Optimization Algorithms for Multiple Constant Multiplications

Levent Aksoy\textsuperscript{1}, Eduardo Costa\textsuperscript{2}, Paulo Flores\textsuperscript{3}, and José Monteiro\textsuperscript{3}

\textsuperscript{1} Istanbul Technical University, Istanbul, Turkey  
levent@ehb.itu.edu.tr
\textsuperscript{2} Catholic University of Pelotas, Pelotas, RS, Brazil  
ecosta@ucpel.tche.br
\textsuperscript{3} INESC-ID / Instituto Superior Técnico, TU Lisbon, Lisbon, Portugal  
\{pff,jcm\}@inesc-id.pt

Abstract. We describe efficient parallel architecture implementations of digital processing systems, namely FIR filters, that require the multiplication of each input sample by a set of constant coefficients. These architectures allow for significant reductions in hardware, and consequently power, by sharing the partial products of the input among the set of multiplications.

We present an exact algorithm for common subexpression elimination that finds the optimum sharing of partial terms in Multiple Constant Multiplications (MCM). We model this problem as a Boolean network that covers all possible partial terms which may be used to generate the set of coefficients in the MCM instance. This problem is cast into a 0-1 Integer Linear Programming (ILP) problem by requiring that the single output of this network is asserted while minimizing the number of gates representing operations in the MCM implementation that evaluate to one. A SAT-based 0-1 ILP solver is used to obtain the exact solution. We argue that for many real problems the size of the problem is within the capabilities of current SAT solvers.

Because performance is often a primary design parameter, we describe how this algorithm can be modified to target the minimum area solution under a user-specified delay constraint.

Additionally, we describe approximate algorithms based on the exact approach with extremely competitive results. We have applied these algorithms on the design of digital filters and present a comprehensive set of results, that evaluate ours and existing approximation schemes against exact solutions.

1 Introduction

In several computationally intensive operations, notably Finite Impulse Response (FIR) filters, the same input is to be multiplied by a set of coefficients, an operation known as Multiple Constant Multiplications (MCM). These operations are common in Digital Signal Processing (DSP) applications and hardwired dedicated architectures are the best option for maximum performance and minimum power consumption.
Constant coefficients allow for a great simplification of the multipliers. When the same input is to be multiplied by a set of constant coefficients, significant reductions in hardware, and consequently power, can be obtained by sharing the partial products of the input among the set of multiplications. This chapter describes an algorithm that optimally solves the maximal sharing of partial terms. Although this problem has been proven to be NP-hard [1], we demonstrate that for many practical instances the size of the problem still allows for the computation of the optimum solution.

In many designs, particularly in DSP systems, performance is a critical parameter. Hence, circuit area is generally expendable in order to achieve a given performance target. The exact algorithm we describe is able to be parametrized with a delay constraint so that only solutions that meet the desired delay are considered. Thus, the obtained solution is the minimum area solution under the specified maximum delay.

This chapter is organized as follows. In Section 2, we give a brief overview of FIR filter implementation that serves as a motivation for the algorithms proposed for the MCM. Section 3 defines the problem that we address and Section 4 introduces main concepts on number representation. The model developed for the exact algorithm is described in Section 5. Section 6 presents how this model can be extended to limit the search to solutions that meet a maximum delay constraint. We describe heuristic algorithms designed for the MCM problem in Section 7. Section 8 presents and discusses a set of results on selected benchmarks. Finally, in Section 9 we summarize the main contributions and give directions for future research.

2 FIR Filter Design

Finite Impulse Response (FIR) filters are of great importance in the Digital Signal Processing (DSP) systems. Their characteristics in linear phase and feed forward implementation make them very useful for building stable high performance filters. The problem of designing FIR filters has received a significant amount of attention during the last decade, because the filters require a large number of multiplications, leading to excessive area, delay and power consumption, even if implemented in a full custom integrated circuit. In this section, we present the main aspects related to the frequency response and architectural design of FIR filters.

FIR filtering is achieved by convolving the input data samples with the desired unit impulse response of the filter. The output $y[n]$ of an $N$-tap FIR filter is given by the weighted sum of the latest $N$ input data samples $x[n]$ as shown in Equation 1.

$$ y[n] = \sum_{i=0}^{N-1} H_i x[n - i] $$

The characteristics of digital filters are often specified in the frequency domain. For frequency selective filters, such as low-pass and band-pass filters, the
specifications are often in the form of tolerance schemes. A typical specification of a low-pass filter is depicted in Figure 1.

In Figure 1, the dashed horizontal lines indicate the tolerance limits. In the pass-band (< $W_p$) and the stop-band (> $W_s$), the magnitude response has a maximum deviation of $s$. The width of the transition band determines how sharp the filter is. The magnitude response decreases monotonically from the pass-band to the stop-band in this region. The value of $s$ and the width of the transition band determine the required number of coefficients, $N$, for the filter.

Using the set of parameters $W_p$, $W_s$, and $s$, the coefficients of the FIR filter are obtained using the Discrete Fourier Transform (DFT) of the required frequency transfer function, with one of many existing windowing methods [2].

In the direct form FIR filter implementation, Figure 2(a), in each clock cycle a new data sample, the delayed input values, and corresponding filter coefficients are applied to each multiplier. The result of each multiplier is added simultaneously, producing a long critical delay and a significant amount of glitching that propagates through all the adders [3].

An alternative fully-parallel architecture [4], called the transposed form, is depicted in Figure 2(b). This architecture presents the same complexity as the direct form, but it involves multiplying all the coefficients by the same input data. Because registers are now between the adders, the maximum delay is the series of a multiplier with an adder. Also, most of the glitching is filtered by the registers, resulting in a significant power reduction compared to the direct form architecture. Thus, although both the direct form and transposed architectures of the FIR filter have the same complexity, the transposed form is preferred due to its higher performance, power efficiency, and more relevant to this chapter, its potential for sharing the partial products of the multiplier modules [5, 6].

In the transposed form architecture of a FIR filter, the same input value is to be multiplied by a set of constant coefficients. This is a situation that is common to several other DSP algorithms, namely the Fast Fourier Transforms (FFT) and DSP transforms, and is generally known as Multiple Constant Multiplications (MCM).
3 Multiple Constant Multiplications

We address the problem of minimizing the hardware required for a parallel multiplication of an input value over a set of constant coefficients. As mentioned, a paradigmatic example of an application where MCM are realized is the implementation of the multiplier block of a digital FIR filter in the transposed form. As illustrated in Figure 3, the main objective is to replace a set of constant multiplications using generic multipliers as presented in Figure 2(b) by a unique MCM block that is able to compute the same output values with less hardware, delay, and power.

