İSTANBUL TECHNICAL UNIVERSITY ★ INSTITUTE OF SCIENCE AND TECHNOLOGY

OPTIMIZATION ALGORITHMS FOR THE MULTIPLE CONSTANT MULTIPLICATIONS PROBLEM

Ph.D. Thesis by
Levent AKSOY

Department : **Electronics and Communication Engineering**

Programme : **Electronics Engineering**

MARCH 2009

İSTANBUL TECHNICAL UNIVERSITY ★ INSTITUTE OF SCIENCE AND TECHNOLOGY

OPTIMIZATION ALGORITHMS FOR THE MULTIPLE CONSTANT MULTIPLICATIONS PROBLEM

Ph.D. Thesis by
Levent AKSOY
(504032202)

Date of submission : **7 November 2008**
Date of defence examination : **11 March 2009**

Supervisor (Chairman) : **Prof. Dr. Ece Olcay GÜNEŞ**
Members of the Examining Committee : **Assis. Prof. Dr. Ahmet ONAT (SU)**
Assoc. Prof. Dr. Serdar ÖZOĞUZ (ITU)
Prof. Dr. Ertuğrul ÇELEBİ (ITU)
Assoc. Prof. Dr. Arda YURDAKUL (BU)

MARCH 2009

İSTANBUL TEKNİK ÜNİVERSİTESİ ★ FEN BİLİMLERİ ENSTİTÜSÜ

BİRDEN FAZLA KATSAYININ ÇARPIMI PROBLEMİ İÇİN OPTİMİZASYON ALGORİTMALARI

DOKTORA TEZİ
Levent AKSOY
(504032202)

Tezin Enstitüye Verildiği Tarih : **7 Kasım 2008**
Tezin Savunulduğu Tarih : **11 Mart 2009**

Tez Danışmanı : **Prof. Dr. Ece Olcay GÜNEŞ**
Diğer Jüri Üyeleri : **Yrd. Doç. Dr. Ahmet ONAT (SÜ)**
Doç. Dr. Serdar ÖZOĞUZ (İTÜ)
Prof. Dr. Ertuğrul ÇELEBİ (İTÜ)
Doç. Dr. Arda YURDAKUL (BÜ)

MART 2009

FOREWORD

When I look back to three years in my Ph.D. research, I see many people whom I would like to express my deepest gratitude for their support, contributions, friendship, encouragement, and wisdom.

First of all, to my family, to my parents and sisters for their support and encouragement during all these years, and to my little nephew and niece for adding joy and happiness into the exhaustive days.

To my advisor, Prof. Dr. Ece Olcay Güneş, for her constant support, dedication, suggestions, and reviews, without whom, this thesis simply would not exist.

Also, to Prof. Dr. Ahmet Onat and Prof. Dr. Serdar Özoğuz for their effort and useful suggestions through my Ph.D. study.

To Prof. José Monteiro who invited me to work as a visiting researcher at ALGOS research unit in INESC-ID where the ideas in this thesis were born. His motivation, constitutive suggestions, inspiring thoughts, fruitful comments, and detailed reviews are always acknowledged.

To Prof. Paulo Flores whom I had a privilege and pleasure to work with. I appreciate his great contributions throughout this work, encouraging ideas, helpful comments, and so many useful suggestions.

Also, to Prof. Eduardo Costa for the pleasant conversations we had, his forthcoming suggestions, and above all, for his friendship.

To Anup Hosangadi, Yevgen Voronenko, Vasco Manquinho, Hossein Sheini, Prof. Oscar Gustafsson, and Prof. Andrew Dempster for providing me their algorithms or the results of their algorithms given in this work, and for fruitful discussions we made on their and our algorithms.

To Prof. Luis Silveira, Jorge Villena, Ana Jesus, Teresa, James, Sandy, Cathy, and many others in Lisbon who made my stay enjoyable, comfortable, and also, exciting.

To all my colleagues and friends in Istanbul for their assistance and friendship.

At last, but not least, to my previous professors, Prof. Dr. Ertuğrul Eriş and Prof. Dr. Ahmet Dervişoğlu, who have great influences on me and my research all over these years.

Without any of them, I think this thesis would not be as it is.

November 2008 Levent AKSOY

TABLE OF CONTENTS

Page

ABBREVATIONS

BCP : Binate Covering Problem
BHM : Bull-Horrocks Modified
CNF : Conjunctive Normal Form
CSA : Carry Save Adder
CSE : Common Subexpression Elimination
CSD : Canonical Signed Digit
DAG : Directed Acyclic Graph
DLL : Davis Logemann Loveland
DSP : Digital Signal Processing
EDA : Electronic Design Automation
FA : Full Adder
FFT : Fast Fourier Transform
FIR : Finite Impulse Response
HA : Half Adder
ILP : Integer Linear Programming
MCM : Multiple Constant Multiplications
MSD : Minimum Signed Digit
NP : Nondeterministic Polynomial time
P : Polynomial time
PB : Pseudo Boolean
PC : Personal Computer
RCA : Ripple Carry Adder
SAT : Satisfiability
SCM : Single Constant Multiplication
VMA : Vector Merging Adder

LIST OF TABLES

LIST OF FIGURES

OPTIMIZATION ALGORITHMS FOR THE MULTIPLE CONSTANT MULTIPLICATIONS PROBLEM

SUMMARY

The Multiple Constant Multiplications (MCM), i.e, the multiplication of a variable by a set of constants, has been a central operation and performance bottleneck in many digital signal processing applications such as, video processing, digital television, data transmission, and wireless communications. Since the design of multiplications is expensive in terms of area, delay, and power consumption in hardware and the values of the constants are known beforehand, the area-delay optimization of the MCM operation has often been accomplished by using the shift-adds architecture.

The last decade has seen much progress in the design of efficient algorithms for the MCM problem, i.e., the implementation of the MCM operation using the fewest number of addition and subtraction operations. The design of efficient algorithms for the MCM problem has also provided motivations to design the MCM operation by taking into account the area, delay, and power consumption objectives, that are the most important and crucial parameters in the design of hardware implementations and directly influence the performance of the implementation.

However, since the MCM problem is a Nondeterministic Polynomial time (NP)-complete problem, the previously proposed exact algorithms have high computational complexity. As finding the exact optimal solution is intractable, almost all existing algorithms are heuristics in nature and the obtained solutions are highly possibly suboptimal due to the local minima.

On the other hand, recent impressive speed-ups of solvers for Boolean satisfiability (SAT) enabled their adaptations to solve Boolean optimization problems that were traditionally handled as instances of 0-1 Integer Linear Programming (ILP) and their applications to new optimization problems in electronic design automation.

In this thesis, the MCM problem and its variants are modeled as 0-1 ILP problems and the exact solutions are found using 0-1 ILP solvers equipped with recent improvements in both areas, SAT and ILP. Also, the problem reduction and model simplification techniques that significantly reduce the size of the 0-1 ILP problem, consequently, increase the performance of the 0-1 ILP solvers, enabling the applications of the exact algorithms to larger size instances are introduced.

Due to the NP-completeness of the MCM problem, naturally, there are more complex instances that the exact algorithms cannot handle. Hence, in this thesis, approximate algorithms that find competitive results with the minimum solutions and obtain better solutions than those of the previously proposed heuristics are also introduced.

BİRDEN FAZLA KATSAYININ ÇARPIMI PROBLEMİ İÇİN OPTİMİZASYON ALGORİTMALARI

ÖZET

Birden fazla katsayının çarpımı (MCM), bir başka deyişle, bir küme içindeki katsayıların bir değişken ile çarpımı, video işleme, sayısal televizyon, bilgi aktarımı ve kablosuz haberleşme gibi birçok sayısal sinyal işleme uygulamalarında performansı etkileyen merkezi bir işlem olmuştur. Donanım içinde çarpma işlemleri alan, gecikme ve güç tüketimi açısından maliyetli olduklarından ve katsayıların değerleri daha önceden bilindiğinden dolayı MCM işleminin alan-gecikme optimizasyonu genellikle ötele-topla mimarisi kullanılarak sağlanmıştır.

Son on yıl, MCM problemi, bir başka deyişle, MCM işleminin en az sayıda toplama ve çıkarma işlemleri kullanılarak gerçeklenmesi, için etkili algoritmaların tasarımındaki oldukça büyük gelişmelere tanıklık etmiştir. MCM problemi için etkili algoritmaların tasarımı, MCM işleminin, donanım tasarımında oldukça önemli ve vazgeçilmez ve aynı zamanda tasarımın başarımını doğrudan etkileyen alan, gecikme ve güç tüketimi ölçütleri de dikkate alınarak tasarlanmasına imkan sağlamıştır.

Yine de, MCM problemi bir belirleyici olmayan polinom (NP)-bütün problem olduğundan dolayı daha önceden önerilen kesin algoritmalar yüksek hesaplama karmaşıklığına sahiptirler. Kesin en iyi sonucu bulmak oldukça zor olduğundan, varolan bütün algoritmaların çoğu sezgisel algoritmalardır ve elde edilen sonuçlar arama uzayı içindeki yerel minimum noktalarının varlığından dolayı büyük bir olasılıkla minimum sonuçlar değildir.

Bunun yanında, Boolean sağlanabilirlik (SAT) problemi için önerilen çözücülerin yakın zamanlardaki etkileyici başarımları daha önceden 0-1 tamsayı doğrusal programlama (ILP) örnekleri olarak ele alınan Boolean optimizasyon problemlerini çözmek için uyarlanmalarına ve elektronik tasarım otomasyonu içinde yeni uygulamaların ele alınmasına olanak sağlamıştır.

Bu tezde, MCM problemi ve onun değişik biçimleri 0-1 ILP problemleri olarak modellenmekte ve kesin sonuçlar SAT ve ILP alanındaki yeni gelişmeler ile donatılmış 0-1 ILP çözücüler kullanılarak bulunmaktadır. Bunun yanında, 0-1 ILP problem boyutunu azaltan, böylelikle 0-1 ILP çözücülerin başarımını arttıran ve kesin algoritmaların geniş boyutlu örneklere uygulanmasına olanak sağlayan problem indirgeme ve model basitleştirme teknikleri sunulmaktadır.

MCM probleminin bir NP-bütün problem olmasından dolayı, doğal olarak kesin algoritmaların ele alamayacakları çok daha karmaşık örnekler bulunmaktadır. Bundan dolayı, bu tez içinde minimum sonuçlar ile rekabet edecek sonuçlar elde edebilen ve daha önceden önerilmiş sezgisel yöntemlerden daha iyi sonuçlar bulabilen yaklaşık algoritmalar sunulmaktadır.

1. INTRODUCTION

1.1 Motivation and Objectives

In several computationally intensive operations, such as Finite Impulse Response (FIR) filters as illustrated in Figure 1.1 and Fast Fourier Transforms (FFT), the same input is multiplied by a set of coefficients, an operation known as Multiple Constant Multiplications (MCM). These operations are typical in Digital Signal Processing (DSP) applications and hardwired dedicated architectures are the best option for maximum performance and minimum power consumption.

However, the design complexity of these applications is dominated by a large number of constant multiplications leading to excessive area, delay, and power consumption even if implemented in a full custom integrated circuit. Since the values of the constants are known beforehand, the constant multiplications can be designed using addition/subtraction and shifting operations in the shift-adds architecture [1]. When the same input is to be multiplied by a set of constant coefficients, significant reductions in hardware can also be obtained by sharing the partial products of the input among the set of multiplications. Since shifts are free in terms of hardware, the MCM problem is defined as finding the minimum number of addition/subtraction operations to implement the constant multiplications. The MCM problem has been proven to be NP-complete in [2].

In the last two decades, many efficient algorithms have been proposed for the optimization of the number of operations in MCM. These methods can be categorized

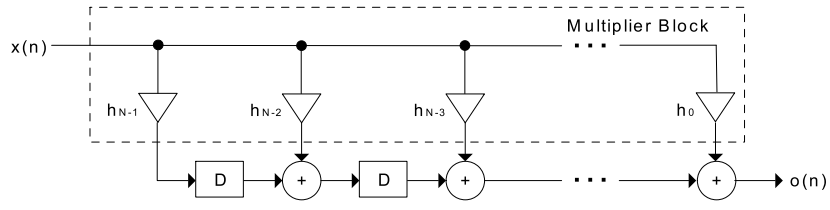


Figure 1.1: Transposed form of a hardwired FIR filter implementation.

in two classes: the Common Subexpression Elimination (CSE) and the graph-based algorithms. The CSE algorithms basically find common non-zero digit patterns on the representations of the constants. The exact CSE algorithms that formalize the MCM problem as a 0-1 Integer Linear Programming (ILP) problem and find the minimum number of operations solution of the MCM problem by maximizing the partial product sharing have been proposed in [3,4]. However, these exact algorithms are not equipped with the problem reduction and model simplification techniques that significantly reduce the 0-1 ILP problem size, consequently, the required time to find the minimum solution. Hence, the exact CSE algorithms can be applied on small size instances. On the other hand, the graph-based algorithms are not restricted to a particular number representation and synthesize the constants iteratively by building a graph. The previously proposed graph-based algorithms have been heuristics and provide no indication on how far from the minimum their solutions are. To the best of our knowledge, there is no exact graph-based algorithm designed for the MCM problem.

The primary objective of this thesis is to introduce exact CSE and graph-based algorithms that can be applied on real size instances of the MCM problem. However, due to the NP-completeness of the MCM problem, there are more complex instances that the exact algorithms find them difficult to obtain the minimum solutions. Hence, the primary objective of this thesis is also to propose approximate algorithms that find similar results with the exact algorithms using a little computational effort.

In many applications, performance is a critical parameter. Hence, circuit area is generally expandable in order to achieve a given performance target. As the delay is dependent on several implementation issues, such as circuit technology, placement, and routing, in the MCM problem the delay is generally considered as the maximum number of addition/subtraction operations in series to produce any constant multiplication [5]. Thus, CSE and graph-based algorithms [5–8] have been proposed to find the fewest number of operations solutions under a delay constraint in MCM. However, the previously proposed algorithms have been based on heuristics and may find suboptimal solutions that are far from the minimum number of operations solutions under a delay constraint.

In the synthesis of the constant multiplications at the gate-level, each addition/subtraction operation implementing a constant multiplication occupies different scale of area based on its architecture. To obtain the minimum area implementation of the MCM, the area cost of each operation should be also considered in the MCM problem. The previously proposed heuristic [9] relies on the ripple carry architecture of addition/subtraction operations including Half Adders (HAs) and Full Adders (FAs), and aims to find the smallest area solutions of the constant multiplications in terms of HAs and FAs. However, the area cost of each operation can be determined more precisely by taking into account specific cases and the minimum area solutions in terms of gate-level metrics can be obtained by modeling the minimization of area problem as a 0-1 ILP problem.

In the algorithms proposed for the MCM problem, an addition/subtraction operation is assumed to be a two-input operation that is generally implemented with Ripple Carry Adders (RCAs) increasing the delay of the computation. On the other hand, Carry-Save Adders (CSAs) are commonly used for high-speed implementation of multi-operand additions. Although there exist mapping techniques [10, 11] that convert addition/subtraction operations into high-speed operations using CSAs, they do not attempt to minimize the number of required CSAs. Also, the previously proposed CSE and graph-based algorithms [12, 13] designed for the optimization of the number of CSA blocks have been heuristics.

The secondary objective of this thesis is to introduce exact CSE algorithms for the minimization of the number of operations under a delay constraint, the minimization of area, and the minimization of the number of CSA blocks.

1.2 Original Contributions

The original contributions of this thesis are given as follows:

- **An alternative exact CSE model for the MCM problem -** In this thesis, the problem reduction and simplification techniques for the exact CSE algorithm of [4] that enables the exact algorithm to be applied on large size instances [14] are introduced. Also, for the MCM problem, an alternative exact CSE model [15] that considers the minimization of the number of operations rather than maximizing

the partial product sharing as considered in [4] is proposed. This model allows the exact CSE algorithm to be applied on more sophisticated optimization problems. Also, an approximate CSE algorithm [16] that can be applied on more complex instances is presented. Furthermore, an exact algorithm [17] that can handle the constants under general number representation and obtains better solutions than those of the exact CSE algorithm [4] is introduced.

- **An exact graph-based algorithm for the MCM problem -** Although the exact CSE algorithms proposed for the MCM problem give good results, their solutions depend on the number representation. Hence, the exact CSE algorithms cannot guarantee their solutions as the minimum solutions when the constant multiplications are not restricted to any particular number representation. In this thesis, an exact graph-based algorithm [18] that finds the minimum number of operations solution of the MCM problem is introduced. Although the proposed exact algorithm is based on a breadth-first search and can be applied on less complex instances, it can handle real size instances in a reasonable time and may find better solutions than those of the prominent graph-based heuristics. Also, an approximate algorithm [19] based on the exact algorithm that finds competitive and better solutions than efficient graph-based heuristics on large size instances is introduced.

- **An exact CSE algorithm for the optimization of the number of operations under a delay constraint in MCM -** In this thesis, the exact CSE algorithm designed for the MCM problem is extended to find the minimum number of operations solution under a delay constraint by using the alternative exact model [20]. In this algorithm, delay constraints are also added to the 0-1 ILP problem so that the minimum number of operations solution does not violate the delay constraint. Also, an approximate CSE algorithm [16] that finds better solutions than the CSE heuristics and competitive results with the exact CSE algorithm is introduced.

- **An exact CSE algorithm for the optimization of area in terms of gate-level metrics in MCM -** In this thesis, addition and subtraction architectures for the constant multiplications based on HAs, FAs, and additional logic gates under

signed and unsigned input are introduced. In the exact CSE algorithm [21], the area cost of each operation is determined by the data given in the design library and the minimum area solutions of constant multiplications are found by using the alternative exact model.

- **Exact and approximate algorithms for the optimization of the number of CSA blocks in MCM -** In this thesis, an exact CSE algorithm designed for the minimization of the number of CSA blocks is presented. Also, an approximate CSE algorithm that can deal with large size instances is introduced. Furthermore, the approximate CSE algorithm is extended to handle the constants under general number representation [22].

1.3 Thesis Organization

The rest of this thesis is organized as follows. Chapter 2 gives the background concepts related with the optimization algorithms designed for the MCM problem. In Chapter 3, initially, we introduce the single constant multiplication (SCM) problem and give an overview of the algorithms designed for the SCM problem. Then, we define the MCM problem and describe the algorithms proposed for the MCM problem. Chapter 4 presents the exact and approximate algorithms designed for the MCM problem. This chapter starts with the introduction of the exact and approximate CSE algorithms. Then, it is followed by the presentation of the exact algorithm that can handle the constants under general number representation. Finally, this chapter ends with the introduction of the exact and approximate graph-based algorithms. In the following three chapters, the exact and approximate CSE algorithms designed for the optimization of area and delay in MCM are introduced. Chapter 5 describes the exact and approximate CSE algorithms designed for the minimization of the number of operations under a delay constraint. In Chapter 6, the exact CSE algorithm designed for the minimization of area of the MCM implementation in terms of gate-level metrics is introduced. Chapter 7 describes the exact and approximate algorithms designed for the minimization of the number of CSA blocks in MCM. Finally, discussions on the proposed algorithms, conclusions, and directions for future work are given in Chapter 8.

2. BACKGROUND

This chapter starts with the description of the number representations and the number representation conversion algorithms. It is followed by the introduction of basic definitions on complexity classes. Also, the Satisfiability (SAT) problem is presented and a generic backtrack search SAT algorithm is described. Then, the 0-1 Integer Linear Programming (ILP) problem is defined. Finally, this chapter ends with the overview of pseudo-Boolean (PB) optimization algorithms.

2.1 Number Representations

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

$$k = \sum_{i=0}^{n-1} c_i 2^i \tag{2.1}$$

where $c_i \in \{1, 0, -1\}$. Hereafter, the digit -1 will be denoted by $\overline{1}$. Observe that the binary signed digit system is a redundant number system, for example, both 0101 and $10\overline{1}\,\overline{1}$ correspond the integer value 5.

The *Canonical Signed Digit* (CSD) representation [23] is a signed digit system that has a unique representation for each integer and verifies two main properties: (i) the number of non-zero digits is minimal, (ii) two non-zero digits are not adjacent. Any n digit number in CSD format has at most $\lceil (n+1)/2 \rceil$ non-zero digits. On average, the number of non-zero digits is reduced by 33% when compared with the binary representation [24]. This representation is widely used in multiplierless implementations of constant multiplications, because it reduces the hardware requirements due to the minimum number of non-zero digits.

Algorithm 2.1 Binary to CSD conversion algorithm. The algorithm takes the binary representation of the constant, b, including n digits and returns the CSD representation of the constant, c, using the conversion table.

Binary2CSD(b, n)

1: $b_n = 0$
2: $b_{n+1} = 0$
3: $state = 0$
4: **for** $i = 0$ to n **do**
5: c_i = get_value_from_table($state$, b_{i+1}, b_i)
6: $state$ = get_next_state_from_table($state$, b_{i+1}, b_i)
7: **return** c

Conversion Table				
Inputs			Outputs	
$state$	b_{i+1}	b_i	c_i	$next\_state$
0	0	0	0	0
0	0	1	1	0
0	1	0	0	0
0	1	1	$\overline{1}$	1
1	0	0	1	0
1	0	1	0	1
1	1	0	$\overline{1}$	1
1	1	1	0	1

There are several techniques to find the CSD representation of a constant. The method described in [25], given in Algorithm 2.1, initially, obtains the binary representation of the constant and then, starts replacing all the sequences found as "01...11" by the sequence "10...0$\overline{1}$" with the same number of digits, while traversing on the digits of the binary representation from the least significant digit to the most significant digit, i.e., from right to left. This procedure uses a conversion table and a state variable to detect the 1s sequences. The method of [26] finds the CSD representation of a constant by traversing in both directions. Also, an efficient method presented in [27] avoids the need to represent the constant in binary and uses the Hamming weight pyramid to find the CSD representation of the constant.

The *Minimal Signed Digit* (MSD) representation [28] 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. The MSD representations of a constant can be computed from its CSD representation by replacing all possible combinations of the sequences "10$\overline{1}$" and "$\overline{1}$01" by the sequences "011" and "0$\overline{1}\,\overline{1}$" respectively by traversing on the digits of the CSD representation from left to right. For each replacement, a new MSD representation is obtained, since the number of non-zero digits is not increased. Algorithm 2.2 presents the procedure described in [28] that computes the MSD representations of the constant from its CSD representation.

As an example, suppose the constant 23 defined in six bits. The representation of 23 in binary, i.e., 010111, includes 4 non-zero digits. The constant is represented as

Algorithm 2.2 CSD to MSD conversion algorithm. The algorithm takes the CSD representation of the constant, c, including n digits and returns the set of MSD representation(s) of the constant, S, including m elements.

P_i : the digit position of the i. MSD representation of the constant in S
CSD2MSD(c, n)

1: $i = 1, m = 1$
2: $S_1 = \{c\}$
3: $P_1 = n - 1$
4: **while** 1 **do**
5: **while** $P_i \geq 2$ **do**
6: **if** $S_i[P_i,\ P_i - 1,\ P_i - 2] = 10\bar{1}$ **then**
7: $m = m + 1, S_m = S_i$
8: S_m = replace_three_digits(P_i, S_m, "011")
9: $P_i = P_i - 2,\ P_m = P_i - 2$
10: **else if** $S_i[P_i,\ P_i - 1,\ P_i - 2] = \bar{1}01$ **then**
11: $m = m + 1, S_m = S_i$
12: S_m = replace_three_digits(P_i, S_m, "0$\bar{1}\,\bar{1}$")
13: $P_i = P_i - 2,\ P_m = P_i - 2$
14: **else**
15: $P_i = P_i - 1$
16: $i = i + 1$
17: **if** $i > m$ **then**
18: **return** S

$10\bar{1}00\bar{1}$ in CSD and both $10\bar{1}00\bar{1}$ and $01100\bar{1}$ denote 23 in MSD with the minimum number of non-zero digits, i.e., 3.

2.2 Complexity Classes

The complexity of a process or an algorithm is a measure of how difficult it is to perform. The study of the complexity of algorithms, also known as complexity theory, deals with the resources required during the computation to solve a given problem. The most common resources are time, i.e., how many steps does it take to solve a problem, and space, i.e., how much memory does it take to solve a problem. The time complexity of a problem, generally determined as a function of the size of the input, is the number of steps taken to solve an instance of the problem using the most efficient algorithm. Table 2.1 compares the CPU time required for solving instances with different time complexity.

To generalize the time complexity of a problem, since the number of computer instructions depends on what machine or language is used, the *Big O* notation is

Table 2.1: Time complexity of problems with different functions.

n	$f(n) = n$	$f(n) = n^2$	$f(n) = 2^n$	$f(n) = n!$
10	0.01 μs	0.1 μs	1 μs	3.63 ms
20	0.02 μs	0.4 μs	1 ms	77.1 years
30	0.03 μs	0.9 μs	1 s	$8.4 * 10^{15}$ years
40	0.04 μs	1.6 μs	18.3 minutes	
50	0.05 μs	2.5 μs	13 days	
100	0.1 μs	10 μs	$4 * 10^{13}$ years	
1000	1 μs	1 ms		

used. For example, if a problem has time complexity $O(n^2)$ on one typical computer, then it will also have complexity $O(n^2)$ on most other computers.

A *decision problem* is a problem where the answer is always *yes* or *no*. As an example, for the problem *is-prime*, an integer is given and the answer indicates whether it is a prime number or not. Decision problems are important, because an arbitrary problem can always be reduced to a decision problem.

Decision problems fall into sets of comparable complexity, called complexity classes. The most well-known complexity classes are *Polynomial time* (P) and *Nondeterministic Polynomial time* (NP). The *complexity class P* is the set of decision problems that can be solved by a deterministic machine with a number of steps bounded by a power of the problem's size. This class of problems can be effectively solved even in the worst cases. On the other hand, the *complexity class NP* is the set of decision problems where a nondeterministic solution can be verified with the number of steps bounded by a power of the problem's size.

The class of P-problems is a subset of the class of NP-problems. The question of whether P is the same set as NP is the most important open question in theoretical computer science, i.e., one of the 7 Millennium Prize Problems[1]. Observe that if P and NP are not equivalent, then finding a solution for NP-problems requires an exhaustive search in the worst case.

The question of whether P = NP motivates the concepts of hard and complete. A set of problems X is *hard* for a set of problems Y if every problem instance in Y can be transformed in polynomial time into some problem instance in X with the same answer. A problem is said to be *NP-hard* if an algorithm for solving it can be

[1] http://www.claymath.org/millennium/

translated into one for solving any other problem in the NP complexity class. A set of problems X is *complete* for a set of problems Y if every problem instance in X is hard for a problem instance in Y, and X is also a subset of Y. Thus, an *NP-complete* problem is both NP-hard, i.e., any other problem in the NP complexity class can be easily translated into this problem, and NP, i.e., a nondeterministic solution is verifiable in polynomial time.

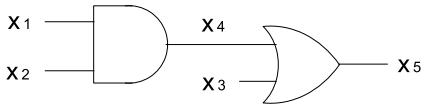
2.3 Boolean Satisfiability

2.3.1 Preliminaries

A *propositional formula* denotes a Boolean function $f : \{0,1\}^n \rightarrow \{0,1\}$. A *Conjunctive Normal Form* (CNF) is a representation of a propositional formula φ consisting of a conjunction of propositional clauses where each *clause* ω is a disjunction of literals, and a *literal* l_j is either a variable x_j or its complement $\overline{x_j}$. Observe that if a literal of a clause assumes value 1, then the clause is satisfied. If all literals of a clause assume value 0, then the clause is unsatisfied.

A *combinational circuit* is a directed acyclic graph (DAG) with nodes corresponding to logic gates and directed edges corresponding to wires connecting the gates. Incoming edges of a node are called *fanins* and outgoing edges are called *fanouts*. The *primary inputs* of the network are the nodes without fanins. The *primary outputs* are the nodes without fanouts. The primary inputs and outputs define the external connections of the network.

The CNF formula of a combinational circuit is the conjunction of the CNF formulas of each gate output, where the CNF formula of each gate denotes the valid input-output assignments to the gate. The derivation of the CNF formulas of logic gates can be found in [29]. As a small example, consider the combinational circuit and its CNF formula given in Figure 2.1. In the formula given in Figure 2.1(b), the first three clauses represent the CNF formula of the AND gate, and the last three clauses denote the CNF formula of the OR gate. Observe from Figure 2.1 that the assignment $x_1 = x_3 = x_4 = x_5 = 0$ and $x_2 = 1$ makes the formula φ equal to 1 indicating a valid assignment. However, the assignment $x_1 = x_3 = x_4 = 0$ and $x_2 = x_5 = 1$ makes the last



(a)

$$\varphi = (x_1 + \overline{x_4}).(x_2 + \overline{x_4}).(\overline{x_1} + \overline{x_2} + x_4).(\overline{x_3} + x_5).(\overline{x_4} + x_5).(x_3 + x_4 + \overline{x_5})$$

(b)

Figure 2.1: (a) A combinational circuit; (b) its CNF formula.

clause of the formula equal to 0, consequently the formula φ, indicating the conflict between the values of the inputs and the output of the OR gate.

2.3.2 Satisfiability problem

The *satisfiability problem* is to find an assignment on n variables of the Boolean formula in CNF that evaluates the formula to 1 or to prove that the formula is equal to the constant 0. The time complexity of the SAT problem in the worst case is $O(2^n)$. The SAT problem is the first problem proven to be NP-complete by Stephen Cook [30].

Boolean SAT is intrinsic to many problems in Electronic Design Automation (EDA). Hence, SAT models and techniques have been applied to EDA problems, such as, circuit delay computation [31], test pattern generation [29], equivalence checking [32], fault diagnosis [33] among many other problems. Also, SAT plays a central role in solving instances of binate covering problems [34–36]. Moreover, SAT is a key issue in other domains including artificial intelligence and operations research [37].

2.3.3 Satisfiability algorithms

The proposed SAT algorithms can be categorized in two classes as incomplete and complete algorithms. The incomplete SAT algorithms based on local search methods [38, 39], simulated annealing technique [40], genetic algorithms [41], and the hybrid of these methods [42, 43] may find a satisfying solution if it exists, but cannot prove that the formula is unsatisfiable if there is no satisfying solution. On the other hand, the complete SAT algorithms can find a satisfying solution if it exists, or otherwise, prove that the formula is equal to constant 0.

Over the years, many efficient SAT algorithms based on the backtrack search algorithm [44], called DLL, have been proposed. The backtrack search algorithm

Algorithm 2.3 A generic backtrack search SAT algorithm. The algorithm takes the Boolean formula φ in CNF and returns a value, SATISFIABLE or UNSATISFIABLE.

```
SAT(φ)
 1: d = 0
 2: while Decide(φ,d) == DECISION do
 3:    if Deduce(φ,d) == CONFLICT then
 4:       β = Diagnose(φ,d)
 5:       if β = −1 then
 6:          return UNSATISFIABLE
 7:       else
 8:          Backtrack(φ,d,β)
 9:          β = d
10:    else
11:       d = d + 1
12: return SATISFIABLE
```

is implemented by a search process that implicitly enumerates the search space of 2^n possible binary assignments to the n variables. The pseudo-code for a generic DLL-based backtrack search algorithm is given in Algorithm 2.3.

Given an SAT problem, formulated as a CNF formula, φ, the *SAT* algorithm conducts a search through the space of all possible assignments to the n problem variables. At each stage of the search, a variable assignment is selected with the *Decide* function. A decision level d is then associated with each selection of an assignment. Implied assignments are identified with the *Deduce* function. Whenever a clause becomes unsatisfied, the *Deduce* function returns a *Conflict* indication which is then analyzed using the *Diagnose* function. The diagnosis of a given conflict returns a backtracking decision level, β, which denotes the decision level to which the search process is required to backtrack to. Afterwards, the *Backtrack* function clears all assignments, both decision and implied assignments, from the current decision level d through the backtrack decision level β. Furthermore, considering that the search process should resume at the backtrack level, the current decision level d becomes β. Finally, the current decision level d is incremented. This process is interrupted whenever the formula is found to be satisfiable or unsatisfiable. The formula is satisfied when all variables are assigned and therefore, all clauses must be satisfied. The formula is unsatisfied when the empty clause is derived, which is implicit when the *Diagnose* function returns -1 as the backtrack level [45].

Important improvements in the generic backtrack search SAT algorithm, such as non-chronological backtracking, conflict-based learning mechanisms, clause deletion policies, branching heuristics, and lazy data structures, have led to efficient SAT algorithms [46–48]. Recent SAT algorithms can handle and solve SAT instances with tens of thousands of variables and millions of clauses in a matter of seconds or minutes [49].

2.4 0-1 Integer Linear Programming

The *0-1 Integer Linear Programming* (ILP) problem is the minimization or the maximization of a linear cost function subject to a set of linear constraints and is generally defined as follows[2]:

$$\text{Minimize} \quad \mathbf{c}^T \cdot \mathbf{x} \tag{2.2}$$

$$\text{Subject to} \quad \mathbf{A} \cdot \mathbf{x} \geq \mathbf{b}, \quad \mathbf{x} \in \{0,1\}^n \tag{2.3}$$

In (2.2), c_j in $\mathbf{c}$ is an integer cost associated with each of the n variables x_j, $1 \leq j \leq n$, in the cost function, and in (2.3), $\mathbf{A} \cdot \mathbf{x} \geq \mathbf{b}$ denotes the set of m linear constraints where $\mathbf{b}, \mathbf{c} \in \mathbb{Z}^n$ and $\mathbf{A} \in \mathbb{Z}^m \times \mathbb{Z}^n$. These linear constraints are commonly referred to as *pseudo-Boolean* (PB) inequalities to distinguish them from those that admit unrestricted integer variables.

A clause to be satisfied in a Boolean CNF formula, $l_1 + \ldots + l_k$, $k \leq n$, can be interpreted as a linear inequality, $l_1 + \ldots + l_k \geq 1$, where $\overline{x_j}$ is represented by $1 - x_j$ as shown in [50]. These linear inequalities are the special cases of the PB constraints, where $a_{ij} \in \{-1, 0, 1\}$ and b_i is equal to 1 minus the total number of the complemented variables in its CNF formula, and are commonly referred to as CNF constraints. For instance, the set of clauses, $(x_1 + x_2 + x_3)$, $(\overline{x_2} + \overline{x_4})$, $(x_1 + \overline{x_3})$, has the equivalent linear inequalities given as follows:

$$\begin{aligned} x_1 + x_2 + x_3 &\geq 1, \\ -x_2 - x_4 &\geq -1, \\ x_1 - x_3 &\geq 0. \end{aligned} \tag{2.4}$$

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

On the other hand, PB constraints represent a natural generalization of CNF constraints and are more expressive than CNF constraints. Thus, a single PB constraint may in some cases correspond to an exponential number of CNF clauses [49]. The techniques used for the conversion of PB constraints to CNF clauses can be found in [51, 52]. For instance, the PB constraint,

$$3x_1 - 2x_2 + 4x_3 \geq 2, \tag{2.5}$$

where $x_1, x_2, x_3 \in \{0,1\}$, corresponds to the Boolean equality constraint,

$$x_1\overline{x_2} + x_3 = 1, \tag{2.6}$$

that can be written in CNF with two clauses as:

$$(x_1 + x_3).(\overline{x_2} + x_3) = 1. \tag{2.7}$$

There are special forms of the 0-1 ILP problem. For example, if every entry in the $m \times n$ matrix A is in the set {0,1} and $b_i = 1, 1 \leq i \leq m$, then the 0-1 ILP problem is an instance of the *unate covering problem*. Moreover, if the entries a_{ij} of **A** belong to {-1,0,1} and $b_i = 1 - |\{a_{ij} : a_{ij} = -1, 1 \leq j \leq n\}|$, then the 0-1 ILP problem is an instance of the *binate covering problem* (BCP).

