less computational effort than the exact breadth-first search algorithm. The proposed algorithms were applied on a comprehensive set of instances including randomly generated instances and FIR filters, and compared with the previously proposed exact CSE algorithm [14] and prominent graph-based heuristics [9,12]. The experimental results clearly indicate that the exact depth-first search algorithm that explores a highly constricted search space determined by the approximate algorithm obtains the minimum solutions of the MCM instances that the exact breadth-first search algorithm finds them difficult to handle and for which all the prominent graph-based heuristics obtain suboptimal solutions.

The rest of the paper is organized as follows: Section 2 gives the background concepts on the MCM problem and Section 3 describes the exact breadth-first search algorithm. The approximate graph-based algorithm is presented in Section 4 and the exact depth-first search algorithm is introduced in Section 5. Afterwards, experimental results are presented and finally, the conclusions are given in Section 7.

2. Background

In this section, initially, we give the main concepts and the problem definition and then, we present an overview of the graph-based algorithms.

Note that since the common input is multiplied by the multiple constants in MCM, the implementation of constant multiplications is equal to the implementation of constants. For example, the constant multiplication $3x = (x \ll 1) + x = ((1 \ll 1) + 1)x$ can be rewritten as $3 = (1 \ll 1) + 1$ by eliminating the variable x from both sides. Hereafter, this notation will be used for the sake of clarity and in this notation, we will refer '1' and the *intermediate constant* to the variable that the constants are multiplied with, i.e., x, and to the *partial product* used in the former notation respectively.

2.1. Definitions

In the MCM problem, the main operation, called *A-operation* in [12], is an operation with two integer inputs and one integer out-

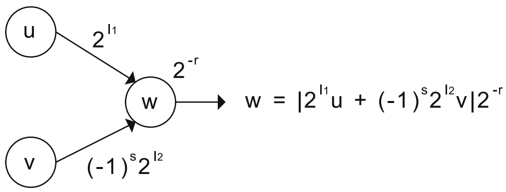


Fig. 3. The representation of the A-operation in a graph.

put that performs a single addition or a subtraction, and an arbitrary number of shifts. It is defined as follows:

$$\begin{aligned} w = A(u, v) &= |(u \ll l_1) + (-1)^s (v \ll l_2)| \gg r \\ &= |2^{l_1} u + (-1)^s 2^{l_2} v| 2^{-r} \end{aligned} \quad (1)$$

where $l_1, l_2 \geqslant 0$ are integers denoting left shifts, $r \geqslant 0$ is an integer indicating the right shift, and $s \in \{0, 1\}$ is the sign that denotes the addition/subtraction operation to be performed. The operation that implements a constant can be represented in a graph where the vertices are labeled with constants and the edges are labeled with the sign and shifts as illustrated in Fig. 3.

In the MCM problem, it is assumed that the shifting operation has no cost, since shifts can be implemented only with wires in hardware. Also, the sign of the constant is assumed to be adjusted at some part of the design and the complexity of an adder and a subtracter is equal in hardware, although the area of an addition/subtraction operation depends on the low-level implementation issues as described in [18]. Thus, in the MCM problem, only positive and odd constants are considered. Observe from Eq. (1) that in the implementation of an odd constant considering any two odd constants at the inputs of an *A-operation*, one of the left shifts, l_1 or l_2, is zero and r is zero, or both l_1 and l_2 are zero and r is greater than zero. When finding an operation to implement a constant, it is necessary to constrain the number of left shifts, otherwise a constant can be implemented in infinite ways. As shown in [9], it is sufficient to limit the shifts by the maximum bit-width of the constants to be implemented, i.e., *bw*, since allowing larger shifts does not improve the solutions obtained with the former limits. In the algorithms introduced in this paper and also, in the graph-based algorithm of [12], the number of shifts is allowed to be at most $bw + 1$. With these considerations, the MCM problem [12] can be also defined as follows:

Definition 1 (*The MCM problem*). Given the target set, $T = \{t_1, \ldots, t_n\} \subset \mathbb{N}$, including the positive and odd unrepeated target constants to be implemented, find the smallest ready set $R = \{r_0, r_1, \ldots, r_m\}$ with $T \subset R$ such that $r_0 = 1$ and for all r_k with $1 \leqslant k \leqslant m$, there exist r_i, r_j with $0 \leqslant i, j < k$ and an A-operation $r_k = A(r_i, r_j)$.

Hence, the number of operations required to be implemented for the MCM problem is $|R| - 1$ as given in [12]. Thus, to find the minimum number of operations solution of the MCM problem, one has to find the minimum number of intermediate constants such that all the constants, target and intermediate, are implemented using a single *A-operation* where its inputs are '1', intermediate, or target constants and the MCM implementation is represented in a directed acyclic graph.

2.2. Related work

For the single constant multiplication problem, an exact algorithm that finds the minimum number of required operations for a constant up to 12 bits was introduced in [22] and it was extended up to 19 bits in [23].

For the MCM problem, four algorithms, 'add-only', 'add/subtract', 'add/shift', and 'add/subtract/shift', were proposed in [8]. The 'add/subtract/shift' algorithm of [8] was modified in [9], called BHM, by extending the possible implementations of a constant, considering only odd numbers, and processing constants in order of increasing single coefficient cost that is evaluated by the algorithm of [22]. A graph-based algorithm, called RAG-n, was also introduced in [9]. RAG-n has two parts: optimal and heuristic. In the optimal part where the initial ready set includes only '1', each target constant that can be implemented using a single *A-operation* whose inputs are in the ready set are found and removed from the target set to the ready set. If there exist unimplemented element(s) left in the target set, the algorithm switches to the heuristic part. In this iterative part of the algorithm, intermediate constants are added to the ready set to implement the target constants. RAG-n initially chooses a single unimplemented target constant with the smallest single coefficient cost evaluated by the algorithm of [22] and then, synthesizes it with a single operation including one(two) intermediate constant(s) that has(have) the smallest value among the possible constants. However, observe that since the intermediate constants are selected for the implementation of a single target constant in each iteration, the intermediate constants chosen in previous iterations may not be shared for the implementation of not-yet synthesized target constants in later iterations, thus yielding a local minimum solution. To overcome this limitation, the graph-based heuristic of [12], called Hcub, includes the same optimal part of RAG-n, but uses a better heuristic that considers the impact of each possible intermediate constant on the not-yet synthesized target constants. In each iteration, for the implementation of a single target constant, Hcub chooses a single intermediate constant that yields the best cumulative benefit over all unimplemented target constants. It is shown in [12] that Hcub obtains significantly better results than BHM and RAG-n.

We make two general observations on a heuristic algorithm designed for the MCM problem. In these observations, $|T|$ denotes the number of elements of the target set to be implemented, i.e., the lowest bound on the number of required operations.

Lemma 1. *If a heuristic algorithm finds a solution with $|T|$ operations, then the found solution is minimum.*

In this case, no intermediate constant is required to implement the target constants. Since the elements of the target set cannot be synthesized using less than $|T|$ operations as shown in [9], then the found solution by the heuristic algorithm is the minimum solution.

Lemma 2. *If a heuristic algorithm that includes an optimal part as RAG-n and Hcub finds a solution with $|T| + 1$ operations, then the found solution is minimum.*

In this case, only one intermediate constant is required to implement the target constants. If the heuristic algorithm cannot find a solution in the optimal part, then it is obvious that at least one intermediate constant is required to find the minimum solution. So, if the found solution includes $|T| + 1$ operations, then it is the minimum solution.