As mentioned in the previous section, since all coefficients are constant, in order to reduce the hardware implementation we can replace a full-fledged multiplier by a set of additions of shifted versions of the input [7]. A bit set to 1 in position $m$ of the coefficient implies the sum of the input ($x$) shifted left
by \( m \) positions. Shifts are free in terms of hardware, hence the hardware required for a multiplication with a constant with \( n \) bits set to 1 are simply \( n - 1 \) adders. Figure 4 presents an example of how \( 11x \) can be implemented using a shift-add approach.

![Diagram](a) ![Diagram](b)

**Fig. 4.** Computation of \( 11x \) using: (a) multiplier; (b) shift-adds.

Each addition generates a partial term. If the same input is to be multiplied by a set of constant coefficients, significant savings can be accomplished by sharing partial terms among the coefficients multiplications. To illustrate this point, consider that we need to implement both \( 7x \) and \( 11x \). Instead of using two adders per coefficient as in Figure 5(a), we can share the adder that generates the value \( 3x \) to obtain an implementation with a total of three adders, Figure 5(b).

![Diagram](a) ![Diagram](b)

**Fig. 5.** Simultaneous computation of \( 7x \) and \( 11x \): (a) no sharing; (b) sharing the partial term \( 3x \).

We make two immediate notes about the sharing of partial terms. The first is that all values obtained through a shift of any partial term can be considered. The second is that the sharing depends on how the coefficients are decomposed, because the partial terms depend on the sequence of additions. Returning to
our example, the sharing exploited in Figure 5(b) was possible, because we used the decomposition $11x = 2^3x + (2^1x + x)$. If instead we had used $11x = (2^3x + 2^1x) + x$, the same level of sharing could be obtained, albeit using the partial term $(2^3x + 2^1x)$, equivalent to $(2^2x + x)$ shifted left by one. However, if the decomposition is $11x = 2^1x + (2^3x + x)$, no sharing is possible with partial terms of $7x$.

This problem can be regarded as a particular case of a more general problem known as Common Subexpression Elimination (CSE) [8].

Definition 1 (Unconstrained maximum sharing problem).
Given a set of constant coefficients, find the minimum number of operations (additions or subtractions) required to implement the MCM block.

In order to consider not only the number of operations (with direct implication in the hardware area) but also the multiplication performance of a given implementation, a delay metric have to be defined. As the delay is dependent on several implementation issues, such as circuit technology, placement, and routing, we consider the delay as number of operators in series to produce any multiplication, generally called as the number of adder-steps [9]. Note that the definition of adder-steps is identical to the definition of level in combinational circuits.

Definition 2 (Number of adder-steps).
The maximum number of operations (additions or subtractions) that a signal traverses from the input to the outputs to produce any multiplication is defined as the number of adder-steps of a given implementation.

Clearly, the maximum number of adder-steps over all coefficients defines the maximum delay of one computation. For example, as shown in Figure 6, $23x$ can be implemented as $23x = 2^4x + (2^2x + (2^1x + x))$ with three adder-steps, or as $23x = (2^4x + 2^2x) + (2^1x + x)$ with two.

The implementation of a coefficient, or a partial term, with $n$ non-zero digits, has a number of adder-steps between the maximum of $n - 1$ (operators layout in cascade) and the minimum of $\lceil \log_2 n \rceil$ (operators layout as a binary tree).

We extend the maximum sharing problem so that we can limit the maximum number of adder-steps of an implementation. The problem of optimizing area under a delay constraint [10] can be defined as the minimization of the number of adders/subtracters such that a user-specified delay is not exceeded.

Definition 3 (Maximum sharing problem under a delay constraint).
Given a set of constant coefficients and a maximum number of adder-steps, find the minimum number of operations (additions or subtractions) required to implement the MCM block so that the user-specified maximum number of adder-steps is not exceeded.
4 Number Representation

In the previous section, all examples use the binary representation for the numerical values, where a number is decomposed as a sum of powers of two. Although this is the numerical representation of choice for computer arithmetic, alternative representations can offer advantages when implementing multiplications with known constants based on shift-adds.

The representation of numbers using a sign digit system makes the use of positive and negative digits. The binary sign digit representation decomposes a number in a set of additions and subtractions of powers of two. Thus, an integer $k$ represented in the binary sign digit system can be written as:

$$k = \sum_{i=0}^{N-1} c_i 2^i$$

where $c_i \in \{1, 0, -1\}$ and $N$ is the number of digits used to represent $k$. Hereafter, the digit $-1$ will be denoted by $\bar{1}$.

The Canonical Signed Digit (CSD) representation [8] is a signed digit system that has a unique representation for each value and verifies two main properties:

1. the number of non-zero digits is minimal,
2. two non-zero digits are not adjacent.

This representation is widely used in multiplierless implementations, because it reduces the hardware requirements due to the minimum number of non-zero digits. Any $N$ digit number in CSD format has at most $(N+1)/2$ non-zero digits, thus requires only that number of operators (adders/subtracters). On average the number of non-zero digits is reduced by 33% when compared with the binary representation [11].
To obtain the CSD representation of a number, one could start processing its binary representation from the least significant digit to the most significant digit and replace repeatedly all the sequences found as 01...1 by the sequence 10...01 with the same number of digits. This procedure is implemented by the pseudo-code presented in Figure 7 which uses a conversion table and a state variable to detect the ones sequences and generate the CSD output [12].

<table>
<thead>
<tr>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>( b_{i+1} )</td>
<td>( b_i )</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

\( b \) is the binary representation of the number

\( N \) is the number of digits used to represent \( b \)

\[
\text{CSD} (b, N) \{
\text{state} = 0;
b_N = 0;
b_{N+1} = 0;
\text{for } i = 0 \text{ to } N \{
\quad c_i = \text{get\_out\_val\_from\_table}(\text{state}, b_{i+1}, b_i);
\quad \text{state} = \text{get\_next\_state\_from\_table}(\text{state}, b_{i+1}, b_i);
\}
\text{return } c_i;
\}
\]

Fig. 7. Pseudo-code to compute the CSD representation of a number using a conversion table.

The Minimum Signed Digit (MSD) representation [13] is obtained by dropping the second property of the CSD representation. Thus, a constant can have several representations under MSD, but all with a minimum number of non-zero digits. For example, suppose the constant 23 defined in six bits. The representation of 23 in binary, 010111, includes 4 non-zero digits. The constant is represented as 101001 in CSD and both 101001 and 011001 denote 23 in MSD with 3 non-zero digits.