Note that in a BCP, each constraint is a CNF constraint and can be interpreted as a propositional clause. Thus, there is an intimate relation between 0-1 ILP and binate covering problems. For every instance of 0-1 ILP problem, there is an instance of BCP with the same satisfying solutions and therefore with the optimum solutions, and vice versa. Given a problem instance, it is not clear a-priori which formulation is better. It is an interesting question to characterize the class of problems that can be better formulated and solved with one technique or the other [53].

2.5 Pseudo-Boolean Optimization Algorithms

In [50], Peter Barth first proposed an approach based on Boolean SAT techniques for solving 0-1 ILP problems that are generally referred to as PB optimization problems. This approach performs a linear search on the possible values of the cost function, starting from the highest, at each step requiring the next computed solution to have a cost lower than the most recently computed upper bound. Whenever a new solution

is found that satisfies all the constraints, the value of the cost function is recorded as the current lowest computed upper bound. If the resulting instance of SAT is unsatisfiable, then the solution to the instance of PB optimization problem is given by the last recorded solution. The algorithm of [52] follows the same approach of [50], but it converts the PB constraints to Boolean clauses efficiently and applies the SAT solver [48], i.e., equipped with the recent improvements in Boolean SAT, iteratively to find a minimal cost assignment. This SAT-based approach focuses primarily on finding solutions for the problem constraints. Therefore, for highly constrained problems these techniques are very effective. However, these algorithms find it difficult to deal with the information from the cost function.

Unlike the SAT-based approach, branch-and-bound algorithms [54, 55] have been proved to be very effective when the instances to be solved are not highly constrained, since they are able to prune the search tree earlier due to estimate of the value of the cost function. In branch-and-bound algorithms, upper bounds on the value of the cost function are identified for each solution to the constraints, and lower bounds on the value of the cost function are estimated considering the current set of variable assignments. The procedures used for lower bound estimation are the approximation of a maximum independent set of constraints [54, 56], linear-programming relaxations [55], and Lagrangian relaxations [57]. For a given PB optimization problem, let *ub* denote the upper bound on the value of the cost function. The search is pruned whenever the lower bound estimation is higher than or equal to *ub*. In this case, it is guaranteed that a better solution cannot be found with the current variable assignments and therefore, the search can be pruned. The algorithms of [54–56,58] designed for the binate covering problem and several integer programming solvers follow this approach.

The hybrid PB optimization algorithms that include efficient SAT and ILP techniques in their structures have been proposed in [59, 60]. The algorithm of [59] incorporates the most significant features from both approaches, namely, the lower bound estimation methods such as linear programming and Lagrangian relaxations, and the reduction techniques from branch-and-bound algorithms, and the search pruning techniques from SAT algorithms. The algorithm of [60] integrates logic-based reasoning and integer programming methods like the cutting plane technique to solve

PB optimization problems. It uses an efficient literal watching strategy and several learning techniques that take advantage of the pruning power of PB constraints while minimizing the overhead.

Although there are many efficient PB solvers [61], in this thesis, we worked with *bsolo* [59], *glpPB* [62], and *minisat+* [52], since they obtained better solutions than other solvers on our instances.[3]

[3]The results of PB solvers on the MCM problems, the MCM problems under a delay constraint, and the minimization of area problems described in this thesis can be reached from the web address, http://atlas.cc.itu.edu.tr/~aksoyl/bench.html. Also, more detailed results on the performance of PB solvers on a comprehensive set of benchmarks can be found at http://www.cril.univ-artois.fr/PB07/.

3. CONSTANT MULTIPLICATIONS

This chapter addresses the problem of efficiently multiplying the known constant(s) with a variable multiplierless, i.e., using the fewest number of addition/subtraction operations, and presents an overview of algorithms designed for the single and multiple constant multiplication problems.

We note that in these problems, the complexity of an adder and a subtracter is assumed to be equal in hardware. It is also assumed that the sign of the constant can be adjusted at some part of the design and the shifting operation has no cost, since shifts can be implemented with only wires in hardware. Hence, the algorithms designed for the single and multiple constant multiplication problems generally focus on the minimization of the number of addition/subtraction operations. However, we note that the structures of these algorithms enable their adaptations to handle the objectives that also take into account the different complexities of an adder and a subtracter, and also, the number of shifts.

3.1 Single Constant Multiplication

The multiplication of a variable by a single known target constant, i.e., t_1, can be decomposed into additions, subtractions, and binary shifts. The problem of finding the decomposition using minimum number of addition/subtraction operations is known as the *Single Constant Multiplication* (SCM) problem and it is proven to be NP-complete in [2]. The SCM problem is similar to the addition chain problem [63] where the constant multiplication is realized using only addition and shift operations. The multiplication by a single constant occurs in many applications such as, multiple precision arithmetic, cryptography, and in the design of compilers.

The lower bound on the minimum number of operations required to implement the SCM is investigated in [64] and is given as follows:

$$\#operations_{lb,SCM} = \lceil log_2 S(t_1) \rceil \quad \textbf{(3.1)}$$

where $S(t_1)$ denotes the number of non-zero digits of t_1 when it is defined under CSD, i.e., the minimum number of non-zero digits required to represent t_1. We note that the given lower bound indicates that the solution of the SCM problem cannot include the number of operations less than the lower bound.

The algorithms designed for the SCM problem is generally categorized in three classes:

- Digit-based methods;
- Common Subexpression Elimination (CSE) algorithms;
- Graph-based algorithms.

A digit-based method defines the constant in a particular number representation and realizes the multiplierless implementation of the constant multiplication from its representation. This method is the fastest, i.e., its computational complexity is linear in the number of digits in the representation of the constant. Thus, the multiplication of the constant including hundred and thousands of digits with a variable can be easily implemented. But, this method is the worst-performing, i.e., its solution is generally far from the minimum implementation. For instance, suppose 1687 is multiplied with the variable x and the constant is represented under binary. Thus, the implementation of $1687x$,

$$1687x = (11010010111)_{bin}x = x_{\ll 10} + x_{\ll 9} + x_{\ll 7} + x_{\ll 4} + x_{\ll 2} + x_{\ll 1} + x, \quad \textbf{(3.2)}$$

requires six addition operations. However, when the constant is defined under CSD representation,

$$1687x = (10\bar{1}01010\bar{1}00\bar{1})_{CSD}x = x_{\ll 11} - x_{\ll 9} + x_{\ll 7} + x_{\ll 5} - x_{\ll 3} - x, \quad \textbf{(3.3)}$$

the constant multiplication requires five operations. Note that the use of CSD representation yields similar or better results than binary representation in the digit-based method, since a constant is represented using minimum number of

non-zero digits in CSD. As shown in [65], the use of binary representation yields a solution with $bw/2+O(1)$ operations on average, where bw denotes the bit-width of the constant. In the use of CSD representation, the average case is determined as $bw/3+O(1)$.

The sharing of partial products among the constant multiplication has a significant impact on the reduction of the number of operations. The CSE algorithms basically find the most-common patterns on the representation of the constants. The CSE heuristic of [66] designed for the SCM problem has the polynomial complexity of $O(bw^3)$ in the worst-case and can be used to find the solution of the SCM problem including large size constants, e.g., 32 bits or 64 bits. Also, the algorithm of [67], initially, represents the constant in double-base number system and then, finds a solution by sharing the partial products, $3x$, $5x$, or $7x$, in a sublinear time. Returning to our example, the solution of the exact CSE algorithm [4], which is described in Section 3.2.1, when the constant is defined under CSD representation includes four operations and is given as follows:

$$\begin{aligned} 3x &= x_{\ll 2} - x, \\ 13x &= 3x_{\ll 2} + x, \\ 23x &= 3x_{\ll 3} - x, \\ 1687x &= 13x_{\ll 7} + 23x. \end{aligned} \tag{3.4}$$

Observe that the common partial product $3x = x_{\ll 2} - x$ identified by the exact CSE algorithm is included in $1687x$ twice when the constant 1687 is defined under CSD representation.

However, the solutions of these algorithms depend on the number representation. Thus, the minimum number of operations solution of the SCM problem cannot be guaranteed by these algorithms, although the constant is represented using minimum number of non-zero digits and the sharing of possible common partial products is utilized. On the other hand, graph-based algorithms are not restricted to a number representation and consider the constant in its decimal value. The graph-based algorithms synthesize constants by building a graph where the vertices are labeled with constants and the edges are labeled with the sign and shifts. The exact graph-based algorithm of [68] proposed for the SCM problem, initially, finds all

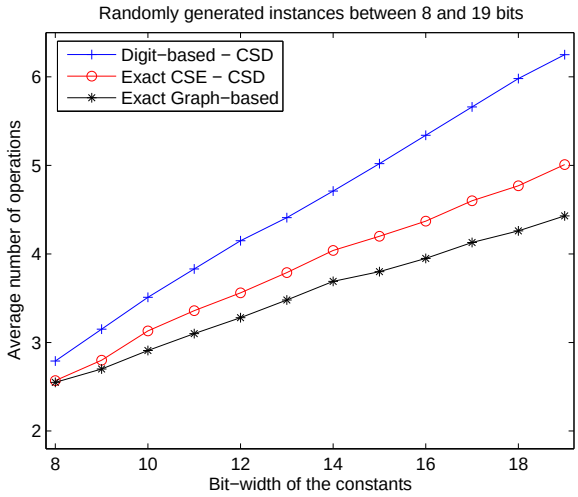


Figure 3.1: Comparison of the algorithms designed for the SCM problem.

possible graph topologies that include at most four operations. Thus, the minimum number of operations implementations of constants up to 12 bits are found by assigning the intermediate constants to the nodes of the networks exhaustively. The method described in [69] introduces simplifications on the graph topologies and extends the exact algorithm of [68] to consider all possible implementations of at most five operations. Thus, for the constants up to 19 bits, the minimum number of operations solutions are obtained. However, the exact graph-based algorithm [69] requires immense computational time as well as memory sources due to its exhaustiveness. The minimum number of operations realization of our example obtained by the exact graph-based algorithm of [69] requires three operations and is given as follows:

$$\begin{aligned} 7x &= x_{\ll 3} - x, \\ 105x &= 7x_{\ll 4} - 7x, \\ 1687x &= 7x_{\ll 8} - 105x. \end{aligned} \tag{3.5}$$

In Figure 3.1, we compare the algorithms proposed for the SCM problem in terms of the number of operations. In this experiment, for each bit-width, bw, between 8 and 19, 200 constants were generated randomly in $\left[2^{bw-1}+1, 2^{bw}-1\right]$. For the digit-based and the exact CSE algorithms, the constants were defined under CSD representation.

Observe from Figure 3.1 that the digit-based method finds worse solutions than those of the exact CSE and graph-based algorithms. Also, note that the difference of average number of operations solutions obtained with the digit-based and CSE algorithms between those of the exact graph-based algorithm increases, as the bit-width of the constant increases. We note that the difference of average number of operations solutions obtained by the digit-based method and the exact graph-based algorithm on constants defined in 19 bit-width reaches to 1.24. This value between the exact CSE and graph-based algorithms is 0.58. This experiment clearly indicates that an exact graph-based algorithm is indispensable to find the minimum number of operations solution.

3.2 Multiple Constant Multiplications

An extension of the SCM problem is the problem of multiplying a variable by a set of target constants, i.e., the target set $T = \{t_1, t_2, \ldots, t_m\}$, in parallel. The implementation of multiple constant multiplications using minimum number of addition/subtraction operations is known as the *Multiple Constant Multiplications* (MCM) problem. Since the MCM problem is the generalization of the SCM problem, it also NP-complete [2]. The MCM problem finds itself and its variants in many applications such as, digital FIR filters, linear signal transforms, image processing, and computer arithmetic.

The lower bound on the minimum number of operations required to implement the MCM is also examined in [64] and is given as follows:

$$\#operations_{lb,MCM} = \min_i \{\lceil log_2 S(t_i) \rceil\} + m - 1 \quad \textbf{(3.6)}$$

where, again, $S(t_i)$ denotes the minimum number of non-zero digits required to represent t_i and m indicates the number of positive and odd unrepeated target constants in the target set T. Hence, the lower bound is equal to the minimum number of operations required to realize the simplest constant plus the number of remaining constants.

However, when the target constants are sorted in ascending order of $S(t_i)$, the given lower bound can be increased as follows:

$$\#operations_{lb,MCM} = \lceil log_2 S(t_i) \rceil + \sum_{i=1}^{m-1} E(S(t_i), S(t_{i+1})) \tag{3.7}$$

where $E(S(t_i), S(t_{i+1}))$ is computed as given in the following.

$$E(S(t_i), S(t_{i+1})) = \begin{cases} 1, & S(t_i) = S(t_{i+1}) \\ \lceil log_2(S(t_{i+1})/S(t_i)) \rceil, & S(t_i) < S(t_{i+1}) \end{cases} \tag{3.8}$$

The latter case, i.e., $S(t_i) < S(t_{i+1})$, in the computation of $E(S(t_i), S(t_{i+1}))$ indicates that it is not possible to compute the target constant with $S(t_{i+1})$ non-zero digits using only one additional operation, if there are only target constants with at most $S(t_i)$ non-zero digits available. Hence, by taking into account this case the lower bound can be increased. Again, we note that the given lower bound indicates that the solution of the MCM problem cannot include the number of operations less than the lower bound.

To obtain a solution of the MCM problem, one may apply one of the algorithms proposed for the SCM problem on each target constant of the MCM problem without taking into account the sharing of partial products in constant multiplications. As an example, suppose the multiplication of multiple constants 11 and 13 by the variable x as given in Figure 3.2(a). Observe from Figure 3.2(b) that the multiplierless implementation without partial product sharing requires four operations. However, the sharing of partial product $9x$ in both multiplications reduces the number of required operations to 3 as illustrated in Figure 3.2(c).

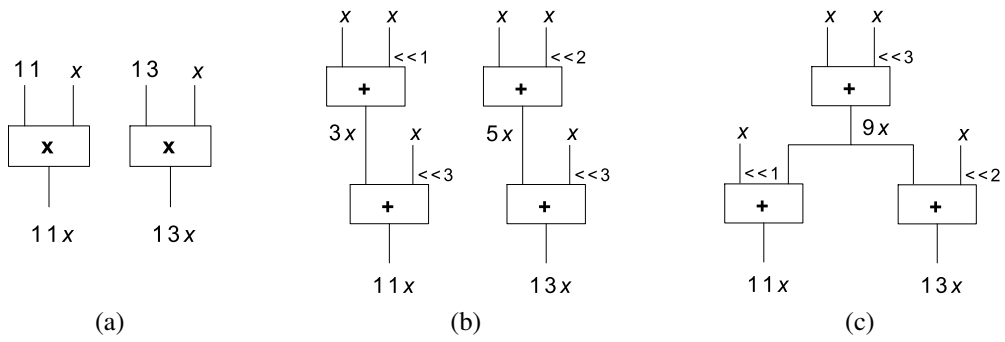


Figure 3.2: (a) Multiple constant multiplications; The shift-adds implementations of MCM: (b) without partial product sharing; (c) with partial product sharing.

The effect of the partial product sharing on the number of required operations in MCM is investigated in Figure 3.3. In this figure, the solutions obtained by the exact

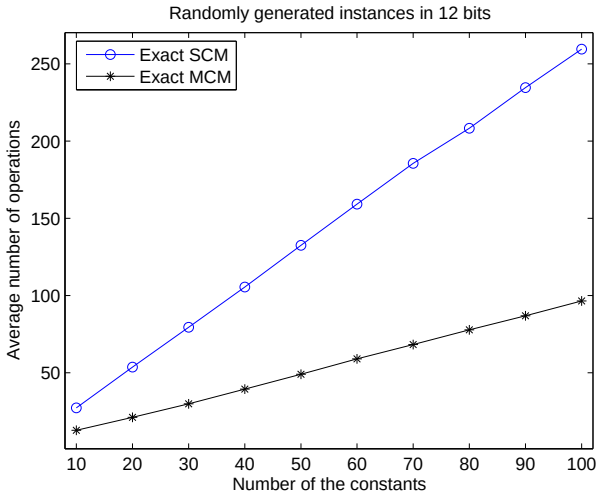


Figure 3.3: Comparison of the exact algorithms designed for the SCM and MCM problems on randomly generated MCM instances.

algorithm [69] designed for the SCM problem without considering the sharing of partial products and the results found by the exact algorithm [18] designed for the MCM problem are given. The experiment set includes randomly generated instances where constants are defined under 12 bit-width. The number of constants ranges between 10 and 100, and we generated 30 instances for each of them. As can be easily observed from Figure 3.3, the partial product sharing significantly reduces the number of required operations indicating its great effectiveness in MCM.

In the following, we give an overview of CSE and graph-based algorithms designed for the MCM problem that consider the partial product sharing. However, we also note that a large amount of work that considers the MCM problem in many applications, specially, in the design of digital FIR filters, has addressed the use of efficient implementations of multiplierless MCM. These methods include the use of different architectures, implementation styles, and constant optimization techniques, e.g., [70–74].

3.2.1 Common subexpression elimination algorithms

In CSE algorithms, initially, the constants are defined under a particular number representation. Then, all possible subexpressions are extracted from the representations of the constants and the "best" subexpression, generally, the most common, is chosen to be shared in constant multiplications. For the example given in Figure 3.2, the sharing of partial product $9x$ illustrated in Figure 3.2(c) 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 common partial product, i.e., $9x = (1001)_{bin}x$, is identified in both multiplications. The CSE algorithms designed for the MCM problem can be categorized in two classes as heuristic and exact algorithms.

The first CSE heuristic based on the CSD representation was introduced in [75] and was applied to the digital FIR filter synthesis. The proposed heuristic defines the constants under CSD representation, finds the two-terms common subexpressions, and then, chooses the one among possible subexpressions according to a benefit function. The benefit function is determined in terms of the number of operations and delay latches in the implementation of the digital FIR filter. Additionally, in [76], the algorithm that implements the constant multiplications using the most common two subexpressions, i.e., $3x$ and $5x$, was also described. The heuristic of [77], similar to the CSE heuristic of [75], initially, defines the constant multiplications as expressions and then, iteratively finds the most common two-term divisor among the possible divisors, i.e., the best divisor, and redefines the expressions by replacing the best divisor in the expressions. The use of different selection criteria for the common subexpressions in CSE algorithms were also described in [78, 79]. However, these algorithms suffer from the fact that once a common subexpression is identified as the "best" common subexpression, the decision cannot be reverted. Thus, these greedy algorithms are easily trapped to the local minima, and consequently, obtain suboptimal solutions. In [80], a CSE algorithm that relaxes the rigidity of the search for common subexpressions by allowing the earlier chosen subexpressions to be replaced with new subexpressions was introduced. This CSE heuristic considers the two-term subexpressions and also, aims to find a solution with the minimum number of adder-steps, i.e., the maximum number of operations in series.

However, the structures of these algorithms allow them to consider only the constants defined under binary or CSD representation that yields a unique representation for a constant. In these algorithms, the CSD representation is generally preferred because, a constant is represented with the minimum number of non-zero digits in CSD, reducing the complexity of the algorithms. In [28], a heuristic algorithm that exploits the redundancy of the MSD representation was proposed. It is shown that the use

of MSD representation yields less number of operations solutions with respect to the solutions obtained under CSD representation. This results from the fact that, in general, there exist several alternatives to represent a given constant in MSD. Consequently, there are more ways to decompose the constant multiplications with different partial products that can be shared with other constant multiplications.

In a recent paper [81], a heuristic algorithm that handles the constants under binary representation rather than CSD and MSD was proposed. In this paper, initially, the most common binary subexpressions, i.e., $3x$, $5x$, $7x$, and $9x$, are determined by analyzing the frequency of occurrences of binary subexpressions on a comprehensive set of FIR filter instances. Then, the constant multiplications are realized using these four subexpressions. It is shown that the use of binary representation that leads a design including only addition operations, achieves significant reductions in hardware when compared to the solutions obtained under CSD representation where the constant multiplications can be implemented using addition or subtraction operations.

The first 0-1 ILP formalization of the common subexpression sharing was introduced in [82]. However, in this method, only the subexpressions with at most two non-zero digits are considered due to the search space of the 0-1 ILP problem. As shown in [3], this model does not yield the minimum number of operations solution due to this limitation. On the other hand, in the exact CSE algorithms of [3, 4], all possible subexpressions are utilized. In these algorithms, initially, the target constants are defined under a number representation, namely, binary, CSD, or MSD. Then, all possible implementations of constant multiplications that can be extracted from the representations of the constants are obtained. In [3], the 0-1 ILP problem is obtained by constructing the cost function and formalizing the implementations of constant multiplications as constraints. To obtain the minimum number of operations solution, a generic branch-and-bound algorithm is used. In [4], all possible implementations of constant multiplications are represented in a Boolean combinational circuit that includes only AND and OR gates. An AND gate in the network represents an operation and an OR gate associated with a constant multiplication gathers all operations that implement the constant multiplication. Then, the 0-1 ILP problem is formed with a cost function to be minimized and constraints to be satisfied. The cost function

is the linear function of optimization variables associated with the partial products. The constraints of the 0-1 ILP problem are obtained by finding the CNF formulas of each gate in the Boolean network and converting each clause in CNF formulas to linear inequalities as described in [50]. The minimum number of operations solution is obtained using a generic SAT-based 0-1 ILP solver.

In [14], problem reduction and model simplification techniques that significantly reduce the size of 0-1 ILP problem obtained by the exact algorithm of [4] were introduced. In this paper, the effect of number representation on the achievable minimum number of operations was also evaluated. It is observed that, as opposed to common usage, CSD representation does not tend to give the minimum number of operation solutions in MCM. This is because, using a single representation of a constant with the minimum number of non-zero digits and both positive and negative signs may produce partial products that are less common in the implementations of constant multiplications. This drawback can be overcome using MSD that considers alternative representations of a constant with the minimum number of non-zero digits. However, the use of binary representation where a constant has a unique representation with more non-zero digits and only positive sign, increases the partial product sharing, and consequently, achieves more promising solutions than the CSD representation.

3.2.2 Extensions to the common subexpression elimination algorithms

It is argued in [83] that being limited to a number representation does not yield the minimum number of operations solutions. This heuristic algorithm obtains much better solutions than the CSE heuristics by extending the possible implementations of constants based on MSD representation. Furthermore, the algorithm of [84] applies the CSE technique of [76] to all signed-digit representations of a constant by taking into account up to k 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 $n+k$ signed digits. This approach is applied to multiple constants using exhaustive searches in [85]. Also, the algorithm of [17] extends the exact CSE algorithm of [4] to handle the constants under general number representation increasing the search space and finds more promising solutions than those of the exact CSE algorithm. Since the

algorithm of [17] has limitations on the implementation of constants to guarantee the solution to be represented in a directed acyclic graph, it does not consider the whole search space as graph-based algorithms.

3.2.3 Graph-based algorithms

In graph-based algorithms, the implementations of the constant multiplications are not restricted with the operations that can be extracted from the number representation of constants as done in CSE algorithms. Thus, the graph-based algorithms are not limited to any particular number representation and are bottom-up methods that construct a graph representing the constant multiplications. Similar to the CSE algorithms, the graph-based algorithms can be classified in two categories as heuristic and exact algorithms.

The first graph-based heuristics designed for the MCM problem, 'add-only', 'add/subtract', 'add/shift', and 'add/subtract/shift', were introduced in [86]. The latter algorithm, i.e., 'add/subtract/shift', was modified in [87], called BHM, by extending the possible implementations of a constant, considering only odd numbers, and processing constants in order of increasing single constant cost that is evaluated by the algorithm of [68]. A graph-based algorithm, called RAG-n, was also introduced in [87]. RAG-n has two parts: optimal and heuristic. In the optimal part, each target constant that can be implemented with a single operation is synthesized. If there exist unimplemented element(s) left in the target set, the algorithm switches to the heuristic part where in each iteration a single unimplemented target constant is synthesized including intermediate constant(s). In its heuristic part, RAG-n chooses an unimplemented target constant with the smallest single constant cost previously computed by the algorithm of [68] and synthesizes it with an operation including intermediate constant(s) that has the smallest value among the possible constants. However, the main drawback of BHM and RAG-n is that the effect of the chosen intermediate constant(s) over the not-yet synthesized target constant implementations is not completely considered. Thus, the intermediate constants chosen for the synthesis of the target constants in previous iterations may not be shared for the implementation of not-yet synthesized target constants in later iterations, yielding a local minimum solution. To increase the possible sharing of intermediate constants,

the graph-based algorithm, called Hcub, that includes the same optimal part of RAG-n, but uses a better heuristic than RAG-n was introduced in [88]. In its heuristic part, Hcub considers the impact of each possible intermediate constant on the not-yet synthesized target constants while implementing a target constant and chooses the one that yields the best cumulative benefit. Also, Hcub is not restricted to the lookup table that includes the single constant cost of constants as RAG-n, thus it is applicable to larger size constants. However, it is limited to the MCM problem instances including up to 200 target constants. The algorithm of [89] follows the similar approach proposed in Hcub and considers alternative intermediate constants while implementing a target constant. It is shown in [88, 89] that these graph-based heuristics obtain better results than the prominent CSE heuristics and also, the graph-based heuristics BHM and RAG-n.

We make two simple observations on RAG-n and Hcub. In these observations, $|T|$ denotes the number of target constants to be implemented, i.e., the lowest bound on the minimum number of operations solution of an MCM problem.

Lemma 3.1: If RAG-n or Hcub find a solution with $|T|$ operations, then the found solution is minimum.

Since the elements of the target set cannot be synthesized using less than $|T|$ operations as shown in [87] and the solution is obtained in the optimal part, then the found solution is the minimum solution. □

Lemma 3.2: If RAG-n or Hcub find a solution with $|T|+1$ operations, then the found solution is minimum.

If a solution cannot be obtained in the optimal part, then it is obvious that at least one additional operation is required to find the minimum solution. So, if the found solution includes $|T|+1$ operations, then it is the minimum solution. □

Note that RAG-n and Hcub 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 RAG-n and Hcub.

Observe that the case described in Lemma 3.1 is general for all algorithms designed for the MCM problem and the cases described in Lemma 3.1 and Lemma 3.2 are general for all graph-based algorithms that include the same optimal part of RAG-n and Hcub. We also note that the solution found by any heuristic algorithm designed for the MCM problem can be determined as minimum, if the number of operations in its solution is equal to the lower bound on the minimum number of operations given in (3.7).

On the other hand, in the approximate graph-based algorithm introduced in [19], rather than synthesizing the target constants once at a time by finding the "best" intermediate constants as done in the previously proposed graph-based algorithms, the proposed algorithm searches the fewest number of intermediate constants such that the target and intermediate constants can be implemented using a single operation at the end of the algorithm. Also, the design of the approximate algorithm in this scheme allows the algorithm to guarantee the minimum solution on more instances than the previously proposed graph-based heuristics. It is shown in [19] that the approximate graph-based algorithm finds competitive and better results than previously proposed graph-based heuristics.

The exact graph-based algorithm that finds the minimum number of operations solution of the MCM problem was introduced in [18]. The exact algorithm is based on the breadth-first search and explores all possible intermediate constants exhaustively. It is shown in [18] that although the exact graph-based algorithm can be applied on medium-size instances, it obtains better results than the prominent graph-based heuristics.

In the following chapter, we introduce the exact and approximate CSE and graph-based algorithms [14–19] designed for the MCM problem and compare with the previously proposed efficient CSE and graph-based algorithms.

4. OPTIMIZATION ALGORITHMS FOR THE MCM PROBLEM

In this chapter, initially, we introduce the exact and approximate CSE algorithms designed for the MCM problem. Then, we extend the exact CSE algorithm to handle the constants under general number representation. Finally, we describe the exact and approximate graph-based algorithms.

We note that since the multiple constants are multiplied by the same input, finding the minimum number of operations solution for the MCM is equivalent to finding the decompositions of multiple constants using minimum number of operations. Hence, in the description of the proposed algorithms for the MCM problem, the latter is favored for the sake of clarity.

4.1 Common Subexpression Elimination Algorithms

In this section, initially, we describe the exact CSE algorithm [15] designed for the MCM problem and then, we present the approximate CSE algorithm [16] based on the exact CSE algorithm that can be applied on more complex instances which the exact CSE algorithm cannot handle.

4.1.1 The exact common subexpression elimination algorithm

The exact CSE algorithm consists of four main steps: Firstly, all possible implementations of constants are extracted from the non-zero digits of the constants defined under a number representation, namely, binary, CSD, or MSD. Secondly, the implementations of constants are represented in a Boolean network that includes only AND and OR gates. Thirdly, the MCM problem is formalized as a 0-1 ILP problem with a cost function to be minimized and constraints to be satisfied. Finally, the minimum number of operations solution is obtained using a generic SAT-based 0-1 ILP solver.

We note that in the exact CSE algorithm, the MCM problem can be modeled as the minimization of the number of operations or the minimization of the number of partial terms, i.e., the maximization of the partial term sharing.

4.1.1.1 Finding the implementations of constants

In the preprocessing phase of the exact algorithm, the target constants to be implemented are converted to positive and then, made odd by successive divisions by 2. The resulting constants are stored in a set called target set, T, without repetition. Thus, the target set includes the minimum number of necessary target constants to be implemented. The part of the exact algorithm where the implementations of the target constants and the partial terms, i.e., also called as intermediate constants in this thesis, are found is given as follows:

1. Take an element from the target set, t_i, find its representation(s) under the given number representation, and store them in a set called S. Form an empty set, O_i, associated with t_i that includes the inputs of all addition/subtraction operations that implement t_i as pairs.

2. For each representation of t_i in the S,

   (a) Compute all non-symmetric partial term pairs that cover the representation of t_i.

   (b) Make each partial term positive and odd.

   (c) Add each partial term pair to the set O_i.

   (d) Add each partial term to the target set, if it does not represent the input that is the constants are multiplied with, i.e., denoted by 1, and is not in the target set.

3. Repeat Step 1 until all elements of the target set are considered.

Observe that the target set that only includes the target constants to be implemented in the beginning of the iterative loop is augmented with the partial terms that are required for the implementation of target constants. Note that all possible implementations of an element in the target set, t_i, are found by decomposing the non-zero digits in the representations of t_i into two partial terms.

$51 = 1000000 + 00\bar{1}010\bar{1} = 1_{\ll 6} - 13$ $\quad$ $51 = 10\bar{1}0000 + 000010\bar{1} = 3_{\ll 4} + 3$
$51 = 00\bar{1}0000 + 100010\bar{1} = -1_{\ll 4} + 67$ $\quad$ $51 = 1000100 + 00\bar{1}000\bar{1} = 17_{\ll 2} - 17$
$51 = 0000100 + 10\bar{1}000\bar{1} = 1_{\ll 2} + 47$ $\quad$ $51 = 100000\bar{1} + 00\bar{1}0100 = 63 - 3_{\ll 2}$

Figure 4.1: Implementations of 51 under CSD representation.

As an example on finding all possible implementations of a constant, suppose 51 as a target constant defined under CSD representation as $10\bar{1}010\bar{1}$ with four non-zero digits. The possible implementations of 51 are given in Figure 4.1.

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 4.1. Observe that after the partial terms required for the implementation of 51 under CSD, i.e., 3, 13, 17, 47, 63, and 67, are found, they are added to the target set without repetition, and their implementations are also found in similar way.

4.1.1.2 Construction of the Boolean network

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

1. Take an element from the target set, t_i.
2. For each pair in O_i, generate a two-input AND gate. The inputs of the AND gate are the elements of the pair, i.e., 1, denoting the input that the constants are multiplied with, or the outputs of the OR gates representing the target constants and partial terms in the network.
3. Generate an OR gate associated with t_i where its inputs are the outputs of the AND gates determined in Step 2.
4. If t_i is a target constant, assign the output of the corresponding OR gate as the output of the network.
5. Repeat Step 1 until all elements in the target set are considered.

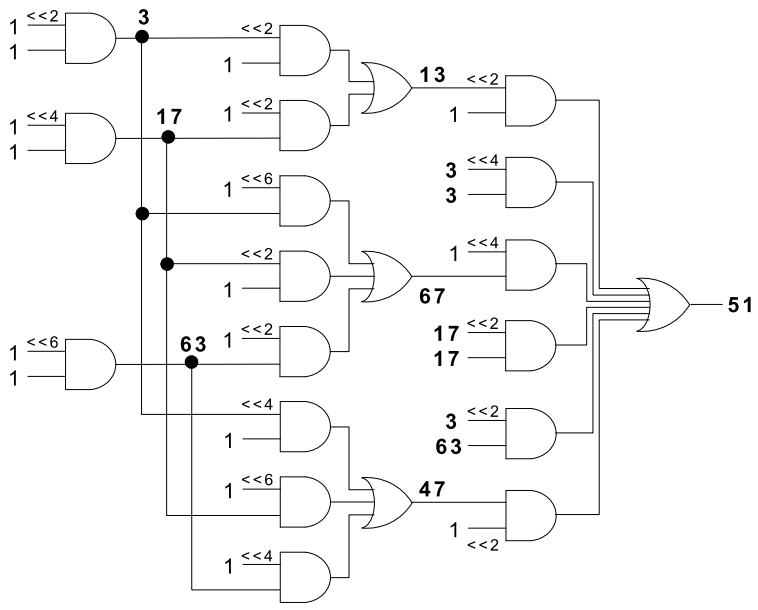


Figure 4.2: The network constructed for the target constant 51 under CSD representation.

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

1. The primary input of the network is the input to be multiplied with the constants denoted by 1.

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 target constant 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 target constants.

The Boolean network generated for the target constant 51 defined in CSD representation is given in Figure 4.2 where the 1-input OR gates for the partial terms 3, 17, and 63 are omitted.

Observe from Figure 4.2 that the network represents all possible implementations of the target constant 51 when it is defined under CSD representation. Note that when constants are defined in CSD or MSD representation, an AND gate represents

an addition/subtraction operation. For example, consider the value 3. Its CSD representation is $10\bar{1}$ and therefore, this value can be obtained with a single subtracter as $1_{\ll 2} - 1$. In MSD, the value 3 can be represented both by 011 and $10\bar{1}$ that can be obtained with an adder as $1_{\ll 1} + 1$ and with a subtracter as $1_{\ll 2} - 1$ respectively. Note that if the constants are defined in binary representation, then each AND gate represents an addition operation.

Also, observe that the exact CSE algorithm can easily 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.

4.1.1.3 Optimization models

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 that yield the same minimum number of operations solution as shown in the following three lemmas.

In *the minimization of the number of operations model*, the optimization variables are associated with the operations that are required for the implementations of target constants and partial terms. Thus, we add a third input denoting an optimization variable to each AND gate that represents an operation in the network. The inclusion of optimization variables is exemplified in Figure 4.3(a) for the implementation of the target constant 51 defined under CSD.

We make a simple observation on the minimization of the number of operations model.