Observe that while the case described in Lemma 1 is general for all algorithms designed for the MCM problem, the case described in Lemma 2 is valid for all algorithms that include an optimal part as RAG-n and Hcub. Note that the RAG-n and Hcub algorithms cannot determine their solutions as minimum if the obtained solutions include the number of operations more than the number of target constants to be implemented plus 1. Because, in this case, the target and intermediate constants are synthesized once at a time in the heuristic parts of the algorithms.

Furthermore, we note that the solution found by a heuristic algorithm can be also determined as minimum if the number of operations in its solution is equal to the lower bound of the MCM problem instance determined by the formula given in [24].

3. The exact breadth-first search algorithm

As described in Section 2.1, the MCM problem is to find the minimum number of intermediate constants such that each target and intermediate constant can be implemented with an operation as given in Eq. (1) where u and v are '1', target, or intermediate constants. This section presents an exact algorithm [20] that finds the minimum number of intermediate constants and therefore, the minimum number of operations solution, by exploring all possible intermediate constant combinations in a breadth-first manner.

3.1. The implementation

In the preprocessing phase of the algorithm, as described in Section 2.1, the target constants are made positive and odd, and added to the target set, T, without repetition. The maximum bit-width of the target constants, bw, is determined. In the main part of the exact algorithm given in Algorithm 1, the ready set that includes the minimum number of elements is computed.

Algorithm 1. The exact breadth-first search algorithm. The algorithm takes the target set, T, including target constants to be implemented and returns the ready set, R, with the minimum number of elements including '1', target, and intermediate constants.

BFSearch (T)
1: $R \leftarrow \{1\}$
2: (R, T) = Synthesize(R, T)
3: **if** $T = \emptyset$
4:    **return** R
5: **else**
6:    $n = 1,\ W_{R_1} \leftarrow R,\ W_{T_1} \leftarrow T$
7:    **while** 1 **do**
8:      $m = n,\ X_R = W_R,\ X_T = W_T$
9:      $n = 0,\ W_R = W_T = [\,]$
10:      **for** $i = 1$ to m **do**
11:        **for** $j = 1$ to $2^{bw+1} - 1$ step 2 **do**
12:          **if** $j \notin X_{R_i}$ and $j \notin X_{T_i}$ **then**
13:            (A, B) = Synthesize$(X_{R_i}, \{j\})$
14:            **if** $B = \emptyset$ **then**
15:              $n = n + 1$
16:              (W_{R_n}, W_{T_n}) = Synthesize(A, X_{T_i})
17:              **if** $W_{T_n} = \emptyset$ **then**
18:                **return** W_{R_n}

Synthesize(R, T)
1: **repeat**
2:    isadded = 0
3:    **for** $k = 1$ to $|T|$ **do**
4:      **if** t_k can be synthesized with the elements of R **then**
5:        isadded = 1
6:        $R \leftarrow R \cup \{t_k\}$
7:        $T \leftarrow T \setminus \{t_k\}$
8: **until** isadded = 0
9: **return** (R, T)

In the *BFSearch*, initially, the ready set including only '1' is formed. Then, the target constants that can be implemented using a single operation with the elements of the ready set are found iteratively and removed to the ready set using the *Synthesize* function. If there exist element(s) in the target set, the intermediate constants to be added to the ready set are considered exhaustively in the infinite loop, line 7 of the algorithm, until there is no element left in the target set. The infinite loop starts with the working array of ready and target sets W_{R_1} and W_{T_1}, i.e., the ready and target sets obtained on the line 2 of the algorithm. Note that the size of the array W that includes the ready and target sets as a pair of elements is denoted by n. In the infinite loop, another working array X is assigned to the array W and its size is represented by m. Then, for each ready set of the array X, all possible intermediate constants are found. Each intermediate constant is added to the associated ready set, and a new ready set is formed. The possible intermediate constants are determined as the positive and odd constants that are not included in the current ready and target sets, X_{R_i} and X_{T_i}, and can be implemented with the elements of the current ready set, as given on the lines 11–14. Note that there is no need to consider the intermediate constants that cannot be implemented with the elements of the current ready set, since all these constants are considered in other ready sets due to the exhaustiveness of the algorithm. When a possible intermediate constant is found, the implications of the possible intermediate constant with the elements of the ready set X_{R_i} on the target set X_{T_i} are determined by the *Synthesize* function and the modified ready and target sets are stored to the array W as a new pair, line 16 of the algorithm. Observe from lines 17–18 of the algorithm that when there is no element left in a target set, the minimum number of operations solution is obtained with the associated ready set.

We note that although it is not stated in Algorithm 1 for the sake of clarity, we avoid from the duplicate intermediate constant combinations by considering the intermediate constants in a sequence.

As a small example, suppose the target set $T = \{307, 439\}$. The iterations in the infinite loop of the *BFSearch* algorithm are sketched in Fig. 4 indicating the working array W with ready and target sets. In this figure, the edges labeled with the intermediate constants represent the inclusions of constants to the ready set. In the first iteration, the intermediate constants that can be implemented using a single operation with the ready set $R = \{1\}$, i.e., 3, 5, 7, 9, 15, 17, 31, 33, 63, 65, 127, 129, 511, 513, 1023, are computed. However, all the possible one intermediate constant combinations, i.e., {1,3}, {1,5},…,{1,1023}, cannot synthesize all the target constants. Then, in the second iteration, the two intermediate constant combinations are considered. Observe that all the target constants are synthesized when the intermediate constant 55 is added to the ready set $R = \{1, 63\}$.

After the ready set with the minimum number of intermediate constants is computed, the final implementation is obtained by synthesizing each target and intermediate constant using a single

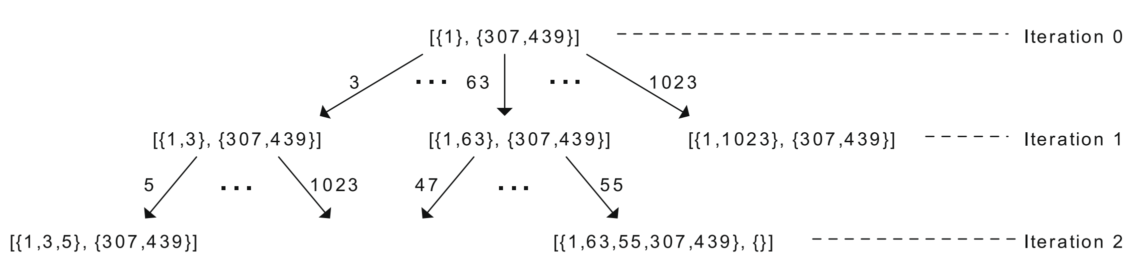
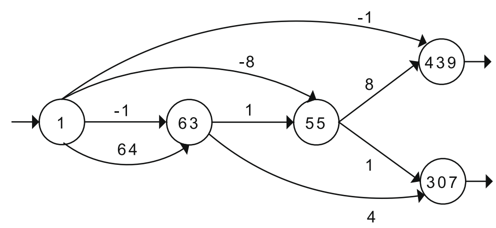
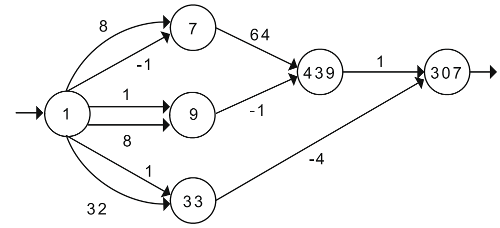


Fig. 4. The flow of the *BFSearch* algorithm for the target constants 307 and 439.