The MSD representations of a number can be computed from its CSD representation by replacing all possible combinations of the sequences 10\( \top \) and \( \top 01 \) by the sequences 011 and \( 0 \text{\_} 1 \) respectively. For each replacement, a MSD represen-
$c$ is the CSD representation of the number $N$ is the number of digits used to represent $c$

$$\text{MSD}(c, N) \{$$
$$\text{MSD}_{\text{set}} = \{c\};$$
$$\text{working}_{\text{set}} = \{c\};$$
$$\text{while } (\text{working}_{\text{set}} \neq \emptyset) \{$$
$$\text{one}_{\text{rep}} = \text{get}_{\text{element}}(\text{working}_{\text{set}});$$
$$\text{for } i = 0 \text{ to } N - 2 \{$$
$$\text{if } (\text{matching}(\text{one}_{\text{rep}}, '011')) \{$$
$$\text{new}_{\text{rep}} = \text{replace}_{\text{three}}_{\text{matching}}_{\text{digits}}(\text{one}_{\text{rep}}, '011');$$
$$\text{working}_{\text{set}} = \text{working}_{\text{set}} \cup \{\text{new}_{\text{rep}}\};$$
$$\text{MSD}_{\text{set}} = \text{MSD}_{\text{set}} \cup \{\text{new}_{\text{rep}}\};$$
$$\}$$
$$\text{if } (\text{matching}(\text{one}_{\text{rep}}, '101')) \{$$
$$\text{new}_{\text{rep}} = \text{replace}_{\text{three}}_{\text{matching}}_{\text{digits}}(\text{one}_{\text{rep}}, '101');$$
$$\text{working}_{\text{set}} = \text{working}_{\text{set}} \cup \{\text{new}_{\text{rep}}\};$$
$$\text{MSD}_{\text{set}} = \text{MSD}_{\text{set}} \cup \{\text{new}_{\text{rep}}\};$$
$$\}$$
$$\}$$
$$\text{return } \text{MSD}_{\text{set}};$$
$$\}$$

Fig. 8. Pseudo-code to compute the MSD representation of a number from its CSD representation.

...tation is obtained since the number of non-zero digits does not increase. Figure 8 presents the pseudo-code of an algorithm that computes the MSD representations of a number from its CSD representation using the described replacement technique. Note that other optimized and alternative algorithms that compute the CSD and MSD representations of a number are presented in [13–15].

The advantage of using the MSD representation for a coefficient results from increasing the possibilities of sharing partial terms between coefficients. This results from the fact that, in general, there exist several alternatives to represent a given coefficient in MSD. Consequently, there are more ways to decompose the coefficient with different partial terms that can be shared with other coefficients. Figure 9 shows all possible implementations that obtain $23x$ using CSD and MSD representations of the constant 23.

5 An Exact Algorithm for the Minimization of the Number of Operations

The Multiple Constant Multiplications (MCM) problem can be represented as a 0-1 Integer Linear Programming (ILP) problem and the minimum number of
In this section, we describe an exact CSE algorithm designed for the minimization of the number of addition/subtraction operations in the multiplier block of a digital FIR filter. The proposed algorithm can handle any type of number representation, namely binary, CSD, or MSD. In the proposed algorithm, initially, all possible implementations of filter coefficients and partial terms are obtained when filter coefficients are defined under a number representation and a Boolean network that represents the implementations of filter coefficients is constructed using only AND and OR gates. In the network, an AND gate represents an addition/subtraction operation and an OR gate representing a constant combines all possible operations that compute the related constant. Then, a 0-1 ILP problem is formed with a cost function to be minimized and constraints to be satisfied. In the 0-1 ILP problem, the cost function is the linear function of the optimization variables that are associated with operations and the constraints
Optimization Algorithms for Multiple Constant Multiplications

are the Pseudo-Boolean (PB) constraints of each gate in the network. Finally, the minimum number of operations solution is obtained using a generic 0-1 ILP solver.

5.1 Finding the Implementations of a Constant

In the preprocessing phase of the algorithm, the filter coefficients are converted to positive and then made odd by successive divisions by 2, since shifts can be implemented using only wires in hardware. The resulting constants are stored in a set called \(Cset\) without repetition. Thus, the \(Cset\) includes the minimum number of necessary coefficients to be multiplied by the filter input. After each element in \(Cset\), \(Cset_i\), is represented under a given number representation, all possible implementations of \(Cset_i\) are found by decomposing the non-zero digits in the representations of \(Cset_i\) into two partial terms. The part of the exact algorithm where the implementations of each filter coefficient and partial term are found is given as follows:

1. Take an element from \(Cset\), \(Cset_i\), find its representations under the given number representation, and store them in a set called \(Rset\).
2. For each representation of \(Cset_i\) in the \(Rset\),
   (a) Compute all non-symmetric partial term pairs that cover the \(Cset_i\).
   (b) Make each partial term positive and odd.
   (c) Add each partial term pair to the set \(Iset_i\) associated with the \(Cset_i\).
   (d) Add each partial term to the \(Cset\), if it does not represent the filter input, i.e., 1, and is not in the \(Cset\).
3. Repeat Step 1 until all elements of the \(Cset\) are considered.

Observe that the \(Cset\) that only includes the filter coefficients in the beginning of the algorithm is augmented with the partial terms that are required for the implementation of filter coefficients. We note that the \(Iset_i\) contains the inputs of all addition/subtraction operations that compute the \(Cset_i\) as a pair.

As an example on finding all possible implementations of a constant, suppose 51 as a filter coefficient defined under CSD representation as 10\(\ll 6\)1010\(\ll 1\) with four non-zero digits. The possible implementations of 51 are given in Figure 10.

\[
\begin{align*}
51 &= 1000000 + 00\overline{1010}\ll 1 - 13 \\
51 &= 10\overline{0000} + 000010\ll 1 = 3\ll 4 + 3 \\
51 &= 00\overline{0000} + 100010\ll 1 = -1\ll 4 + 67 \\
51 &= 1000100 + 00\overline{0010}\ll 1 = 17\ll 2 - 17 \\
51 &= 0000100 + 10\overline{0000}\ll 1 = 1\ll 2 + 47 \\
51 &= 100000\ll 1 + 00\overline{0100} = 63 - 3\ll 2
\end{align*}
\]

Fig. 10. Implementations of 51 under CSD representation.