Lemma 4.1: In the optimum solution, 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 another 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. For the remaining AND gates, one suffices to set the output

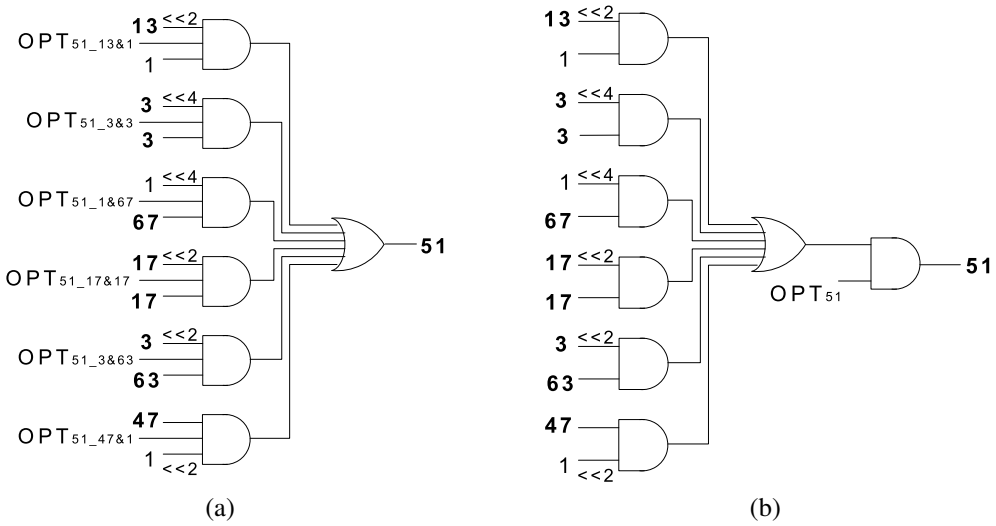


Figure 4.3: Addition of optimization variables in the network: (a) the minimization of the number of operations model; (b) the minimization of the number of partial terms model.

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. □

Note that in this model, the solution to the minimization of the cost function will indicate directly which operations are required. Thus, in the realization of target constants and partial terms, the operations whose optimization variables evaluate to 1 are synthesized.

In *the minimization of the number of partial terms model*, optimization variables are associated with the partial terms that are required to implement the target constants. Thus, we add a 2-input AND gate for each OR gate representing a target constant or a partial term in the network, where one input is the output of the OR gate and the other is the optimization variable. The inclusion of optimization variables is exemplified in Figure 4.3(b) for the implementation of the target constant 51 defined under CSD.

We make a simple observation on the minimization of the number of partial terms model.

Lemma 4.2: If the optimization variable evaluates to 1 in the optimum solution, then the output of the corresponding OR gate evaluates to 1.

Since the cost function to be minimized is the linear function of the optimization variables, if the optimization variable evaluates to 1, then the output of the

corresponding OR gate is required in the optimum solution. Otherwise, the optimization variable could be set to 0 and we would have a better solution, which is a contradiction. □

Note that the converse is not true. If a pair of partial terms that can be combined to generate a partial term with a single operation is available, then the output of the OR gate will evaluate to 1. However, this does not mean that this particular constant is going to be computed, i.e., the optimization variable associated with the constant will evaluate to 1. Thus, in the realization of each target constant and partial term with an optimization variable set to 1 in the optimum solution, one of the associated operations whose AND gate output is set to 1 is synthesized. Since there may be more than one available implementation of the constant, we choose the one that yields the minimum delay of the constant multiplications defined in terms of the number of operations in series, i.e., generally known as the number of adder-steps. Thus, by traversing from primary inputs to primary outputs, the minimum delay synthesis of the found minimum number of operations solution is realized.

Observe that the minimization of the number of partial terms in the exact model is equivalent to the maximization of the partial term sharing.

The following conclusion can be drawn from Lemma 4.1 and 4.2.

Lemma 4.3: The minimization of the number of operations is equivalent to the minimization of the number of partial terms.

In the minimization of the number of operations, we obtain one operation per partial term as given by Lemma 4.1.

In the minimization of the number of partial terms, if the optimization variable at the output of an OR gate evaluates to 1, then one of the AND gates whose output evaluate to 1 is selected arbitrarily. Thus, we obtain one operation per required partial term, i.e., also the same as the number of optimization variables set to 1 as given by Lemma 4.2.

Thus, in both approaches, since we have a one to one correspondence between operation and partial term, and since both solutions are optimum, they have to yield the same cost. □

Although these models yield the same solution in terms of the number of operations, they put their own characteristics into the 0-1 ILP problem. In the minimization of the number of operations model, the optimum solution indicates directly which operations to be implemented. For the MCM problem, this is not so relevant, since it is indifferent which of the available operations is used to compute a target constant or a partial term. However, as will be discussed in Chapter 5-7, it is essential in more sophisticated optimization problems. However, by using the minimization of the number of operations model, the number of optimization variables is increased with respect to the minimization of the number of partial terms model. As shown in Section 4.1.3.3, although this may signify an increased difficulty for SAT-based 0-1 ILP solvers that solve the PB optimization problem iteratively using an efficient SAT-engine, the SAT-based 0-1 ILP solvers that are also equipped with ILP methods perform well on these problems.

4.1.1.4 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, the performance of a generic 0-1 ILP solver can be increased. The following rules can be applied to remove unnecessary inputs from the gates and redundant gates from the network.

1. Since there is no need to implement the primary input, we assign 1 value to the variable indicating the primary input and propagate this value to remove unnecessary gates in the network.

2. Since the implementation of target constants is aimed, we assign 1 value to the outputs of OR gates and AND gates representing the target constants in the minimization of the number of operations model and in the minimization of the number of partial terms model respectively, and make these implications.

3. If an operation includes two identical partial terms at the inputs, then we may remove one of them from the inputs.

4. If the requirements of an operation are more stringent than another operation that implements the same constant, then we may remove it. For example, in Figure 4.2, for the implementation of 51, 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.

The impact of these simplifications depends heavily on the particular instance. They may yield few simplifications in the network or an immediate solution, hence avoiding the 0-1 ILP solver altogether. The impact of these simplifications on the network of Figure 4.2 are presented in Figure 4.4 and 4.5 for the minimization of the number of operations model and the minimization of the number of partial terms model respectively.

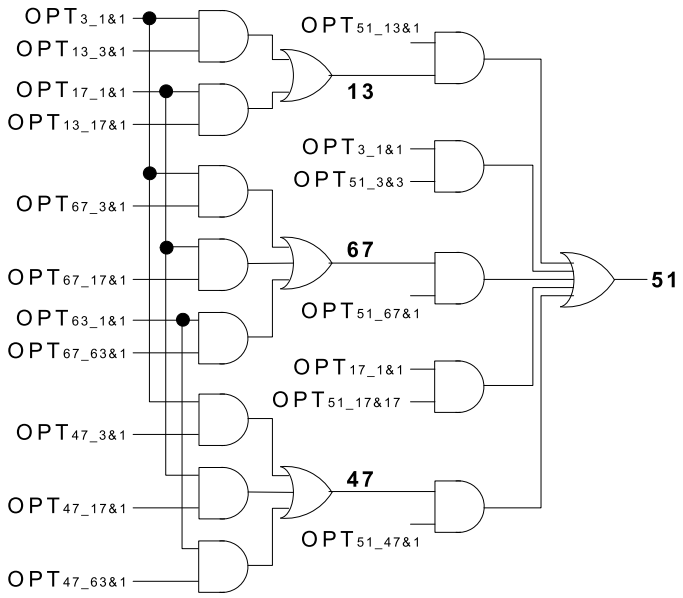


Figure 4.4: Simplification of the network of Figure 4.2 after optimization variables for minimizing the number of operations are added.

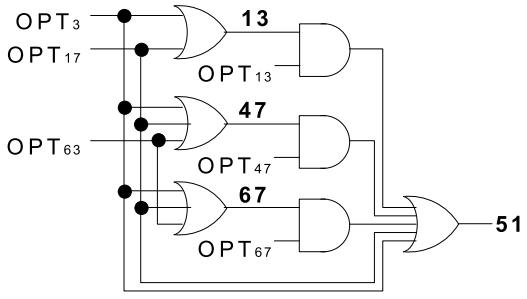


Figure 4.5: Simplification of the network of Figure 4.2 after optimization variables for minimizing the number of partial terms are added.

Additionally, during the construction of the network, the issues described in [90] that speed-up a generic SAT-based 0-1 ILP solver are also considered.

4.1.1.5 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 straight-forward. The cost function is formed as a linear function of optimization variables where the cost value of each optimization variable is 1. The constraints of the 0-1 ILP problem are obtained by finding the CNF formulas of each gate in the network and expressing each clause in CNF formulas as a linear inequality as described in [50]. For example, a 2-input AND gate, $c = a \wedge b$, is translated to CNF as $(a+\overline{c})(b+\overline{c})(\overline{a}+\overline{b}+c)$ and converted to PB constraints as follows:

$$\begin{aligned} a-c &\geq 0 \\ b-c &\geq 0 \\ -a-b+c &\geq -1 \end{aligned} \tag{4.1}$$

Also, the outputs of the OR gates associated with the target constant are set to 1 in the 0-1 ILP problem, since the implementation of target constants is aimed. Thus, the obtained model can serve as an input to a generic 0-1 ILP solver.

4.1.1.6 Analysis of 0-1 ILP problem complexity

In this section, we present the complexity analysis of the MCM problem in terms of the size of the Boolean network, i.e., the number of AND and OR gates, and the size of the 0-1 ILP problem, i.e., the number of variables, constraints, and optimization variables, generated by the exact CSE algorithm under both models.

The complexity analysis is based on a single target constant represented with n non-zero digits. As a special case, suppose the constant is represented in binary with all n digits set to 1. In this case, the Boolean network includes all partial terms with b bits, $b \leq n$, set to 1. Thus, all target constants that include the number of 1 bits less than n are considered in the network. Hence, for n-bit target constants in any representation, the complexity of the problem is bounded above by the case of a single coefficient with all the n bits set to 1. Also, in this analysis, network

simplification techniques described in Section 4.1.1.4 are not taken into account thus, the upper-bounds on the problem complexity are obtained.

In the minimization of the number of operations model[1], the number of gates in the network is given as follows:

$$\#_{\text{ORs}} = \sum_{i=2}^{n} \binom{n}{n-i} \tag{4.2}$$

$$\#_{\text{ANDs}} = \sum_{i=2}^{n} (2^{i-1}-1) \binom{n}{n-i} \tag{4.3}$$

The number of variables, constraints, and optimization variables of the 0-1 ILP problem for this model is given as follows:

$$\#_{\text{vars}} = \#_{\text{optvars}} + \#_{\text{ORs}} + \#_{\text{ANDs}} + 1 \tag{4.4}$$

$$\#_{\text{cons}} = \overbrace{4\#_{\text{ANDs}}}^{\text{due to ANDs}} + \overbrace{(\#_{\text{ANDs}} + \#_{\text{ORs}})}^{\text{due to ORs}} \tag{4.5}$$

$$\#_{\text{optvars}} = \#_{\text{ANDs}} \tag{4.6}$$

Observe from (4.4) that the number of variables is the number of optimization variables plus the number of gates of the network plus 1, i.e., the primary input of the network. As can be easily observed from (4.6), the number of optimization variables is the number of AND gates in the network. Note that the number of constraints given by (4.5) is the number of constraints due to the AND gates plus the number of constraints due to the OR gates in the network. Since the number of fanins of each AND gate is three in this model, the number of constraints obtained by the AND gates is $4\#_{\text{ANDs}}$. Since each AND gate output is the fanin of an OR gate in the network, the number of constraints obtained by the OR gates is the number of AND and OR gates in the network.

In the minimization of the number of partial terms model[2], the number of gates in the network is given as follows:

$$\#_{\text{ORs}} = \sum_{i=2}^{n} \binom{n}{n-i} \tag{4.7}$$

$$\#_{\text{ANDs}} = \#_{\text{ORs}} + \sum_{i=2}^{n} (2^{i-1}-1) \binom{n}{n-i} \tag{4.8}$$

[1]Recall that in the minimization of the number of operations model, the optimization variables are associated with the operations, i.e, the AND gates in the network.

[2]Recall that in the minimization of the number of partial terms model, the optimization variables are associated with the partial terms, i.e, the OR gates in the network.

Observe from (4.2) and (4.7) that the number of OR gates in both optimization models is the same. However, in the minimization of the number of partial terms model, the number of AND gates is augmented with the number of OR gates with respect to (4.3). Because, in this model, an additional 2-input AND gate is included for each constant represented by the OR gates in the network.

The number of variables, constraints, and optimization variables of the 0-1 ILP problem for this model is given as follows:

$$\#_{\text{vars}} = \#_{\text{optvars}} + \#_{\text{ORs}} + \#_{\text{ANDs}} + 1 \quad \textbf{(4.9)}$$

$$\#_{\text{cons}} = \overbrace{3\#_{\text{ANDs}}}^{\text{due to ANDs}} + \overbrace{(\#_{\text{ANDs}} + \#_{\text{ORs}})}^{\text{due to ORs}} \quad \textbf{(4.10)}$$

$$\#_{\text{optvars}} = \#_{\text{ORs}} \quad \textbf{(4.11)}$$

Observe from (4.11) that the number of optimization variables is the number of OR gates in the network. Also, since the number of fanins of each AND gate is two in this model, the number of constraints due to the AND gates is $3\#_{\text{ANDs}}$ as given in (4.10).

Table 4.1 and 4.2 give the size of the Boolean network in terms of the number of AND and OR gates, and the size of the 0-1 ILP problem in terms of the number of variables, constraints, and optimization variables for a single constant with different values of n bits, all set to 1, for the minimization of the number of operations model and the minimization of the number of partial terms model respectively.

Although we can observe the exponential growth in the size of Boolean network and 0-1 ILP problem in both models, the size of the 0-1 ILP problem for up to $n = 12$ is within the reach of current 0-1 ILP solvers. In practice, constants with 12 bits set to 1 may suffice for many real problems. Observe that the exact algorithm can be efficiently applied to larger constants, when they are defined in CSD or MSD representations, since the constants are represented with minimum number of non-zero digits under CSD and MSD. We also note that the network simplifications described in Section 4.1.1.4 reduce the 0-1 ILP problem size significantly, especially, in the model of minimizing the number of partial terms [14], hence allowing the exact algorithm to be applied to larger size problem instances.

Table 4.1: Upper bounds on the size of network and 0-1 ILP problem in the minimization of the number of operations model.

n	$\#_{\text{ORs}}$	$\#_{\text{ANDs}}$	$\#_{\text{vars}}$	$\#_{\text{cons}}$	$\#_{\text{optvars}}$
4	11	25	62	136	25
6	57	301	660	1,562	301
8	247	3,025	6,298	15,372	3,025
10	1,013	28,501	58,016	143,518	28,501
12	4,083	261,625	527,334	1,312,208	261,625
14	16,369	2,375,101	4,766,572	11,891,874	2,375,101
16	65,519	21,457,825	42,981,170	107,354,644	21,457,825

Table 4.2: Upper bounds on the size of network and 0-1 ILP problem in the minimization of the number of partial terms model.

n	$\#_{\text{ORs}}$	$\#_{\text{ANDs}}$	$\#_{\text{vars}}$	$\#_{\text{cons}}$	$\#_{\text{optvars}}$
4	11	36	59	155	11
6	57	358	473	1,489	57
8	247	3,272	3,767	13,335	247
10	1,013	29,514	31,541	119,069	1,013
12	4,083	265,708	273,875	1,066,915	4,083
14	16,369	2,391,470	2,424,209	9,582,249	16,369
16	65,519	21,523,344	21,654,383	86,158,895	65,519

4.1.2 The approximate common subexpression elimination algorithm

Although the exact CSE algorithm presented in the previous section can be applied effectively to relatively large size MCM problems, the execution time to obtain the minimum number of operations solution does tend to grow exponentially with the size of 0-1 ILP problem, limiting its application to more complex instances. Thus, heuristic algorithms that obtain similar results with the minimum solution using very little computational resources are indispensable for the problem instances that the exact algorithms cannot handle.

In this section, we describe the approximate algorithm called ASSUME-A [16] designed for the optimization of the number of operations in MCM. The approximate algorithm we propose uses as the underlying model the Boolean network generated by the exact algorithm, as described in Section 4.1.1.2. In the approximate algorithm, the constants are synthesized one at a time using a single operation selected among the set of possible operations. In the selection of an operation for the implementation of a constant, initially, the implementation costs of all operations are found by considering the not-yet synthesized constants and then, the operation that has the minimum

implementation cost is chosen to implement the constant. The advantages of the proposed algorithm are the use of the network that has the view of all the possible manners a constant can be synthesized and the use of a selection criteria that also considers the not-yet synthesized constants while choosing an operation to implement a constant. These properties make the approximate algorithm quite different from the heuristics that find pairs of the most common non-zero digits [76] or the two-term common subexpressions [77]. Since the heuristics of [76] and [77] build the constants starting at the most simple (in the number of non-zero digits) to the most complex by combining existing partial terms, this bottom-up approach yields a much more limited view of the search space.

In the preprocessing phase of ASSUME-A, by traversing the Boolean network from primary inputs to primary outputs, the *min_adder* and *max_level* values of each operation and constant are computed. The *min_adder* is the minimum number of operations that are required to implement an operation or a constant. The *min_adder* value of the primary input is 0. The *min_adder* value of a constant, i.e., represented by an OR gate, is determined by finding the minimum of the *min_adder* values of operations, i.e., the AND gates, that implement the constant. The *min_adder* value of an operation is the sum of the *min_adder* values of its inputs plus 1, if the inputs are different; otherwise, it is the *min_adder* value of an input plus 1. As an example, consider the network given in Figure 4.2 with the target constant 51 defined under CSD. The *min_adder* value of the constant 51 is 2, determined, for instance, by the operations $51 = 3_{\ll 4} + 3$ and $3 = 1_{\ll 2} - 1$. The *max_level* is the maximum number of operations in series to implement a constant. Again, consider the network of Figure 4.2. The *max_level* of the target constant 51 is 3, determined, for instance, by the operation $51 = 13_{\ll 2} - 1$ that can be implemented with maximum 2 operations in series.

In a similar manner to the algorithms of [87] and [88], ASSUME-A has two main parts: optimal and heuristic. The algorithm is as follows:

1. Store the pre-processed positive and odd target constants in a set called *Aset* without repetition and label them as unimplemented.

2. The *optimal* part: For each element labeled as unimplemented in *Aset*, *Aset*(i), if *Aset*(i) is implemented in the network with an operation whose inputs are either primary inputs or are in *Aset*, then synthesize the element with this operation and label it as implemented.

3. If there are no more elements labeled as unimplemented in *Aset*, return the solution and stop.

4. The *heuristic* part: Take an unimplemented element from *Aset*, *Aset*(i), that has the lowest *max_level* value.

5. For each operation, $O(j)$, that implements *Aset*(i), set its cost value, $C(j)$, to its *min_adder* value, as determined in the preprocessing phase and for each unimplemented element in *Aset*, *Aset*(k), with $i \neq k$:

   (a) Determine $C_{before}(k)$ by finding the *min_adder* value of *Aset*(k), when the *min_adder* values of the elements in *Aset* are assigned to 0. $C_{before}(k)$ is the cost of implementation of *Aset*(k) at this phase of the algorithm, since all elements in *Aset* will be implemented at the end of the algorithm.

   (b) Determine $C_{after}(k)$ as done in (a), but also *assume* that the inputs of $O(j)$ are in *Aset*. $C_{after}(k)$ is the cost of implementing *Aset*(k), if *Aset*(i) is synthesized with $O(j)$ at this phase of the algorithm.

   (c) Update the cost value, $C(j)$, as $C(j) = C(j) - (C_{before}(k) - C_{after}(k))$.

6. After the cost value of each operation, $C(j)$, has been computed, select the operation to synthesize *Aset*(i) that has the minimum cost. If there are operations that have the same minimum cost, select the operation that has the minimum *min_adder* value among these operations. Label *Aset*(i) as implemented.

7. Add each input of the selected operation to *Aset*, provided that they do not already exist in *Aset*, and label them as unimplemented. Go to Step 2.

We note that in the first iteration, the elements of *Aset* are the target constants and in later iterations, *Aset* may include the partial terms needed for the synthesized operations. Observe that all elements of *Aset* are implemented at the end of the algorithm. Also, we note that if all elements of *Aset* are implemented in the

optimal part, then the global minimum solution is obtained. If an element of *Aset* is implemented in the heuristic part, the local minimum solution is obtained.

4.1.3 Experimental results

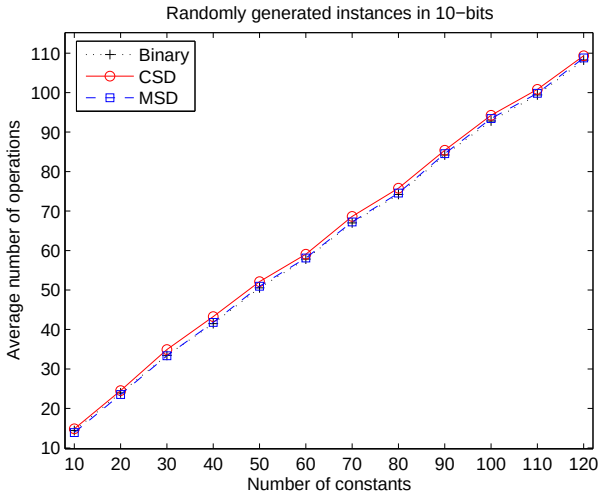
This section starts with the investigation of the effect of number representation on the achievable minimum number of operations in MCM. It is followed by the examination of the effect of the problem reduction techniques on 0-1 ILP problem size and the performance of the SAT-based 0-1 ILP solver. Then, it includes the comparison of the performance of the SAT-based 0-1 ILP solvers that have different algorithmic architectures on the optimization models used in the exact CSE algorithm. Finally, this section ends with the comparison of the exact and approximate CSE algorithms introduced in this work with the previously proposed prominent CSE heuristics.

4.1.3.1 The effect of number representation on the achievable minimum number of operations

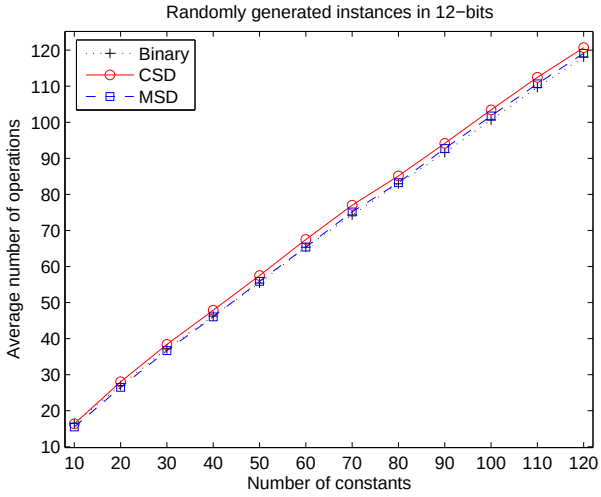
For this experiment, we used randomly generated instances where constants are defined in 10, 12, and 14 bit-widths. In this experiment set, the number of constants ranges between 10 and 120, and we generated 30 instances for each of them. In this experiment, constants are defined under binary, CSD, and MSD representations. Figure 4.6(a)-(c) present the results of the exact CSE algorithm on randomly generated instances where constants are defined under 10, 12, and 14 bits respectively.

As can be observed from Fig. 4.6(a)-(c), using binary representation yields better results than CSD on average. Also, as the number and bit-width of the constants increase, the difference of the number of operations on average between CSD and binary representations tends to increase. While the difference of the average number of operations under 10 bit-widths on problem instances including 120 constants between CSD and binary is 1.1, this value on problem instances including 120 constants under 14 bit-widths is 3.9.

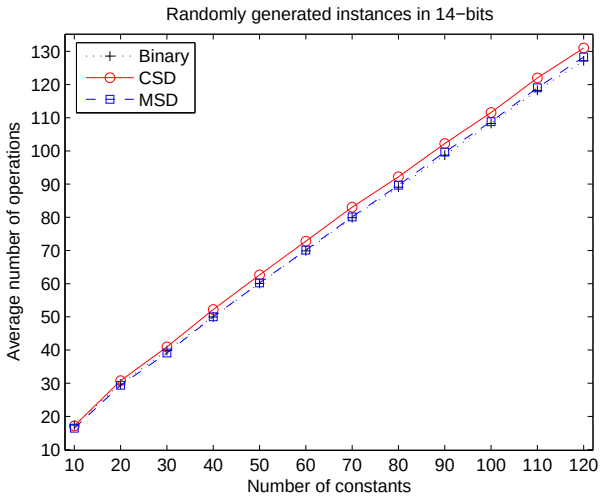
Also, we note that the exact solutions obtained under binary and MSD are quite similar. However, as the number and bit-width of constants are increased, using binary representation achieves better solutions than MSD. For example, while the



(a)



(b)



(c)

Figure 4.6: Results of the exact CSE algorithm under binary, CSD, and MSD representations on randomly generated instances: (a) Constants in 10 bits; (b) Constants in 12 bits; (c) Constants in 14-bits.

difference of the average number of operations under 10 bit-widths on problem instances with 120 constants between MSD and binary is 0.6, this value on problem instances with 120 constants under 14 bit-widths reaches to 1.1.

In this experiment, we observe that as opposed to common usage, CSD representation does not tend to give the minimum number of operation solutions in MCM. Because, using a single representation of a constant with the minimum number of non-zero digits and both positive and negative signs may produce partial terms that are less common in the implementations of constants. This drawback can be overcome using MSD that considers alternative representations of a constant with the minimum number of non-zero digits. However, we observe that binary representation achieves more promising solutions than CSD, since using a unique representation of a constant with more non-zero digits and only positive sign increases the partial term sharing. Also, we note that the use of binary representation becomes more effective on finding the minimum number of operation solutions, as the number and bit-width of constants increase.

However, the main disadvantage of using binary representation is that the design can be obtained in greater number of adder-steps than those of designs when constants are defined under CSD or MSD. The average number of adder-steps of the exact solutions obtained on randomly generated instances in 14 bit-widths under binary, CSD, and MSD are presented in Fig. 4.7. We note that while the average number of adder-steps of solutions obtained under binary representation on problem instances with 120 constants is 6.1, this value is 4.6 for both CSD and MSD.

This experiment clearly indicates that the MSD representation that also yields better solutions than CSD representation should be used when seeking solutions with less number of adder-steps.

For this experiment, we also used FIR filters where the filter coefficients were computed with the *remez* algorithm in MATLAB. The filter specifications are given in Table 4.3 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.

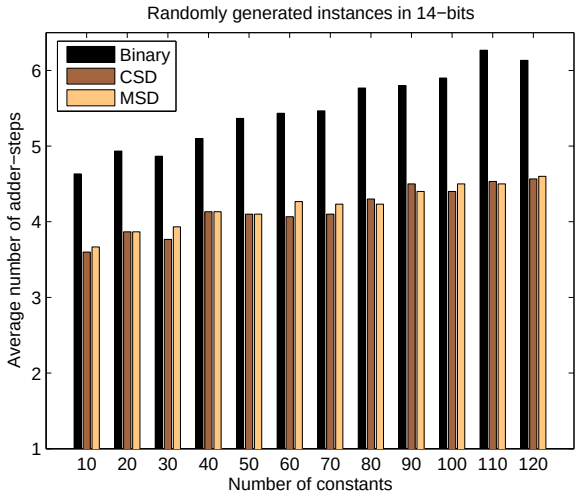


Figure 4.7: Comparison of the number of adder-steps of solutions obtained under binary, CSD, and MSD representations.

Table 4.3: Characteristics of the FIR filters.

Filter	pass	stop	#tap	width
1	0.10	0.15	200	16
2	0.10	0.15	240	16
3	0.10	0.25	180	16
4	0.10	0.25	200	16
5	0.10	0.20	240	16
6	0.10	0.20	300	16
7	0.15	0.25	200	16
8	0.15	0.25	240	16
9	0.20	0.25	240	16
10	0.20	0.25	300	16

The 0-1 ILP problem sizes of filter instances when filter coefficients are defined under binary, CSD, and MSD representations are given in Table 4.4. In this table, *vars*, *cons*, and *optvars* stand for the number of variables, constraints, and optimization variables respectively, when the MCM problem is formalized under the minimization of the number of partial terms model.

As can be easily observed from Table 4.4, the size of 0-1 ILP problems under binary representation is generally larger than the size of problems defined under CSD and MSD. This is because the binary representation of a constant includes more non-zero digits. Also, observe that the use of MSD representation yields larger size 0-1 ILP problems than those of instances when coefficients are defined under CSD representation, since a constant may have several MSD representations with the minimum number of non-zero digits including the CSD representation of the constant.

Table 4.4: 0-1 ILP problem sizes of the FIR filter instances.

Filter	Binary			CSD			MSD		
	vars	cons	optvars	vars	cons	optvars	vars	cons	optvars
1	3862	13550	944	633	1427	316	2103	6877	602
2	9904	38038	1500	618	1460	289	1776	5024	623
3	16226	67753	1433	1833	6014	476	10054	38972	1354
4	15992	63884	1928	1210	3545	420	656	1460	333
5	6808	27119	873	827	2174	329	2606	8127	751
6	13581	55759	1012	1121	3059	417	2778	8862	763
7	2413	8674	567	371	808	188	434	1043	200
8	2781	10119	642	394	824	221	861	2272	370
9	140	162	119	231	344	166	348	562	227
10	171	289	122	126	147	109	152	211	119
Total	71878	285347	9140	7364	19802	2931	21768	73410	5342

Table 4.5: Summary of results of the exact CSE algorithm on the FIR filter instances.

Filter	Binary			CSD			MSD		
	adder	step	CPU	adder	step	CPU	adder	step	CPU
1	81	7	7	83	5	0.1	82	5	0.7
2	86	6	5.5	88	5	0	87	5	0.2
3	52	5	14.2	56	4	1.3	53	5	20
4	92	7	7	94	5	0.1	93	5	0.1
5	65	6	22	66	4	0.1	66	5	8.4
6	71	6	3.1	74	5	0.2	72	4	0.3
7	62	5	0.1	65	4	0.1	64	4	0
8	71	5	0.1	73	4	0	72	4	0.1
9	79	7	0	80	4	0	80	4	0
10	82	7	0	84	4	0	84	4	0
Total	741	61	59	763	44	1.9	753	45	29.8

The results of the exact CSE algorithm on the FIR filter instances when filter coefficients are defined under binary, CSD, and MSD are presented in Table 4.5. In this table, *adder* denotes the number of addition/subtraction operations and *step* denotes the maximum number of operations in series needed to synthesize the filter coefficients. Also, *CPU* is the CPU time in seconds required for the SAT-based 0-1 ILP solver *minisat+* [52] to compute the exact solutions on a personal computer (PC) with Intel Xeon at 3.16GHz with 8GB of main memory. Since the CPU time required to construct the network in the preprocessing phase and to synthesize the MCM in the post-processing phase are also negligible, CPU only indicates the run time of *minisat+*.

As can be observed from Table 4.5, the use of binary representation yields better solutions than solutions obtained under CSD and MSD representation on every instances. We note that using binary representation leads solutions less than 2 and

Table 4.6: The effect of problem reduction techniques on 0-1 ILP problem size and performance of the SAT-based 0-1 ILP solver.

Filter	[4]				This work			
	vars	cons	optvars	CPU	vars	cons	optvars	CPU
1	60416	194552	1595	405	3862	13550	944	7
2	79707	262024	1886	278	9904	38038	1500	5.5
3	59069	191764	1510	678.2	16226	67753	1433	14.2
4	129530	444146	2366	503.1	15992	63884	1928	7
5	63076	207012	1519	52.3	6808	27119	873	22
6	58286	188294	1533	44.9	13581	55759	1012	3.1
7	47004	154086	1142	3.6	2413	8674	567	0.1
8	32044	98816	1048	3	2781	10119	642	0.1
9	133493	461220	2300	2.8	140	162	119	0
10	116294	405186	1880	2.6	171	289	122	0
Avg.	100%	100%	100%	100%	9.2%	10.9%	54.5%	3.0%

1 operation on average with respect to CSD and MSD respectively. However, the delay of the filter designs is increased when compared to the solutions obtained under CSD and MSD. Also, we note that since the CSD representation of a constant includes the minimum number of non-zero digits yielding 0-1 ILP problems in a smaller size, the exact solutions can be found in less amount of CPU time than binary and MSD representations.

4.1.3.2 The effect of problem reduction techniques on 0-1 ILP problem size

The effect of using problem reduction techniques described in Section 4.1.1.4 on the filter instances given in Table 4.3 is shown in Table 4.6. In this table, the exact CSE algorithm introduced in Section 4.1.1 is compared with the previously proposed exact CSE algorithm [4], which is not equipped with the problem reduction techniques, in terms of the size of 0-1 ILP problems they generated and the runtime of the SAT-based 0-1 ILP solver *minisat+* on these 0-1 ILP problems. We note that the results of the exact CSE algorithms on filter instances are obtained when filter coefficients are defined under binary representation.

As can be easily observed from Table 4.6, a 0-1 ILP problem that represents an MCM problem can be obtained in a smaller problem size, when the problem reduction techniques are applied. On these filter instances, while the number of variables and constraints is reduced by almost 90%, the number of optimization variables is reduced

by 45%. Also, the reduction of problem size enables the SAT-based 0-1 ILP solver, *minisat+*, to obtain the exact solutions with a very low computational effort.

4.1.3.3 Comparison of SAT-based 0-1 ILP solvers on optimization models

For this experiment, we used FIR filters where filter coefficients were computed with the *remez* algorithm in MATLAB. The specifications of filters are presented in Table 4.7.

Table 4.7: Characteristics of the FIR filters.

Filter	pass	stop	#tap	width
1	0.20	0.25	120	8
2	0.10	0.25	100	10
3	0.15	0.25	40	12
4	0.20	0.25	80	12
5	0.24	0.25	120	12
6	0.15	0.25	60	14
7	0.15	0.20	60	14
8	0.10	0.15	60	14
9	0.10	0.15	100	16

The 0-1 ILP problem size of the proposed models, i.e., the minimization of the number of partial terms and the minimization of the number of operations models, for the filter coefficients defined under MSD representation are given in Table 4.8.

As can be seen from the Table 4.8, the use of the minimization of the number of partial terms model in the formalization of an MCM problem as a 0-1 ILP problem achieves a smaller size 0-1 ILP problem with respect to the minimization of the number of operations model due to the network simplifications.

On the problem instances given in Table 4.8, we compare the SAT-based 0-1 ILP solvers *bsolo* and *minisat+*, that have different algorithmic structures, in terms of CPU time required to find a solution. The results are given in Table 4.9 where *CPU* denotes the CPU time of SAT-based 0-1 ILP solvers in seconds under a PC with dual Pentium Xeon at 2.4GHz, with 4GB of main memory, running Linux. The allowed CPU time for the 0-1 ILP solvers was 3600 seconds. In this table, the italic results indicate that a suboptimal solution rather than the minimum is obtained in the given CPU time limit.

As can be seen from Table 4.9, *bsolo* finds the minimum solutions for all instances under both models where *minisat+* cannot conclude with the minimum solution for

Table 4.8: 0-1 ILP problem sizes of the proposed optimization models.

Filter	Minimization of the Number of Partial Terms			Minimization of the Number of Operations		
	vars	cons	optvars	vars	cons	optvars
1	10	10	10	247	347	144
2	76	97	56	635	1027	345
3	151	298	80	1327	2387	677
4	93	139	64	1926	3331	1023
5	34	34	34	1142	1769	651
6	107	144	74	4324	8547	2153
7	205	455	93	2250	4828	1062
8	546	1405	200	3915	8542	1856
9	4010	14880	779	26778	55489	13329
Total	5232	17462	1390	42544	86267	21240

Table 4.9: Run time comparison of the SAT-based 0-1 ILP solvers.