(a)



(b)

Fig. 5. The results of algorithms for the target constants 307 and 439: (a) four operations with the *BFSearch* algorithm; (b) five operations with Hcub.

operation. Fig. 5a presents the solution of the *BFSearch* algorithm. Also, the solution of Hcub is given in Fig. 5b. Observe that since Hcub synthesizes the target constants iteratively by including intermediate constants, the intermediate constants chosen for the implementation of target constants in previous iterations may not be shared in the implementation of target constants in later iterations.

Hence, we can make the following observation on the *BFSearch* algorithm.

Lemma 3. *The solution obtained by the BFSearch algorithm yields the minimum number of operations solution.*

If the BFSearch algorithm returns a solution on the line 4 of the algorithm, then no intermediate constant is required to implement the target constants. Similar to the conclusion drawn in Lemma 1, *each target constant can be implemented using a single operation whose inputs are '1' or target constants as ensured by the Synthesize function and the number of required operations to implement the target constants is* $|T|$.

If the BFSearch algorithm returns a solution on the line 18 of the algorithm, then intermediate constants are required to implement the target constants. In this case, the number of required operations to implement the target constants is $|T|$ *plus the number of intermediate constants. Because each element of the ready set, except '1', is guaranteed to be implemented using a single operation by the Synthesize function and all possible intermediate constant combinations are explored exhaustively in a breadth-first manner, the obtained ready set yields the minimum number of operations solution.*

3.2. Complexity analysis

The complexity of the *BFSearch* algorithm depends on both the number of considered ready sets in each iteration, i.e., n in the Algorithm 1, and the maximum bit-width of the target constants, i.e., bw, since the number of considered intermediate constant combinations increases as bw increases. Table 1 presents the maximum number of ready sets exploited by the *BFSearch* algorithm including up to four intermediate constants for a single target constant in between 10 and 14 bit-width. The worst case values given in Table 1 were observed from the *BFSearch* algorithm during the search for the minimum number of operations solutions of the single constants. The exponential growth of the search space can be clearly observed when the number of iterations increases. This is simply because, the inclusion of an intermediate constant to a ready set in the current iteration increases the number of possible intermediate constants to be considered in the next iteration.

Table 1
Upper bounds on the number of ready sets exploited by the exact algorithm for the implementation of a single constant under different bit-widths.

bw	#Ready sets considered in iterations				
	1	2	3	4	Total
10	19	648	30,428	19,000,657	19,031,752
11	21	810	43,761	57,559,925	57,604,517
12	23	990	60,435	165,546,959	165,608,407
13	25	1188	80,907	458,873,308	458,955,428
14	27	1404	105,462	1,230,677,125	1,230,784,018

Note that the maximum number of considered intermediate constant combinations in finding the minimum solutions of the single constants under the same bit-width may be different. For example, the constant 833 in 10 bit-width can be implemented using three operations, $3=(1\ll2)-1, 13=(3\ll2)+1$, and $833=(13\ll6)+1$ with two intermediate constants, i.e., 3 and 13. However, the constant 981 defined under the same bit-width requires four operations, $3=(1\ll2)-1,\ 5=(1\ll2)+1,\ 43=(5\ll3)+3$, and $981=(1\ll10)-43$, thus with three intermediate constants, 3, 5, and 43. In the former case, the maximum number of considered intermediate constant combinations is 19 + 648 = 667 and in the latter case, this value is 19 + 648 + 30428 = 31095. Also, we note that the minimum number of operations solution for a single constant is generally obtained using much less considerations than the worst case.

Observe that there are cases where the existence of multiple constants may reduce the complexity of the search space. For example, consider the target constants 43 and 981. In this case, there is no need to try all the three intermediate constant combinations, since the two intermediate constant combination {3,5} yields the minimum solution. Therefore, the values of the target constants in an MCM instance determine the complexity of the search space to be explored, that directly effects the performance of the algorithm. However, we observe that the *BFSearch* algorithm can obtain the minimum solutions for some MCM instances in a reasonable time. These instances require, in general, less than four intermediate constants to generate all the target constants.

4. The approximate graph-based algorithm

As can be observed from Section 3.2, there are still some complex MCM problem instances for which the *BFSearch* algorithm cannot find the minimum solution in a reasonable time. Hence, heuristic algorithms that find solutions close to the minimum and better solutions than those of the previously proposed prominent heuristics are always indispensable.

In this section, we present an approximate graph-based algorithm [21] that is based on the *BFSearch* algorithm. Similarly to the exact algorithm, we compute all possible intermediate constants that can be synthesized with the current set of implemented constants in each iteration. But, to cope with more complex instances, we select a single intermediate constant that synthesizes the largest number of target constants, and continue the search with the chosen constant. By doing so, we reduce the search space by selecting the "best" intermediate constant at each search level, as opposed to keeping all the valid possibilities until the minimum solution is found. Observe that the approach of the approximate algorithm to the MCM problem is different from that of RAG-n and Hcub, where in each iteration, they select a target constant and synthesize it by finding the "best" intermediate constant. We also note that the implementation of the approximate algorithm in this scheme enables itself to guarantee the minimum solution on more instances than RAG-n and Hcub as will be shown in Sections 4.2 and 6.

4.1. The implementation

The main part of the approximate algorithm is given in Algorithm 2. We note that the preprocessing phase and the *Synthesize* function used in the approximate algorithm are the same as those described in the exact breadth-first search algorithm.

Algorithm 2. The approximate graph-based algorithm. The algorithm takes the target set, *T*, including target constants to be implemented and returns the ready set, *R*, that includes '1', target, and intermediate constants.

```
ApproximateSearch(T)
1:  R ← {1}
2:  (R, T) = Synthesize(R, T)
3:  if T = ∅ then
4:     return R
5:  else
6:       while 1 do
7:          for j = 1 to 2^(bw+1) − 1 step 2 do
8:             if j ∉ R and j ∉ T
9:                (A, B) = Synthesize(R, {j})
10:               if B = ∅ then
11:                  (A, B) = Synthesize(A, T)
12:                  if B = ∅
13:                     A = RemoveRedundant(A)
14:                     return A
15:                  else
16:                     cost_j = EvaluateCost(B)
17:            Find the intermediate constant, ic, with the minimum
               cost among all possible constants j
18:            R ← R ∪ {ic}
19:            (R, T) = Synthesize(R, T)
EvaluateCost(B)
1:  cost = 0
2:  for k = 1 to |B| then
3:     cost = cost + SingleCoefficientCost(b_k)
4:  return cost
RemoveRedundant(A)
1:  for k = 1 to |A| do
2:     if a_k is an intermediate constant then
3:        A ← A \ {a_k}
4:        (A, B) = Synthesize({1}, A)
5:         if B ≠ ∅ then
6:           A ← A ∪ {a_k}
7:  return A
```