We note that the duplications of implementations that can be obtained with the commutative law of the addition/subtraction operation, such as 63 - 3\ll 2 = -3\ll 2 + 63, and that contain the same positive and odd partial term pair at the inputs of an operation, such as 1\ll 6 - 13 = 13\ll 2 - 1, are not listed in Figure 10.
Observe that after the partial terms required for the implementation of 51 are found, they are made positive and odd, are added to the Cset without repetition, and their implementations are also found in similar way.

5.2 Construction of the Boolean Network

After all possible implementations of filter coefficients and partial terms are found, these implementations are represented in a Boolean network that only includes AND and OR gates. The part of the algorithm where the Boolean network is constructed is as follows:

1. Take an element from the Cset, Cset_i.
2. For each partial term pair in the Iset_i, generate a two-input AND gate where its inputs are the filter inputs or the outputs of the OR gates representing the filter coefficients and partial terms.
3. For the Cset_i, generate an OR gate where its inputs are the outputs of the AND gates determined in Step 2.
4. If the Cset_i is a filter coefficient, assign the output of the corresponding OR gate as a primary output of the network.
5. Repeat Step 1 until all elements in the Cset are considered.

The properties of the Boolean network that represents the implementations of filter coefficients and partial terms are as follows:

1. Primary inputs of the network are the filter inputs or their shifted versions.
2. An AND gate in the network represents an addition/subtraction operation and has two inputs.
3. An OR gate in the network represents a filter coefficient or a partial term and combines all possible implementations of the constant.
4. The primary outputs of the network are the OR gate outputs associated with the filter coefficients.

The Boolean network generated for the filter coefficient 51 defined in CSD representation is given in Figure 11 where the 1-input OR gates for the partial terms 3, 17, and 63 are omitted.

We note that when constants are defined in CSD or MSD representation, an AND gate represents an addition/subtraction operation and when constants are defined in binary representation, an AND gate represents an addition operation. Also, observe that the exact algorithm can handle the constants defined in MSD representation that achieves alternative representations of a constant, since all the implementations of a constant are simply the inputs of an OR gate representing the constant.

In the conversion of the MCM problem to a 0-1 ILP problem, we need to include the optimization variables to the network, so that the cost function to be minimized, i.e., the linear function of the optimization variables, can be constructed. To do this, the optimization variables can be associated with operations or partial terms as described in [19] and [17] respectively. In this work,
we present the minimization of the number of operations model where the optimization variables are associated with the operations that are required for the implementations of filter coefficients and partial terms. In this model, we add the third input representing an optimization variable to each AND gate in the network. Hence, the solution to the minimization of the cost function will indicate directly which operations are required for the optimum solution. We can make a simple observation for this model.

**Lemma 1.** The number of optimization variables set to 1 among the AND gates that feed the same OR gate is 1.

We note that any optimization variable in an AND gate with one other input set to 0 will necessarily be 0. Otherwise, we have a contradiction as setting it to 0 would be a solution with a lower cost function. For the remaining AND gates, one suffices to set the output of the OR gate to 1. Hence, only one optimization variable over those AND gates will be 1 in order to minimize the cost function.

\[ \square \]

### 5.3 Network Simplifications

Problem reduction techniques can also be used to reduce the size of the network, and consequently, the size of the 0-1 ILP problem, thus increasing the performance of the generic 0-1 ILP solver. The following rules can be applied to remove unnecessary inputs from the gates and gates from the network.
1. Since there is no need to implement the filter input, we assign 1 value to the filter input and propagate this value to remove unnecessary gates in the network.

2. Since the implementation of filter coefficients is aimed, we assign 1 value to the OR gates representing filter coefficients in the network and make these implications.

3. If an operation includes two identical partial terms at the inputs, one of them can be removed from the inputs.

4. If the requirements of an operation are more stringent than another operation that generates the same constant, we may remove it. For example, in Figure 11, the operation $51 = 63 - 3 \ll 2$ requires partial terms 63 and 3, whereas the operation $51 = 3 \ll 4 + 3$ only requires the partial term 3, thus we may eliminate the former, because if the partial term 3 is available, we can always use the latter.

Figure 12 presents the Boolean network for the filter coefficient 51 under CSD representation of Figure 11, after the optimization variables are included for each operation and problem reduction techniques are applied. Observe that by adding the optimization variables and applying the problem reduction techniques, the primary inputs of the network are determined as the optimization variables of the operations that can be implemented with a single operation whose inputs are filter inputs or their shifted versions and the number of inputs of an AND gate can be two or three.

5.4 Conversion to 0-1 ILP Problem

After the Boolean network is constructed, the conversion of the MCM problem into a 0-1 ILP problem is then straightforward. The cost function is formed as a linear function of optimization variables where the cost value of each optimization variable is 1. The outputs of the OR gates associated with the filter coefficients, i.e., the primary outputs of the network, are set to 1, since the implementation of filter coefficients is aimed. The constraints of the 0-1 ILP problem are obtained by finding the conjunctive normal form (CNF) formulas of each gate in the network and expressing each clause in CNF formulas as a linear inequality as described in [20]. For example, a 2-input AND gate, $c = a \land b$, is translated to CNF as $(a + \overline{c})(b + \overline{c})(\overline{a} + \overline{b} + c)$ and converted to PB constraints as follows:

$$a - c \geq 0$$
$$b - c \geq 0$$
$$-a - b + c \geq -1$$

5.5 Analysis of 0-1 ILP Problem Size

When a coefficient in binary with $n$ bits all set to 1 is given, the Boolean network will include all partial terms with $\leq n$ bits set to 1. Hence, the complexity of
the 0-1 ILP problem is bounded above by the case of a single coefficient with all the $n$ bits set to 1, since any additional coefficient with less than $n$ bits will be computed with a subset of the existing partial terms. Observe that in this case, the number of coefficients is not very relevant. The following complexity analysis is done under this assumption without taking into account the effect of the problem reduction techniques described in Section 5.3. Hence, an upper bound on the size of the 0-1 ILP problem is obtained.