Filter	Minimization of the Number of Partial Terms				Minimization of the Number of Operations			
	bsolo		minisat+		bsolo		minisat+	
	adder	CPU	adder	CPU	adder	CPU	adder	CPU
1	10	0.1	10	0	10	0.2	10	0.1
2	18	0	18	0	18	0.2	18	0.1
3	16	0.1	16	0	16	0.5	16	0.7
4	29	0	29	0	29	1.4	29	0.5
5	34	0.1	34	0	34	0.3	34	0.3
6	22	0.1	22	0	22	6.8	*22*	*3600.1*
7	34	0.1	34	0.1	34	4	34	19.5
8	33	9.8	33	0.1	33	27.4	33	60.8
9	49	380.6	49	4.3	49	1974.8	*59*	*3600.1*
Total	245	390.9	245	4.5	245	2015.6	*255*	*7282.2*

Filter 6 and 9 under the minimization of the number of operations model in an hour. We note that even if the solution of *minisat+* on Filter 6 includes the minimum number of operations, it could not prove that the found solution is the minimum solution in the given CPU time limit. However, we note that the minimization of the number of partial terms model is more appropriate for *minisat+*, since this model includes less number of optimization variables with respect to the minimization of the number of operations model. Also, the minimization of the number of operations model is more appropriate for *bsolo* than *minisat+*, since *bsolo* incorporates problem reduction techniques from both sides, i.e., SAT and branch-and-bound algorithms.

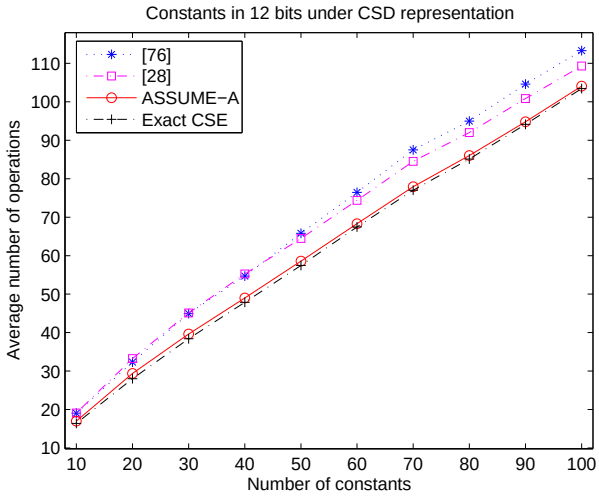


Figure 4.8: Comparison of the exact and heuristic algorithms on randomly generated instances.

4.1.3.4 Comparison of CSE algorithms

In this experiment, we compare the results of the exact and approximate CSE algorithms introduced in this chapter with those of the CSE heuristics of [76] and [28], which we have also implemented, and those of the CSE heuristic [77], that have been provided by Anup Hosangadi.

As the first experiment set, we used randomly generated instances defined in 12 bit-widths. The number of constants ranges between 10 and 100, and there exist 30 instances for each set. Figure 4.8 presents the results of CSE algorithms when constants are defined under CSD representation.

We note from Figure 4.8 that while the average number of operations between ASSUME-A and the exact CSE algorithm is almost 1 on all instances, the average number of operations between the heuristic of [28] and the exact algorithm reaches up to 7.4 operations. Also, since the heuristic of [76] 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. This experiment clearly indicates that the exact CSE algorithm finds better solutions than the heuristic algorithms and among these heuristics, ASSUME-A finds much better solutions than the CSE heuristics of [76] and [28].

Table 4.10: Summary of results of algorithms on the FIR filter instances.

Filter	CSD						MSD					
	[76]		ASSUME-A		Exact		[28]		ASSUME-A		Exact	
	adder	step	adder	step	adder	step	adder	step	adder	step	adder	step
1	10	3	10	2	10	2	10	3	10	3	10	2
2	18	3	18	3	18	3	18	3	18	3	18	3
3	19	3	16	3	16	3	18	4	16	3	16	3
4	30	3	29	3	29	4	29	4	29	4	29	3
5	36	3	34	3	34	3	34	3	34	3	34	3
6	25	3	23	3	23	4	22	4	22	4	22	4
7	35	3	35	3	35	3	35	3	34	3	34	3
8	37	4	35	3	35	3	36	4	34	4	33	3
9	55	4	52	4	51	4	51	5	49	4	49	4
Total	265	29	252	27	251	29	253	33	246	31	245	28

As the second experiment set, we used the filter instances introduced in Table 4.7. Table 4.10 compares the exact and approximate CSE algorithms with the CSE heuristics [28, 76] when filter coefficients are defined under CSD and MSD representations.

As can be observed from Table 4.10, while ASSUME-A finds similar solutions with the exact algorithm, it finds better solutions than other heuristics on overall filter instances. While the average difference of the number of operations between [28] and the exact algorithm is almost 1, the average difference of the number of operations between the heuristic of [76] and the exact CSE algorithm is greater than 1.

As the third experiment set, we used filter instances introduced in [6] to find out the limitations of the exact CSE algorithm. Table 4.11 presents these filter instances where filter coefficients are defined in 24 bit-width. In this table, the 0-1 ILP problem size of each filter instance when coefficients are defined under CSD representation is also presented. We note that the MCM problem is formalized under the minimization of the number of partial terms model.

Table 4.12 compares the results of the exact CSE algorithm with those of the CSE heuristics [28,76,77]. We note that the SAT-based 0-1 ILP solver, *minisat+*, was used to obtain the exact solutions on a PC with dual Pentium Xeon at 2.4GHz, with 4GB of main memory, and the allowed CPU-time was determined as 1 day. The italic results indicate that a suboptimal solution rather than the minimum is obtained in the given CPU time limit. We note that the results of heuristic algorithms are obtained with a very low computational effort.

Table 4.11: Characteristics of filter instances and 0-1 ILP problem sizes.

Filter	Filter Specifications					0-1 ILP Problem Size		
	Type	pass	stop	#tap	width	vars	cons	optvars
1	Butterworth	0.25	0.3	20	24	50732	202698	2158
2	Elliptical	0.25	0.3	6	24	3410	12303	350
3	Least Square	0.25	0.3	41	24	58652	230136	3269
4	Park Mc-Clennan	0.25	0.3	28	24	20572	77736	1703
5	Butterworth	0.27	0.2875	71	24	81641	324765	3984
6	Elliptical	0.27	0.2875	8	24	27614	108062	1266
7	Least Square	0.27	0.2875	172	24	46959	183081	4037
8	Park Mc-Clennan	0.27	0.2875	119	24	74334	294575	4905
9	Elliptical	0.27	0.29	13	24	34969	137129	1746
10	Least Square	0.27	0.29	326	24	38742	150786	3802
11	Park Mc-Clennan	0.27	0.29	189	24	55816	218351	4820

Table 4.12: Summary of results of the exact and heuristic algorithms.

Filter	[77]		[76]		[28]		ASSUME-A		Exact		
	adder	step	adder	step	adder	step	adder	step	adder	step	CPU
1	26	4	26	7	31	5	24	4	21	5	6244.2
2	10	3	11	7	11	4	11	5	10	4	27.7
3	58	4	61	7	67	6	52	4	*77*	*4*	*86400.1*
4	45	4	46	7	48	6	43	4	*45*	*4*	*86400.1*
5	61	4	57	6	61	6	54	4	*63*	*4*	*86400.1*
6	14	4	15	7	16	5	16	5	12	5	2387.7
7	178	4	167	5	203	6	156	5	*228*	*5*	*86400.1*
8	136	4	137	6	158	6	124	5	*192*	*4*	*86400.1*
9	24	4	24	6	27	6	23	4	*23*	*4*	*86400.1*
10	266	4	238	5	240	6	211	5	*249*	*5*	*86400.1*
11	199	4	204	6	223	5	176	4	*247*	*5*	*86400.1*
Total	1017	43	986	69	1085	61	890	49	1167	49	$>$ 8 days

In this experiment, we observe that the minimum solutions of three out of 11 filters, i.e., Filter 1, 2, and 6, are obtained in the CPU-time limit. However, the minimum solutions of eight filters could not be found in one day. We note that even if the size of the 0-1 ILP problem obtained for Filter 1 is greater than that of Filter 4, a minimum solution could not be obtained for Filter 4. This shows that the size of the 0-1 ILP problem and the hardness of the 0-1 ILP problem depend heavily on the filter coefficients. We observe that for the filter instances where the minimum solutions could not be found in the CPU-time limit, the obtained solution by the exact algorithm can be far from the solutions that are obtained by a heuristic algorithm, e.g., Filter 7 and 8. On overall instances, ASSUME-A finds the best solutions among these algorithms. This experiment also shows that the use of a heuristic algorithm

is indispensable, when an exact algorithm could not conclude with the minimum solution.

4.1.4 Conclusions

In this section, we introduce an exact CSE algorithm where the MCM problem can be formalized in either the minimization of the number of operations model or the minimization of the number of partial terms model. The problem reduction techniques that significantly reduce the 0-1 ILP size, consequently, the CPU time of a generic 0-1 ILP solver required to find the minimum solution, are also included in the exact algorithm. It is shown by the experimental results that the exact CSE algorithm can be applied on real-size instances.

Since there exist instances that the proposed exact CSE algorithm finds them difficult to obtain the minimum solutions, we also introduce an approximate CSE algorithm that finds competitive results with the minimum solutions and better solutions than those of the previously proposed prominent CSE heuristics.

An interesting and original result shown by the exact CSE algorithm is that the binary representation allows for a greater amount of sharing, hence, producing more area-efficient implementations for the MCM problems than the CSD representation that is commonly preferred. However, we note that when seeking area and delay efficient solutions, the MSD representation should be used.

4.2 Minimum Number of Operations under General Number Representation

In this section, we extend the exact CSE algorithm introduced in Section 4.1.1 to handle multiple constants under general number representation as described in [17]. Since the implementations of a constant are not limited to any number representation in the proposed algorithm, we increase the search space, allowing our algorithm to be significantly more effective in the optimization of the number of operations. To help the search in this larger solution space, we consider the minimization of the number of partial terms model described in Section 4.1.1.3, the problem reduction techniques described in Section 4.1.1.4, and also, the model simplification methods.

4.2.1 Implementations of constants under general number representation

The solution of the exact CSE algorithm depends on number representation, since all possible implementations of constants are extracted from the non-zero digits of the constants representations. While it is true that there is a higher probability of a representation with the 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 the existing partial terms and lead to a better solution.

However, in general number representation, finding the operations that implement a constant has some limitations, since it must be ensured that the obtained solution has no feedback. To illustrate the problem, consider the target constants 7, 11, and 19 to be implemented. The possible implementations of these constants under general number representation are given in Figure 4.9.

Implementations of 7	Implementations of 11	Implementations of 19
$7 = 1 + 6 = 1 + 3_{\ll 1}$	$11 = 1 + 10 = 1 + 5_{\ll 1}$	$19 = 1 + 18 = 1 + 9_{\ll 1}$
$7 = 2 + 5 = 1_{\ll 1} + 5$	$11 = 2 + 9 = 1_{\ll 1} + 9$	$19 = 2 + 17 = 1_{\ll 1} + 17$
...	...	...
$7 = 11 - 4 = 11 - 1_{\ll 2}$	$11 = 7 + 4 = 7 + 1_{\ll 2}$	$19 = 7 + 12 = 7 + 3_{\ll 2}$
...	...	...
$7 = 19 - 12 = 19 - 3_{\ll 2}$	$11 = 19 - 8 = 19 - 1_{\ll 3}$	$19 = 11 + 8 = 11 + 1_{\ll 3}$
...	...	...

Figure 4.9: Implementations of 7, 11, and 19 in general number representation.

If all operations listed in Figure 4.9 are accepted for the target constants, a minimum solution that includes a feedback loop can be obtained. For example, $7 = 11 - 1_{\ll 2}$, $11 = 19 - 1_{\ll 3}$, and $19 = 1_{\ll 3} + 11$. To avoid these feedback loops, only addition operations can be considered or additional constraints that break the loops should be added to the 0-1 ILP problem. Since more promising results are obtained with both addition and subtraction operations and the number of additional constraints grows exponentially with the number of partial terms, neither approaches were used. Instead, for each target constant, t_i, odd numbers between 1 and $2^{\lceil log_2(t_i) \rceil + 1} - 1$ are sorted in ascending order of the number of non-zero digits in their CSD representations in a set called *Nset*. In fact, *Nset* is a set where the odd numbers are ordered according to the number of operations required to implement each single constant when it is defined in CSD. Also, in *Nset*, the constants can also be sorted according to their single constant cost values obtained by the algorithm of [69].[3] After *Nset* is formed, the operations for a target constant are found by traversing from the first element to the element before the target constant in *Nset* and assigning each element to the first input of an addition operation with positive and negative sign. The operation that implements the target constant is accepted, if its second input is placed in a lower position than the position of the target constant in *Nset*. By doing so, the implementations of a target constant that can be considered in the exact CSE algorithm under CSD and MSD representations are also extracted in the exact algorithm under general number representation, and furthermore, the implementations of a target constant that cannot be extracted from the non-zero digits combinations of the constant representation are also obtained. As an example, consider the target constant 51, i.e., $10\bar{1}010\bar{1}$ in CSD. Suppose 23, that is defined in CSD as $10\bar{1}00\bar{1}$ and is located in a lower position than that of 51 in *Nset*, is assigned to the first input of addition operations with positive and negative sign. The operations, $51 = 23 + 7_{\ll 2}$ and $51 = -23 + 37_{\ll 1}$, are accepted for the implementation of 51, since the second inputs, i.e., the partial terms 7 and 37, are located before 51 in *Nset*. For the target constant 51, we note that the partial term 23 cannot be considered in the exact CSE algorithm when the target constant is defined under CSD representation. Also, under general number representation, the partial terms

[3]This approach was also implemented and it was observed that the obtained results were similar to those of the described approach.

may include equal number of non-zero digits, which cannot be encountered in the exact CSE algorithm where the partial terms are simply obtained by decomposing the non-zero digits in the representations of constants. Again, for the implementation of the target constant 51, the operation $51 = 43 + 1_{\ll 3}$ thus, the partial term 43, i.e., $10\bar{1}0\bar{1}0\bar{1}$ in CSD, cannot be considered in the exact CSE algorithm under binary, CSD, or MSD representation. Thus, the use of *Nset* enables us to avoid feedback loops and increases the possible sharing of partial terms by providing more possible implementations of a constant with respect to the number of operations obtained under any particular number representation in the exact CSE algorithm.

4.2.2 The exact algorithm under general number representation

As shown in Lemma 4.3, the minimization of the number of operations in MCM is equivalent to finding the minimum number of partial terms such that each target constant and partial term can be implemented using a single operation whose inputs are target constants, found partial terms, or the input that is the constants are multiplied with, denoted by 1. The exact algorithm under general number representation is based on the model of minimizing the number of partial terms, since this model formalizes the MCM problem as a 0-1 ILP problem with smaller size than the model of minimizing the number of operations. Also, we include model simplification techniques in the network for the minimization of the number of partial terms required to implement the target constants. The proposed exact algorithm follows the same steps of the exact CSE algorithm described in Section 4.1.1.

In the preprocessing phase of the algorithm, after the target constants are made positive and odd, they are stored without repetition in a target set, T, and labeled as unimplemented. The part of the algorithm where the partial terms are found for each element in the target set is as follows:

1. Take an unimplemented element from the target set, t_i. Form an empty set of arrays called P_i associated with t_i. P_i will contain all partial terms that are required to implement t_i.

2. Find an operation that implements t_i;

(a) Find the positive and odd unrepeated inputs of the operation that are neither a target constant nor 1, and store them in an empty array called *Iarray*. Hence, observe that *Iarray* may be an empty set, or it may contain a single partial term or a pair of partial terms.

(b) If *Iarray* is empty, then make P_i empty and go to Step 5. In this case, t_i can be implemented with an operation whose inputs are target constants or 1, and this is the minimum cost implementation of t_i.

(c) If *Iarray* is not empty, then check for each array of P_i, $P_i(k)$, if $P_i(k) \subseteq Iarray$. If *Iarray* is included in P_i, then go to Step 3.

(d) If *Iarray* is not empty, then check for each array of P_i, $P_i(k)$, if $Iarray \subset Pset_i(k)$. If *Iarray* dominates $P_i(k)$, then delete $P_i(k)$.

(e) Add *Iarray* to P_i.

3. Repeat Step 2 until all the implementations of t_i are considered.

4. Add each partial terms of P_i to the target set, if it is not 1 and is not in the target set, and label it as unimplemented.

5. Label t_i as implemented and repeat Step 1 until all elements in the target set are labeled as implemented.

We note that the operations that implement the constants, Step 2 of the algorithm, are found as described in Section 4.2.1. The partial terms required to implement the constants are extracted from the inputs of the operations. Since, there is no need to implement the input that is the constants are multiplied with, denoted by 1, and the aim is to implement the target constants, the partial terms are determined as the constants that are neither 1 nor the target constants. Thus, it is obvious that a pair of partial terms or a single partial term simply represents an operation implementing the constant.

After all partial terms required to implement each target constant and partial term are found, these implementations are represented in a combinational network that includes only AND and OR gates. An OR gate, representing a target constant or a partial term, combines all partial terms that can be used for the synthesis of the associated target constant or partial term. An AND gate, representing a pair of partial

terms, combines two partial terms. The primary inputs of the network are the target constants and partial terms that can be implemented with an operation whose inputs are 1 or target constants. The primary outputs of the network are the outputs of the OR gates associated with the target constants.

After the Boolean network is constructed, additional hardware, i.e., a 2-input AND gate with an optimization variable for each partial term, is added to the network, as done in the minimization of the number of partial terms model. Then, the MCM problem is modeled as a 0-1 ILP problem with a cost function to be minimized and constraints to be satisfied. The cost function is the linear function of the optimization variables that represent the partial terms. The constraints of the 0-1 ILP problem are obtained by assigning the optimization variables that represent the target constants to 1 and expressing each clause in CNF formulas of each gate in the network as a linear inequality.

We note that the proposed algorithm is an exact algorithm under general number representation with its limitations on considering all possible implementations of the constant, since the MCM problem is formalized as a 0-1 ILP problem and the solution is obtained using a generic 0-1 ILP solver. However, the proposed algorithm is not an exact graph-based algorithm since all possible implementations of a constant are not considered due to the limitations described in Section 4.2.1.

4.2.3 Experimental results

In this section, we present the results of the exact algorithm under general number representation on randomly generated and FIR filter instances and compare with the solutions of the exact CSE algorithm when constants are defined under binary, CSD, and MSD representations.

As the first experiment set, randomly generated instances where the number of constants ranges between 10 and 100, and the constants are defined in 12 bit-widths were used. We generated 30 instances for each number of constants. We compare the results of exact algorithm under binary, CSD, MSD, and general number representations in Fig. 4.10.

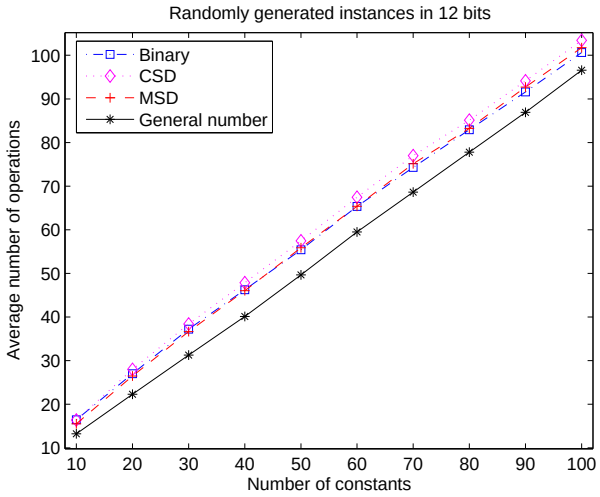


Figure 4.10: Comparison of the solutions obtained under binary, CSD, MSD, and general number representations.

In this experiment, we observe that the maximum difference of average number of operations obtained under binary, CSD, and MSD with respect to solutions obtained under general number representation is 5.8, 8.7, and 6.5 respectively. This clearly shows the advantage of using general number representation over a particular number representation when searching for the maximal sharing of partial terms in the optimization of the number of operations.

As the second experiment set, the FIR filters where the coefficients were computed using the *remez* algorithm in MATLAB were used. The filter specifications are given in Table 4.13.

Table 4.13: Characteristics of the FIR filters.

Filter	pass	stop	#tap	width
1	0.15	0.25	40	12
2	0.20	0.25	80	12
3	0.24	0.25	120	12
4	0.15	0.25	60	14
5	0.15	0.20	60	14
6	0.10	0.15	60	14
7	0.10	0.15	100	16
8	0.15	0.25	120	16
9	0.10	0.15	160	16

The 0-1 ILP problem size of the exact algorithm on filter instances when coefficients are defined under CSD, MSD, and general number representations are presented in Table 4.14.

Table 4.14: 0-1 ILP problem sizes of the FIR filter instances.

Filter	CSD			MSD			General Number		
	vars	cons	optvars	vars	cons	optvars	vars	cons	optvars
1	77	119	50	151	302	80	21231	96222	475
2	61	83	47	92	137	64	28	28	28
3	34	34	34	34	34	34	34	34	34
4	168	345	95	107	146	74	20	20	20
5	241	562	107	203	466	93	29	29	29
6	331	799	137	541	1446	200	17647	77268	556
7	938	3131	259	4009	16037	779	45	45	45
8	511	1271	218	673	1918	239	1722	5489	449
9	1866	6671	467	9510	40050	1384	70	70	70
Total	4227	13015	1414	15320	60534	2947	40826	179205	1706

As can be easily observed from Table 4.14, on some instances, such as Filter 7 and 9, when filter coefficients are defined under general number representation, each filter coefficient can be implemented using a single operation where the inputs are filter coefficients or the filter input. This occurs because the general number representation considers more possible implementations that can cover the coefficients. In this case, there is no need to represent the problem as an optimization problem. Thus, the 0-1 ILP problem size can be much smaller under general number representation with respect to those of 0-1 ILP problems obtained by the exact CSE algorithm under a particular number representation. On the other hand, the 0-1 ILP problem size can be too large under general number representation when partial terms are required to implement the coefficients, since the number of possible implementations of the coefficients are increased with the use of general number representation. For instance, on Filter 1, only two coefficients need to be synthesized with the partial terms.

The results of the exact algorithm under binary, CSD, MSD, and general number representations are given in Table 4.15. In this table, *CPU* is the CPU time in seconds required for the SAT-based 0-1 ILP solver, *minisat+*, to compute the exact solutions on a PC with dual Pentium Xeon at 2.4GHz, with 4GB of main memory, running Linux.

As can be easily observed from Table 4.15, the solutions obtained under general number representation are superior than the solutions obtained under binary, CSD, or MSD representation. We note that the reduction in the number of operations obtained by using general number representation is 9%, 10%, and 8% on average, and up to a

Table 4.15: Summary of the results of the exact algorithm under different number representations on the FIR filter instances.

Filter	Binary			CSD			MSD			General Number		
	adder	step	CPU	adder	step	CPU	adder	step	CPU	adder	step	CPU
1	17	5	0	16	3	0	16	3	0	15	4	1.2
2	29	5	0	29	3	0	29	4	0	28	3	0
3	35	5	0	34	3	0	34	3	0	34	3	0
4	23	4	0.1	23	3	0	22	3	0	20	4	0
5	32	5	0	35	4	0	34	3	0	29	4	0
6	34	5	1.2	35	4	0	33	3	0.1	29	5	0.1
7	51	5	10.8	51	4	0.2	49	4	12.8	45	5	0
8	53	5	1.7	54	4	0.3	53	4	0.2	48	4	0.4
9	75	5	3.2	77	5	4.7	77	4	63.9	70	4	0
Total	349	44	17.0	354	33	5.2	347	31	77.0	318	36	1.7

maximum of 15%, 17%, and 15%, according to binary, CSD, and MSD representation respectively. Also, we note that despite a larger search space in general number representation, the exact solution is found using very low computational effort by *minisat+*. Observe that although the solutions including less number of operations are obtained under general number representation, the delay of the solutions is increased with respect to the solutions found by the exact CSE algorithm when coefficients are defined under CSD and MSD representation.

4.2.4 Conclusions

In this section, we extend the exact CSE algorithm described in Section 4.1.1 to handle the constants under general number representation. In this algorithm, the possible implementations of the constants are extracted from the decimal values of the constants rather than the non-zero digits of the constant representation in the exact CSE algorithm. Thus, the number of possible implementations of a constant, consequently, the possible sharing of partial terms, is increased. To deal with the increased search space, we also include the problem reduction and model simplification techniques in the exact algorithm. It is shown by the experimental results that the exact algorithm under general number representation can be applied on real-size instances and finds better solutions than those of the exact CSE algorithm under a particular number representation.

4.3 Graph-based Algorithms

In this section, initially, we give the main concepts in graph-based algorithms and redefine the MCM problem. Then, we introduce the exact graph-based algorithm [18] designed for the MCM problem and then, we present the approximate graph-based algorithm [19] that can handle more complex instances.

4.3.1 Preliminaries

In the graph-based algorithms, the main operation, called *A-operation* in [88], is an operation with two integer inputs and one integer output that performs a single addition or a subtraction, and an arbitrary number of shifts. It is defined as follows:

$$w = A(u,v) = |(u \ll l_1) + (-1)^s (v \ll l_2)| \gg r = |2^{l_1}u + (-1)^s 2^{l_2} v| 2^{-r} \quad \textbf{(4.12)}$$

where $l_1, l_2 \geq 0$ are integers denoting left shifts, $r \geq 0$ 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 Figure 4.11. Recall that in the MCM problem, the complexity of an adder and a subtracter is assumed to be equal in hardware. It is also assumed that the sign of the constant can be adjusted at some part of the design and the shifting operation has no cost, since shifts can be implemented with only wires in hardware. Thus, in the MCM problem, only positive and odd constants are considered. Observe from (4.12) that in the implementation of an odd constant considering odd constants at the inputs, one of the left shifts, l_1 or l_2, is zero and r is zero, or l_1 and l_2 are zero and r is greater than zero. We note that in CSE algorithms, the latter case is not taken into account, since the implementations of a positive and odd constant are extracted from the non-zero digit combinations of the constant defined under a number representation. However, the latter case allows to consider more possible implementations of a constant thus, enables the graph-based algorithms to find a solution with the fewest number of operations. For instance, suppose the target constants 27, 41, and 67 to be implemented. Note that the target constants require intermediate constant(s), or partial term(s), to be synthesized. At a decision level, suppose the intermediate constant 33 is considered. Then, the operations, $33 = 1_{\ll 5} + 1$, $41 = 33 + 1_{\ll 3}$, $67 = 33_{\ll 1} + 1$, and $27 = (41 + 67)_{\gg 2}$,

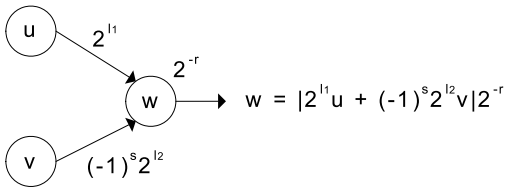


Figure 4.11: The representation of the A-operation in a graph.

that implement the target and intermediate constants can be obtained. Observe that the target constant 27 is synthesized with an operation that includes odd constants at its inputs that cannot be considered in the exact CSE algorithm.

In finding an operation to implement a constant, it is necessary to constrain the left shifts, l_1 and l_2, otherwise a constant can be implemented in infinite ways. As shown in [87], it is sufficient to limit the shifts by the maximum bit-width of the constants to be implemented, i.e., *bw*, and allowing larger shifts than *bw* does not improve the solutions obtained with the former limits. In the proposed exact algorithm and also, in the algorithm of [88], the number of shifts is allowed to be at most $bw+1$.

Thus, the MCM problem can also be defined as follows:

Definition 4.1: THE MCM PROBLEM. Given the target set including the positive and odd unrepeated target constants, $T = \{t_1, \ldots, t_m\} \subset \mathbb{N}$, find the smallest ready set $R = \{r_0, r_1, \ldots, r_n\}$ with $T \subset R$ such that $r_0 = 1$ and for all r_k with $1 \leq k \leq m$, there exist r_i, r_j with $0 \leq i, j < k$ and an operation $r_k = A(r_i, r_j)$.

Hence, the number of operations required to implement the MCM is $|R| - 1$ [88].

4.3.2 The exact graph-based algorithm

The MCM problem is to find the minimum number of intermediate constants such that each constant, target and intermediate, can be implemented with an operation as given in (4.12) where *u* and *v* are 1, target, or intermediate constants. It is obvious that the minimum number of intermediate constants, thus the minimum number of operations solution of the MCM problem, can be found using a breadth-first search [18]. In the preprocessing phase of the exact algorithm, 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

Algorithm 4.1 The exact 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, bw)

1: $R \leftarrow \{1\}$
2: (R, T) = Synthesize(R, T)
3: **if** $T = \emptyset$ **then**
4: **return** R
5: **else**
6: $n = 1$, $W_{R_1} \leftarrow R$, $W_{T_1} \leftarrow T$
7: **while** 1 **do**
8: $m = n$, $\mathrm{X_R} = \mathrm{W_R}$, $\mathrm{X_T} = \mathrm{W_T}$
9: $n = 0$, $\mathrm{W_R} = \mathrm{W_T} = [\,]$
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: $X_{R_i} \leftarrow X_{R_i} \cup \{j\}$
16: $n = n + 1$
17: (W_{R_n}, W_{T_n}) = Synthesize(X_{R_i}, X_{T_i})
18: **if** $W_{T_n} = \emptyset$ **then**
19: **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)

given in Algorithm 4.1, the ready set that includes the minimum number of elements is computed.

In *BFSearch* function, initially, the target constants that can be implemented with the elements of the ready set, $\{1\}$, are found iteratively and removed to the ready set using the *Synthesize* function, i.e., the lines 1-2, as done in the optimal parts of RAG-n and Hcub. If there is no element left in the target set, then the minimum number of operations solution is obtained. Otherwise, the intermediate constants to be added to the ready set are considered exhaustively in the infinite loop, i.e., the line 7 of the algorithm, until all the target constants are synthesized. The infinite loop starts with

the 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 including ready and target sets as a pair is denoted by n. Then, in the infinite loop, another array X is assigned to the array W and its size is represented with m. In an iteration of the infinite loop, for each ready set of the array X, the possible intermediate constants are found and added to the associated ready set forming new ready sets. The possible intermediate constants are determined from 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, i.e., the lines 11-14 of *BFSearch* function. Note that there is no need to consider the 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. After the intermediate constant is added to the ready set X_{R_i}, its implications on the target set X_{T_i} are found by the *Synthesize* function and the modified ready and target sets are stored to the array W as a new pair, i.e., the line 17 of *BFSearch* function.

The flow of the algorithm in two iterations is sketched in Figure 4.12 indicating the array W at the end of iterations. In this figure, the edges labeled with the intermediate constants represent the inclusions of constants to the ready set. An intermediate constant is denoted by ic_{abc} where a, b, and c denote the number of iteration it is included, the index of the ready set it is added, and its index in the iteration respectively.

Observe from Figure 4.12 that the exact algorithm explores the search space in a breadth-first manner. In each iteration, each ready set is augmented with a single intermediate constant. For example, while the ready set W_{R_1} at the end of the second iteration includes 1, the intermediate constants ic_{111}, ic_{211}, and the target constants that can be implemented with the elements of W_{R_1}, the associated target set W_{T_1} consists of the target constants have not been implemented by the elements of W_{R_1} so far. Hence, when there is no element left in a target set, the minimum number of operations solution is obtained with the associated ready set, i.e., the lines 18-19 of *BFSearch* function.

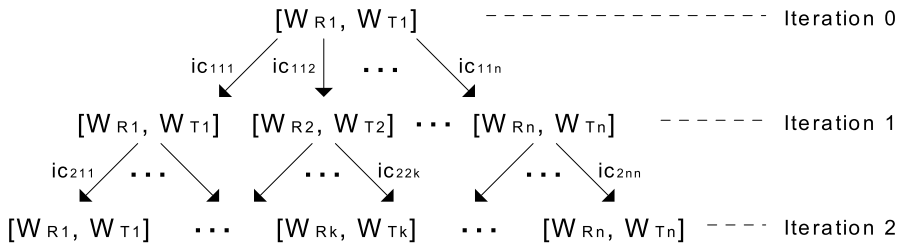


Figure 4.12: The flow of the exact algorithm in two iterations.

We make a simple observation on the exact graph-based algorithm. In this observation, $|T|$ denotes the number of unrepeated positive and odd target constants to be implemented.

Lemma 4.4: The solution obtained by the Algorithm 4.1 yields the minimum number of operations solution.

If a solution is returned on the line 4 of the *BFSearch* function, then no intermediate constant is required to implement the target constants. Hence, each target constant can be implemented using a single operation whose inputs are 1 or target constants as ensured by the *Synthesize* function. In this case, the number of required operations to implement the target constants is $|T|$. Because the target constants cannot be implemented using less than $|T|$ operations as shown in [87], the obtained ready set yields the minimum solution.

If a solution is returned on the line 19 of the *BFSearch* function, then intermediate constant(s) 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 constant(s). Because each element of the ready set, except 1, is guaranteed to be implemented using a single operation and all possible intermediate constants are considered in a breadth-first manner, the obtained ready set yields the minimum number of operations solution. □

As can be easily observed from Lemma 4.4, after the ready set including minimum number of intermediate constants is obtained by the *BFSearch* algorithm, the minimum number of operations implementation of the MCM problem can be realized by synthesizing the target and intermediate constants using a single operation whose inputs are 1, target, or intermediate constants as given in (4.12).

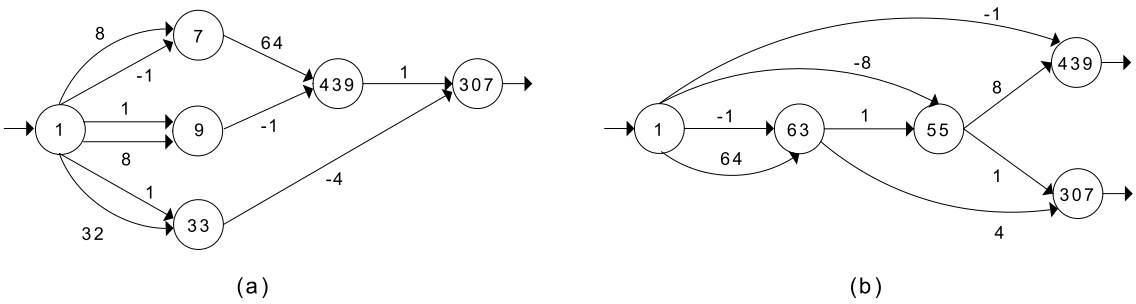


Figure 4.13: The results of algorithms for the target constants 307 and 439: (a) 5 operations with Hcub; (b) 4 operations with the exact algorithm.

As a small example, suppose the target set including 307 and 439. Figure 4.13 presents the solutions obtained by Hcub and the exact graph-based algorithm. As can be easily observed from Figure 4.13(a), since Hcub synthesizes each target constant in an iteration by including an intermediate constant, the intermediate constants included for the implementation of target constants in previous iterations may not be shared in the implementation of target constants in later iterations, although Hcub is particularly designed for this case. In the exact graph-based algorithm, initially, it is observed that the target constants cannot be implemented using a single operation whose inputs are the elements of the ready set, i.e., $\{1\}$. Then, in the first iteration, the intermediate constants that can be implemented using a single operation with the elements of the ready set $\{1\}$, i.e., $3, 5, \ldots, 1023$, are found. However, all the possible ready sets including one intermediate constant, i.e., $\{1,3\}, \{1,5\}, \ldots, \{1,1023\}$, also cannot synthesize all the target constants. In the second iteration, for each ready set obtained in the first iteration, the intermediate constants that can be implemented with the elements of the associated ready set are found and added to the associated ready set. As can be observed from Figure 4.13(b), all the target constants are synthesized when the intermediate constant 55 is added to the ready set $\{1,63\}$, i.e., one of the ready sets obtained in the first iteration of the exact algorithm.