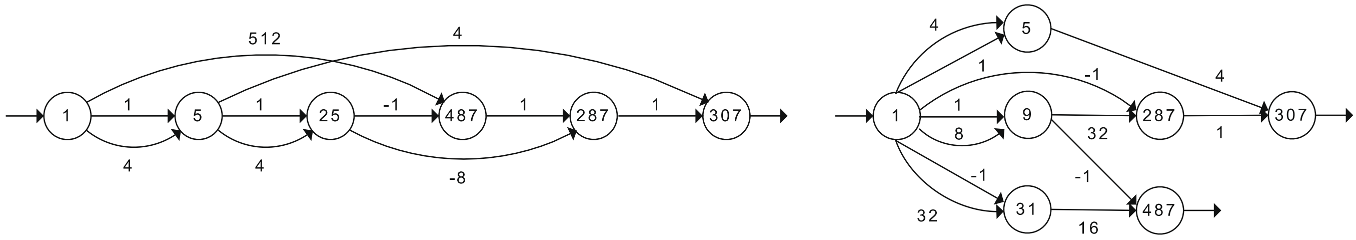
As done in the optimal part of RAG-n and Hcub, the *ApproximateSearch* initially forms the ready set including only '1' and then, the target constants that can be implemented with the elements of the ready set using a single operation are found and removed to the ready set iteratively using the *Synthesize* function. If there exist unimplemented constant(s) in the target set, then in each iteration of the infinite loop, line 6 of the algorithm, an intermediate constant is added to the ready set until there is no element left in the target set. As done in the *BFSearch* algorithm, the *ApproximateSearch* algorithm considers the positive and odd constants that are not included in the current ready and target sets and can be implemented with the elements of the current ready set using a single operation as possible intermediate constants. Note that the working ready and target sets in each iteration are denoted by *A* and *B* respectively. After the possible intermediate constant is included into the working ready set, its implications on the current target set are found by the *Synthesize* function. If there exist unimplemented target constants in the working target set, by using the *EvaluateCost* function, the implementation cost of the not-yet synthesized target constants is determined in terms of the single coefficient cost computed as in [23] and is assigned to the cost value of the intermediate constant. After the cost value of each possible intermediate constant is found, the one with the minimum cost value is chosen and added to the current ready set, and the target constants that can be implemented with the elements of the updated ready set are found. The infinite loop is interrupted whenever there is no element left in the working target set, thus the solution is obtained with the working ready set. However, note that by adding an intermediate constant to the ready set in each iteration, the previously added intermediate constants can be redundant due to the recently added constant. Hence, the *RemoveRedundant* function is applied on the final ready set to remove redundant intermediate constants. After the ready set that consists of the fewest number of constants is obtained, each element in the ready set, except 1, are synthesized with a single operation whose inputs are the elements of the ready set.

As an example, suppose the target set $T = \{287, 307, 487\}$. Fig. 6 presents the solutions obtained by the *ApproximateSearch* and Hcub algorithms. In this example, the *ApproximateSearch* algorithm chooses the intermediate constant 5 that can be implemented as $5 = (1 \ll 2) + 1$ with the current ready set $R = \{1\}$ and adds it to the ready set in the first iteration. Then, the intermediate constant 25 that can be implemented as $25 = (5 \ll 2) + 5$ with the current ready set $R = \{1, 5\}$ is chosen to be included into the ready set in the second iteration. As can be observed from Fig. 6a, all the target constants are synthesized with the current ready set $R = \{1, 5, 25\}$. As can be observed from Fig. 6a and b, the intermediate constant selection heuristic used in the *ApproximateSearch* algorithm (i.e., selecting the "best" intermediate constant for the implementation of the most of the target constants) may yield better solutions than the intermediate constant selection heuristic used in Hcub (i.e., selecting the "best" intermediate constant for the implementation of a single target constant taking into account the not-yet synthesized constants). We note that the solution found by the *ApproximateSearch* algorithm is the minimum number of operations solution, as determined by Lemma 4 given in Section 4.2 where main characteristics of the *ApproximateSearch* algorithm are introduced.

We also note that the removal of redundant intermediate constants is never considered in the previously proposed graph-based heuristics. Hence, their solutions may include redundant constants. For instance, consider the target constants 287 and 411 to be implemented. The solution of Hcub presented in Fig. 7a includes four operations with the intermediate constants 9 and 31. However, the intermediate constant 9 is redundant, as determined by the *RemoveRedundant* function, since only the intermediate constant 31 can be used to synthesize the target constants 287 and 411 as shown in Fig. 7b. We note that this is also the minimum number of operations solution for the implementation of the target constants 287 and 411 guaranteed by the exact algorithm.

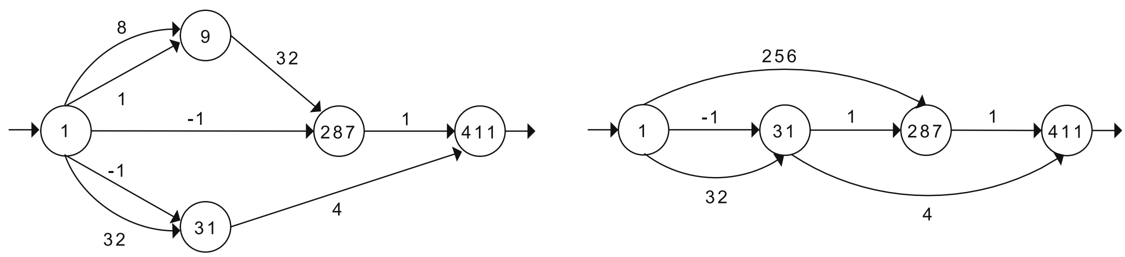
4.2. The characteristics of the approximate algorithm

The observations given in Lemmas 1 and 2 are also valid for the *ApproximateSearch* algorithm given in Algorithm 2, since it includes the same optimal part as Hcub and RAG-n. Furthermore, the following conclusions can be drawn for the *ApproximateSearch* algorithm.



(a) (b)

Fig. 6. The results of algorithms for the target constants 287, 307, and 487: (a) five operations with the approximate algorithm; (b) six operations with Hcub.



(a) (b)

Fig. 7. The implementations of the target constants 287 and 411: (a) four operations with Hcub; (b) three operations after using the *RemoveRedundant* function.

Lemma 4. *If the ApproximateSearch algorithm finds a solution with $|T| + 2$ operations, then the found solution is minimum.*

In this case, two intermediate constants are required to implement the target constants. Note that the case described in Lemma 1 is checked on the lines 2–3 of the algorithm and the case described in Lemma 2 is explored exhaustively on the lines 7–16 of the algorithm in the first iteration. Hence, if there exist unimplemented target constants at the end of the first iteration, then the minimum solution requires at least one more intermediate constant, thus totally two intermediate constants. So, if ApproximateSearch algorithm finds a solution including $|T| + 2$ operations, then it is the minimum solution.

It is obvious that if the *ApproximateSearch* algorithm finds a solution including more than $|T| + 2$ operations, then it cannot guarantee the found solution is minimum, since all possible intermediate constant combinations including more than one constant are not explored exhaustively.

Lemma 5. *If the minimum solution of an MCM instance includes up to $|T| + 1$ operations, then the ApproximateSearch algorithm always finds the minimum solution.*

For the MCM instances including $|T|$ operations in their minimum solutions, the ApproximateSearch algorithm and also, RAG-n and Hcub, always find the minimum solution due to their optimal part.

For the MCM instances including $|T| + 1$ operations in their minimum solutions, the ApproximateSearch algorithm always finds the minimum solution, since it considers all possible one intermediate combinations exhaustively on the lines 7–16 of the algorithm in the first iteration.

Note that for the MCM instances including $|T| + 1$ operations in their minimum solutions, RAG-n and Hcub do not always find the minimum solution (for an example, see Fig. 7), since in this case, the solutions are obtained in their heuristic parts.

Hence, the following conclusion can be drawn from Lemma 5.

Lemma 6. *If the ApproximateSearch algorithm cannot guarantee its solution as the minimum solution, then the lower bound on the minimum number of operations solution is $|T| + 2$.*

In this case, the ApproximateSearch algorithm finds a solution including more than $|T| + 2$ operations. Since the minimum solution of the MCM instance includes greater than or equal to $|T| + 2$ operations due to Lemma 5, the lower bound on the minimum number of operations solution is $|T| + 2$.

On the other hand, if RAG-n and Hcub algorithms cannot guarantee the minimum solution on an MCM instance, then they implicitly state that the lower bound on the minimum number of operations solution is $|T| + 1$.