For a given constant with $n$ bits all set to 1, the total number of gates in the Boolean network is given by:

$$G_{or}(n) = \sum_{i=3}^{n} \left( \begin{array}{c} n - 1 \\ i - 1 \end{array} \right) = \sum_{i=3}^{n} \prod_{k=1}^{i-1} \frac{n-k}{k}$$

$$G_{and}(n) = n - 1 + \sum_{i=3}^{n} (2^{i-1} - 1) \left( \begin{array}{c} n - 1 \\ i - 1 \end{array} \right) = n - 1 + \sum_{i=3}^{n} (2^{i-1} - 1) \prod_{k=1}^{i-1} \frac{n-k}{k}$$

As we cast the MCM problem into a 0-1 ILP problem, the relevant complexity parameters are: the number of variables, clauses, and optimization variables. The number of optimization variables is simply the number of AND gates, $\#opt\_vars(n) = G_{and}(n)$. The total number of variables is given by the total number of gates in the network, plus the primary inputs, i.e., $\#vars(n) =$
Finally, the number of clauses can be computed by noting that, for each logic gate, the number of clauses is the number of gate inputs plus one. All AND gates in the network have three inputs, hence each contributes with 4 clauses. Although the number of inputs to the OR gates varies, we note that for a given level, the total number of inputs to all the OR gates at that level is the number of AND gates. Thus, the total number of clauses is:

\[
\text{due to ANDs} \quad \text{due to ORs} \\
\#\text{clauses} = 4G_{\text{and}}(n) + (G_{\text{and}}(n) + G_{\text{or}}(n)) = 5G_{\text{and}}(n) + G_{\text{or}}(n)
\]

Table 1 gives the size of the Boolean network in terms of the number of OR and AND gates, and the size of the 0-1 ILP problem in terms of the number of clauses, variables, and optimization variables with different values of \( n \). Although the size of the 0-1 ILP problem grows exponentially with \( n \), we again note that the complexity analysis is based on a single coefficient with all bits set to 1 and the effect of network simplifications that may reduce the size of the problem significantly, hence allowing the exact algorithm to be applied to much larger designs is not considered, i.e., the worst case analysis. However, we note that the size of the 0-1 ILP problem for \( n = 12 \) is still within reach of current 0-1 ILP solvers thus, can be solved exactly. Also, observe that when a single coefficient is defined under CSD or MSD representation, a smaller size 0-1 ILP problem is obtained than that of binary, since both representations of a constant include less number of non-zero digits than the binary representation of a constant.

<table>
<thead>
<tr>
<th>( n )</th>
<th>#OR</th>
<th>#AND</th>
<th>#clauses</th>
<th>#vars</th>
<th>#opt_vars</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>120</td>
<td>2,059</td>
<td>10,415</td>
<td>2,187</td>
<td>2,059</td>
</tr>
<tr>
<td>10</td>
<td>502</td>
<td>19,171</td>
<td>96,357</td>
<td>19,683</td>
<td>19,171</td>
</tr>
<tr>
<td>12</td>
<td>2,036</td>
<td>175,099</td>
<td>877,531</td>
<td>177,147</td>
<td>175,099</td>
</tr>
<tr>
<td>14</td>
<td>8,178</td>
<td>1,586,131</td>
<td>7,938,833</td>
<td>1,594,323</td>
<td>1,586,131</td>
</tr>
<tr>
<td>16</td>
<td>32,752</td>
<td>14,316,139</td>
<td>71,613,447</td>
<td>14,348,907</td>
<td>14,316,119</td>
</tr>
</tbody>
</table>

It is also interesting to note that when the MCM problem is converted to a 0-1 ILP problem using the minimization of the number of partial terms model as described in [17], a smaller size 0-1 ILP problem can be obtained with respect to the 0-1 ILP problem obtained with the minimization of the number of operations model. However, optimization problems, such as the minimization of the number of operations under a delay constraint [19] as introduced in Section 6 and the minimization of area in digital filters in terms of gate-level metrics [21], can be described as 0-1 ILP problems under the minimization of the number of operations model, since its minimum solution directly points the operations to be implemented.
6 Minimization of the Number of Operations under a Delay Constraint

In this section, we introduce the exact algorithm designed for the MCM problem under a delay constraint as described in [19]. The algorithm is based on the minimum number of operations model described in Section 5.2 and the delay is considered as the number of adder/subtracters in series to produce any constant multiplication. Since the definition of adder-steps is identical to the definition of level in combinational circuits, in the following we use both definitions interchangeably.

6.1 Computing the Levels of Operations and Partial Terms

In general, a partial term can be implemented with operations that have different adder-steps. Therefore, we can define a range of levels for each partial term, and consequently, a range of levels for the operations that compute this partial term. For example, a partial term with \( n \) non-zero digits can be implemented in a tree of operations with the minimum latency, i.e., \( \lceil \log_2 n \rceil \) adder-steps, and in a chain of operations with the maximum latency, i.e., \( n - 1 \) adder-steps. In the Boolean network, an OR gate associated with the partial term gathers all of these operations. So, a partial term can be generated with the number of adder-steps ranging from its minimum to maximum latency implementations. As can be seen from Figure 11, the filter coefficient 51 can be implemented with minimum 2 and maximum 3 adder-steps, determined, for instance, by 51 = \( 3 \cdot 4 + 3 \) which has a minimum and a maximum of 2 adder-steps and by 51 = \( 13 \cdot 2 - 1 \) which has a minimum and a maximum of 3 adder-steps.

The proposed algorithm can find the minimum number of operations solution with either the minimum delay that the network can have, \( \text{min\_delay} \), or a user-specified maximum delay constraint, \( \text{user\_delay} \). After the Boolean network is constructed, we compute the minimum level (\( \text{min\_level} \)) and maximum level (\( \text{max\_level} \)) values of each operation and partial term in the network by traversing the network from primary inputs to primary outputs. Then, we find the \( \text{min\_delay} \) value by computing the maximum of the \( \text{min\_level} \) values of the primary outputs. By setting \( \text{user\_delay} = \text{min\_delay} \) as the maximum delay constraint, the algorithm that we propose is an exact algorithm that gives the minimum number of operations with the minimum delay. Naturally, if the user sets \( \text{user\_delay} < \text{min\_delay} \), no solution is possible.

6.2 Finding the Delay Constraints

In the algorithm, initially, we find the paths in the network that exceed the maximum delay constraint using the information on minimum and maximum levels of operations and partial terms and then, for each path, we add a delay constraint to the 0-1 ILP problem to prevent all the operations in the path from being selected in the final solution. The part of the algorithm where the paths that exceed the \( \text{user\_delay} \) are found is as follows:
1. **Preprocessing phase**: Determine the primary outputs of the network that have \( \text{max} \_\text{level} \) values higher than the \( \text{user} \_\text{delay} \) and store them in a set called \( \text{Pset} \).