The complexity of search space in the exact graph-based algorithm is dependent on both the number of considered ready sets and the maximum bit-width of the target constants, i.e., *bw*, since the number of considered ready sets increases as *bw* increases. Table 4.16 presents the number of ready sets exploited by the exact graph-based algorithm including up to 4 intermediate constants when *bw* is in between 8 and 14. The exponential growth of the search space can be clearly observed

Table 4.16: Upper bounds on the number of ready sets exploited by the exact graph-based algorithm under different bit-widths.

bw	#ready sets considered in iterations				
	1	2	3	4	Total
8	15	378	12,398	1,668,403	1,681,194
9	17	504	20,118	5,897,424	5,918,063
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	1,188	80,907	458,873,308	458,955,428
14	27	1,404	105,462	1,230,677,125	1,230,784,018

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.

We note that the complexity of the search space also depends on the target constants to be implemented in an MCM instance. There are cases where multiple constants may reduce the complexity of the search space. For example, consider the single target constant 981 defined in 10 bit-width. The minimum number of operations implementation of 981 requires four operations, i.e., $3 = 1_{\ll 2} - 1$, $5 = 1_{\ll 2} + 1$, $43 = 5_{\ll 3} + 3$, and $981 = 1_{\ll 10} - 43$, thus three intermediate constants, 3, 5, and 43. To find this minimum solution, in the worst case, a total of 31905, i.e., $19 + 648 + 30428$, ready sets must be considered. Now, suppose the multiple target constants 43 and 981. In this case, the minimum number of operations solution is found in two iterations with the ready set, $\{1,3,5\}$, including two intermediate constants, i.e., in the worst case, $19 + 648 = 667$ ready sets are exploited. As can be observed from this example, the number of ready sets exploited by the exact algorithm depends heavily on the target constants to be implemented. Hence, the exact algorithm can be efficiently applied on instances including large number of constants as shown in Section 4.3.4. Also, note that the minimum solution is generally obtained before the total number of ready sets are considered. Hence, Table 4.16 presents the upper bounds on the number of ready sets exploited by the exact graph-based algorithm. We note that the exact algorithm can obtain the minimum solutions of the MCM instances that require less than 5 intermediate constants in a reasonable time.

4.3.3 The approximate graph-based algorithm

In this section, we introduce an approximate algorithm [19] based on the exact graph-based algorithm described in the previous section that can be applied on large size instances. The preprocessing phase of the approximate algorithm is similar to that of the exact algorithm, where the target constants are made positive and odd, added to the target set, T, without repetition, and the maximum bit-width of the target constants, bw, is determined. The main part of the approximate algorithm is given in Algorithm 4.2.

In the *ApproximateSearch* function, initially, the ready set including only 1 is formed as given on the line 1 of the algorithm. Then, the target constants that can be implemented with the elements of the ready set using a single operation are found iteratively and removed to the ready set using the *Synthesize* function. If there exist unimplemented constant(s) in the target set, then in each iteration of the infinite loop, i.e., the line 6 of the algorithm, an intermediate constant is added to the ready set until there is no element left in the target set. The approximate 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 as possible intermediate constants, as seen on the lines 7-10 of the algorithm. Note that the ready and target sets denoted by A and B represent the working ready and target sets respectively. Then, each possible intermediate constant is added to the working ready set and its implications on the current target set are found by the *Synthesize* function. If there exist unimplemented target constants in the working target set, the implementation cost of the unimplemented target constants is found in terms of the single constant cost evaluated in [69] and is assigned to the cost value of the intermediate constant, as given on line 17 of the algorithm. After the cost value of each intermediate constant is found, the one with the minimum cost is chosen to be added to the current ready set and the target constants that can be implemented with the elements of the 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

Algorithm 4.2 The approximate 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, bw)

```
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 then
9:                 (A, B) = Synthesize(R, {j})
10:                if B = ∅ then
11:                    A ← A ∪ {j}
12:                    (A, B) = Synthesize(A, T)
13:                    if B = ∅ then
14:                        A = RemoveRedundant(A)
15:                        return A
16:                    else
17:                        cost_j = EvaluateCost(B)
18:        Find the intermediate constant, ic, with the minimum cost among all possible
           intermediate constants, j.
19:        R ← R ∪ {ic}
20:        (R, T) = Synthesize(R, T)
```

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 ← R ∪ {t_k}
7:             T ← T \ {t_k}
8: until isadded = 0
9: return (R, T)
```

EvaluateCost(T)

```
1: cost = 0
2: for k = 1 to |T| do
3:     cost = cost + SingleConstantCost(t_k)
4: return cost
```

RemoveRedundant(R)