Also, it is obvious that if the *ApproximateSearch* algorithm cannot guarantee the minimum solution, then the upper bound on the minimum number of operations solution is determined as the number of operations in its solution. Thus, the lower and upper bounds of the search space obtained by the *ApproximateSearch* algorithm can be used to direct the search in finding the minimum solution as described in the following section.

5. The exact depth-first search algorithm

The exact breadth-first search algorithm introduced in Section 3 has a main drawback: it lacks pruning techniques that reduce the search space and consequently, the required time to find the minimum solution. On the other hand, the approximate algorithm presented in Section 4 gives promising lower and upper bounds on the search space to be explored by an exact algorithm.

In this section, we introduce an exact algorithm that searches the minimum solution in a depth-first manner with the use of lower and upper bounds obtained by the approximate algorithm. In the proposed exact depth-first search algorithm, initially, a solution is found using the approximate algorithm. If the approximate algorithm can guarantee its solution as the minimum number of operations solution, then the minimum solution is returned, otherwise the depth-first search is applied. At the end of the depth-first search, the minimum solution that is better than the solution obtained by the approximate algorithm is found or it is proved by exploring all possible intermediate constant combinations that the solution found by the approximate algorithm is, in fact, the minimum solution.

The main part of the exact depth-first search algorithm is given in Algorithm 3. Again, the preprocessing phase and the *Synthesize* function used in the exact depth-first search algorithm are the same as those described in the exact breadth-first search algorithm.

Algorithm 3. The exact depth-first search algorithm. The algorithm takes the target set, *T*, including target constants to be implemented and returns the ready set, *R*, with the minimum number of elements including '1', target, and intermediate constants.

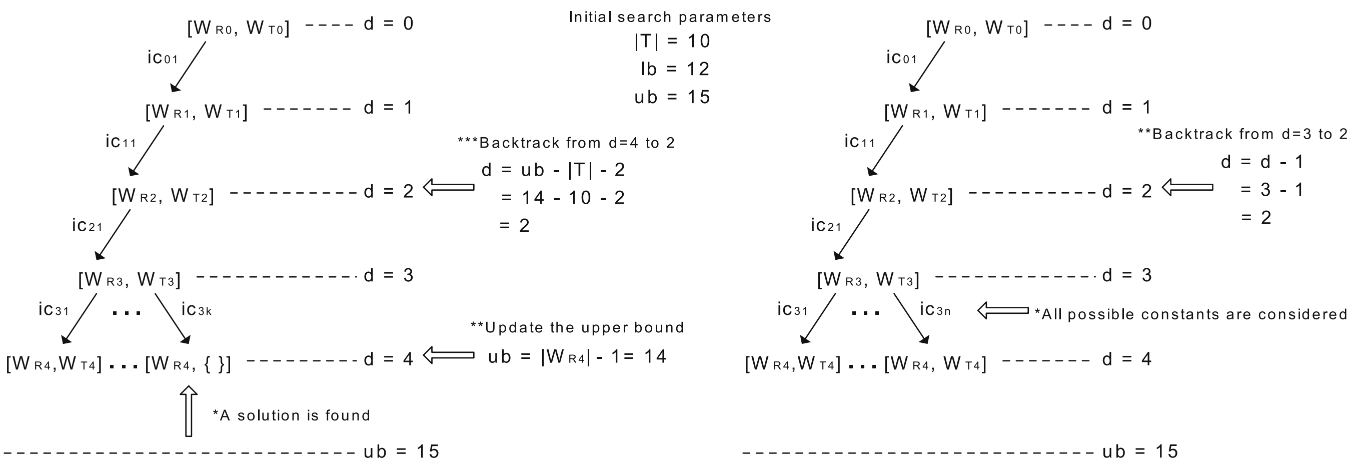
DFSearch(T)
1: R = ApproximateSearch(T)
2: **if** $|R| - 1 \leqslant |T| + 2$
3:   **return** R
4: **else**
5:     $lb = |T| + 2,\ ub = |R| - 1,\ d = 0,\ ic\_list_0 = 1$
6:     (W_{R_0}, W_{T_0}) = Synthesize({1}, T)
7:     **while** 1 **do**
8:       **if** $|T| + d + 1 \leqslant ub - 1$
9:         ic = FindIntermediateConstant$(ic_list_d, W_{R_d}, W_{T_d})$
10:          **if** ic **then**
11:            $ic\_list_d = ic,\ d = d + 1$
12:            (W_{R_d}, W_{T_d}) = Synthesize $(W_{R_{d-1}} \cup \{ic\}, W_{T_{d-1}})$
13:            **if** $W_{T_d} = \emptyset$ **then**
14:              $R = W_{R_d},\ ub = |R| - 1$
15:              **if** lb = ub **then**
16:                **return** R
17:              **else**
18:                $d = ub - |T| - 2$
19:            **else**
20:            **if** CLB $(W_{R_d} \setminus \{1\} \cup W_{T_d}) \geqslant ub$ **then**
21:                $d = d - 1$
22:              **else**
23:                $ic\_list_d = 1$
24:          **else**
25:            $d = d - 1$
26:        **else**
27:          $d = d - 1$
28:        **if** $d = -1$ **then**
29:          **return** R

FindIntermediateConstant(ic, R, T)
1: **while** 1 **do**
2:   $ic = ic + 2$
3:   **if** $ic > 2^{bw+1} - 1$ **then**
4:     **return** 0
5:   **else**
6:     **if** $ic \notin R$ and $ic \notin T$ **then**
7:       (A, B) = Synthesize(R, {ic})
8:       **if** $B = \emptyset$ **then**
9:         **return** ic

In the *DFSearch*, initially, the approximate algorithm given in Algorithm 2 is applied. If the approximate algorithm returns the minimum number of operations solution that is guaranteed by Lemmas 1, 2 and 4, then the *DFSearch* algorithm is returned with the solution of the approximate algorithm. Otherwise, the initial lower and upper bound values of the search space, denoted by *lb* and *ub* respectively, are determined. The decision level or the depth of search space, i.e., *d*, is set to 0. An array denoted by *ic_list* that includes the intermediate constants considered at each decision level is formed and its value at the decision level 0 is assigned to 1. The working ready and target sets at decision level 0, i.e., W_{R_0} and W_{T_0}, are obtained using the *Synthesize* function. In the infinite loop, the search space is explored in a depth-first manner up to *ub*-1 decision level, since we have a solution including *ub* number of operations. Hence, the condition given on the line 8 of the algorithm avoids to make unnecessary moves beyond *ub*-1 during the search. At each decision level, the *FindIntermediateConstant* function is applied to find the positive and odd intermediate constant that is not included in the current working ready and target sets and can be implemented using a single operation with the elements of the current working ready set. If an intermediate constant is found, it is stored to the intermediate constant list at that decision level, ic_list_d. Then, the next decision level working ready and target sets are simply obtained when the intermediate constant is included in the current ready set and its implications are found on the current working target set by the *Synthesize* function. Whenever all the elements of the target set are synthesized, a better solution than the one found so far is obtained and *ub* is updated. In this case, if the number of operations in the found solution is equal to *lb*, the infinite loop is interrupted indicating that the obtained solution is the minimum solution. Otherwise, the search is backtracked to the $ub - |T| - 2$ decision level. Fig. 8a illustrates this backtrack when a better solution is found during the depth-first search. For this example, suppose that the number of target constants to be implemented, i.e., $|T|$, is 10, and *lb* and *ub* values obtained by the approximate algorithm are determined as 12 and 15 respectively. In this figure, ic_{di} denotes the intermediate constant considered at the decision level *d* with its index *i*.