2. For each element in the \( \text{Pset} \), \( \text{Pset}_i \),
   (a) If an operation that computes \( \text{Pset}_i \) has \( \text{min} \_\text{level} \) value higher than \( \text{user} \_\text{delay} \), remove this operation from the network. Because it can never be used to meet the \( \text{user} \_\text{delay} \).
   (b) Otherwise, if the operation has \( \text{max} \_\text{level} \) value higher than \( \text{user} \_\text{delay} \), add this operation to a set called \( \text{path}_j \) as an initial node and also add this operation to a set called \( \text{Oset} \) with a target level, \( \text{user} \_\text{delay} - 1 \), and the associated path identifier, \( j \).

3. **Main loop**: Remove an operation from the \( \text{Oset} \) with its target level, \( \text{target} \), and the associated path identifier, \( j \). For each input of the operation, \( P_k \), i.e., a partial term,
   (a) If an operation that implements \( P_k \) has \( \text{min} \_\text{level} \) value higher than \( \text{target} \), add this operation to \( \text{path}_j \) as a terminal node, and construct the path.
   (b) Otherwise, if an operation has \( \text{max} \_\text{level} \) value higher than \( \text{target} \), form a new path by adding this operation to the \( \text{path}_j \). Also, insert this operation into the \( \text{Oset} \) with its target level, \( \text{target} - 1 \), and a path identifier.

4. Repeat Step 3 until there is no element left in the \( \text{Oset} \).

Observe that in the preprocessing phase of the algorithm, the \( \text{Pset} \) includes the filter coefficients that can be implemented in a greater delay than \( \text{user} \_\text{delay} \) and at the end of the preprocessing phase, the initial nodes of the paths that violate the \( \text{user} \_\text{delay} \) constraint are found. We also note that in the main loop, the paths are constructed in a breadth-first manner and the \( \text{Oset} \) includes the operations that are the last nodes of the paths have not been constructed yet.

As an example, suppose that the filter coefficient represented with the output of the OR gate \( A \) is to be implemented in 5 adder-steps, i.e., the \( \text{user} \_\text{delay} \), as given in Figure 13. In this figure, optimization variables are omitted and the relevant paths are highlighted for the sake of clarity. The operations and partial terms are labeled with letters inside the gates and the \( \text{min} \_\text{level} \) and \( \text{max} \_\text{level} \) values are given with a \( \text{min} \text{-max} \) pair above the gates. \( \text{path} \) includes the operations that exceed the \( \text{user} \_\text{delay} \), determined when traversing the network from the primary output to the primary inputs.

In the preprocessing phase, the operation \( B \) is added to the initial node of \( \text{path}_1 \) and to the \( \text{Oset} \) with its target level value 4 and path identifier 1, since its \( \text{max} \_\text{level} \) value is higher than the \( \text{user} \_\text{delay} \). Observe that while the operation \( C \) never meets the user specified delay constraint and thus, can be removed from the network, the operation \( D \) never violates the user specified delay constraint. In the main loop, the operation \( B \) with its target level, \( \text{target}(B)=4 \), and associated path identifier, 1, is removed from the \( \text{Oset} \). Suppose that the partial term \( E \) is considered as the input of \( B \). The operation \( H \) is added to the \( \text{path}_1 \) as a terminal node and the path is constructed, since the operation \( H \) can be implemented in minimum 5 adder-steps that exceeds the \( \text{target}(B) \). Also, a new path, \( \text{path}_2 \), is
Fig. 13. An illustrative example of determining the paths that exceed the maximum delay constraint.

formed by inserting the operation $F$ to the path $1$, since the max-level value of the operation $F$ is higher than $\text{target}(B)$ indicating that there is an operation(s) that cause greater delay than the user_delay with the operations in this path. So, the operation $F$ with its target level $\text{target}(F)=\text{target}(B)-1$ and associated path identifier, 2, is added to the $\text{Oset}$. We note that the operation $G$ is not considered to be added to the path $1$, because it can be implemented in maximum of 4 adder-steps that does not exceed the $\text{target}(B)$ value. Observe that all the paths are found when there is no element left in the $\text{Oset}$.

After all paths that violate user_delay have been found, for each path, a delay constraint, $-\text{optvar}_1 - \text{optvar}_2 - \ldots - \text{optvar}_m \geq 1 - m$, where $\text{optvar}_j$, $1 \leq j \leq m$, denotes the optimization variable of an operation in the path and $m$ is the number of operations in the path, is added to the 0-1 ILP problem. The delay constraints express that the operations in the path must not be included together in the solution. This guarantees that the solution to be found by the 0-1 ILP solver respects the delay constraints and allows for the possible sharing of partial terms in the paths with other partial terms not in the critical paths. Finally, using the same cost function and the constraints obtained from the Boolean network together with these delay constraints are given to the 0-1 ILP solver to find a solution with the minimum number of operations.

7 Heuristic Algorithms

Although the exact algorithms presented in the two previous sections can be applied effectively to relatively large MCM problems, the execution time does tend to grow exponentially, limiting its application to more complex instances. Thus, heuristic algorithms are necessary to find solutions on MCM instances that the exact algorithm cannot cope with.
There have been a number of proposed techniques on the optimization of area in a digital FIR filter, mostly based on finding common digit patterns in the coefficients. These methods range from the graph-based coefficient synthesis techniques [22, 23], and incorporation of two-term common subexpressions [24] to exhaustive enumeration of all possible digit patterns [25].

In this section, initially, we describe an intuitive heuristic algorithm [26] that maximizes the sharing of partial terms in MCM where constants are defined under MSD representation. Then, we extend this algorithm for general coefficient representation, under which only the numerical values of the coefficients are considered, hence without being limited to any number representation [27].

### 7.1 Maximal Sharing under MSD Representation

Park et al propose the usage of the MSD representation for the coefficients in [13]. The proposed algorithm exploits the redundancy of the MSD representation by choosing heuristically the MSD representation of the coefficients that leads to a maximal sharing in the implementation of efficient FIR filters. The algorithm we describe augments the search conditions of [13], thus is significantly more effective in area optimization.