```
1: for k = 1 to |R| do
2:     if r_k is an intermediate constant then
3:         R ← R \ {r_k}
4:         (R, T) = Synthesize({1}, R)
5:         if T ≠ ∅ then
6:             R ← R ∪ {r_k}
7: return R
```

is applied on the final ready set to remove the 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, is synthesized with a single operation whose inputs are the elements of the ready set.

We make some simple observations on the approximate algorithm. In these observations, again, $|T|$ denotes the number of unrepeated positive and odd target constants to be implemented, i.e., the lowest bound on the minimum number of operations solution.

Lemma 4.5: If the approximate 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. Because the elements of the target set cannot be synthesized using less than $|T|$ operations as shown in [87], if the approximate algorithm finds a solution including $|T|$ operations, then the found solution is the minimum solution. □

Lemma 4.6: If the approximate algorithm 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. Because the case described in Lemma 4.5 is checked on the lines 2-3 of the algorithm, if there exist unimplemented target constants, then the minimum solution requires at least one intermediate constant. So, if the solution found by the approximate algorithm includes $|T|+1$ operations, then it is the minimum solution. □

Lemma 4.7: If the approximate 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. Because the case described in Lemma 4.5 is checked on the lines 2-3 of the algorithm and the case described in Lemma 4.6 is explored exhaustively on the line 7 of the algorithm, if there exist unimplemented target constants at the end of the first iteration, then the minimum solution requires at least two intermediate constants. So, if the solution found by the approximate algorithm includes $|T|+2$ operations, then it is the minimum solution. □

It is obvious that if the approximate algorithm finds a solution including more than $|T|+2$ operations, then the approximate algorithm cannot guarantee the found solution is minimum, since all possible intermediate constant combinations including more than two constants are not explored exhaustively in the algorithm. However, observe that the bound on the minimum number of operations solution determined by the approximate algorithm can be increased when the exhaustive search is applied on these cases.

Recall from Lemmas 3.1 and 3.2 that the graph-based heuristics RAG-n [87] and Hcub [88] can guarantee the minimum solution if the solutions found by these heuristics include the number of operations up to $|T|+1$.

The following conclusion can be drawn from the Lemmas 4.5-4.7.

Lemma 4.8: If the approximate 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 approximate algorithm finds a solution including more than $|T|+2$ operations. Since the cases described in Lemma 4.5 and 4.6 are searched exactly and the case described in Lemma 4.7 is not explored exhaustively in the approximate algorithm, the solution of the MCM problem cannot include $|T|$ and $|T|+1$ operations, and may include $|T|+2$ operations. Hence, in this case, the lower bound on the minimum number of operations solution is $|T|+2$. □

We note that when RAG-n and Hcub cannot guarantee the minimum solution, they can ensure that the lower bound on the minimum number of operations solution is $|T|+1$.

Also, note that when the approximate algorithm cannot guarantee its solution as the minimum solution, the upper bound on the minimum number of operations solution can also be determined as the number of operations in its solution. Thus, the solution of the approximate algorithm can be used to obtain highly constricted lower and upper bounds on the minimum number of operations solution such that an exact depth-first algorithm can find the minimum solution in a reasonable time by searching in a narrow search space.

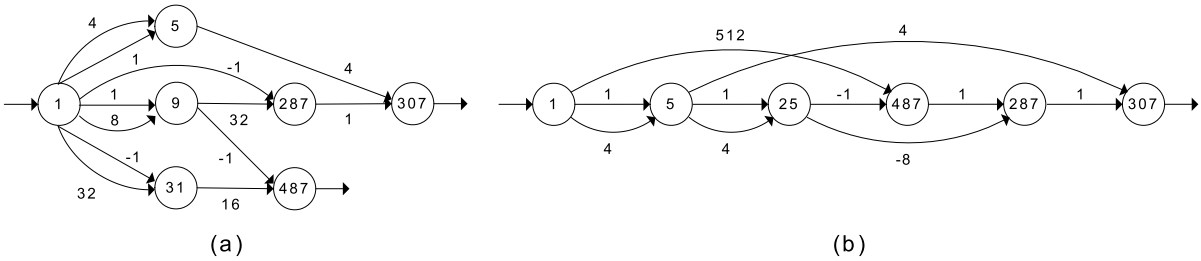


Figure 4.14: The results of algorithms for the target constants 287, 307, and 487: (a) 6 operations with Hcub; (b) 5 operations with the approximate algorithm.

As a small example on the approximate algorithm, suppose the target set including 287, 307, and 487. Figure 4.14 presents the results obtained by Hcub and the approximate algorithm. Observe from Figure 4.14(a) that since Hcub synthesizes target constants once at a time by including intermediate constants, it may find a worse solution than the approximate algorithm. On the other hand, in each iteration of the approximate algorithm, an intermediate constant that can be implemented with the elements of the current ready set is added to the ready set. On this example, the intermediate constants 5 and 25 are added to the ready set in the first and second iterations respectively, Figure 4.14(b). The intermediate constant chosen in each iteration is the constant that implements more not-yet synthesized target constants with the elements of the current ready set using a single operation. Note that the target constants that can be implemented with the elements of the current ready set are removed from the target set to the ready set in each iteration. Hence, the approximate algorithm may obtain better solutions than Hcub. Observe from Lemma 4.7 that the approximate algorithm also ensures the minimum solution on this example.

We also note that in the previously proposed graph-based heuristics, once intermediate constant(s) is selected for the implementation of a single target constant in one iteration, it cannot be reverted although new intermediate constants are included in later iterations. Hence, the final solution of the prominent graph-based heuristics may include redundant intermediate constants. For example, consider the target constants 287 and 411 to be implemented. The solution of Hcub is presented in Figure 4.15(a) including four operations with the intermediate constants 9 and 31. However, as can be easily observed from Figure 4.15(b), the intermediate constant 9 is redundant, determined by the *RemoveRedundant* function, since the target constants 287 and 411 can be synthesized with the intermediate constant 31 only. Thus, by

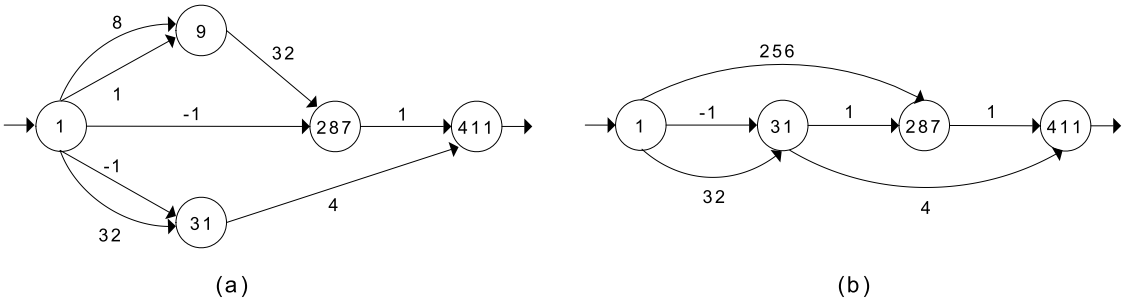


Figure 4.15: The implementations of the target constants 287 and 411: (a) 4 operations with Hcub; (b) 3 operations after using the *RemoveRedundant* function.

using the *RemoveRedundant* function in the approximate algorithm, the redundant intermediate constants can be eliminated from the final solution yielding a fewer number of operations solution.

4.3.4 Experimental results

In this section, we present the results of the exact and approximate graph-based algorithms. Initially, we compare the exact graph-based algorithm with the exact CSE algorithm described in Section 4.1.1 and the exact algorithm under general number representation given in Section 4.2. Then, the exact and approximate graph-based algorithms are compared with the previously proposed graph-based heuristics of [87] and [88]. The graph-based heuristics were obtained from [91].

As the first experiment set, we used randomly generated instances where constants were defined under 12 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. Figure 4.16 compares the solutions obtained by the exact algorithms when constants are defined under binary, CSD, MSD, and general number representations with the minimum number of operations solutions obtained by the exact graph-based algorithm.

As can be easily observed from Figure 4.16, the solutions obtained by the exact CSE algorithm are far from the minimum number of operations solutions, since the implementations of constants in the exact CSE algorithm are restricted to the number representation. 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 5.7, 7.5, and 5.8 respectively on overall 300 instances. Also,

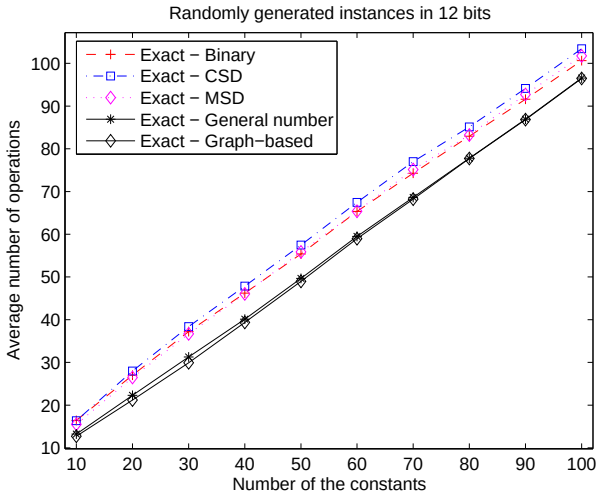


Figure 4.16: Comparison of the solutions of the exact CSE algorithm and exact algorithm under general number representation with the minimum number of operations solutions.

observe that the exact algorithm under general number representation obtains similar results with the minimum number of operations solutions. However, since the exact algorithm under general number representation cannot consider the whole search space as the exact graph-based algorithm, it may yield suboptimal solutions. We note that the average difference of the number of operations solutions between the exact algorithm under general number representation and the exact graph-based algorithm is 0.6 on overall 300 instances.

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 4.17. We note that Filter 11 was used as an example filter in [8,64].

Table 4.17: Characteristics of the FIR filters.

Filter	pass	stop	#tap	width
1	0.10	0.15	40	14
2	0.10	0.15	80	16
3	0.10	0.25	30	14
4	0.10	0.25	80	16
5	0.10	0.20	40	14
6	0.10	0.20	80	16
7	0.15	0.25	40	14
8	0.15	0.25	60	16
9	0.20	0.25	40	14
10	0.20	0.25	60	16
11	0.25	0.30	25	12

Table 4.18: Summary of results of the graph-based algorithms on the FIR filter instances.

Filter	$\|T\|$	LBs [64]		RAG-n [87]		Hcub [88]		Approximate		Exact		
		adder	step	adder	step	adder	step	adder	step	adder	step	CPU
1	19	20	3	24	10	23	7	22	9	22	8	126.8
2	39	40	3	44	9	42	8	41	10	41	9	25.3
3	14	14	3	19	5	16	5	16	5	16	5	42.3
4	33	33	3	37	5	34	5	34	5	34	5	1.1
5	18	19	3	22	5	20	5	20	5	20	5	4.5
6	36	37	3	40	5	38	5	37	6	37	6	0.6
7	19	19	3	22	5	21	7	21	7	21	5	2.9
8	29	29	3	33	7	31	7	31	7	31	7	17.5
9	19	20	3	25	5	21	6	21	7	21	7	28.9
10	29	30	3	34	6	31	7	31	7	31	7	20.4
11	13	14	3	17	9	16	7	16	8	16	8	210.8
Total	268	275	33	317	71	293	69	290	76	290	72	481.1

Table 4.18 presents the results of the graph-based algorithms. In this table, $|T|$ denotes the number of positive and odd unrepeated filter coefficients, i.e., the lowest bound on the number of operations, and *LBs* indicates the lower bounds on the number of operations and the number of adder-steps, obtained by the formulas given in [64]. Also, *CPU* denotes the required CPU time in seconds of the exact algorithm implemented in MATLAB to obtain the minimum solution on a PC with 2.4GHz Intel Core 2 Quad CPU and 3.5GB memory. We note the solutions of the heuristics and the approximate algorithm are obtained in a few seconds.

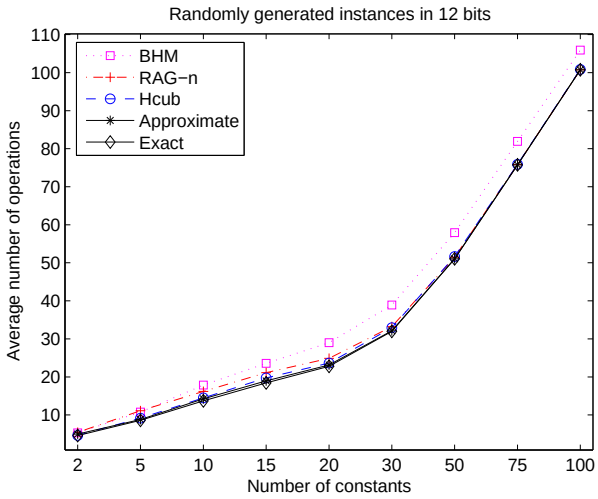
As can be easily observed from Table 4.18, the exact algorithm finds the minimum number of operations solutions with a little computational effort, since the minimum solutions require at most three extra intermediate constants for the implementation of filter coefficients. Observe that the CPU time required to find the minimum solution increases, as the minimum number of the required intermediate constants increases. Note that the approximate algorithm obtains solutions same as those of the exact algorithm in terms of the number of operations. According to the Lemmas 4.5, 4.6, and 4.7, the approximate algorithm guarantees the minimum solution on 9 filter instances. The filter instances that the approximate algorithm cannot guarantee the minimum solutions are Filter 1 and 11. However, the solutions of the approximate algorithm on these filter instances are the minimum solutions ensured by the solutions of the exact algorithm on these instances. Also, we note that Hcub finds similar results with the exact algorithm, but it obtains worse solutions on Filter 1, 2, and 6, and Hcub determines only its solution on Filter 4 as the minimum solution. On the other hand,

RAG-n obtains suboptimal results on all filter instances that are far from the minimum solutions. Also, observe that the lower bound on the minimum number of required operations can only be used to determine the solution of the approximate algorithm on Filter 6 as the minimum solution, although it is also proven to be minimum by the given lemmas. This is because the formula given in (3.7) computes a lower bound on the minimum number of operations close to the lowest bound, i.e., $|T|$. Thus, this experiment indicates that an exact algorithm is indispensable to ensure the minimum solution.

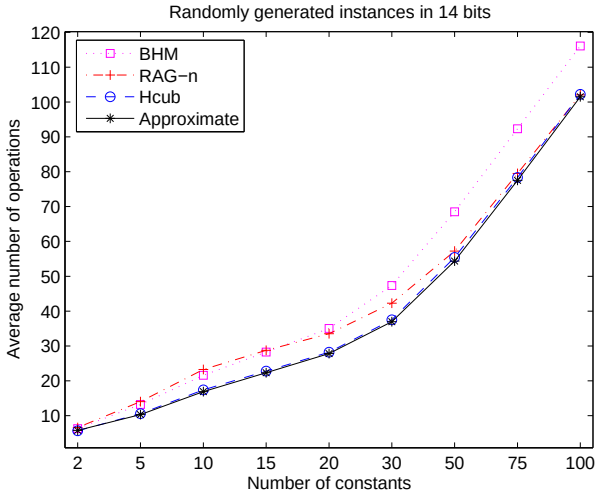
As can be observed from Table 4.18, the approximate algorithm finds the fewest number of operations solution of a filter instance in a greater number of adder-steps with respect to its lower bound, indicating, in general, the traditional tradeoff between area and delay. This is simply because the sharing of intermediate constants in MCM generally increases the logic depth of constant multiplications as shown in [5]. However, we note that the proposed approximate algorithm can be easily modified to find the fewest number of operations solution under a delay constraint as described in [5, 8]. In this case, only the intermediate constants that do not violate the delay constraint must be considered in the algorithm.

As the third experiment set, we used randomly generated instances where the constants were defined in between 10 and 16 bit-width. We tried to generate hard instances to distinguish the algorithms clearly. Hence, under each bit-width, i.e., 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 1890 instances. Figure 4.17 presents the results of the algorithms only on randomly generated hard instances defined under 12, 14, and 16 bit-width.

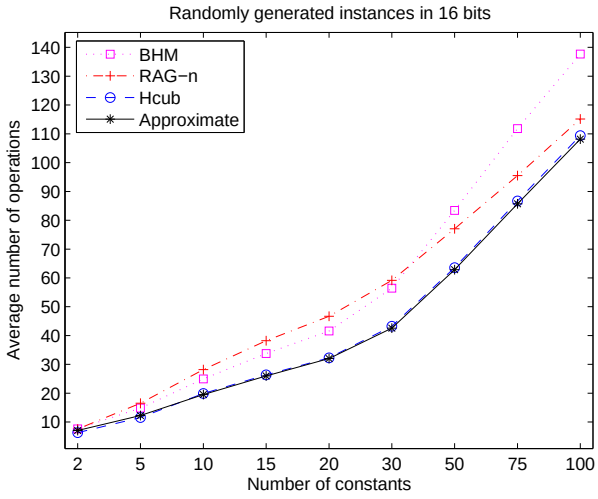
We note that the exact graph-based algorithm was applied on randomly generated hard instances defined in 10, 11, and 12 bit-width, since the solutions of all these instances can be obtained in a reasonable time. As can be easily observed from Figure 4.17(a), the graph-based heuristics, except BHM, obtain competitive results with those of the exact algorithm. However, we note that on the instances with 30 constants defined in 12 bits, the difference of the average number of operations between the approximate and exact algorithms is 0.5, and this value on the same instances between Hcub and



(a)



(b)



(c)

Figure 4.17: Results of graph-based algorithms on randomly generated hard instances: (a) Constants in 12 bits; (b) Constants in 14 bits; (c) Constants in 16-bits.

Table 4.19: Summary of results of the graph-based algorithms on randomly generated hard instances.

$>$	**BHM** [87]	**RAG-n** [87]	**Hcub** [88]	**Approximate**
BHM [87]	0	410	3	15
RAG-n [87]	1209	0	101	15
Hcub [88]	1751	1215	0	173
Approximate	1738	1319	688	0

the exact algorithm is almost 1. Also, on the instances with 15 constants defined in 12 bits, the difference of the average number of operations between RAG-n and the exact algorithm is 2.7.

As can be easily observed from Figure 4.17(b)-(c), the approximate and Hcub algorithms obtain significantly better results than RAG-n and BHM as the bit-width increases. Also, observe that the approximate algorithm obtains better solutions than Hcub as the number of constants increases. For example, while the difference of the average number of operations between the approximate algorithm and Hcub is 0.7 on the instances with 2 constants defined in 16 bits, this value between Hcub and the approximate algorithm reaches to 1.2 on the instances with 100 constants defined in 16 bits. This is because the number of considered intermediate constants is increased with the number of constants to be implemented, yielding better solutions in the approximate algorithm.

The results of graph-based heuristic algorithms on overall 1890 instances are summarized in Table 4.19 where $X>\mathbf{Y}$ denotes the number of instances that the algorithm X finds better solutions than the algorithm $\mathbf{Y}$. As can be easily observed from Table 4.19, the number of instances that the approximate algorithm finds better solutions than Hcub is 688, while the number of instances that Hcub obtains better solutions than the approximate algorithm is 173 on overall 1890 instances. When the approximate algorithm is compared with BHM and RAG-n, it finds better solutions than these heuristics on 1738 and 1319 instances respectively. We also note that the number of instances that the approximate algorithm guarantees the minimum solution is 701, i.e., 37% of the experiment set, and the number of instances that RAG-n and Hcub ensure the minimum solution is 394 and 386 respectively on overall 1890 instances.

This experiment clearly indicates that an approximate algorithm that guarantees the minimum solution on more instances is indispensable to ensure the minimum solution of the MCM problem where an exact algorithm cannot be applied.

4.3.5 Conclusions

In this section, we introduce an exact graph-based algorithm that searches the minimum number of operations solution of the MCM problem in a breadth-first manner. Unlike the exact CSE algorithm, the proposed exact graph-based algorithm is independent from the number representation used for constants. The experimental results show that the exact graph-based algorithm can be efficiently applied on low complex instances of real size FIR filters.

Also, we present an approximate graph-based algorithm that finds the fewest number of intermediate constants such that the target and intermediate constants can be synthesized using a single operation at the end of the algorithm, rather than synthesizing the target constants once at a time by including intermediate constant(s). The design of the approximate algorithm in this scheme allows the algorithm to guarantee the minimum solution on more instances than the previously proposed graph-based heuristics. It is shown by the experimental results that the proposed approximate algorithm finds competitive and better solutions than the previously proposed prominent graph-based heuristics.

As future work, we are currently working on the implementation of an exact depth-first search algorithm that takes the upper and lower bounds from the solution of the approximate algorithm and seeks the optimal solution in a highly constricted search space.

5. OPTIMIZATION OF AREA UNDER A DELAY CONSTRAINT

In many designs, particularly in DSP systems, performance is an important and crucial parameter. Hence, circuit area is generally expendable in order to achieve a given performance target. In this chapter, we address the problem of finding the fewest number of operations implementation of MCM under a delay constraint. Although the delay is dependent on several implementation issues, such as circuit technology, placement, and routing, the delay in MCM is generally considered as the number of adder-steps, which denotes the maximal number of adders/subtracters in series to produce any constant multiplication [5].

The number of adder-steps of the MCM implementation has a significant impact on the switching speed, consequently, the power consumption due to the switching [8]. Because, longer paths in the design allow glitches to propagate via many operations. Hence, the implementation of the MCM with the minimum number of adder-steps also leads a design that consumes lower power.

In this chapter, initially, we present the background concepts and give the problem definition. Then, we introduce exact and approximate CSE algorithms designed for the minimization of operations under a delay constraint. We note that since the definition of adder-steps is identical to the definition of levels in combinational circuits, we use both definitions interchangeably in this chapter.

5.1 Background

A single constant represented 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, or in a chain of operations with the maximum latency, i.e., $n-1$ adder-steps. For example, the constant multiplication $23x$, defined in binary as $(10111)_{bin}x$, 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 adder-steps as shown in Figure 5.1.

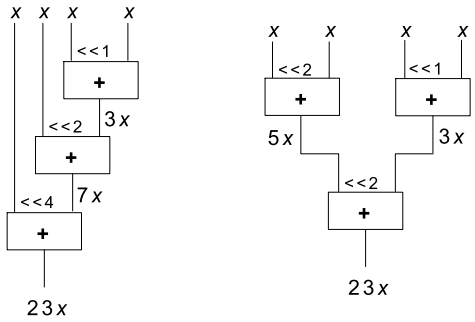


Figure 5.1: Two implementations of $23x$: (a) $23x = 2^4x + (2^2x + (2^1x + x))$, with three adder-steps; (b) $23x = (2^4x + 2^2x) + (2^1x + x)$, with two adder-steps.

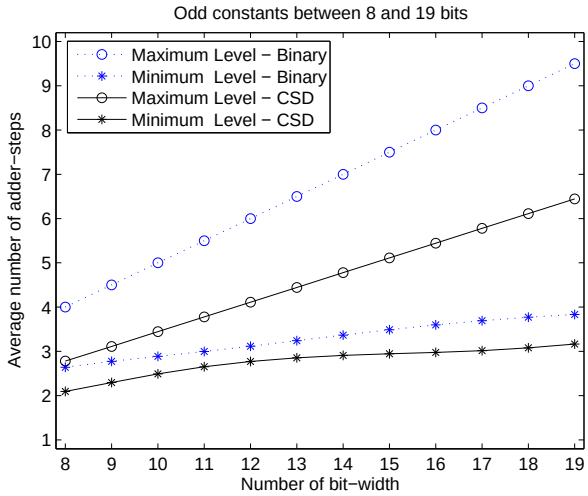


Figure 5.2: Comparison of the number of adder-step of constants between 8 and 19 bit-width defined in binary and CSD.

The effect of number representation on the number of adder-steps of a single constant multiplication is presented in Figure 5.2 that compares the average minimum and maximum number of adder-steps of odd constants between 8 and 19 bit-width when they are represented in binary and CSD. As can be easily observed from Figure 5.2, the use of binary representation in SCM yields greater delay than CSD, since the binary representation of a constant generally includes more non-zero digits than those of CSD.

Obviously, the maximum of the minimum number of adder-steps of each constant in an MCM problem defines the minimum delay of the multiple constant multiplications. Hence, for a target set, $T = \{t_1, t_2, t_3, \ldots, t_m\}$, the lowest bound on the number of

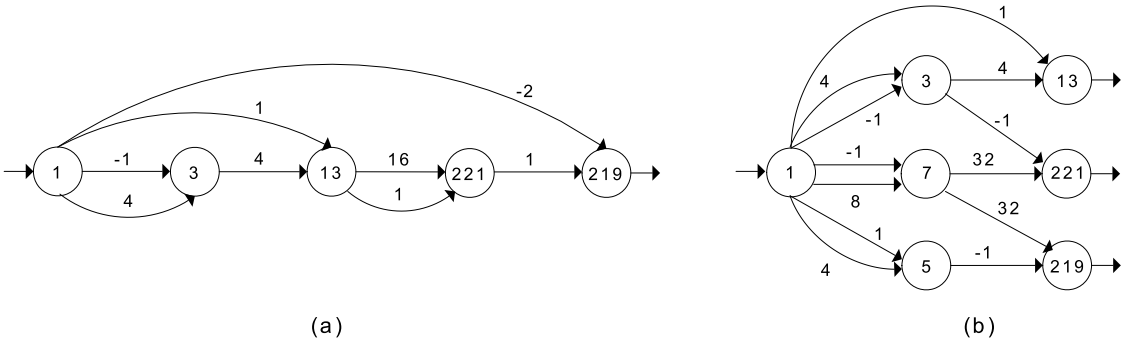


Figure 5.3: The implementation of the target set $\{3, 13, 219, 221\}$: (a) with 4 adder-steps; (b) with the minimum number of adder-steps.

adder-steps in MCM [5, 64] is determined as

$$\#adder - step_{lb,MCM} = \max_{t_i}\{\lceil log_2 S(t_i)\rceil\} \tag{5.1}$$

where $S(t_i)$ is the number of non-zero digits in the CSD representation of the target constant t_i.

As an example, consider a set of target constants, $T = \{3, 13, 219, 221\}$, to be implemented. We note that the minimum number of adder-steps of the MCM implementation is 2 as computed by (5.1). Figure 5.3(a) presents the minimum number of operations solution. Observe that the solution includes 4 operations with 4 adder-steps. However, the minimum number of operations solution under the minimum number of adder-steps constraint given in Figure 5.3(b) includes 6 operations with 2 adder-steps. Observe that the minimum number of operations solution under the minimum number of adder-steps constraint includes the number of operations equal to, or generally, greater than that of the minimum number of operations solution.

Thus, the minimization of the number of operations under a delay constraint problem can be defined as follows:

Definition 5.1: THE MCM PROBLEM UNDER A DELAY CONSTRAINT. Given a set of target constants and a maximum number of adder-steps, find the minimum number of additions/subtractions operations required to implement the MCM such that the user-specified maximum number of adder-steps is not exceeded.

Despite the large number of techniques proposed for the optimization of the number of operations, there are not many methods that also consider the delay of the design,

which is essential for high-speed systems. In the nonrecursive CSE algorithm of [92], contrary to the CSE heuristics of [75, 93], the subexpressions extracted from a constant multiplication are not shared with those of different constant multiplications. This modification leads to independent structures of constant multiplications where the relation between the number of operations and the number of adder-steps is compromised by the proposed method. In the CSE heuristic of [6], that is based on the CSE heuristic of [77] designed for the MCM problem, the subexpressions that meet the desired delay are considered among possible subexpressions in the implementations of the expressions. The same approach is also applied in the graph-based heuristic of [7] to control the delay of the MCM while synthesizing the constants in each iteration. In the graph-based heuristic of [94], three methods that reduce the number of adder-steps are applied in BHM [87] and RAG-n [87]. On the other hand, the graph-based heuristic of [8], initially, finds a solution including generally more number of operations but, with a small number of adder-steps and then, reduces the number of operations without increasing the number of adder-steps in an iterative loop. Obviously, in this heuristic, the final solution with a smaller number of adder-steps is dependent on the initial solution obtained by the graph-based heuristic designed for the MCM problem. To the best of our knowledge, there is no exact algorithm proposed for the MCM problem under a delay constraint.

5.2 The Exact Common Subexpression Elimination Algorithm

In this section, we introduce the exact CSE algorithm [20], which is obtained by parameterizing the exact CSE algorithm designed for the MCM problem described in Section 4.1.1 with a delay constraint so that, only the implementations that meet the desired delay are considered.

The algorithm includes similar steps with the exact CSE algorithm given in Section 4.1.1. Initially, the target constants are made positive and odd, and stored in a target set. Then, the possible implementations of target constants and partial terms are found when constants are defined under a number representation. Note that the exact CSE algorithm can handle constants defined under binary, CSD, and MSD. After all possible implementations of constants are found, these implementations are represented in a Boolean network that includes only AND and OR gates as given in

Section 4.1.1.2. Then, as described in the following two sections, the paths that exceed the delay constraint are found in the network. The MCM problem under a delay constraint is formalized as a 0-1 ILP problem under the minimization of the number of operations model described in Section 4.1.1.3. The cost function of the 0-1 ILP problem is determined as the linear function of optimization variables representing the operations. The constraints of the 0-1 ILP problem are the constraints obtained from the network and the constraints obtained from the paths that violate the delay constraint. Finally, a generic 0-1 ILP solver is used to find the minimum number of operations solution.

5.2.1 Computing the levels of operations in the Boolean network

In the Boolean network that represents the implementations of constants, i.e., partial terms and target constants, it can be easily observed that a constant can be implemented with operations that have different number of adder-steps. Therefore, we can define a range of levels for each constant, and consequently, a range of levels for the operations that compute this constant. In the Boolean network, an OR gate associated with the constant gathers all of these operations. So, a constant can be synthesized with the number of adder-steps ranging from its minimum to maximum latency implementations. As can be seen from Figure 4.2, the target constant 51 defined in CSD can be implemented with minimum 2 and maximum 3 adder-steps, determined, for instance, by $51 = 3_{\ll 4} + 3$ which has a minimum and a maximum of 2 adder-steps and by $51 = 13_{\ll 2} - 1$ which has a minimum and a maximum of 3 adder-steps.

We note that the proposed exact CSE algorithm can find the minimum number of operations solution with either the minimum delay that the network can have, *min_delay*, or a user-specified maximum delay constraint, *user_delay*. Observe from (5.1) that when constants are defined under CSD or MSD representation, the *min_delay* is equal to the minimum delay of the MCM.

After the Boolean network is constructed, we compute the minimum level, *min_level*, and maximum level, *max_level*, values of each operation and constant by traversing the network from primary inputs to primary outputs. Then, we find the *min_delay* value by computing the maximum of the *min_level* values of the primary outputs. By

setting $user\_delay = min\_delay$ 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 $user\_delay < min\_delay$, no solution is possible.

5.2.2 Finding the delay constraints

The paths that exceed the maximum delay constraint are found using the information on minimum and maximum levels of operations and constants. The part of the algorithm where the paths that exceed the *user_delay* are found is as follows:

1. *Preprocessing phase*: Determine the primary outputs of the network that have *max_level* values higher than *user_delay* and store them in a set called *Pset*.

2. For each element in the *Pset*, $Pset_i$,

   (a) If an operation that computes $Pset_i$ has *min_level* value higher than *user_delay*, remove this operation from the network. Because it can never be used to meet the *user_delay*.

   (b) Otherwise, if the operation has *max_level* value higher than *user_delay*, add this operation to a set called $path_j$ as an initial node and also, add this operation to a set called *Oset* with a *upper_level* value, i.e., $user\_delay-1$, and the associated path identifier, j.

3. *Main loop*: Remove an operation from the *Oset* with its upper level value, *upper_level*, 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 *min_level* value higher than *upper_level*, add this operation to $path_j$ as a terminal node, and construct the path.

   (b) Otherwise, if an operation has *max_level* value higher than *upper_level*, form a new path by adding this operation to the $path_j$. Also, insert this operation into the *Oset* with its upper level value, $upper\_level - 1$, and a path identifier.

4. Repeat Step 3 until there is no element left in the *Oset*.

Observe that in the preprocessing phase of the algorithm, the *Pset* includes the target constants that can be implemented in a greater delay than *user_delay* and at the end of the preprocessing phase, the initial nodes of the paths, i.e., the operations that violate the *user_delay* constraint are found. We also note that in the main loop, the paths are constructed in a breadth-first manner and the *Oset* includes the operations that are the last nodes of the paths have not been determined yet.

As an example, suppose that the target constant represented with the output of the OR gate *A* is to be implemented in 5 adder-steps, i.e., the *user_delay*, as given in Figure 5.4. 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 *min_level* and *max_level* values are given with a *min-max* pair above the gates. Note that the *path* includes the operations that exceed the *user_delay*, determined when traversing the network from the primary output to the primary inputs.

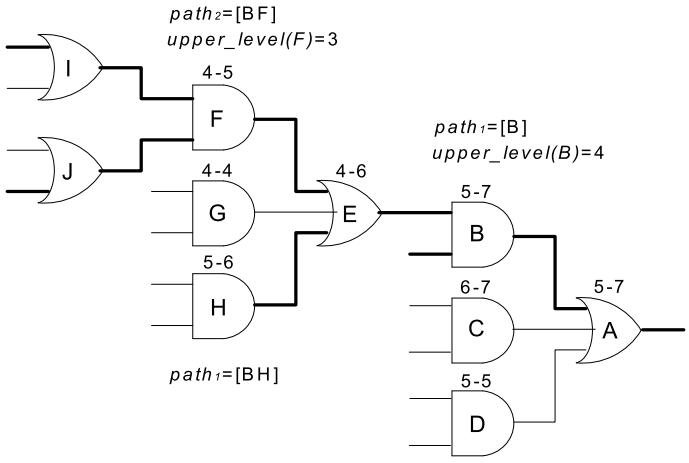


Figure 5.4: An illustrative example on determining the paths that exceed the maximum delay constraint.

In the preprocessing phase, the operation *B* is added to the initial node of $path_1$ and to the *Oset* with its upper level value 4 and path identifier 1, since its *max_level* value is higher than the *user_delay*. Observe that while the operation *C* never meets the *user_delay* and thus, can be removed from the network, the operation *D* never violates the *user_delay*. In the main loop, the operation *B* with its upper level value, $upper\_level(B) = 4$, and associated path identifier, 1, is removed from the *Oset*. Suppose that the partial term *E* is considered as the input of *B*. The operation *H* is

added to the $path_1$ as a terminal node and the path is constructed, since the operation H can be implemented in minimum 5 adder-steps that exceeds the *upper_level(B)*. Also, a new path, $path_2$, is formed by inserting the operation F to the $path_1$, since the *max_level* value of the operation F is higher than *upper_level(B)* indicating that there is(are) operation(s) that cause greater delay than the *user_delay* with the operations in this path. Thus, the operation F with its upper level $upper\_level(F) = upper\_level(B) - 1$ and associated path identifier, 2, is added to the *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 *upper_level(B)* value.

After all paths that violate the *user_delay* have been determined, for each path, a delay constraint, $-OPT_1 - OPT_2 - \ldots - OPT_k \geq 1 - k$, where OPT_j, $1 \leq j \leq k$, denotes the optimization variable of an operation in the path and k is the number of operations in the path, is added to the 0-1 ILP problem. Each delay constraint expresses 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, the 0-1 ILP problem with the cost function to be minimized, i.e., the linear function of optimization variables representing operations, and the constraints obtained from the Boolean network together with these delay constraints is given to the 0-1 ILP solver to find a solution with the minimum number of operations. Thus, the obtained minimum number of operations solution guarantees that it does not violate the delay constraint. Observe that without these delay constraints the problem to be solved is simply the MCM problem.

5.3 The Approximate Common Subexpression Elimination Algorithm

Although the exact CSE algorithm can find the minimum number of operations solutions under a delay constraint in MCM on real size instances as shown in Section 5.4, naturally, there are instances that the exact algorithm find them difficult to cope with. Hence, in this section, we introduce an approximate CSE algorithm [16] that can find competitive results with the exact solutions and can deal with the

instances that the exact CSE algorithm cannot conclude with the minimum solutions in a reasonable time.

The approximate CSE algorithm, called ASSUME-D, synthesizes each constant with an operation one at a time, similar to the ASSUME-A described in Section 4.1.2, but ASSUME-D considers only the operations that meet the desired delay in the implementation of a constant. Just as the exact version, ASSUME-D can also find a solution with either the minimum delay of the network, *min_delay*, or a maximum user-specified delay constraint, *user_delay*.

Again, as done in ASSUME-A, the algorithm starts by traversing the Boolean network to obtain the *min_adder*, *min_level*, and *max_level* values of each operation and partial term. As described in Section 4.1.2, the *min_adder* denotes the minimum number of operations that are required to implement an operation or a constant in the network. As defined in Section 5.2, the *min_delay* is determined as the maximum of the *min_level* values of the primary outputs. A minimum delay solution can be obtained when the *user_delay* is assigned to the *min_delay*.

In each iteration of ASSUME-D, a target constant or a partial term is synthesized in a top-down approach that yields more possible implementations of a constant while controlling the delay. The algorithm is as follows:

1. Store the pre-processed positive and odd target constants in a set called *Dset* and label them as unimplemented. Assign the *delay_limit* value of each element in *Dset* to *user_delay*.

2. Take an element labeled as unimplemented from *Dset*, *Dset*(i), that has the highest *max_level* value. Store the operations that implement *Dset*(i) and whose *min_level* values do not exceed the *delay_limit*(i) in an empty set called *Oset*.

3. If *Dset*(i) can be implemented with an operation in *Oset* whose inputs are primary inputs or are in *Dset*, then synthesize *Dset*(i) with this operation and label it as implemented. Determine the delay limit of each input of the operation, j, as $delay\_limit(j) = \min(delay\_limit(j), delay\_limit(i) - 1)$.

4. Otherwise, choose an operation from *Oset* to synthesize *Dset*(i) as done in steps 5 and 6 of ASSUME-A given in Section 4.1.2, and label it as implemented.

If the input(s) of the operation is in *Dset*, then assign the delay limit of the input, j, $delay\_limit(j) = \min(delay\_limit(j), delay\_limit(i) - 1)$. If not, add this element to *Dset*, label it as unimplemented, and assign its delay limit value to $delay\_limit(i) - 1$.

5. If there is an element left labeled as unimplemented in *Dset*, go to step 2, otherwise return the solution.

5.4 Experimental Results

In this section, we present the results of the proposed exact and approximate CSE algorithms on randomly generated and FIR filter instances, and compare with those of the CSE heuristics [6, 92] and the graph-based heuristics [7, 8].

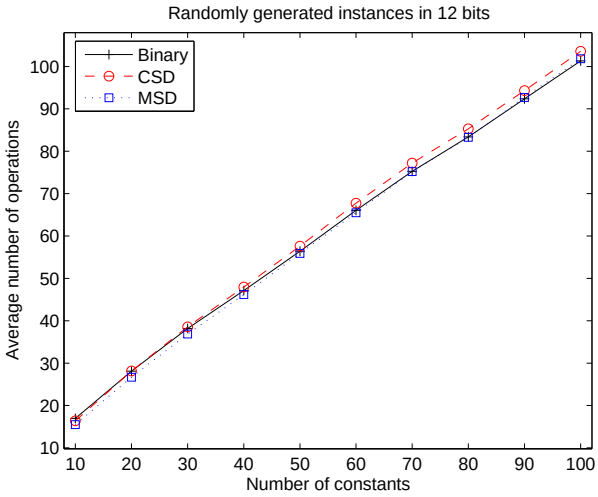
This section starts with the investigation of the effect of number representation on the minimum number of operations under the minimum delay constraint. It is followed by the comparison of CSE algorithms. Then, the performance of SAT-based 0-1 ILP solvers on this problem is examined. Finally, this section ends with the comparison of the exact CSE algorithm with the graph-based heuristics.

We note that in the exact and approximate CSE algorithms, the delay constraint was set to the minimum delay of the network, i.e., $user\_delay = min\_delay$.

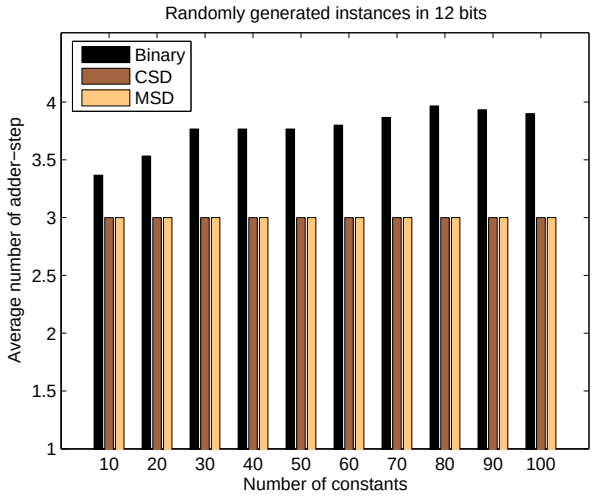
5.4.1 The effect of number representation on the achievable minimum number of operations under a delay constraint

In this experiment, we used randomly generated instances where the 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. Figure 5.5(a)-(b) presents the average number of operations and the average number of adder-steps of the solutions, respectively, obtained by the exact CSE algorithm when constants are defined under binary, CSD, and MSD.

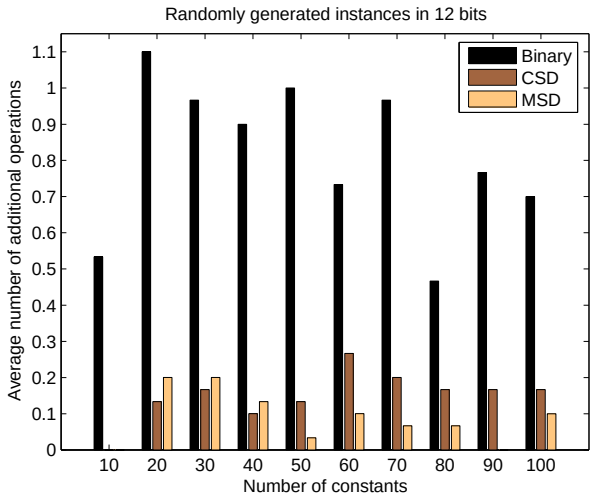
As can be observed from Figure 5.5(a), the use of MSD representation yields better solutions than those obtained under CSD representation, since the alternative representations of the constants in MSD increase the possible sharing of partial terms. Also, observe from Figure 5.5(b) that since both CSD and MSD representations define



(a)



(b)



(c)

Figure 5.5: The results of the exact CSE algorithm under binary, CSD, and MSD representation: (a) The average number of operations; (b) The average number of adder-steps; (c) The average number of additional operations to obtain the minimum delay solutions.

the constants using the minimum number of non-zero digits, the minimum number of adder-steps solutions are obtained using these representations. On the other hand, the use of binary representation leads similar solutions with those obtained under MSD representation in terms of the number of operations. However, as can be easily observed from Figure 5.5(b), the delay of the solutions obtained under binary representation is greater than those of the solutions obtained under MSD. This is because the binary representation of a constant generally includes more non-zero digits than that of MSD.

In Figure 5.5(c), the average number of additional operations required to obtain the minimum delay solution, i.e., the difference of the results presented in the graphs in Figure 5.5(a) and Figure 4.6(b), is also presented. As can be easily observed from Figure 5.5(c), under the CSD and MSD representations, the cost of obtaining the minimum number of operations under the minimum delay solution is negligible. Under the binary representation, obtaining the minimum delay solutions requires more additional operations than CSD and MSD, although there are more alternative sharing patterns that reduce the overhead for the binary representation for instances including large number of constants.

5.4.2 Comparison of CSE algorithms

In this experiment, we used randomly generated instances where the 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. Figure 5.6 gives the results of the exact and approximate CSE algorithms when constants are defined under CSD representation.

In this experiment, we observe that ASSUME-D finds solutions with almost 2 additional operations on average compared to the exact solutions. This experiment clearly indicates that the exact algorithm is indispensable to find the minimum number of operations under the delay constraint.

In this experiment, we also used FIR filter instances presented in Table 4.7. The results of the CSE algorithms are given in Table 5.1. Note that the results of the CSE heuristic [6] were provided by Anup Hosangadi. We also note that since ASSUME-D found the same solutions in terms of the number of operations as those

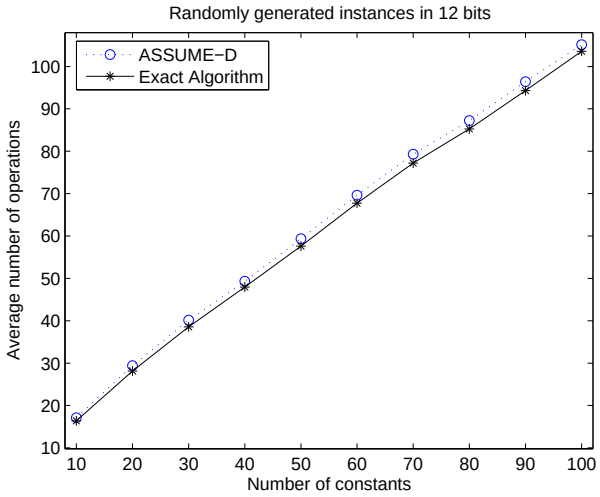


Figure 5.6: Comparison of the exact and approximate CSE algorithms for the minimization of the number of operations under a delay constraint.

of the exact algorithm on all filter instances when filter coefficients are defined under CSD representation, its results were not included in this table.