In the *DFSearch*, if the CLB (*ComputeLowerBound*) function that determines the lower bound on a set of constants using the formula given in [24] returns a value that is equal to or greater than the current upper bound, i.e., when the condition given on the line 20 of the algorithm is met, then a backtrack to the previous decision level occurs. This simply states that with the elements in the current ready and target sets, to be found solution will have for sure a number of operations equal to or greater than the one that has been found so far. Also, if all possible intermediate constants are considered at a decision level, i.e., when the condition given on the line 24 of the algorithm is met, the search backtracks to the previous decision level as illustrated in Fig. 8b. Note that whenever the decision level is -1, the infinite loop is interrupted indicating that the depth-first search is completed.

In the *DFSearch* algorithm, we avoid from the duplicate intermediate constant combinations by paying attention to the intermediate constants considered at each decision level. We note that in the worst case, the complexity of the *DFSearch* algorithm, i.e., the number of considered intermediate constant combinations, is the same as that of the *BFSearch* algorithm. However, observe that a better initial lower bound enables the *DFSearch* algorithm to conclude the search in an earlier decision level, when a solution is found. A better initial upper bound helps the algorithm to search up to a smaller decision level where a better solution can be found. Also, observe from the *FindIntermediateConstant* function that the *DFSearch* algorithm, at each decision level, branches with an intermediate constant that has a smaller value, since it is more possible to implement the target constants with smaller intermediate constants. This branching method enables the algorithm to explore less number of intermediate constant combinations in finding a better solution. Hence, the *DFSearch* algorithm, in general, finds the minimum solution of an MCM problem instance by exploring significantly less search space and requiring much less memory than the *BFSearch* algorithm.



(a) (b)

Fig. 8. Backtracks in the *DFSearch* algorithm: (a) when a better solution is found; (b) when all possible constants are considered.

6. Experimental results

In this section, we present the results of the exact and approximate algorithms on randomly generated and FIR filter instances, and compare with those of the previously proposed exact CSE algorithm [14] and the graph-based heuristics [9,12]. The graph-based heuristics were obtained from [25].

As the first experiment set, we used uniformly distributed randomly generated instances where constants were defined under 14 bit-width. The number of constants ranges between 10 and 100, and we generated 30 instances for each of them. Thus, the experiment set includes 300 instances. Fig. 9 presents the solutions obtained by the exact CSE algorithm of [14] under binary, CSD, and MSD representations and the exact graph-based algorithm.

As can be observed from Fig. 9, the solutions obtained by the exact CSE algorithm [14] are far from the minimum number of operations solutions, since the possible implementations of constants are limited with the number representation in the exact CSE algorithm. We note that on these instances, the average difference of the number of operations solutions between the exact CSE algorithm under binary, CSD, and MSD representations, and the exact graph-based algorithm is 8.5, 10.8, and 8.6 respectively. Since both CSE and graph-based algorithms are exact, we can clearly state that the graph-based algorithms obtain much better solutions than the CSE algorithms.

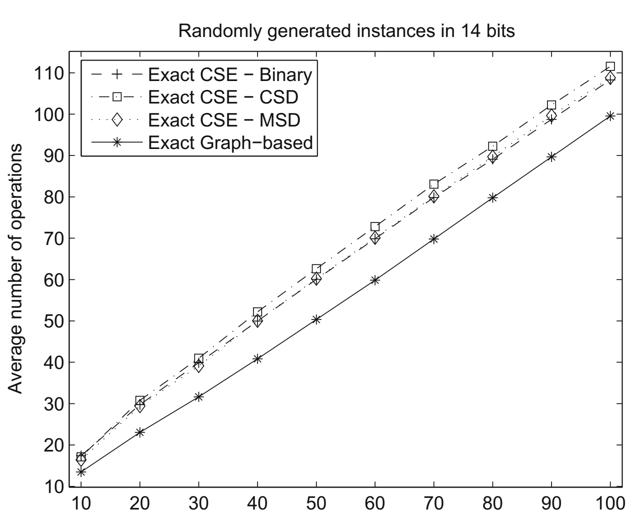


Fig. 9. Results of exact CSE and graph-based algorithms on randomly generated instances defined under 14 bit-width.

As the second experiment set, we used FIR filter instances where filter coefficients were computed with the *remez* algorithm in MATLAB. The specifications of filters are presented in Table 2 where: *Pass* and *Stop* are normalized frequencies that define the passband and stopband respectively; *#Tap* is the number of coefficients; and *width* is the bit-width of the coefficients.

The results of the algorithms are presented in Table 3 where *Adder* denotes the number of operations and *Step* indicates the maximum number of operations in series, i.e., generally known as adder-steps. In this table, $|T|$ denotes the number of unrepeated positive and odd filter coefficients, i.e., the lowest bound on the number of operations, and the *LBs* indicates the lower bounds on the number of operations and the number of adder-steps obtained by the formulas given in [24].

As can be easily observed from Table 3, while the graph-based heuristic RAG-n obtains suboptimal solutions that are far away from the minimum, Hcub and approximate algorithms find solutions close to the minimum. We note that the approximate algorithm finds similar or better solutions than RAG-n and Hcub, and according to Lemmas 1, 2 and 4, it guarantees the minimum solution on two filter instances, i.e., Filter 5 and 8. Also, according to Lemmas 1 and 2, the graph-based heuristics RAG-n and Hcub cannot guarantee any of their solutions as the minimum solution. Note that the lower bound [24] on the minimum number of required operations can only be used to determine the solution of the approximate algorithm on Filter 8 as the minimum solution, although it is also proven to be minimum by Lemma 2. Also, observe from Lemma 6 that the lower bounds obtained by the

Table 2
Filter specifications.

Filter	Pass	Stop	#Tap	Width
1	0.10	0.15	40	14
2	0.10	0.12	40	14
3	0.15	0.20	30	14
4	0.20	0.25	30	14
5	0.10	0.15	80	16
6	0.15	0.20	60	16
7	0.20	0.25	40	16
8	0.10	0.20	80	16

Table 3
Summary of results of graph-based algorithms on FIR filter instances.

Filter	$\|T\|$	LBs [24]		RAG-n [9]		Hcub [12]		Approximate		Exact	
		Adder	Step	Adder	Step	Adder	Step	Adder	Step	Adder	Step
1	19	20	3	24	10	23	7	22	9	22	8
2	19	20	3	27	6	24	6	23	7	23	7
3	14	14	3	24	9	19	7	18	6	17	11
4	14	14	3	23	5	18	7	18	7	17	8
5	39	40	3	44	9	42	8	41	10	41	10
6	29	29	3	36	10	32	10	32	5	31	11
7	19	20	3	28	7	24	7	23	12	23	10
8	36	37	3	40	5	38	5	37	6	37	6
Total	189	194	24	246	61	220	57	214	62	211	71

approximate algorithm, i.e., $|T| + 2$, on all filters except Filter 5 and 8, are better than those of [24]. On the other hand, the exact algorithm finds better solutions than all the graph-based heuristics on Filter 3, 4, and 6. Hence, this experiment indicates that an exact algorithm is indispensable to ensure the minimum solution, since there are real-size instances that all the prominent graph-based heuristics cannot conclude with the minimum solution.

As can be observed from Table 3, the graph-based algorithms designed for the MCM problem find the fewest number of operations solution with a greater number of adder-steps according to its lower bound, indicating the traditional tradeoff between area and delay. This occurs because, the sharing of intermediate constants in MCM generally increases the logic depth of the MCM design as shown in [26]. However, we note that the exact algorithms can be easily modified to find the fewest number of operations solution under a delay constraint as done in [14,26,27]. In this case, only the intermediate constants that do not violate the delay constraint must be considered.