We have two sets: $C_{set}$ maintains all MSD representations of the coefficients not yet covered; $P_{atset}$ is the set with the partial terms found so far. Before being inserted into $C_{set}$, all MSD representations are shifted right such that the least significant bit is 1, and also any duplicates that may appear in this process are eliminated. $P_{atset}$ is initialized with a single element, the value 1. We then enter a loop where all shifted versions of elements in $P_{atset}$ are pair-wised added and subtracted, as defined by the following steps:

1. remove all coefficients in $C_{set}$ that have the same MSD representation as a shifted value of an element in $P_{atset}$.
2. remove all coefficients in $C_{set}$ whose MSD representation can be obtained by adding or subtracting shifted versions of two elements in $P_{atset}$. Insert the removed elements into $P_{atset}$.
3. remove all coefficients in $C_{set}$ whose MSD representation can be obtained by adding or subtracting shifted versions of three elements in $P_{atset}$. Insert the removed elements into $P_{atset}$. If no element was removed from $C_{set}$ in the previous steps, go to Step 4. Otherwise, go to Step 1.
4. check which of the partial terms obtained by adding or subtracting shifted versions of two elements in $P_{atset}$ maximally matches a subset of bits of an MSD representation. Register the combination as a new partial term in $P_{atset}$ and insert a new element obtained by removing the new partial term from the selected MSD representation into $C_{set}$. Go to Step 1.

This loop is repeated until there are no more coefficients in $C_{set}$.

In this algorithm, all pairwise combinations are valid. In the case of [13], the algorithm does not consider a combination of shifted elements of $P_{atset}$ with non-zero bits in the same position.
Figure 14 presents an example of shifting and combination of two elements of the Patset. This combination in particular is performed in Step 2 of the algorithm. Figure 14(a) shows the behavior of the algorithm of [13]. When the first combination is performed, there are no conflicts between the elements. Thus, the addition and subtraction of the elements are obtained. When the first element is shifted left, there is a conflict between elements, where the second least significant bits of each element are equal to 1 simultaneously. Thus, the results from the addition and subtraction are not considered. Figure 14(b) shows the behavior under the algorithm described above. The conflict between elements is not a deterrent and we are able to consider two new subexpressions with these combinations.

### 7.2 Maximum Sharing under General Representation

Although the bit-level representation is convenient for pattern matching, it does pose a limitation in that not all partial terms are considered for sharing. We make the observation that the minimum area solution, in general, is not obtained using all coefficient representations with the minimum number of non-zero digits. In fact, we claim that the representation used for the coefficients is irrelevant and during the optimization process numerical values of the coefficients should be used. While it is true that there is a higher probability of a representation with a minimal number of non-zero digits being selected for the optimized solution, it is also true that there are situations where a non-minimal representation may fit better with existing partial terms and lead to a better solution.

For example, the coefficient 29 is represented in MSD as 100101 and 100011. Suppose that at some stage of the search process we have the partial terms 100011 and 101 (both valid MSD representations, for values 35 and 3, respectively). Existing methods will not be able to use these terms as no combination of them yields a valid MSD pattern for 29. Yet, we can use a subtracter to obtain 35 − 3 = 29, which represents the coefficient value. Hence, we can implement 29 with a single operator, whereas that was not possible with just MSD.
two elements from Patset
0 1 1 0 1 0
0 1 1 0 1 1 addition
0 0 0 0 1 0 shifting the first element
0 1 1 0 1 0
0 1 1 0 0 0 subtraction
0 1 1 0 1 0
0 1 1 0 0 0 subtraction

(a)

two elements from Patset
5
26
31 result from addition
-21 result from subtraction

(b)

Fig. 15. Example of shift and combination of elements: (a) using a bit-level representation; (b) using general number representation.

By using numerical values for the coefficients and partial terms we increase the search space significantly, allowing our algorithm to be more effective in area optimization.

The algorithm for the general representation follows the same steps presented for the MSD representation. However, we go one step further to note that we really don’t care about the bit patterns, we are only concerned with the numerical value. Hence, we don’t need to store all the representations of a given value, only its numerical value. During the search process, instead of combining and matching bit patterns, what we do is to add or subtract shifted versions of already found partial terms, and compare the numerical value with the value of the coefficients we need to implement. The algorithm for maximum sharing under a general number representation also uses the two sets, \(C_{set}\) and \(P_{set}\), with similar meaning as before, and is implemented with the following steps:

1. remove all coefficients in \(C_{set}\) that have the same decimal representation as a shifted value of an element in \(P_{set}\).
2. remove all coefficients in \(C_{set}\) whose decimal representation can be obtained by adding or subtracting shifted versions of two elements in \(P_{set}\). Insert the removed elements into \(P_{set}\).
3. remove all coefficients in \(C_{set}\) whose decimal representation can be obtained by adding or subtracting shifted versions of three elements in \(P_{set}\). Insert the removed elements into \(P_{set}\). If no element was removed from \(C_{set}\) in the previous steps, go to Step 4. Otherwise, go to Step 1.
4. check which of the partial terms obtained by adding or subtracting shifted versions of two elements in \(P_{set}\) maximally matches a subset of bits of a decimal representation. Register the combination as a new partial term in \(P_{set}\) and insert a new element obtained by removing the new partial term from the selected decimal representation into \(C_{set}\). Go to Step 1.

This loop is repeated until there are no more coefficients in \(C_{set}\).

Figure 15 presents an example of shifting and combination of two elements of the \(P_{set}\). Figure 15(a) shows the behavior of the algorithm explained in the
previous section. Figure 15(b) shows the behavior of the algorithm under the general representation. As should be observed, we are not concerned with the conflict between elements, since the elements are represented in decimal form. Thus, similar to the algorithm designed for the MSD representation, we are also able to consider new subexpressions with these combinations.

8 Experimental Results

In this section, we present results obtained with the exact CSE algorithms for the unconstrained maximum sharing problem, as well as the same problem under a delay constraint and also results obtained with the heuristic algorithms presented in this chapter. In the algorithms designed for maximum sharing problem under a delay constraint, we set the user_delay to the min_delay, i.e., we find the minimum area under minimum delay solutions.

As the first experiment set, we used randomly generated instances where constants are defined in 12 bit-width. The number of constants ranges between 10 and 100, and for each of them we generated 30 instances. On this benchmark, we compare the exact CSE algorithms with several previously proposed CSE heuristics, namely with the heuristics of [8, 13, 28], which we have also implemented.

Initially, we compare the effect of different number representations, i.e., binary, CSD, and MSD, on the minimum number of operations and delay solutions. The results of the exact algorithms on unconstrained maximum sharing problem and maximum sharing problem under a delay constraint are given in Figures 16 and 17, respectively.