As can be easily observed from Table 5.1, ASSUME-D finds similar results with those of the exact algorithm, and better solutions than the CSE heuristic of [6]. The difference of the number of operations between the CSE heuristic [6] and the exact CSE algorithm is 1.8 operations on average. Also, observe that the use of MSD representation yields better solutions in terms of the number of operations than those obtained under binary and CSD representations. While the minimum delay solutions are obtained when constants are defined under CSD or MSD representation, the binary representation yields solutions in a greater delay than those of solutions obtained under CSD or MSD.

Also, in this experiment, we used FIR filters given in Table 4.11. The results of the approximate algorithm and the CSE heuristics of [6, 92] when filter coefficients are defined under CSD representation are presented in Table 5.2. In this table, *original* denotes the results of the minimum delay implementations of the coefficient multiplications when the sharing of partial products is not considered. We note that the results of the CSE heuristics were provided by Anup Hosangadi.

As can be easily observed from Table 5.2, the CSE heuristic of [92] obtains solutions that are far from those of the CSE heuristic [6] and the approximate algorithm. On

Table 5.1: Summary of results of algorithms on the FIR filter instances.

Filter	Binary				CSD				MSD			
	ASSUME-D		Exact		[6]		Exact		ASSUME-D		Exact	
	adder	step	adder	step	adder	step	adder	step	adder	step	adder	step
1	10	3	10	3	11	2	10	2	10	2	10	2
2	18	3	18	3	18	3	18	3	18	3	18	3
3	18	3	18	3	18	3	16	3	16	3	16	3
4	29	3	29	3	30	3	29	3	29	3	29	3
5	36	3	35	3	35	3	34	3	34	3	34	3
6	25	3	25	3	26	3	23	3	22	3	22	3
7	33	4	32	4	36	3	35	3	34	3	34	3
8	36	4	34	4	37	3	35	3	35	3	33	3
9	53	4	51	4	58	3	52	3	49	3	49	3
Total	258	30	252	30	269	26	252	26	247	26	245	26

Table 5.2: Summary of results of the CSE heuristics on the filter instances.

Filter	Original		[92]		[6]		ASSUME-D	
	adder	step	adder	step	adder	step	adder	step
1	106	4	59	4	23	4	23	4
2	26	3	16	3	10	3	11	3
3	238	4	146	4	54	4	53	4
4	158	4	102	4	41	4	41	4
5	234	4	129	4	55	4	56	4
6	44	4	27	4	14	4	13	4
7	868	4	527	4	163	4	154	4
8	656	4	378	4	128	4	127	4
9	82	4	49	4	24	4	24	4
10	1414	4	917	4	229	4	219	4
11	988	4	652	4	182	4	175	4
Total	4814	43	3002	43	923	43	896	43

the other hand, ASSUME-D obtains better solutions than the CSE heuristic [6] on overall these filter instances, i.e., less than 2.4 operations on average.

5.4.3 Comparison of SAT-based 0-1 ILP solvers

In this experiment, we used FIR filter instances presented in Table 4.7. The 0-1 ILP problem size of filter instances when constants are defined under MSD representation are presented in Table 5.3. In this table, while *cons* indicates the total number of constraints, *delay cons* denotes the number of delay constraints in the 0-1 ILP problem. The run-time of SAT-based 0-1 ILP solvers, *bsolo* and *minisat+*, on the filter instances are also presented in this table where *CPU* denotes the CPU time in seconds required for the SAT-based 0-1 ILP solvers to find the minimum solution under a PC with dual Pentium Xeon at 2.4GHz, with 4GB of main memory, running

Table 5.3: 0-1 ILP problem sizes of the FIR filters and run-time performance of the SAT-based 0-1 ILP solvers.

Filter	0-1 ILP Problem Size				bsolo		minisat+	
	vars	cons	delay cons	optvars	adder	CPU	adder	CPU
1	247	372	25	144	10	0.2	10	0
2	635	1075	48	345	18	0.2	18	0.6
3	1327	2546	159	677	16	0.4	16	2.1
4	1926	3616	285	1023	29	2.7	29	0.7
5	1142	1897	128	651	34	0.3	34	0.3
6	4324	10127	1580	2153	22	25.4	*22*	*3600.1*
7	2250	5081	253	1062	34	5.2	34	8.9
8	3915	9230	688	1856	33	41.6	33	29.2
9	26778	71670	16181	13329	49	1332.2	*53*	*3600.1*

Linux. The allowed CPU time for the 0-1 ILP solvers was 3600 seconds and again, the italic results indicate that a suboptimal solution rather than the minimum is obtained in the given CPU time limit.

As can be observed from Table 5.3, the number of delay constraints that guarantee the minimum number of operations solution to be obtained in minimum number of adder-steps is less than 10% of the total number of constraints on filter instances, except on Filter 6 and 9 where these values are 15.6% and 22.6% respectively. Also, observe that *bsolo*, which is also equipped with ILP techniques obtains the minimum number of operations solutions with a very little computational effort and *minisat+* obtains a suboptimal result on Filter 9 in a given CPU time limit.

5.4.4 Comparison of the CSE and graph-based algorithms

In this experiment, we used FIR filter instances presented in Table 4.7. Table 5.4 compares the solutions of the exact CSE algorithm when coefficients are represented under MSD with those of the graph-based heuristics [7, 8]. The results of the graph-based heuristic [7] were taken from its paper [7] and the results of the graph-based heuristic [8] were provided by Andrew Dempster.

As can be observed from Table 5.4, although the graph-based algorithms [7, 8] do not always ensure the minimum delay solution, they find better solutions in terms of the number of operations than the exact CSE algorithm on the instances obtained with the minimum delay, e.g., Filter 2. This is simply because the implementations of the constants considered in these algorithms are not limited with

Table 5.4: Summary of results of the graph-based heuristics and the exact CSE algorithm on the FIR filter instances.

Filter	[7]		[8]		Exact CSE MSD	
	adder	step	adder	step	adder	step
1	10	3	10	2	10	2
2	17	3	17	3	18	3
3	17	4	16	3	16	3
4	28	5	28	3	29	3
5	34	3	34	3	34	3
6	20	4	20	4	22	3
7	29	6	30	3	34	3
8	28	6	29	4	33	3
9	47	6	47	4	49	3
Total	230	40	231	29	245	26

only the implementations that can be extracted from the number representations of constants.

5.5 Conclusions

In this chapter, we introduce an exact CSE algorithm designed for the MCM problem under a delay constraint. The exact algorithm is based on the exact CSE algorithm designed for the MCM problem. To guarantee the minimum number of operations solution under the minimum delay, the paths that violate the delay constraint in the network are found, expressed as PB constraints, and added to the 0-1 ILP problem that models the MCM problem. It is shown by the experimental results that delay can be minimized practically with no area overhead in a reasonable time for real-sized instances.

In this chapter, we also present an approximate algorithm that can handle more complex instances. It is shown by the experimental results that the approximate algorithm finds competitive results with the exact CSE algorithm and better solutions than the CSE heuristics.

As future work, we are currently working on a graph-based algorithm designed for the MCM problem under a delay constraint that can guarantee the minimum number of adder-steps solutions and also, can find better solutions in terms of the number of operations than the previously proposed graph-based heuristics.

6. OPTIMIZATION OF AREA AT GATE-LEVEL

Although the exact algorithms designed for the MCM problem introduced in Chapter 4 find the minimum number of operations solutions, these solutions may not yield the minimum area solutions when implemented at the gate level. In this chapter, we address the problem of finding the minimum area of MCM in terms of gate-level metrics.

To find the minimum area solutions of the MCM problem, one has to consider the area of addition/subtraction operations during the synthesis of MCM and take into account these cost values in the selection of operations that implement the constant multiplications. We note that the area of an operation in the design of MCM implemented at the gate-level depends on:

- the type of the operation (addition or subtraction),
- the shifted input (minuend or subtrahend) in a subtraction,
- the number of shifts at the inputs,
- the position of the operation in the architecture (that influences the number of bits),
- the range and type of numbers considered (unsigned or signed).

Thus, we define the minimization of area problem as follows:

Definition 6.1: MINIMIZATION OF AREA AT THE GATE-LEVEL. Given a set of target constants, find a set of operations that implement the MCM such that the constant multiplications are synthesized using minimum area in terms of gate-level metrics.

Despite the large number of algorithms designed for the minimization of the number of operations, there is only a graph-based heuristic [9] that considers the number of half adders (HAs) and full adders (FAs) required to implement the operations when

finding a solution that implements the MCM. However, the selection criteria in the proposed heuristic is based on only the number of HAs and FAs needed to synthesize the constant multiplications and special cases are not taken into account. To the best of our knowledge, there is no exact algorithm designed for the minimization of area in MCM in terms of gate-level metrics.

In this chapter, we introduce the actual architectures based on HAs and FAs for addition and subtraction operations under unsigned and signed inputs in MCM, and describe the exact CSE algorithm [21] that finds the minimum area solution of the MCM problem. The exact CSE algorithm is based on the minimization of the number of operations model where optimization variables are associated with the operations. The minimization of area problem is formalized as a 0-1 ILP problem with a cost function to be minimized and constraints to be satisfied. The cost function of the 0-1 ILP problem is determined as a linear function of optimization variables representing operations, where the cost value of each optimization variable is the area of the associated operation computed in terms of gate-level metrics. The constraints of the 0-1 ILP problem are obtained from the possible implementations of constants represented in a Boolean network.

6.1 Addition and Subtraction Architectures under Unsigned and Signed Input

In this section, we describe the implementations of addition and subtraction operations in the design of MCM and determine the cost of each operation in terms of HAs, FAs, and logic gates in a given technology library. Since the shifts are free in terms of hardware, the constants are considered as odd numbers in MCM algorithms. So, there are three different types of operations:

1. $A_{\ll S_A} + B_{\ll S_B}$ (an adder where $S_A = 0$, $S_B = S$)

2. $A_{\ll S_A} - B_{\ll S_B}$ (a subtracter where $S_A = S$, $S_B = 0$)

3. $A_{\ll S_A} - B_{\ll S_B}$ (a subtracter where $S_A = 0$, $S_B = S$)

In given operations, A and B represent the constants at the inputs of the operation, S_A and S_B denote the number of shifts on the inputs A and B respectively. Note that we can always consider one of S_A and S_B is zero and the other is greater than zero. When

both are greater than zero, one can make one of them (the smallest) zero by shifting the output of the operation by the smallest times.

The parameters considered during the computation of the operation cost are given as follows:

- S: the number of shifts
- n_A: the number of bits of the input A
- n_B: the number of bits of the input B
- n_m: minimum number of bits at the inputs: $min(n_A + S_A, n_B + S_B)$
- n_M: maximum number of bits at the inputs: $max(n_A + S_A, n_B + S_B)$

We note that the number of the bits at the inputs of an operation also depends on the bit-width of the input that is the constants are multiplied with, denoted by N.

The cost of each operation is determined considering unsigned and signed numbers, since these lead to different implementations due to the sign extension and is formulated by considering the overlap between inputs and taking into account specific cases.

6.1.1 Addition operation $A + B_{\ll S}$

The cost of implementation of an addition operation in terms of HAs and FAs is given in Table 6.1.

Table 6.1: The cost of an $A + B_{\ll S}$ operation.

Cost Parameters	Unsigned Input	Signed Input
#FA	n_m - S - 1	n_M - S - 1
#HA	n_M - n_m + 1	1

In Figure 6.1, examples on unsigned and signed input models are given. Observe that larger number of shifts at the input achieves smaller area, since shifts are implemented with only wires in the design. Note that when the number of FAs is negative in the unsigned input case, no hardware is needed and the operation can be implemented with only wires as illustrated in the second example on unsigned input. This situation

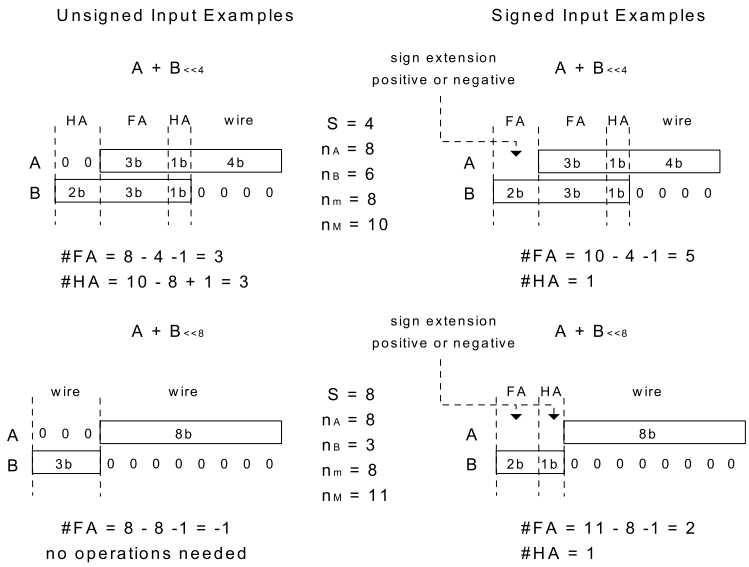


Figure 6.1: Examples on the computation of the area cost of an $A + B_{\ll S}$ operation.

occurs when the number of shifts of the operand B is equal to or higher than the number of bits of the operand A. In the signed input case, this situation never occurs, due to the sign extension of the operand A.

6.1.2 Subtraction operation $A_{\ll S} - B$

The cost of implementation of this subtraction operation is given in Table 6.2. A subtraction operation is implemented using 2's complement, i.e., $A + \overline{B} + 1$. So, the number of inverters, #inv, is included in the cost of subtraction operations. Additionally, a different type of HA denoted by HA$'$ is introduced. HA$'$ is the special implementation of an FA when one of the inputs is 1 as opposed to the HA, i.e., another special implementation of FA, when one of the inputs is 0. In the FA, suppose the input B_i is 1. Then, the sum (S_i) and carry (C_{i+1}) outputs are the functions of the input A_i and the carry input (C_i) given as $S_i = \overline{C_i \oplus A_i}$ and $C_{i+1} = C_i + A_i$.

Table 6.2: The cost of an $A_{\ll S} - B$ operation.

Cost Parameters	Unsigned Input	Signed Input
#FA	n_B - S	n_A
#HA	S - 1	S - 1
#HA$'$	n_A + S - n_B	0
#inv	n_B	n_B

In Figure 6.2, examples on unsigned and signed input models are given. Observe that in both examples, the first digit of the result is the first digit of the operand B and the

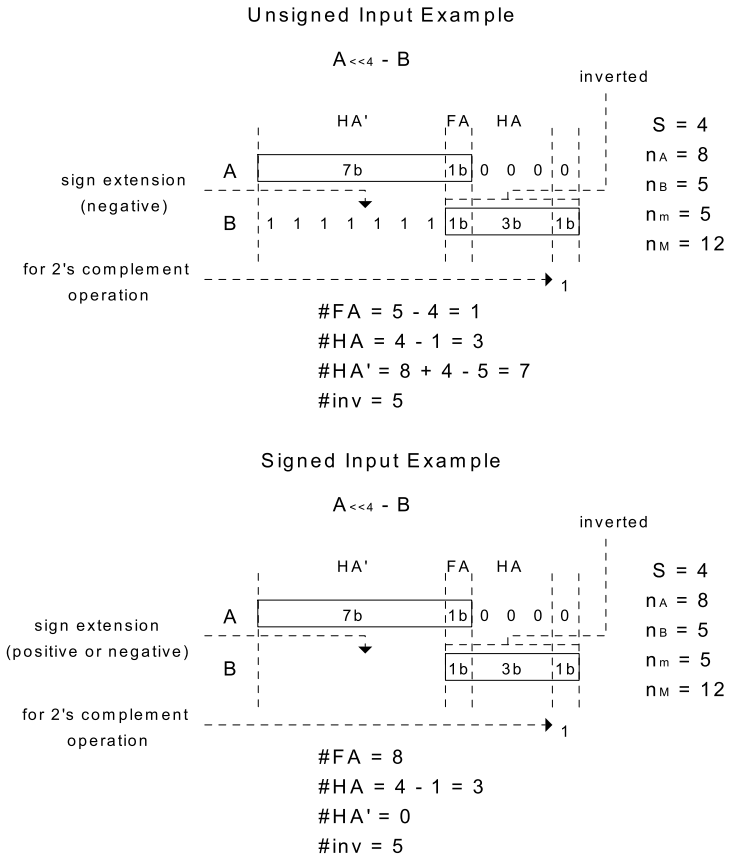


Figure 6.2: Examples on the computation of the area cost of an $A_{\ll S} - B$ operation.

carry taken to the second digit, in this example to the input of HA, is the complement of the first digit of B. So, only one inverter is needed to obtain the first digit of the result.

6.1.3 Subtraction operation $A - B_{\ll S}$

The cost of implementation of this subtraction operation is given in Table 6.3. Observe that the cost of the operation is computed without HAs as opposed to the $A_{\ll S} - B$ subtraction operation.

Table 6.3: The cost of an $A - B_{\ll S}$ operation.

Cost Parameters	Unsigned Input	Signed Input
#FA	n_B - 1	n_M - S -1
#HA′	n_A - n_B - S +1	1
#inv	n_B	n_B

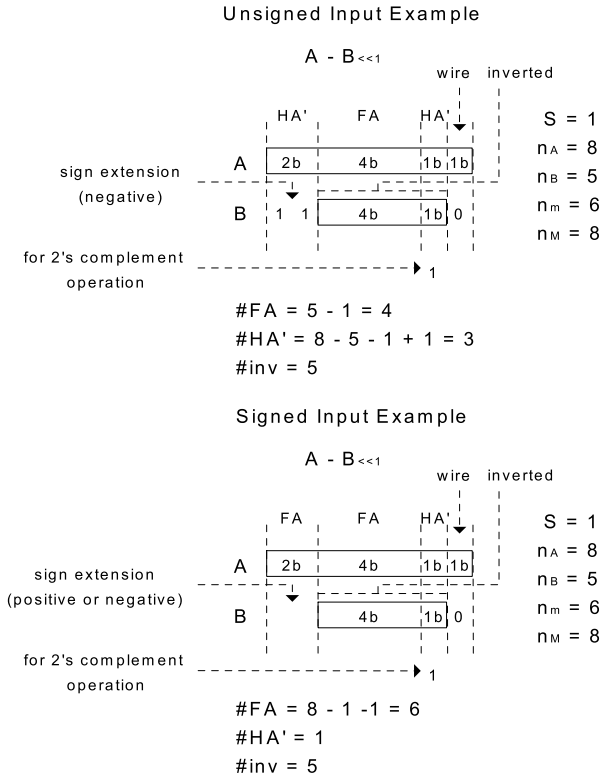


Figure 6.3: Examples on the computation of the area cost of an $A - B_{\ll S}$ operation.

In Figure 6.3, examples on unsigned and signed input models are given. Observe that the shifts can be fully utilized by starting addition with the first digit of the inverted operand B resulting in a smaller area.

6.2 The Exact Common Subexpression Elimination Algorithm

In this section, we present the exact CSE algorithm designed for the minimization of area and describe the combinational network constructed by the algorithm. We note that the proposed exact CSE algorithm follows the same steps of the exact CSE algorithm designed for the MCM problem given in Section 4.1.1. In the exact algorithm, the constants can be defined under any type of number representation, namely, binary, CSD, or MSD.

Initially, all possible implementations of each target constant and also partial term are obtained by decomposing the non-zero digits in the representation(s) of the constant as described in Section 4.1.1.1. Then, the Boolean network that represents the possible implementations of the constants is constructed as given in Section 4.1.1.2. The Boolean network includes only AND and OR gates, where an AND gate in the

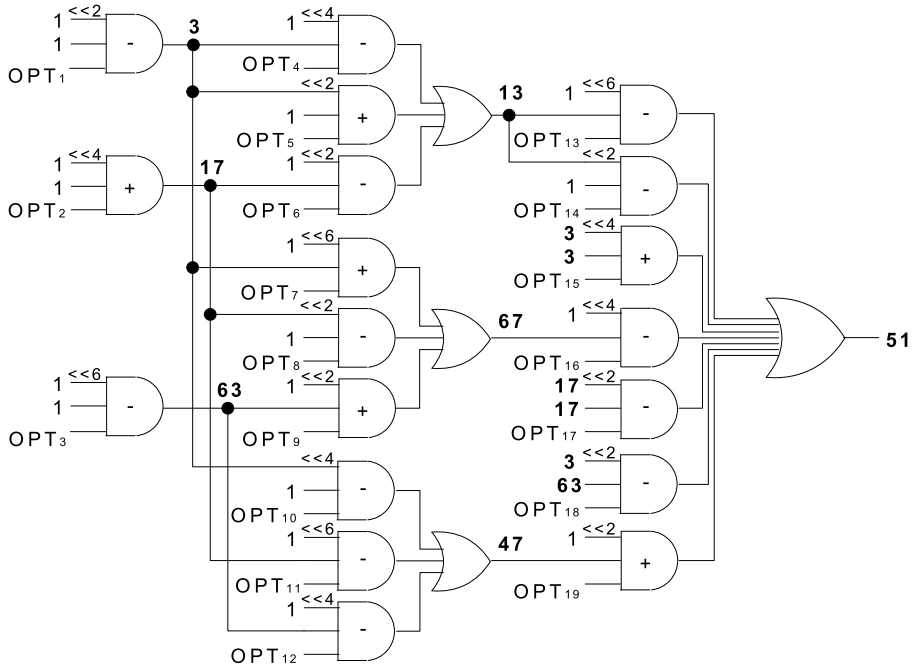


Figure 6.4: The network generated for the target constant 51 in CSD.

network represents an addition/subtraction operation and an OR gate represents a constant and combines all operations that implement the constant. The minimization of area problem is formalized under the minimization of the number of operations model. Hence, the optimization variables associated with the operations are added to the AND gates in the network as the third input. As an example, suppose the target constant 51 defined in CSD. The Boolean network generated for the target constant is given in Figure 6.4, where the 1-input OR gates for the partial terms 3, 17, and 63 are omitted and the type of each operation, i.e., an adder or a subtracter, is given inside of each AND gate. As can be easily observed, the target constant 51 defined under CSD has 7 alternative implementations including additions and subtractions.

Observe from Figure 6.4 that the operations associated with the optimization variables OPT_4 and OPT_5 have the same positive and odd inputs, similar to the operations with the optimization variables OPT_{13} and OPT_{14}. Although these operations are included in the network, because they have different area cost as described in Section 6.1, we note that only one of these operations that has the minimum area cost among these operations can be kept for the implementation of the constant, and the other can be deleted from the network.

After the Boolean network is constructed, the formalization of the minimization of area problem as a 0-1 ILP problem is then straight-forward. Initially, the cost function of the 0-1 ILP problem is determined as the linear function of optimization variables representing operations, where the cost value of each optimization variable is the area cost of each operation computed as described in Section 6.1. Then, the constraints of the 0-1 ILP problem are determined as the PB constraints of each gate in the network. Also, we assign 1 value to the variables denoting the outputs of OR gates associated with the target constants and to the variables representing the input that is the constants are multiplied with. Observe that if the cost value of each optimization variable is assigned to 1, then the problem turns to the minimization of the number of operations problem.

Finally, a generic 0-1 ILP solver is used to find the minimum area solution of the MCM problem. Note that the minimum solution directly indicates the operations to be synthesized to implement the MCM.

6.3 Experimental Results

In this section, we present the results of the exact CSE algorithm that are obtained with the minimum number of operations and the minimum area objectives. The data associated with the HA, HA$'$, FA, and an inverter were taken from UMC Logic $0.18\mu m$ Generic II library and are given in Table 6.4. We note that while inv, HA, and FA were the primitive gates of the library, HA$'$ was implemented using XNOR and OR gates in the library as defined in Section 6.1.2.

Table 6.4: Experimental settings.

Gate	Area (μm^2)	Max Delay (ns)	Max Delay Carry (ns)
inv	6	0.06	–
HA	32	0.185	0.137
HA$'$	35	0.185	0.085
FA	58	0.261	0.194

As an experiment set, we used filter instances where the filter coefficients were computed using the *remez* algorithm in MATLAB. The specifications of filters are given in Table 6.5. In the exact CSE algorithm, the filter coefficients are defined in MSD representation.

Table 6.5: Filter specifications.

Filter	*pass*	*stop*	*#tap*	*width*
1	0.20	0.25	120	8
2	0.10	0.25	100	10
3	0.15	0.25	40	12
4	0.20	0.25	80	12
5	0.24	0.25	120	12
6	0.15	0.25	60	14
7	0.15	0.20	60	14
8	0.15	0.20	100	16
9	0.10	0.15	60	14
10	0.10	0.15	100	16

The exact solutions with the minimum number of operations and minimum area objectives are obtained on filter instances for unsigned and signed input models, when the bit widths of the filter inputs, i.e., N, are 8, 16, and 24. The results on unsigned and signed models are given in Table 6.6 and Table 6.7, respectively. In these tables, *adder* denotes the number of operations, *area* and *delay* denote the total area and the delay of the multiplier block of the filter, respectively. We note that after a solution is found with any given objective by the exact algorithm, the area and delay results are obtained with the described gate-level metrics.

In this experiment, we observe that a minimum solution in terms of the number of operations may not give the minimum area design, e.g., Filter 10 when $N = 8$ for the unsigned and signed input models. Also, even if the number of operations in the solutions obtained under both objectives are the same, the solution found under the minimum number of operations objective may yield larger area in the design, e.g., Filter 5 when $N = 8$ for the unsigned and signed input models. As can be seen from experimental results, the reduction in area can be 13.5% and 12.5% on average for unsigned and signed models respectively. We note that since the optimization of area yields the optimization of the number of HAs and FAs, the delay of the design is also decreased on most of the filter instances.

This experiment clearly indicates that the minimum number of operations solution of an MCM problem does not yield the minimum area solution at the gate-level. To obtain the minimum area solution, the actual area data of addition/subtraction operations implementing the MCM should be added into the optimization problem.

Table 6.6: Experimental results on unsigned input model.

N	8 bits						16 bits						24 bits					
Objective	# Operations			Area			#Operations			Area			# Operations			Area		
Filter	adder	area	delay	adder	area	delay	adder	area	delay	adder	area	delay	adder	area	delay	adder	area	delay
1	10	5114	4.2	10	4477	4.9	10	9994	7.3	10	9213	9.6	10	14874	10.4	10	13949	14.3
2	18	9465	7.1	18	7898	5.7	18	18249	11.7	18	16394	10.3	18	27033	16.4	18	24890	15.0
3	16	8847	6.7	17	8407	5.8	16	16751	11.4	16	16218	10.4	16	24655	16.0	16	23978	15.1
4	29	16462	6.9	29	13537	6.0	29	30638	11.2	29	27410	12.2	29	44814	16.2	29	41202	18.4
5	34	17645	7.1	34	15310	6.6	34	33949	11.7	34	31326	11.3	34	50253	16.4	34	47342	15.9
6	22	12673	8.4	23	12008	7.3	22	23457	14.6	22	22816	12.4	22	34241	20.8	22	33456	17.1
7	34	20310	7.5	35	17518	7.0	34	36854	12.2	34	33871	12.2	34	53398	16.8	34	50162	16.4
8	47	27798	7.8	51	24092	7.3	47	51003	14.1	47	47901	12.8	47	73819	20.3	47	70311	19.0
9	33	20787	8.2	35	16998	6.2	33	37140	14.4	35	34134	10.8	33	53316	20.6	34	50769	16.2
10	49	29474	8.6	54	25497	7.6	49	53714	15.6	49	50145	15.5	49	77458	21.8	49	73649	21.7
Avg. (%)	100	100	100	104.8	86.5	88.9	100	100	100	100.7	92.8	94.7	100	100	100	100.3	94.7	96.2
Min. (%)	–	–	–	100.0	81.8	75.5	–	–	–	100.0	89.5	75.1	–	–	–	100.0	91.9	78.7
Max. (%)	–	–	–	110.2	95.0	117.3	–	–	–	106.1	97.3	131.7	–	–	–	103.0	97.7	136.9

Table 6.7: Experimental results on signed input model.

N	8 bits						16 bits						24 bits					
Objective	# Operations			Area			#Operations			Area			# Operations			Area		
Filter	adder	area	delay	adder	area	delay	adder	area	delay	adder	area	delay	adder	area	delay	adder	area	delay
1	10	5924	4.6	10	5152	4.1	10	10804	7.7	10	9888	7.2	10	15684	10.8	10	14624	10.3
2	18	11098	7.8	18	9050	5.7	18	19882	12.5	18	17546	10.4	18	28666	17.1	18	26042	15.1
3	16	10552	7.3	16	10054	6.1	16	18456	12.0	16	17814	10.8	16	26360	16.6	16	25574	15.4
4	29	19333	7.5	29	15944	6.2	29	33509	12.1	29	29736	12.5	29	47685	16.8	29	43528	18.7
5	34	20682	7.9	34	17884	6.7	34	36986	12.5	34	33900	11.3	34	53290	17.2	34	49916	16.0
6	22	15396	9.4	23	14721	8.0	22	26180	15.6	22	25612	13.3	22	36964	21.9	22	36252	18.0
7	34	24625	8.5	35	21248	7.3	34	41169	13.2	34	37762	13.2	34	57713	17.8	34	54066	17.8
8	47	34149	9.2	50	29950	8.5	47	56965	15.4	47	52609	13.4	47	79781	21.6	47	74993	19.6
9	33	24375	9.2	36	21216	6.7	33	40551	15.4	34	37822	12.1	33	56727	21.6	34	54078	16.8
10	49	36280	10.5	52	31837	8.5	49	60024	16.7	49	55719	16.2	49	83768	22.9	49	79223	22.4
Avg. (%)	100	100	100	103.8	87.5	82.9	100	100	100	100.3	92.4	90.5	100	100	100	100.3	94.2	92.3
Min. (%)	–	–	–	100.0	81.5	72.8	–	–	–	100.0	88.3	78.9	–	–	–	100.0	90.8	77.8
Max. (%)	–	–	–	109.1	95.6	93.3	–	–	–	103.0	96.5	102.6	–	–	–	103.0	98.1	111.1

Table 6.8: Effect of the bit widths of filter input over area on unsigned input model.

Filter	D_8^8	D_{16}^8	D_{24}^8	D_8^{16}	D_{16}^{16}	D_{24}^{16}	D_8^{24}	D_{16}^{24}	D_{24}^{24}
1	4477	4477	4477	9213	9213	9213	13949	13949	13949
2	7898	7898	7898	16394	16394	16394	24890	24890	24890
3	8407	8458	8458	16850	16218	16218	25026	23978	23978
4	13537	13618	13618	27532	27410	27410	41324	41202	41202
5	15310	15310	15310	31326	31326	31326	47342	47342	47342
6	12008	12176	12176	23064	22816	22816	34120	33456	33456
7	17518	17567	17650	34489	33871	33906	51257	50175	50162
8	24092	25469	25543	49379	47901	47927	73523	70333	70311
9	16998	17137	17932	34262	34134	34513	50934	50806	50769
10	25497	26641	26641	51776	50145	50145	77408	73649	73649
Avg. (%)	100	102.1	102.7	101.7	100	100.2	102.3	100.02	100
Min. (%)	–	100.0	100.0	100.0	–	100.0	100.0	100.0	–
Max. (%)	–	105.7	106.0	103.9	–	101.1	105.1	100.1	–

To find the effect of the bit width of the filter input, N, on the area of the multiplier block of the FIR filters, we have also conducted an experiment under the unsigned input model. In this experiment, initially, the optimum area filter implementations, i.e., the set of operations that yields the minimum area, for N is 8, 16, and 24 bits is obtained. Then, these implementations are synthesized at the gate-level when N is 8, 16, and 24. The area of the designs are presented in Table 6.8, where D_{opt}^{syn}, $opt, syn \in \{8, 16, 24\}$ represents the area of the design optimized for area when the bit width of the filter input is *opt* and synthesized when the bit width of the filter input is *syn*.

In this experiment, we observe that the bit width of the filter input affects the minimum area solutions slightly in the proposed approach. This result can be interpreted as follows. For instance, suppose the set of operations that yields the minimum area when the bit width of the filter input is 16 has been obtained and these operations have been designed at gate-level when the bit width of the filter input is 8 and 24. The area of these designs are denoted by D_{16}^8 and D_{16}^{24} respectively. When these values are compared with their optimum values, i.e., D_8^8 and D_{24}^{24} respectively, it can be easily observed that the area overhead according to the minimum area designs is only 2.1% and 0.02% on average respectively. Thus, once the set of operations that yields the minimum area under a specific number of bit widths of the filter input was found, it could be designed for any number of bit-widths of the filter input and the

area overhead with respect to the optimum area solution would be small. We note that similar results are observed on the signed input model.

6.4 Conclusions

In this chapter, an exact CSE algorithm that minimizes the area of multiple constant multiplications at the gate-level is proposed and demonstrated on the synthesis of the multiplier block of digital FIR filters. In the exact algorithm, the area value of each operation is defined in terms of gate-level metrics and is assigned to the cost value of each operation in the cost function of the 0-1 ILP problem. The area of addition/subtraction operations are obtained from their actual architectures proposed in this work. We present results indicating that if the objective is limited to the minimization of the number of operations, the actual hardware implementation can be far from optimum. Although the experimental results are based on FIR filters, this method can be directly applied to any system that includes multiple constant multiplications.

7. OPTIMIZATION OF AREA IN HIGH-SPEED DIGITAL FIR FILTERS

In the algorithms designed for the MCM problem described in Chapter 4, an addition/subtraction operation is assumed to be a 2-input operation that is generally implemented using ripple carry adder (RCA) blocks [9, 21] that yield great latency in the implementation of MCM. In high-speed applications, particularly in DSP systems, carry-save adder (CSA) blocks are preferred to RCA blocks taking into account the increase in area. This chapter addresses the problem of finding the fewest number of CSA blocks for the implementation of the MCM that achieves a high throughput.

In this chapter, initially, we introduce the background concepts and give the problem definition. Then, we present the exact CSE algorithm [22] designed for the minimization of the number of CSA blocks problem. Also, we introduce an approximate CSE algorithm [22] based on the proposed exact CSE algorithm that can deal with large size instances. Since the solutions obtained by the proposed CSE algorithms depend on the number representation, in the approximate algorithm, we further increase the number of possible implementations of constants using a general number representation allowing our algorithm to be more effective in area optimization [22].

7.1 Background

A CSA block has three inputs and two outputs, i.e., sum (S) and carry (C). The two outputs together form the result. An n-bit CSA block includes n FAs. Since there is no need to propagate the carry as required in an RCA block, Figure 7.1(a), the latency of an addition is equal to the gate delay of an FA, Figure 7.1(b).

The implementation of a digital FIR filter that achieves a high throughput as described in [95] is illustrated in Figure 7.2 where each addition represents a CSA block and the filter output is obtained using a vector merging adder (VMA). The proposed exact

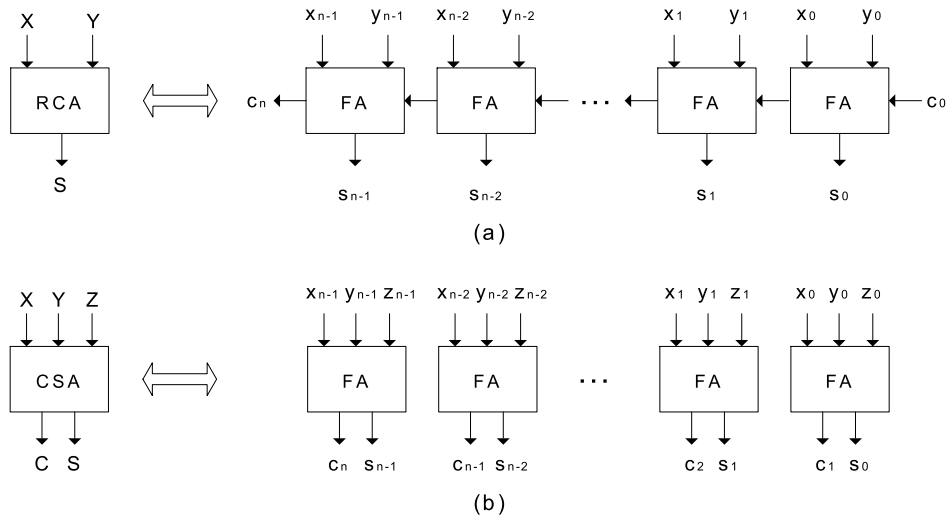


Figure 7.1: Addition architectures: (a) Ripple carry adder block; (b) Carry-save adder block.

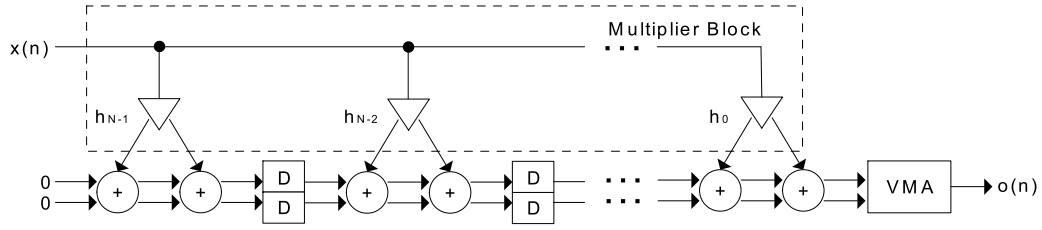


Figure 7.2: The implementation of the transposed form of a high-speed digital FIR filter.

and approximate algorithms are demonstrated on the design of the multiplier block of a high-speed FIR filter given in Figure 7.2.

Thus, the minimization of the number of CSA blocks problem is defined as follows:

Definition 7.1: MINIMIZATION OF THE NUMBER OF CSA BLOCKS. Given a set of filter coefficients, find the minimum number of CSA blocks that implement the coefficient multiplications in the multiplier block of a digital FIR filter.

To design the multiplier block of a digital FIR filter using CSA blocks, one may, initially, find a set of operations implemented using RCA blocks, i.e., the operations have two inputs, and then, transform these RCA blocks into CSA blocks using a mapping technique. To do this, a technique that converts RCA blocks into CSA blocks for the implementation of MCM in a digital FIR filter is described in [13]. In this scenario, three different cases may occur:

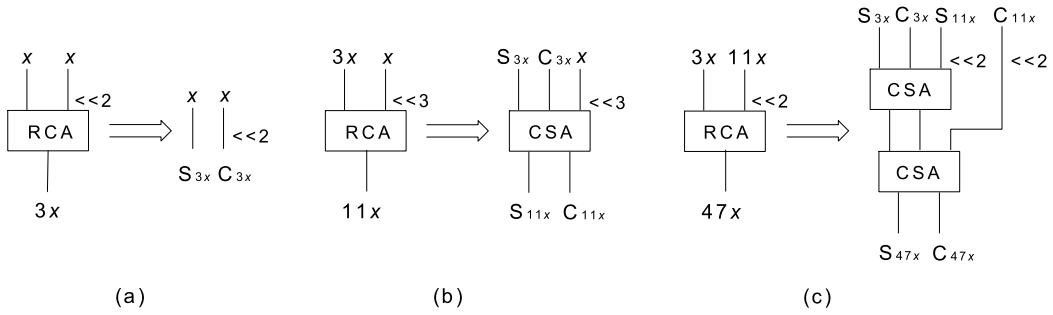


Figure 7.3: Conversion of RCA operations to CSA operations in MCM.

- If both inputs of an RCA block are filter inputs, then no CSA block is required, since these two inputs can be represented as sum and carry outputs, Figure 7.3(a).
- If only one of the inputs of an RCA block is the filter input, then one CSA block is required, Figure 7.3(b).
- If an RCA block has inputs that are not filter inputs, then two CSA blocks are required, Figure 7.3(c).

In Figure 7.4, we compare the solutions obtained by the exact CSE algorithm [22] designed for the minimization of the number of CSA blocks with the solutions obtained by the described RCA to CSA conversion technique [13]. We note that the minimum number of RCA blocks solutions were obtained using the exact CSE algorithm of [14]. In this experiment, the constants were generated randomly in 12 bit-width and defined under CSD representation. The number of constants varies between 10 and 100, and each set includes 30 instances.

As can be easily observed from Figure 7.4, the mapping from RCAs to CSAs yields suboptimal solutions that are far from the solutions obtained by the exact CSE algorithm [22]. This experiment clearly indicates that taking into account of the CSA block architecture in the optimization is indispensable to find the minimum area solution.

The mapping techniques, similar to the RCA to CSA conversion technique of [13], proposed for the design of more complex arithmetic circuits using CSA blocks can be found in [10, 11]. However, we note that these techniques also do not focus on the optimization of the number of CSA blocks.

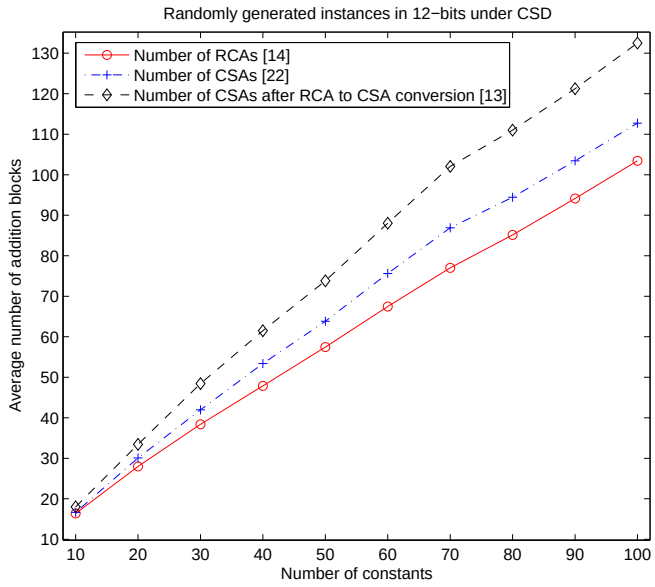


Figure 7.4: Comparison of the minimum number of RCA and CSA blocks solutions with the solutions obtained using the RCA to CSA conversion technique.

The algorithms designed for the optimization of the number of CSA blocks can be categorized in two classes: CSE and graph-based algorithms. The CSE heuristic of [12], initially, defines the coefficient multiplications in expressions and then, iteratively extracts all possible three-term divisors from the expressions, finds the best divisor, i.e., the most common divisor, among these divisors, and redefines the expressions by replacing the best divisor in the expressions with two terms, i.e., sum and carry outputs of a CSA block. The graph-based heuristic of [13] includes optimal and heuristic parts. In the optimal part, the coefficients that can be implemented using one CSA block are synthesized. If there exist unrealized coefficients after the optimal part, then in the heuristic part, an unrealized coefficient is synthesized with two CSA blocks or with its minimum number of CSA block implementation obtained from [96] by including intermediate constant(s). It is shown in [13] that the graph-based heuristic finds better solutions than a CSE heuristic, since it considers more possible implementations of constants. To the best of our knowledge, there is no exact algorithm proposed for the optimization of the number of CSA blocks in MCM.

7.2 An Exact Common Subexpression Elimination Algorithm

In this section, we introduce the exact CSE algorithm that can handle coefficients defined under binary, CSD, or MSD representation. The algorithm has three main parts: (i) generation of all possible implementations of filter coefficients using CSA blocks, (ii) construction of the Boolean network that represents the implementations of coefficients, (iii) formalization of the minimization of the number of CSA blocks problem as a 0-1 ILP problem.

7.2.1 Generation of operations

In the preprocessing phase of the algorithm, initially, an empty set called *Cset* is formed and the filter input, denoted by 1, is added to the *Cset* and labeled as implemented. Then, filter coefficients are converted to positive and made odd by successive divisions by 2. The resulting coefficients are stored in *Cset* without repetition and labeled as unimplemented. The *Cset* includes the filter input and unrepeated positive and odd coefficients to be implemented. In the following iterative loop, for each element labeled as unimplemented in *Cset*, all possible operations that implement the constant are found and their cost values are determined as the number of required CSA blocks.

1. Take an unimplemented element from *Cset*, $Cset_i$, and find its representation(s) under a given number representation. Form an empty set called $Oset_i$ that includes the required operations for the implementation of $Cset_i$ with their positive and odd inputs.

2. If the representation of $Cset_i$ includes 2 non-zero digits, then label $Cset_i$ as implemented and go to Step 6. In this case, the implementation of $Cset_i$ requires no CSA block.

3. If the representation of $Cset_i$ includes 3 non-zero digits, then store the positive and odd inputs of the operation in $Oset_i$ and assign the cost value of the operation as 1. Label $Cset_i$ as implemented and go to Step 6. In this case, the $Cset_i$ can be implemented using a single CSA block and this is the minimum cost implementation.

4. Otherwise, on each representation of $Cset_i$, find all possible operations that implement $Cset_i$. Store the positive and odd inputs of operations in $Oset_i$ and determine the cost values of the operations.

5. Label $Cset_i$ as implemented, add all elements of $Oset_i$ to *Cset* without repetition, and label them as unimplemented.

6. Repeat Step 1 until all elements in *Cset* are labeled as implemented.

Observe that the *Cset* that includes the filter input, and positive and odd filter coefficients to be implemented in the beginning of the iterative loop is augmented with partial terms that are required to implement the coefficients in later iterations. In finding the operations that implement a constant including more than 3 non-zero digits,[1] i.e., the Step 4 of the iterative loop, the non-zero digits are decomposed into two parts such that one of them includes more than two non-zero digits, and the other includes one or more than two non-zero digits. Each partition including more than two non-zero digits forms a partial term. A partial term is represented with its sum and carry outputs at the inputs of the operation, since a partial term requires CSA block(s) to be implemented. If one of the inputs of an operation is the filter input, then the cost value of the operation is determined as 1 CSA block. If none of the inputs is the filter input, then the cost value of the operation is 2 CSA blocks. As a small example, suppose that 51 is given as the filter coefficient and defined in CSD as $10\bar{1}010\bar{1}$. The implementations of 51 are given in Figure 7.5.

$$
\begin{aligned}
&\mathrm{I}_1 : S\&C_{51} = 10\bar{1}0100 + 000000\bar{1} = 13_{\ll 2} - 1 = S\&C_{13\ll 2} - 1\\
&\mathrm{I}_2 : S\&C_{51} = 10\bar{1}000\bar{1} + 0000100 = 47 + 1_{\ll 2} = S\&C_{47} + 1_{\ll 2}\\
&\mathrm{I}_3 : S\&C_{51} = 100010\bar{1} + 00\bar{1}0000 = 67 - 1_{\ll 4} = S\&C_{67} - 1_{\ll 4}\\
&\mathrm{I}_4 : S\&C_{51} = 00\bar{1}010\bar{1} + 1000000 = -13 + 1_{\ll 6} = -S\&C_{13} + 1_{\ll 6}
\end{aligned}
$$

Figure 7.5: Implementations of 51 in CSD using CSA blocks.

Observe that since the partial terms, i.e., 13, 47, and 67, include 3 non-zero digits, they are represented with sum and carry outputs. Because one of the inputs of these operations is the filter input, denoted by 1, the cost value of each operation is 1. Note that the implementation I_4 is redundant, since it includes the same positive and odd constants at the inputs as the implementation I_1 and both of them require 1 CSA

[1] Recall that a CSA block has three inputs and two outputs, i.e., sum (S) and carry (C), and these two outputs together indicate a value.

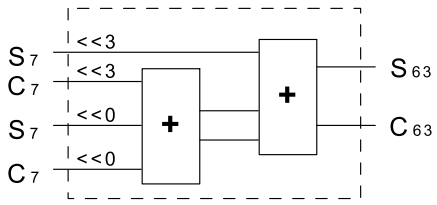


Figure 7.6: Implementation of 63 in binary using 2 CSA blocks.

block. Note that after the partial terms are found, they are added to the *Cset* without repetition, and their implementations are found as described above.

As an example on an operation that requires 2 CSA blocks, consider the constant 63 in binary, 111111. One of the operations that implements $S\&C_{63}$, i.e., $S\&C_{7\ll 3} + S\&C_7$, requires 2 CSA blocks as illustrated in Figure 7.6.

7.2.2 The Boolean network

After all possible implementations of filter coefficients and partial terms are obtained, the Boolean network that includes AND and OR gates is constructed. The primary input of the network is the filter input. An AND gate in the network represents an operation that includes CSA block(s). The output of an AND gate represents the sum and carry outputs of the CSA block implementing a constant. An OR gate associated with a coefficient or a partial term gathers all operations that implement the constant. The output of an OR gate denotes the sum and carry outputs representing a constant. The primary outputs of the network are the OR gate outputs associated with the coefficients. The network generated for the coefficient 51 defined under CSD is given in Figure 7.7, where 1-input OR gates for the partial terms, 13, 47, and 67, and the redundant implementation of 51 are omitted.

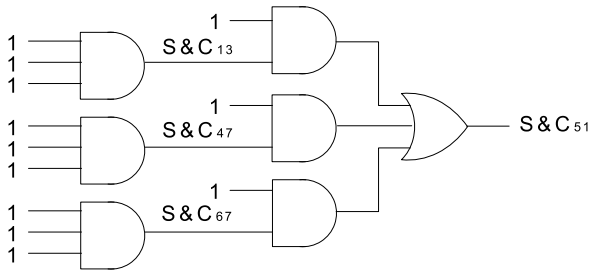


Figure 7.7: The Boolean network constructed for the coefficient 51 in CSD.

Note that 51 can be implemented using 2 CSA blocks and the Boolean network represents all possible implementations using CSA blocks with all possible partial terms that can be extracted from the CSD representation of 51.

In the conversion of the minimizing the number of CSA blocks problem to a 0-1 ILP problem, we need to include optimization variables to the network so that the cost function can be constructed. To do this, we use the minimization of the number of operations model described in Section 4.1.1.3, where the optimization variables are associated with the operations, i.e., to each AND gate in the network, an extra input denoting an optimization variable is added.