On this experiment set, we also compare the runtime performance of the *BFSearch* and *DFSearch* algorithms in Tables 4 and 5 respectively. In these tables, *Crs* indicates the number of considered ready sets in the breadth-first and depth-first search parts of the exact algorithms, and *CPU* denotes the CPU time in seconds of the exact algorithms implemented in MATLAB to obtain a solution on a PC with 2.4 GHz Intel Core 2 Quad CPU and 3.5GB memory. We note that the CPU time limit was determined as 1 h. For the *DFSearch* algorithm, the initial lower and upper bounds of the search space were determined by RAG-n, Hcub, and approximate algorithms. Note that the graph-based heuristics obtained their solutions using a little computational effort, thus their execution time were not taken into account in the *DFSearch* algorithm. In these tables, an italic result indicates that the exact algorithm is ended due to the CPU time limit returning with the best solution found so far.

As can be easily observed from Tables 4 and 5, although the *BFSearch* algorithm is proven to be a complete algorithm by Lemma 3, i.e., given all the required computational resources, the *BFSearch* algorithm guarantees to find the minimum solution, it cannot conclude with a solution on Filter 2, 3, 4, 6, and 7 due to the CPU time limit. On the other hand, the *DFSearch* algorithm initially obtains a solution using a graph-based heuristic, i.e., it never returns without a solution. Then, with the lower and upper bounds of the search space determined by the solution of the graph-based heuristic, it finds the minimum solution that is better than that of the graph-based heuristic or proves that the solution of the graph-based heuristic is the minimum solution by exploring all the search space. For example, the solutions of RAG-n on all filter instances include totally 246 operations as given in Table 3 and RAG-n cannot guarantee any of its solution as minimum. However, the *DFSearch* algorithm using the solutions of RAG-n to compute the bounds of search space finds the minimum solution on all filter instances except Filter 2 with totally 212 operations and in almost 2 h. As can be easily observed from Table 5, the quality of the graph-based heuristic solution, i.e., the quality of lower and upper bounds of search space, effects the performance of the *DFSearch* algorithm significantly. For example, with the solutions of the approximate algorithm that are close to the minimum, the *DFSearch* algorithm concludes with ensuring the minimum solution on all instances

Table 4
Runtime performance of the *BFSearch* algorithm on FIR filter instances.

Filter	Adder	#Crs	CPU
1	22	14,098	136.28
2	–	*236,289*	*3600.01*
3	–	*145,418*	*3600.01*
4	–	*142,308*	*3600.01*
5	41	2986	26.44
6	–	*253,029*	*3600.01*
7	–	*70,003*	*3600.01*
8	37	8	0.53
Total	100	864,139	18,163.30

Table 5
Runtime performance of the *DFSearch* algorithm on FIR filter instances.

Filter	With RAG-n			With Hcub			With approximate		
	Adder	#Crs	CPU	Adder	#Crs	CPU	Adder	#Crs	CPU
1	22	14,281	10.52	22	14,138	10.34	22	3834	2.38
2	*24*	*6,814,991*	*3600.01*	*24*	*6,832,247*	*3600.01*	23	2,157,429	1042.61
3	17	4,321,079	1586.08	17	4,309,727	1595.97	17	3,261,389	1162.19
4	17	63,870	30.7	17	23,530	7.86	17	23,530	7.91
5	41	9079	9.27	41	2993	3.05	41	0	0
6	31	1,871,348	1460.39	31	1,706,210	1328.80	31	1,702,822	1326.48
7	23	1,375,750	756.75	23	1,300,795	617.73	23	695,115	250.61
8	37	38	0.16	37	9	0.14	37	0	0
Total	212	14,470,436	7453.88	212	14,189,649	7163.90	211	7,844,119	3792.18

Table 6
Summary of results of the graph-based heuristics on hard instances.

>	BHM [9]	RAG-n [9]	Hcub [12]	Approximate
BHM [9]	0	173	3	9
RAG-n [9]	964	0	96	6
Hcub [12]	1231	700	0	62
Approximate	1235	802	442	0

and using less computational effort (the number of considered ready sets and CPU time) than those obtained with the solutions of RAG-n and Hcub.

As the third experiment set, we used randomly generated instances where constants were defined in between 10 and 14 bit-width. We tried to generate hard instances to distinguish the algorithms clearly. Hence, under each bit-width (*bw*), the constants were generated randomly in $[2^{bw-2}+1, 2^{bw-1}-1]$. Also, the number of constants were determined as 2, 5, 10, 15, 20, 30, 50, 75, and 100 and we generated 30 instances for each of them. Thus, the experiment set includes 1350 instances.

The results of graph-based heuristic algorithms on overall 1350 instances are summarized in Table 6 where $X > \mathbf{Y}$ denotes the number of instances that the algorithm X finds better solutions than the algorithm **Y**. As can be easily observed from Table 6, the number of instances that the approximate algorithm finds better solutions than Hcub is 442, while the number of instances that Hcub obtains better solutions than the approximate algorithm is 62 on overall 1350 instances. When the approximate algorithm is compared with RAG-n and BHM, it finds better solutions than these heuristics on 802 and 1235 instances respectively. It is observed that the instances that the approximate algorithm finds worse solutions than the prominent graph-based heuristics generally include small number of constants, e.g., 2, under large bit-widths, e.g., 14. We note that this is simply because of the greedy heuristic used in the approximate algorithm where, in each iteration, an intermediate constant that synthesizes the largest number of target constant is chosen. We also note that the number of instances that the approximate algorithm guarantees the minimum solution is 699 (51.8% of the experiment set), and the number of instances that RAG-n and Hcub ensure the minimum solution is 394 (29.2% of the experiment set) and 386 (28.6% of the experiment set) respectively on overall 1350 instances. Hence, this experiment indicates that the approximate algorithm guarantees the minimum solution on more instances than the prominent graph-based heuristics.

We also applied the *DFSearch* algorithm with the bounds resulting from both Hcub and the approximate algorithm on 651 instances whose solutions were not guaranteed to be the minimum by these algorithms. Again, the CPU time limit was set to 1 h. We note that the *DFSearch* algorithm ensured the minimum solution on 473 instances obtaining better solutions than both Hcub and the approximate algorithm on 195 instances. However, the *DFSearch* algorithm could not conclude with the minimum solution on 178 instances due to the CPU time limit. On these 178 instances, we observed that the difference between the best upper bound obtained by Hcub and the approximate algorithm and the lowest bound is equal to or greater than 6, that means, in the worst case, all intermediate constant combinations including 5 or more than 5 constants must be considered to complete the search. This experiment clearly indicates that although the *DFSearch* algorithm can be applied on real-size instances and finds better solutions than the graph-based heuristics, there are still instances that the *DFSearch* algorithm cannot return the minimum solution in a reasonable time.