We can observe that these three representations yield about the same solutions for instances with few constants. For instances with larger number of constants, the CSD representation achieves worse solutions than the binary and MSD representations, requiring more than 2 additional operations on average. Binary and MSD representations yield very similar results, with the binary performing better as the number of constants increases. This demonstrates that having a third digit, the signed digit, while desirable in representing one or a few constants, creates a more varied set of patterns that limits the amount of sharing for a larger number of constants. This is partially overcome by the redundancy in the MSD representation.

We compare the minimum delay solutions achievable with different number representations for the maximum sharing problem under a delay constraint in Figure 18. We observe that the CSD and MSD representations provide solutions with at most three operations in series, while the binary representation on average requires more operations in series, and this number increases with the number of constants. Hence, the minimum delay solutions presented in Figure 17, while similar in area, have a much smaller delay in the cases of the CSD and MSD.

We also compare the exact solutions with the heuristics, [8, 13, 28], for unconstrained maximum sharing problem and with the heuristic of [28] for the
Fig. 16. Comparison of the number representations on the unconstrained maximum sharing problem.

Fig. 17. Comparison of number representations on the maximum sharing problem under a delay constraint.
maximum sharing problem under a delay constraint on randomly generated instances where constants are represented in CSD. The results are given in Figures 19 and 20.

In this experiment, we observe that for the unconstrained maximum sharing problem, while the average number of operations between the heuristic of [28] and the exact algorithm is almost 1 on all instances, the average number of operations between the heuristic of [13] and the exact algorithm reaches up to 7.4 operations. Also, since the heuristic of [8] is a greedy algorithm that finds the most common subexpression in each iteration of the algorithm, it is easily trapped to the local minima on instances that include more than 40 constants. On the instances with 100 constants, the average number of operations between this heuristic and the exact algorithm is almost 10 operations. For the maximum sharing problem under a delay constraint, the heuristic of [28] finds solutions with almost 2 additional operations on average compared to the exact solutions. This clearly shows that exact algorithms find better solutions than the heuristic algorithms and among the heuristics, the heuristic of [28] finds much better solutions than the heuristics of [8] and [13].

As the second experiment set, we used FIR filter instances where filter coefficients were computed with the \texttt{remez} algorithm in \texttt{matlab}. The specifications of filters are presented in Table 2 where: \textit{pass} and \textit{stop} are normalized frequencies that define the passband and stopband respectively; \#\texttt{tap} is the number of coefficients; and \textit{width} is the bit-width of the coefficients.

We compare the exact CSE algorithm and heuristic algorithms presented in this chapter on unconstrained maximum sharing problem in Table 3. In this
Fig. 19. Comparison of the exact and heuristic algorithms for the unconstrained maximum sharing problem.

Fig. 20. Comparison of the exact and heuristic algorithms for the maximum sharing problem under a delay constraint.
Table 2. FIR filter specifications.

<table>
<thead>
<tr>
<th>Filter</th>
<th>pass</th>
<th>stop</th>
<th>#tap</th>
<th>width</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.20</td>
<td>0.25</td>
<td>120</td>
<td>8</td>
</tr>
<tr>
<td>2</td>
<td>0.10</td>
<td>0.25</td>
<td>100</td>
<td>10</td>
</tr>
<tr>
<td>3</td>
<td>0.15</td>
<td>0.25</td>
<td>40</td>
<td>12</td>
</tr>
<tr>
<td>4</td>
<td>0.20</td>
<td>0.25</td>
<td>80</td>
<td>12</td>
</tr>
<tr>
<td>5</td>
<td>0.24</td>
<td>0.25</td>
<td>120</td>
<td>12</td>
</tr>
<tr>
<td>6</td>
<td>0.15</td>
<td>0.25</td>
<td>60</td>
<td>14</td>
</tr>
<tr>
<td>7</td>
<td>0.15</td>
<td>0.20</td>
<td>60</td>
<td>14</td>
</tr>
<tr>
<td>8</td>
<td>0.10</td>
<td>0.15</td>
<td>60</td>
<td>14</td>
</tr>
<tr>
<td>9</td>
<td>0.10</td>
<td>0.15</td>
<td>100</td>
<td>16</td>
</tr>
</tbody>
</table>

Table 3. Comparison of algorithms on FIR filter instances for unconstrained maximum sharing problem.

<table>
<thead>
<tr>
<th>Filter</th>
<th>Exact CSE Algorithm</th>
<th>Heuristic Algorithms</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Binary</td>
<td>CSD</td>
</tr>
<tr>
<td></td>
<td>adder</td>
<td>step</td>
</tr>
<tr>
<td>1</td>
<td>10</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>18</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>17</td>
<td>5</td>
</tr>
<tr>
<td>4</td>
<td>29</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>35</td>
<td>4</td>
</tr>
<tr>
<td>6</td>
<td>23</td>
<td>4</td>
</tr>
<tr>
<td>7</td>
<td>32</td>
<td>5</td>
</tr>
<tr>
<td>8</td>
<td>34</td>
<td>4</td>
</tr>
<tr>
<td>9</td>
<td>51</td>
<td>5</td>
</tr>
</tbody>
</table>

Table, adder denotes the number of operations and step denotes the maximum number of operations in series that implements the MCM.

In this experiment, we observe that MSD representation yields better solutions than that of binary and CSD on the exact algorithm. However, the heuristic algorithm that augments the MSD representations of the coefficients increasing the possibilities of partial term sharing obtains better results than the exact CSE algorithm under MSD, such as on filters 2, 7, 8, and 9. Also, the heuristic algorithm under general number representation finds superior results than other algorithms, since it is not limited to any number representation.

9 Conclusions

We have described an exact algorithm that computes the minimum number of adder/subtractor modules in the implementation of MCM structures by maximizing the sharing of common subexpressions. The algorithm can handle binary, CSD, and MSD representations for the coefficients. Delay constraints can be included in the model so that a user-specified delay can be accommodated. We presented results on FIR filter instances where we demonstrate that the exact algorithm can be applied to real-sized problems.
Since the proposed algorithm is exact, the results allow for some interesting conclusions. Contrary to intuition, and common practice, we demonstrated that the binary representation allows for a greater amount of sharing, hence producing more area-efficient implementations for MCM problems than the CSD and MSD representations. However, when seeking minimum delay solutions, the MSD representation should be used.

We have also shown that the minimum area solution cannot be obtained when coefficients are defined under a number representation. The number of possible implementations of coefficients can be increased by using numerical values for the coefficients allowing an algorithm to be more effective in area optimization. However, the exact CSE algorithm described in this chapter can be extended to handle general number representation of coefficients by using the techniques described in [16, 29].

As future work, we are currently working on the implementation of an exact graph-based algorithm.

References