7. Conclusions

In this article, we introduced exact and approximate graph-based algorithms designed for the MCM problem for which only heuristic algorithms have been proposed due to the NP-hardness of the problem. In this content, we presented an exact breadth-first search algorithm that is capable of finding the minimum number of operations solution of real mid-size instances. To cope with more complex instances that the exact algorithm cannot handle, we introduced an approximate algorithm based on the exact algorithm that finds competitive solutions with the minimum solutions and also obtains better lower and upper bound values of the search space. Furthermore, we proposed an exact depth-first search algorithm that is equipped with search pruning techniques and incorporates the approximate algorithm to determine the lower and upper bounds of the search space. These techniques enable the exact depth-first search algorithm to be applied on large size instances. The proposed algorithms, exact and approximate, have been applied on a comprehensive set of instances including randomly generated and FIR filter instances, and compared with the exact CSE algorithm and the prominent graph-based heuristics. The following conclusions were drawn from the experimental results: (i) the exact graph-based algorithm finds significantly better results than the exact CSE algorithm; (ii) the proposed approximate algorithm obtains competitive and better solutions, and computes better lower and upper bound values of the search space than the existing graph-based heuristic algorithms; (iii) the proposed exact depth-first search algorithm can find the minimum solutions of the MCM problem instances that the previously proposed exact breadth-first search algorithm cannot handle and for which all the prominent graph-based heuristics find suboptimal solutions or cannot guarantee the minimum solutions.

References

[1] H. Nguyen, A. Chatterjee, Number-splitting with shift-and-add decomposition for power and hardware optimization in linear DSP synthesis, IEEE Transactions on VLSI 8 (4) (2000) 419–424.
[2] P. Cappello, K. Steiglitz, Some complexity issues in digital signal processing, IEEE Transactions on Acoustics, Speech, and Signal Processing 32 (5) (1984) 1037–1041.
[3] M. Potkonjak, M. Srivastava, A. Chandrakasan, Multiple constant multiplications: efficient versatile framework algorithms for exploring common subexpression elimination, IEEE TCAD 15 (2) (1996) 151–165.
[4] R. Hartley, Subexpression sharing in filters using canonic signed digit multipliers, IEEE TCAS II 43 (10) (1996) 677–688.
[5] R. Pasko, P. Schaumont, V. Derudder, S. Vernalde, D. Durackova, A new algorithm for elimination of common subexpressions, IEEE TCAD 18 (1) (1999) 58–68.
[6] I-C. Park, H-J. Kang, Digital filter synthesis based on minimal signed digit representation, in: Proceedings of DAC, 2001, pp. 468–473.
[7] R. Mahesh, A.P. Vinod, A new common subexpression elimination algorithm for realizing low-complexity higher order digital filters, IEEE TCAD 27 (2) (2008) 217–229.
[8] D. Bull, D. Horrocks, Primitive operator digital filters, IEE Proceedings G: Circuits, Devices and Systems 138 (3) (1991) 401–412.
[9] A. Dempster, M. Macleod, Use of minimum-adder multiplier blocks in FIR digital filters, IEEE TCAS II 42 (9) (1995) 569–577.
[10] K. Muhammad, K. Roy, A graph theoretic approach for synthesizing very low-complexity high-speed digital filters, IEEE TCAD 21 (2) (2002) 204–216.
[11] O. Gustafsson, H. Ohlsson, L. Wanhammar, Improved multiple constant multiplication using a minimum spanning tree, in: Proceedings of Asilomar Conference on Signals, Systems and Computers, 2004, pp. 63–66.
[12] Y. Voronenko, M. Püschel, Multiplierless multiple constant multiplication, ACM Transactions on Algorithms 3 (2) (2007).
[13] O. Gustafsson, L. Wanhammar, ILP modelling of the common subexpression sharing problem, in: Proceedings of ICECS, 2002, pp. 1171–1174.
[14] L. Aksoy, E. Costa, P. Flores, J. Monteiro, Exact and approximate algorithms for the optimization of area and delay in multiple constant multiplications, IEEE TCAD 27 (6) (2008) 1013–1026.
[15] A. Dempster, M. Macleod, Using all signed-digit representations to design single integer multipliers using subexpression elimination, in: Proceedings of ISCAS, 2004, pp. 165–168.

[16] A. Dempster, M. Macleod, Digital filter design using subexpression elimination and all signed-digit representations, in: Proceedings of ISCAS, 2004, pp. 169–172.
[17] E. Costa, P. Flores, J. Monteiro, Exploiting general coefficient representation for the optimal sharing of partial products in MCMs, in: Proceedings of SBCCI, 2006, pp. 161–166.
[18] K. Johansson, O. Gustafsson, L. Wanhammar, Bit-level optimization of shift-and-add based FIR filters, in: Proceedings of ICECS, 2007, pp. 713–716.
[19] O. Gustafsson, A. Dempster, L. Wanhammar, Multiplier blocks using carry-save adders, in: Proceedings of ISCAS, 2004, pp. 473–476.
[20] L. Aksoy, E.O. Gunes, P. Flores, An exact breadth-first search algorithm for the multiple constant multiplications problem, in: Proceedings of IEEE Norchip Conference, 2008, pp. 41–46.
[21] L. Aksoy, E.O. Gunes, An approximate algorithm for the multiple constant multiplications problem, in: Proceedings of SBCCI, 2008, pp. 58–63.
[22] A. Dempster, M. Macleod, Constant integer multiplication using minimum adders, IEE Proceedings – Circuits, Devices and Systems 141 (5) (1994) 407–413.
[23] O. Gustafsson, A. Dempster, K. Johansson, M. Macleod, L. Wanhammar, Simplified design of constant coefficient multipliers, Circuits, Systems, and Signal Processing 25 (2) (2006) 225–251.
[24] O. Gustafsson, Lower bounds for constant multiplication problems, IEEE TCAS II: Analog and Digital Signal Processing 54 (11) (2007) 974–978.
[25] Spiral webpage, <http://www.spiral.net>.
[26] H-J. Kang, I-C. Park, FIR filter synthesis algorithms for minimizing the delay and the number of adders, IEEE TCAS II: Analog and Digital Signal Processing 48 (8) (2001) 770–777.
[27] A. Dempster, S. Demirsoy, I. Kale, Designing multiplier blocks with low logic depth, in: Proceedings of ISCAS, 2002, pp. 773–776.

Levent Aksoy received his B.Sc. degree in Electronics and Communications Engineering, from Yıldız Technical University in 2000. He received his M.Sc. and Ph.D. degree from the Institute of Science and Technology, Istanbul Technical University (ITU) in 2003 and 2009 respectively. Since 2001, he has been a Research Assistant with the Division of Circuits and Systems, Faculty of Electrical and Electronics Engineering, ITU. During 2005–2006, he was a Visiting Researcher with the Algorithms for Optimization and Simulation Research Unit, Instituto de Engenharia de Sistemas e Computadores (INESC-ID), Lisbon, Portugal.
His research interests include satisfiability algorithms, pseudo-Boolean optimization, and electronic design automation problems.

Ece Olcay Güneş received her B.Sc. degree in Electronics and Communications Engineering from the Faculty of Electrical and Electronics Engineering, Istanbul Technical University, Turkey. She received her M.Sc. and Ph.D. degrees in 1991 and 1998, respectively, from the Institute of Science and Technology of the same university. She is currently a full Professor at the Electronics and Communications Department in ITU.
Her main research interests are analog circuit design, current-mode circuits and logic design.

Paulo Flores received the five-year Engineering degree, M.Sc., and Ph.D. degrees in Electrical and Computer Engineering from the Instituto Superior Técnico, Technical University of Lisbon, Lisbon, Portugal, in 1989, 1993, and 2001, respectively. Since 1990, he has been teaching at Instituto Superior Técnico, Technical University of Lisbon, where he is currently an Assistant Professor in the Department of Electrical and Computer Engineering. He has also been with the Instituto de Engenharia de Sistemas e Computadores (INESC-ID), Lisbon, since 1988, where he is currently a Senior Researcher.
His research interests are in the area of embedded systems, test and verification of digital systems, and computer algorithms, with particular emphasis on optimization of hardware/software problems using satisfiability (SAT) models.Dr. Flores is a member of the IEEE Circuit and Systems Society.