7.2.3 Conversion to 0-1 ILP problem

After the Boolean network is constructed, the modeling of the minimization of the number of CSA blocks problem as a 0-1 ILP problem is then straight-forward. The cost function is determined as the linear function of optimization variables associated with the operations where the cost value of each optimization variable is the number of CSA blocks required to implement each operation, i.e., 1 or 2. The outputs of OR gates associated with the filter coefficients are assigned to 1 value, since the implementation of the coefficients is aimed. The variable representing the filter input is also assigned to 1. The constraints of the 0-1 ILP problem are obtained by finding the CNF formulas of each gate in the network and expressing each clause in CNF formulas as a linear inequality as described in [50]. Thus, the obtained model can serve as an input to a generic 0-1 ILP solver that is used to obtain the minimum number of CSA blocks solution.

7.3 Approximate Algorithms

In this section, initially, we present the approximate CSE algorithm and then, we introduce the approximate algorithm that can handle the coefficients under general number representation.

7.3.1 The approximate common subexpression elimination algorithm

Although the exact CSE algorithm presented in the previous section can be applied on real size instances as shown in Section 7.4, naturally, there are more complex

instances that the exact algorithm cannot handle. It is because the required computation time of a generic 0-1 ILP solver to find the minimum solution tends to increase as the size of the 0-1 ILP problem increases. However, the problem size can be significantly reduced by considering only the operations that require 1 CSA block in finding the implementations of the constants, i.e., the operations whose one of the inputs is the filter input or its shifted versions. The proposed approximate CSE algorithm differs from the exact algorithm only in finding the implementations of the coefficients as described in Section 7.2.1, where the operations that require 2 CSA blocks are also considered.

In Figure 7.8, we present the average area overhead between the solutions of the approximate and exact CSE algorithms on instances where constants are generated randomly in 12 bit-width. In this experiment, the number of constants ranges between 10 and 100, and each set includes 30 instances. The results of the exact and approximate algorithms are obtained when constants are defined under binary, CSD, and MSD.

As can be observed from Figure 7.8, the solutions obtained with the approximate algorithm are quite similar to the solutions of the exact CSE algorithm, less than 0.7 CSA block on average, since the operations with 2 cost value are rarely implemented in the exact algorithm. Observe that as the number of constants increases, the difference of the solutions obtained with the approximate and exact algorithms decreases, since the partial term sharing increases with the number of constants. Also, we note that the approximate algorithm obtains solutions on 300 instances with only one extra CSA block with respect to the exact CSE algorithm when constants are defined under MSD representation.

7.3.2 The approximate algorithm under general number representation

Since the results of CSE algorithms depend on the number representation, the minimum number of operations solutions cannot be obtained using an exact CSE algorithm as shown in Chapter 4. We extend the approximate CSE algorithm to handle coefficients in general number representation as described in Section 4.2. The proposed approximate algorithm under general number representation differs from the approximate CSE algorithm only in considering the operations that can be used

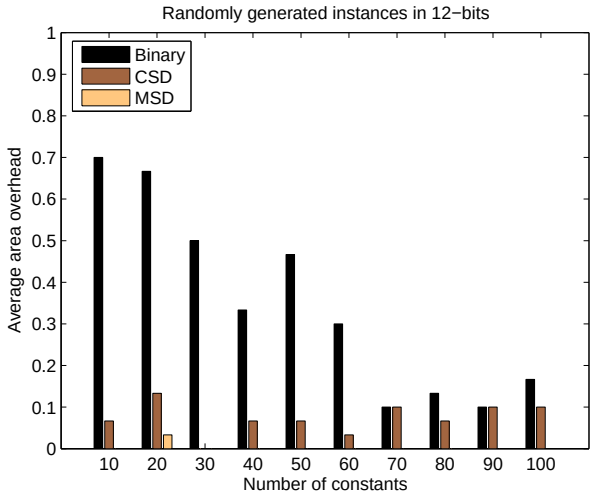


Figure 7.8: Area overhead between the approximate and exact CSE algorithms on randomly generated instances.

to implement the constants. In the proposed algorithm, initially, the positive and odd numbers between 1 and $2^{bw+1}-1$, where *bw* denotes the maximum bit-width of coefficients, are sorted in ascending order of the number of non-zero digits in CSD and stored in a set called *Nset*. Then, for each coefficient and partial term, the operations that implement the constant are generated by assigning the filter input and its shifted versions with positive and negative sign to the first input of the operations and finding the partial terms required to implement the constant. The sum and carry outputs of the CSA block that implements the partial term are assigned to the other two inputs of the operations. The valid operations are determined as the operations that include partial terms located in the *Nset* before the position of the constant in the *Nset* to guarantee the solution to be acyclic as described in Section 4.2. For example, again, consider 51 as a filter coefficient. All implementations of 51 generated under general number representation are given in Figure 7.9.

$$
\begin{aligned}
&I_1 : S\&C_{51} = 1 + 25_{\ll 1} = 1 + S\&C_{25\ll 1} && I_8 : S\&C_{51} = -1_{\ll 3} + 59 = -1_{\ll 3} + S\&C_{59}\\
&I_2 : S\&C_{51} = -1 + 13_{\ll 2} = -1 + S\&C_{13\ll 2} && I_9 : S\&C_{51} = 1_{\ll 4} + 35 = 1_{\ll 4} + S\&C_{35}\\
&I_3 : S\&C_{51} = 1_{\ll 1} + 49 = 1_{\ll 1} + S\&C_{49} && I_{10} : S\&C_{51} = -1_{\ll 4} + 67 = -1_{\ll 4} + S\&C_{67}\\
&I_4 : S\&C_{51} = -1_{\ll 1} + 53 = -1_{\ll 1} + S\&C_{53} && I_{11} : S\&C_{51} = 1_{\ll 5} + 19 = 1_{\ll 5} + S\&C_{19}\\
&I_5 : S\&C_{51} = 1_{\ll 2} + 47 = 1_{\ll 2} + S\&C_{47} && I_{12} : S\&C_{51} = -1_{\ll 5} + 83 = -1_{\ll 5} + S\&C_{83}\\
&I_6 : S\&C_{51} = -1_{\ll 2} + 55 = -1_{\ll 2} + S\&C_{55} && I_{13} : S\&C_{51} = 1_{\ll 6} - 13 = 1_{\ll 6} - S\&C_{13}\\
&I_7 : S\&C_{51} = 1_{\ll 3} + 43 = 1_{\ll 3} + S\&C_{43} && I_{14} : S\&C_{51} = -1_{\ll 6} + 115 = -1_{\ll 6} + S\&C_{115}
\end{aligned}
$$

Figure 7.9: Implementations of 51 under general number representation.

We note that the operations I_4, I_{12}, and I_{14} are not accepted for the implementation of 51, since the locations of the partial terms in these implementations, i.e., 53, 83, and 115, in *Nset* are beyond the position of 51 in *Nset*. Also, observe that the implementation I_{13} is redundant, since it includes the same positive and odd inputs as the implementation I_2. Thus, there are 10 implementations of 51 including different partial terms that may increase the partial term sharing in MCM. As can be easily observed, by using the general number representation we find the same implementations that can be obtained in the approximate CSE algorithm under any number representation and furthermore, we consider the implementations of constants that cannot be obtained with the non-zero digit combinations under a number representation of the constants.

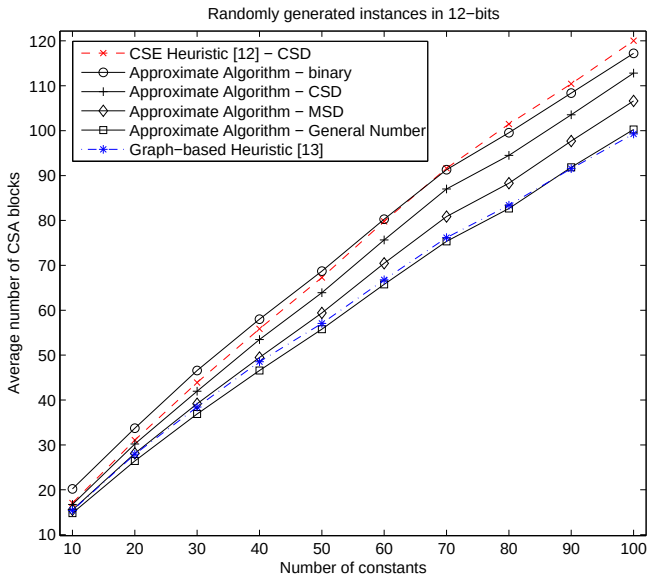
After the implementations of filter coefficients and the partial terms are obtained, the Boolean network is constructed and the 0-1 ILP problem that defines the minimization of the number of CSA blocks is formed as described in Section 7.2. Finally, the minimum solution is found using a generic 0-1 ILP solver.

7.4 Experimental Results

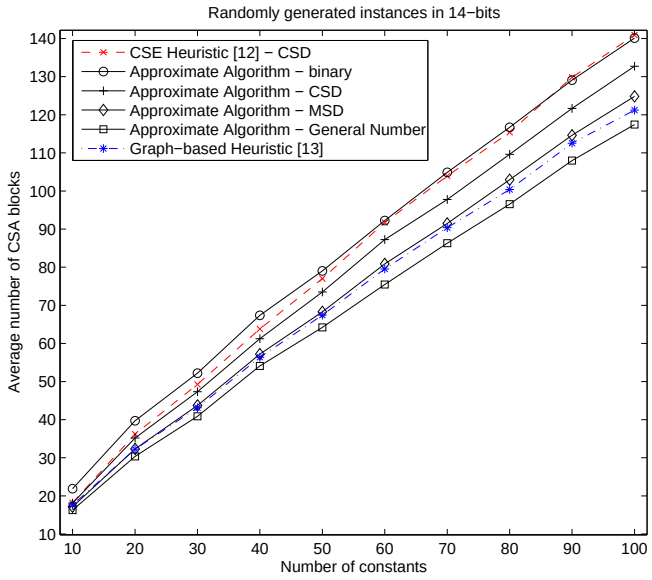
In this section, we present the results of the exact and approximate algorithms on randomly generated and filter instances, and compare with the results of heuristics of [12, 13]. The CSE heuristic of [12] was also implemented and the graph-based heuristic of [13] was provided by Oscar Gustafsson.

As the first experiment set, we used randomly generated instances where constants are defined in 12 and 14 bit-width. The number of constants ranges between 10 and 100, and each set includes 30 instances, totally 600 instances. The results of the approximate algorithm and previously proposed heuristics on randomly generated instances defined in 12 and 14 bits are presented in Figure 7.10(a) and (b) respectively.

As can be easily observed from Figure 7.10, the representation of constants in MSD yields better solutions than those of binary and CSD. Also, the approximate CSE algorithm gives superior results than the CSE heuristic of [12]. The difference of average number of CSA blocks between the results of CSE heuristic [12] and the approximate algorithm in CSD reaches up to 7.2 and 8.2 on instances with 100



(a)



(b)

Figure 7.10: Comparison of the heuristic algorithms on randomly generated instances: (a) Constants in 12 bit-width; (b) Constants in 14 bit-width.

Table 7.1: 0-1 ILP problem sizes of the FIR filter instances.

Filter	Exact CSE Algorithm			Approximate Algorithm					
	MSD			MSD			General Number		
	vars	cons	optvars	vars	cons	optvars	vars	cons	optvars
1	506	809	271	126	165	96	5583	11138	2777
2	823	1345	426	84	74	74	3102	5882	1630
3	515	795	275	65	47	63	6638	12948	3398
4	1744	3108	871	141	205	105	9145	18357	4538
5	851	1362	442	199	275	145	24623	49916	12116
6	1581	2678	791	264	401	186	17479	35337	8615
7	11742	22726	5658	1768	4689	876	426650	876949	207350
8	8854	17091	4272	1338	3244	696	138176	283099	67358
9	27261	57079	13164	3117	8777	1397	539405	1106400	262733
Total	53877	106993	26170	7102	17877	3638	1170801	2400026	570515

constants under 12 and 14 bit-width respectively. Since the graph-based heuristic of [13] is not limited to any number representation, it obtains better solutions than CSE algorithms. However, it is interesting to note that the approximate algorithm under MSD gives competitive solutions with the graph-based heuristic of [13] on instances with small number of constants. On the other hand, the approximate algorithm under general number representation gives better solutions than the graph-based heuristic on average, except the instances with 90 and 100 constants in 12 bit-width. The maximum difference of the average number of CSA blocks solutions between the graph-based heuristic and the approximate algorithm under general number representation is obtained as 4.5 CSA blocks on instances with 90 constants under 14 bit-width.

As the second experiment set, we used FIR filter instances presented in Table 4.13. The size of 0-1 ILP problems obtained with the exact and approximate algorithms under MSD and general number representations are presented in Table 7.1.

As can be easily observed from Table 7.1, the approximate CSE algorithm generates smaller size 0-1 ILP problems with respect to the 0-1 ILP problems obtained by the exact CSE algorithm. Also, we note that the size of 0-1 ILP problems can be very large when coefficients are defined under general number representation, e.g., Filter 7 and 9, since more possible implementations of coefficients are considered under general number representation. It is worth to mention that current 0-1 ILP solvers can deal with such large size problems as shown in Table 7.2.

Table 7.2: Summary of results of algorithms on the FIR filter instances.

Filter	[12]	Exact CSE Algorithm			Approximate Algorithm					[13]
	CSD	CSD	MSD		CSD	MSD		General Number		
	CSA	CSA	CSA	CPU	CSA	CSA	CPU	CSA	CPU	CSA
1	16	16	16	0.1	16	16	0.1	15	2.7	16
2	30	28	27	0.2	28	27	0.1	25	0.3	25
3	31	31	31	0.1	31	31	0.1	31	0.6	31
4	25	25	21	0.3	25	21	0.1	21	3.9	22
5	36	35	34	0.3	35	34	0.1	34	13.5	35
6	36	34	32	0.5	34	32	0.1	32	6.8	34
7	60	60	55	52.1	60	55	0.5	53	4652.1	54
8	62	61	57	47.6	63	57	0.5	53	937.2	55
9	88	85	82	754.9	86	83	6.0	78	27165.9	81
Total	384	375	355	856.1	378	356	7.3	342	32783	353

The results of algorithms on filter instances are presented in Table 7.2. In this table, *CSA* represents the number of CSA blocks and *CPU* is the required CPU time in seconds of the 0-1 ILP solver, *glpPB* [62], to find the minimum solution on a PC with Intel Xeon at 3.16GHz and 8GB memory.

As can be observed from Table 7.2, the exact CSE algorithm finds similar or better solutions than the CSE heuristic of [12]. The difference of solutions between the CSE heuristic and the exact CSE algorithm under CSD is 1 CSA block on average. Also, the exact CSE algorithm finds better solutions than the graph-based heuristic of [13] on Filter 4, 5, and 6 when coefficients are defined under MSD. We note that the approximate algorithm finds similar solutions with the exact algorithm under MSD using less computational effort. Observe that the approximate algorithm under general number representation obtains the same or better results than the graph-based heuristic of [13]. The difference of solutions between the graph-based heuristic and the approximate algorithm under general number representation is 1.2 CSA blocks on average.

7.5 Conclusions

In this chapter, we introduce an exact CSE algorithm designed for the minimization of the number of CSA blocks in the implementation of the multiplier block of a digital FIR filter. We also present an approximate algorithm based on the exact CSE algorithm that considers limited implementations of the coefficients. It is shown that the approximate algorithm obtains similar results with the exact CSE algorithm using

a little computational effort. Furthermore, the approximate algorithm is extended to handle coefficients under general number representation that achieves more possible implementations of the coefficients. The proposed algorithms in this chapter, exact and approximate, have been tested on randomly generated and FIR filter instances and it is shown that the proposed algorithms can be applied on real size instances. Also, we compare our algorithms with the previously proposed heuristics. It is observed from the experimental results that the exact and approximate algorithms give much better results than the CSE heuristic and the approximate algorithm under general number representation obtains more promising solutions than the graph-based heuristic.

8. DISCUSSIONS AND CONCLUSIONS

In this thesis, the problem of finding the fewest number of addition/subtraction operations to implement the multiple constant multiplications has been addressed. We resorted to the previously proposed exact CSE algorithm that computes the minimum number of operations solution of the MCM by modeling the MCM problem as a 0-1 ILP problem when constants are defined under a number representation. To extend the applicability of the exact CSE algorithm to larger size instances, problem reduction and model simplification techniques that significantly reduce the search space were introduced. It was shown by the experimental results that the exact CSE algorithm can be easily applied on real size instances including up to 16 bit-width constants with these techniques. We note that the minimum solution of the MCM problem instances including large number of constants, e.g., greater than 30 constants in 16 bit-width, can be obtained in a reasonable time by the exact CSE algorithm, since large number of constants increases the possible partial term sharing. To cope with more complex instances that the exact CSE algorithm cannot handle, such as MCM instances including 24 bit-width constants, we introduced an approximate algorithm that gives competitive solutions with the exact algorithm and significantly better results than the previously proposed heuristics. However, since the approximate CSE algorithm considers more parameters to make a better decision in each iteration of its heuristic part, the run-time of the approximate algorithm is generally greater than those of the previously proposed CSE heuristics.

Since the results of the CSE algorithms depend on the number representation, the exact CSE algorithm was extended to handle the constants under general number representation. It was observed from the experimental results that the proposed algorithm under general number representation obtains significantly better solutions than those of the exact CSE algorithm under any number representation, namely, binary, CSD, or MSD. However, this advantage comes with the increase in the size of ILP problems consequently, in the required time of the 0-1 ILP solver to find the

minimum solution. However, we note that the exact algorithm under general number representation can be easily applied on MCM instances including up to 14 bit-width constants.

We note that since the performance and applicability of the exact algorithms are related with the 0-1 ILP solvers used to find the minimum solution and there is a tremendous effort on the design of highly efficient 0-1 ILP solvers, specially on SAT-based 0-1 ILP solvers, we believe that the MCM problems that cannot be solved in a reasonable time or cannot be handled by the current 0-1 ILP solvers will be easily solved in near future.

Furthermore, an exact graph-based algorithm that can be applied on less complex instances including up to 14 bit-width constants was introduced. Because finding the minimum number of operations solution of the MCM problem is intractable on more complex instances, an approximate graph-based algorithm based on the exact graph-based algorithm was proposed. It was shown by the experimental results that the proposed approximate algorithm finds similar solutions with the exact algorithm, and obtains competitive and better results than the prominent graph-based heuristics. We note that the approximate graph-based algorithm can be easily applied on instances including up to 18 bit-width constants. Same as the approximate CSE algorithm, the approximate graph-based algorithm finds a solution using more computational effort than the efficient graph-based heuristics. This is simply because, finding the "best" decision at each iteration that gives the "best" final solution requires to consider more parameters.

From all the experiments, we also observed that the constants as themselves in the MCM problem is another parameter that determines the MCM problem as a hard problem for the exact, approximate, and heuristic algorithms additional to other parameters, i.e., the number of constants and their size.

To apply the exact CSE algorithm on more sophisticated problems such as, the minimization of the number of operations under a delay constraint, the minimization of area in terms of gate-level metrics, and the minimization of the number of CSA blocks in MCM, we also introduced an alternative optimization model.

To design an exact CSE algorithm for the MCM problem under a delay constraint, the delay constraints were determined and added to the 0-1 ILP problem that is formalized to find the minimum number of operations solutions. To cope with large size instances, an approximate CSE algorithm that finds competitive solutions with the exact CSE algorithm and better solutions than the efficient CSE heuristics was introduced.

To design an exact CSE algorithm that minimizes the area of the MCM in terms of gate-level metrics, we relied on the actual architectures based on HAs and FAs for addition and subtraction operations under unsigned and signed input models and formalized the minimization of area problem as a 0-1 ILP problem. It was shown by the experimental results that if the objective is limited to the minimization of the number of operations, the actual area of the hardware implementation can be far from the optimum.

To design an exact CSE algorithm for the minimization of the number of CSA blocks in MCM, initially, all possible implementations of constants using CSA blocks were found when constants are defined under a number representation and then, the optimization problem was modeled as a 0-1 ILP problem. Also, an approximate CSE algorithm that considers the limited implementations of constants was presented. Furthermore, the approximate CSE algorithm was extended to handle the constants under general number representation. It was shown by the experimental results that the exact and approximate algorithms give much better results than the CSE heuristic and the approximate algorithm under general number representation obtains more promising solutions than the prominent graph-based heuristic.

As future work, we plan to apply the proposed algorithms to the systems that include MCM such as, two-dimensional digital FIR filters and linear DSP transforms. Also, we plan to apply the approximate graph-based algorithm designed for the MCM problem to more sophisticated MCM problems.

REFERENCES

[1] **Nguyen, H. and Chatterjee, A.**, 2000. Number-Splitting with Shift-and-Add Decomposition for Power and Hardware Optimization in Linear DSP Synthesis, *IEEE Transactions on VLSI*, **8(4)**, 419–424.

[2] **Cappello, P. and Steiglitz, K.**, 1984. Some Complexity Issues in Digital Signal Processing, *IEEE Transactions on Acoustics, Speech, and Signal Processing*, **32(5)**, 1037–1041.

[3] **Gustafsson, O. and Wanhammar, L.**, 2002. ILP Modelling of the Common Subexpression Sharing Problem, *Proceedings of International Conference on Electronics, Circuits and Systems*, pp. 1171–1174.

[4] **Flores, P.**, **Monteiro, J. and Costa, E.**, 2005. An Exact Algorithm for the Maximal Sharing of Partial Terms in Multiple Constant Multiplications, *Proceedings of International Conference on Computer-Aided Design*, pp. 13–16.

[5] **Kang, H.J. and Park, I.C.**, 2001. FIR Filter Synthesis Algorithms for Minimizing the Delay and the Number of Adders, *IEEE Transactions on Circuits and Systems II: Analog and Digital Signal Processing*, **48(8)**, 770–777.

[6] **Hosangadi, A.**, **Fallah, F. and Kastner, R.**, 2005. Simultaneous Optimization of Delay and Number of Operations in Multiplierless Implementation of Linear Systems, *Proceedings of International Workshop on Logic Synthesis*.

[7] **Costa, E.**, **Flores, P. and Monteiro, J.**, 2005. Maximal Sharing of Partial Terms in MCM under Minimal Signed Digit Representation, *Proceedings of IEEE European Conference on Circuit Theory and Design*, pp. 221–224.

[8] **Dempster, A.**, **Demirsoy, S. and Kale, I.**, 2002. Designing Multiplier Blocks with Low Logic Depth, *Proceedings of IEEE International Symposium on Circuits and Systems*, pp. 773–776.

[9] **Johansson, K.**, **Gustafsson, O. and Wanhammar, L.**, 2005. A Detailed Complexity Model for Multiple Constant Multiplication and an Algorithm to Minimize the Complexity, *Proceedings of IEEE European Conference on Circuit Theory and Design*, pp. 465–468.

[10] **Kim, T.**, **Jao, W. and Tjiang, S.**, 1998. Circuit Optimization using Carry-Save-Adder Cells, *IEEE Transactions on Computer-Aided Design of Integrated Circuits*, **17(10)**, 974–984.

[11] **Verna, A. and Ienne, P.**, 2004. Improved Use of the Carry-Save Representation for the Synthesis of Complex Arithmetic Circuits, *Proceedings of International Conference on Computer-Aided Design*, pp. 791–798.

[12] **Hosangadi, A.**, **Fallah, F. and Kastner, R.**, 2006. Optimizing High Speed Arithmetic Circuits using Three-Term Extraction, *Proceedings of Design, Automation and Test in Europe Conference*, pp. 1294–1299.

[13] **Gustafsson, O.**, **Dempster, A. and Wanhammar, L.**, 2004. Multiplier Blocks using Carry-Save Adders, *Proceedings of IEEE International Symposium on Circuits and Systems*, pp. 473–476.

[14] **Aksoy, L.**, **Gunes, E.**, **Costa, E.**, **Flores, P. and Monteiro, J.**, 2007. Effect of Number Representation on the Achievable Minimum Number of Operations in Multiple Constant Multiplications, *Proceedings of IEEE Workshop in Signal Processing Systems*, pp. 424–429.

[15] **Aksoy, L.**, **Costa, E.**, **Flores, P. and Monteiro, J.**, 2008. Exact and Approximate Algorithms for the Optimization of Area and Delay in Multiple Constant Multiplications, *IEEE Transactions on Computer-Aided Design of Integrated Circuits*, **27(6)**, 1013–1026.

[16] **Aksoy, L.**, **Costa, E.**, **Flores, P. and Monteiro, J.**, 2006. ASSUMEs: Heuristic Algorithms for Optimization of Area and Delay in Digital Filter Synthesis, *Proceedings of International Conference on Electronics, Circuits and Systems*, pp. 748–751.

[17] **Aksoy, L.**, **Costa, E.**, **Flores, P. and Monteiro, J.**, 2007. Minimum Number of Operations under a General Number Representation for Digital Filter Synthesis, *Proceedings of IEEE European Conference on Circuit Theory and Design*, pp. 252–255.

[18] **Aksoy, L.**, **Gunes, E. and Flores, P.**, 2008. An Exact Breadth-First Search Algorithm for the Multiple Constant Multiplications Problem, *Proceedings of IEEE Norchip Conference*, pp. 41–46.

[19] **Aksoy, L. and Gunes, E.**, 2008. An Approximate Algorithm for the Multiple Constant Multiplications Problem, *Proceedings of Symposium on Integrated Circuits and Systems Design*, pp. 58–63.

[20] **Aksoy, L.**, **Costa, E.**, **Flores, P. and Monteiro, J.**, 2006. Optimization of Area under a Delay Constraint in Digital Filter Synthesis using SAT-based Integer Linear Programming, *Proceedings of Design Automation Conference*, pp. 669–674.

[21] **Aksoy, L.**, **Costa, E.**, **Flores, P. and Monteiro, J.**, 2007. Optimization of Area in Digital FIR Filters using Gate-Level Metrics, *Proceedings of Design Automation Conference*, pp. 420–423.

[22] **Aksoy, L. and Gunes, E.**, 2008. Area Optimization Algorithms in High-Speed Digital FIR Filter Synthesis, *Proceedings of Symposium on Integrated Circuits and Systems Design*, pp. 64–69.

[23] **Avizienis, A.**, 1961. Signed-digit Number Representation for Fast Parallel Arithmetic, *IRE Transactions on Electronic Computers*, **EC-10**, 389–400.

[24] **Garner, H.**, 1965. Number Systems and Arithmetic, *Advances in Computers*, **6**, 131–194.

[25] **Reitwiesner, G.**, 1960. Binary Arithmetic, *Advances in Computers*, **1**, 261–265.

[26] **Backenius, E.**, **Sall, E. and Gustafsson, O.**, 2006. Bidirectional Conversion to Minimum Signed-Digit Representation, *Proceedings of IEEE International Symposium on Circuits and Systems*, pp. 2413–2416.

[27] **Xu, F.**, **Chang, C.H. and Jong, C.C.**, 2004. HWP: A New Insight into Canonical Signed Digit, *Proceedings of IEEE International Symposium on Circuits and Systems*, pp. 201–204.

[28] **Park, I.C. and Kang, H.J.**, 2001. Digital Filter Synthesis Based on Minimal Signed Digit Representation, *Proceedings of Design Automation Conference*, pp. 468–473.

[29] **Larrabee, T.**, 1992. Test Pattern Generation Using Boolean Satisfiability, *IEEE Transactions on Computer-Aided Design of Integrated Circuits*, **11(1)**, 4–15.

[30] **Cook, S.**, 1971. The Complexity of Theorem-Proving Procedures, *Proceedings of Third Annual ACM Symposium on Theory of Computing*, pp. 151–158.

[31] **Devadas, S.**, **Keutzer, K. and Malik, S.**, 1993. Computation of Floating Mode Delay in Combinational Circuits: Practice and Implementation, *IEEE Transactions on Computer-Aided Design of Integrated Circuits*, **12(12)**, 1923–1936.

[32] **Mishchenko, A.**, **Chatterjee, S.**, **Brayton, R. and Een, N.**, 2006. Improvements to Combinational Equivalence Checking, *Proceedings of International Conference on Computer-Aided Design*, pp. 836–843.

[33] **Smith, A.**, **Veneris, A.**, **Ali, M. and Viglas, A.**, 2005. Fault Diagnosis and Logic Debugging using Boolean Satisfiability, *IEEE Transactions on Computer-Aided Design of Integrated Circuits*, **24(10)**, 1606–1621.

[34] **Flores, P.**, **Neto, H. and Marques-Silva, J.**, 2001. An Exact Solution to the Minimum Size Test Pattern Problem, *ACM Transactions on Design Automation of Electronic Systems*, **6(4)**, 629–644.

[35] **Fuhrer, R. and Nowick, S.**, 1998. Exact Optimal State Minimization for 2-Level Output Logic, *Proceedings of International Workshop on Logic Synthesis*.

[36] **Taylor, B. and Pileggi, L.**, 2007. Exact Combinatorial Optimization Methods for Physical Design of Regular Log, *Proceedings of Design Automation Conference*, pp. 344–349.

[37] **Zhang, H.**, 1997. SATO: An Efficient Propositional Prover, *Proceedings of International Conference on Automated Deduction*, pp. 272–275.

[38] **Selman, B.**, **Levesque, H. and Mitchell, D.**, 1992. A New Method for Solving Hard Satisfiability Problems, *Proceedings of Tenth National Conference on Artificial Intelligence*, pp. 440–446.

[39] **Selman, B.**, **Kautz, H. and Cohen, B.**, 1994. Noise Strategies for Improving Local Search, *Proceedings of Twelfth National Conference on Artificial Intelligence*, pp. 337–343.

[40] **Spears, V.**, 1996. Simulated Annealing for Hard Satisfiability Problems, *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, **26**, 533–558.

[41] **Gottlieb, J.**, **Marchiori, E. and Rossi, C.**, 2002. Evolutionary Algorithms for the Satisfiability Problem, *Evolutionary Computation*, **10(1)**, 35–50.

[42] **Lardeux, F.**, **Saubion, F. and Hao, J.K.**, 2006. GASAT: A Genetic Local Search Algorithm for the Satisfiability Problem, *Evolutionary Computation*, **14(2)**, 223–253.

[43] **Aksoy, L. and Gunes, E.**, 2006. An Evolutionary Local Search Algorithm for the Satisfiability Problem, *Lecture Notes in Computer Science*, **3949**, 185–193.

[44] **Davis, M.**, **Logemann, G. and Loveland, D.**, 1962. A Machine Program for Theorem-Proving, *Communications of the Association for Computing Machinery*, **5**, 394–397.

[45] **Lynce, I.**, 2004. *Propositional Satisfiability: Techniques, Algorithms and Applications*, Ph.D. thesis, Universidade Técnica de Lisboa, Instituto Superior Técnico.

[46] **Marques-Silva, J. and Sakallah, K.**, 1999. GRASP: A Search Algorithm for Propositional Satisfiability, *IEEE Transactions on Computers*, **48(5)**, 506–521.

[47] **Moskewicz, M.**, **Madigan, C.**, **Zhao, Y.**, **Zhang, L. and Malik, S.**, 2001. Chaff: Engineering an Efficient SAT Solver, *Proceedings of Design Automation Conference*, pp. 530–535.

[48] **Een, N. and Sorensson, N.**, 2003. An Extensible SAT-solver, *Proceedings of International Conference on Theory and Applications of Satisfiability Testing*, pp. 502–518.

[49] **Aloul, F.**, **Ramani, A.**, **Markov, I. and Sakallah, K.**, 2002. Generic ILP versus Specialized 0-1 ILP: An Update, *Proceedings of International Conference on Computer-Aided Design*, pp. 450–457.

[50] **Barth, P.**, 1995. *A Davis-Putnam Based Enumeration Algorithm for Linear Pseudo-Boolean Optimization*, Technical report, Max-Planck-Institut Fur Informatik.

[51] **Warners, J.**, 1998. A Linear-time Transformation of Linear Inequalities into Conjunctive Normal Form, *Information Processing Letters*, **68(2)**, 63–69.

[52] **Een, N. and Sorensson, N.**, 2006. Translating Pseudo-Boolean Constraints into SAT, *Journal on Satisfiability, Boolean Modeling and Computation*, **2**, 1–26.

[53] **Villa, T.**, 1995. *Encoding Problems in Logic Synthesis*, Ph.D. thesis, EECS Department, University of California, Berkeley.

[54] **Coudert, O.**, 1996. On Solving Covering Problems, *Proceedings of Design Automation Conference*, pp. 197–202.

[55] **Liao, S. and Devadas, S.**, 1997. Solving Covering Problems using LPR-Based Lower Bounds, *Proceedings of Design Automation Conference*, pp. 117–120.

[56] **Manquinho, V. and Marques-Silva, J.**, 2002. Search Pruning Techniques in SAT-based Branch-and-Bound Algorithms for the Binate Covering Problem, *IEEE Transactions on Computer-Aided Design of Integrated Circuits*, **21(5)**, 505–516.

[57] **Ahuja, T.**, **Magnanti, T. and Orlin, J.**, 1993. *Network Flows: Theory, Algorithms, and Applications*, Prentice Hall.

[58] **Villa, T.**, **Kam, T.**, **Brayton, R. and Sangiovanni-Vincentelli, A.**, 1997. Explicit and Implicit Algorithms for Binate Covering Problems, *IEEE Transactions on Computer-Aided Design of Integrated Circuits*, **16(7)**, 677–691.

[59] **Manquinho, V. and Marques-Silva, J.**, 2005. Effective Lower Bounding Techniques for Pseudo-Boolean Optimization, *Proceedings of Design, Automation and Test in Europe Conference*, pp. 660–665.

[60] **Sheini, H. and Sakallah, K.**, 2006. Pueblo: A Hybrid Pseudo-Boolean SAT Solver, *Journal on Satisfiability, Boolean Modeling and Computation*, **2**, 61–96.

[61] **Pseudo-Boolean Evaluation PB'07 website**, <*http://www.cril.univ-artois.fr/PB07/*>, accessed at 02.05.2009.

[62] **glpPB website**, <*http://www.eecs.umich.edu/∼hsheini/*>, accessed at 02.05.2009.

[63] **Knuth, D.**, 1969. *The Art of Computer Programming: Seminumerical Algorithms Volume 2*, Addison-Wesley.

[64] **Gustafsson, O.**, 2007. Lower Bounds for Constant Multiplication Problems, *IEEE Transactions on Circuits and Systems II: Analog and Digital Signal Processing*, **54(11)**, 974–978.

[65] **Wu, H. and Hasan, M.**, 1999. Closed-form Expression for the Average Weight of Signed-Digit Representations, *IEEE Transactions on Computers*, **48(8)**, 848–851.

[66] **Lefevre, V.**, 2001. *Multiplication by an Integer Constant*, Technical report, Institut National de Recherche en Informatique et en Automatique.

[67] **Dimitrov, V.**, **Imbert, L. and Zakaluzny, A.**, 2007. Multiplication by a Constant is Sublinear, *Proceedings of the IEEE Symposium on Computer Arithmetic*, pp. 261–268.

[68] **Dempster, A. and Macleod, M.**, 1994. Constant Integer Multiplication using Minimum Adders, *IEE Proceedings - Circuits, Devices, and Systems*, **141(5)**, 407–413.

[69] **Gustafsson, O.**, **Dempster, A. and Wanhammar, L.**, 2002. Extended Results for Minimum-adder Constant Integer Multipliers, *Proceedings of IEEE International Symposium on Circuits and Systems*, pp. 73–76.

[70] **Jain, R.**, **Yang, P. and Yoshino, T.**, 1991. FIRGEN: A Computer-Aided Design System for High Performance FIR Filter Integrated Circuits, *IEEE Transactions on Signal Processing*, **39(7)**, 1655–1668.

[71] **Mehendale, M.**, **Sherlekar, S. and Venkatesh, G.**, 1995. Techniques for Low Power Realization of FIR Filters, *Proceedings of Design Automation Conference*, pp. 404–416.

[72] **Samueli, H.**, 1989. An Improved Search Algorithm for the Design of Multiplierless FIR Filters with Power-of-Two Coefficients, *IEEE Transactions on Circuits and Systems*, **36(7)**, 1044–1047.

[73] **Nannarelli, A.**, **Re, M. and Cardarilli, G.**, 2001. Tradeoffs between Residue Number System and Traditional FIR Filters, *Proceedings of IEEE International Symposium on Circuits and Systems*, pp. 305–308.

[74] **Muhammad, K. and Roy, K.**, 2002. A Graph Theoretic Approach for Synthesizing Very Low-Complexity High-Speed Digital Filters, *IEEE Transactions on Computer-Aided Design of Integrated Circuits*, **21(2)**, 204–216.

[75] **Hartley, R.**, 1991. Optimization of Canonic Signed Digit Multipliers for Filter Design, *Proceedings of IEEE International Symposium on Circuits and Systems*, pp. 1992–1995.

[76] **Hartley, R.**, 1996. Subexpression Sharing in Filters using Canonic Signed Digit Multipliers, *IEEE Transactions on Circuits and Systems II*, **43(10)**, 677–688.

[77] **Hosangadi, A.**, **Fallah, F. and Kastner, R.**, 2005. Reducing Hardware Complexity of Linear DSP Systems by Iteratively Eliminating Two-Term Common Subexpressions, *Proceedings of Asia and South Pacific Design Automation Conference*, pp. 523–528.

[78] **Potkonjak, M.**, **Srivastava, M. and Chandrakasan, A.**, 1996. Multiple Constant Multiplications: Efficient and Versatile Framework and Algorithms for Exploring Common Subexpression Elimination, *IEEE Transactions on Computer-Aided Design of Integrated Circuits*, **15(2)**, 151–165.

[79] **Pasko, R.**, **Schaumont, P.**, **Derudder, V.**, **Vernalde, S. and Durackova, D.**, 1999. A New Algorithm for Elimination of Common Subexpressions, *IEEE Transactions on Computer-Aided Design of Integrated Circuits*, **18(1)**, 58–68.

[80] **Xu, F.**, **Chang, C.H. and Jong, C.C.**, 2005. Contention Resolution Algorithm for Common Subexpression Elimination in Digital Filter Design, *IEEE Transactions on Circuits and Systems II: Express Briefs*, **52(10)**, 695–700.

[81] **Mahesh, R. and Vinod, A.**, 2008. A New Common Subexpression Elimination Algorithm for Realizing Low-Complexity Higher Order Digital Filters, *IEEE Transactions on Computer-Aided Design of Integrated Circuits*, **27(2)**, 217–229.

[82] **Yurdakul, A. and Dundar, G.**, 1999. Multiplierless Realization of Linear DSP Transforms by Using Common Two-Term Expressions, *The Journal of VLSI Signal Processing*, **22(3)**, 163–172.

[83] **Costa, E.**, **Flores, P. and Monteiro, J.**, 2006. Exploiting General Coefficient Representation for the Optimal Sharing of Partial Products in MCMs, *Proceedings of Symposium on Integrated Circuits and Systems Design*, pp. 161–166.

[84] **Dempster, A. and Macleod, M.**, 2004. Using All Signed-Digit Representations To Design Single Integer Multipliers Using Subexpression Elimination, *Proceedings of IEEE International Symposium on Circuits and Systems*, pp. 165–168.

[85] **Dempster, A. and Macleod, M.**, 2004. Digital Filter Design using Subexpression Elimination and All Signed-Digit Representations, *Proceedings of IEEE International Symposium on Circuits and Systems*, pp. 169–172.

[86] **Bull, D. and Horrocks, D.**, 1991. Primitive Operator Digital Filters, *IEE Proceedings G: Circuits, Devices and Systems*, **138(3)**, 401–412.

[87] **Dempster, A. and Macleod, M.**, 1995. Use of Minimum-Adder Multiplier Blocks in FIR Digital Filters, *IEEE Transactions on Circuits and Systems II*, **42(9)**, 569–577.

[88] **Voronenko, Y. and Püschel, M.**, 2007. Multiplierless Multiple Constant Multiplication, *ACM Transactions on Algorithms*, **3(2)**.

[89] **Han, J.H. and Park, I.C.**, 2008. FIR Filter Synthesis Considering Multiple Adder Graphs for a Coefficient, *IEEE Transactions on Computer-Aided Design of Integrated Circuits*, **27(5)**, 958–962.

[90] **Velev, M.**, 2004. Efficient Translation of Boolean Formulas to CNF in Formal Verification of Microprocessors, *Proceedings of Asia and South Pacific Design Automation Conference*, pp. 310–315.

[91] **Spiral website**, <*http://www.spiral.net*>, accessed at 02.05.2009.

[92] **Peiro, M.**, **Boemo, E. and Wanhammar, L.**, 2002. Design of High-Speed Multiplierless Filters using a Nonrecursive Signed Common Subexpression Algorithm, *IEEE Transactions on Circuits and Systems II: Analog and Digital Signal Processing*, **49(3)**, 196–203.

[93] **Potkonjak, M.**, **Srivastava, M. and Chandrakasan, A.**, 1994. Efficient Substitution of Multiple Constant Multiplication by Shifts and Additions using Iterative Pairwise Matching, *Proceedings of Design Automation Conference*, pp. 189–194.

[94] **Kang, H.J.**, **Kim, H. and Park, I.C.**, 2000. FIR Filter Synthesis Algorithms for Minimizing the Delay and the Number of Adders, *Proceedings of International Conference on Computer-Aided Design*, pp. 51–54.

[95] **Hawley, R.**, **Wong, B.**, **Lin, T.J.**, **Laskowski, J. and Samueli, H.**, 1996. Design Techniques for Silicon Compiler Implementations of High-Speed FIR Digital Filters, *IEEE Journal of Solid-State Circuits*, **31(5)**, 656–667.

[96] **Gustafsson, O.**, **Ohlsson, H. and Wanhammar, L.**, 2001. Minimum-Adder Integer Multipliers using Carry-Save Adders, *Proceedings of IEEE International Symposium on Circuits and Systems*, pp. 709–712.

CURRICULUM VITA

Candidate's full name:	Levent Aksoy
Place and date of birth:	İstanbul, 19.09.1976
Permanent Address:	Tepeüstü Temel sokak No:65 Küçükçekmece, 34290, İstanbul
Universities and Colleges attended:	B.Sc., Yıldız Technical University M.Sc., İstanbul Technical University

Publications:

- **Aksoy, L.** and Gunes, E.O., 2003. COM_TEST: A Test Pattern Generation System, *Proceedings of International Conference on Electrical and Electronics Engineering*, pp. 112-116.
- **Aksoy, L.** and Sengor, N.S., 2004. Mikroşerit Hat Endüktans Büyüklüğünün Bilgi Tabanlı Yapay Sinir Ağları ile Modellenmesi, *Proceedings of National Conference on Electrical and Electronics Engineering*, pp. 17-21.
- **Aksoy, L.** and Tekin, O.A., 2005. Hybridization of Local Search Methods with a Simple Genetic Algorithm for the Satisfiability Problem, *Proceedings of International Symposium on Innovations in Intelligent Systems and Applications*, pp. 235-238.
- **Aksoy, L.** and Gunes, E.O., 2005. An Evolutionary Local Search Algorithm for the Satisfiability Problem, *Proceedings of Turkish Symposium on Artificial Intelligence and Neural Networks*, pp. 222-230.
- **Aksoy, L.** and Gunes, E.O., 2006. An Evolutionary Local Search Algorithm for the Satisfiability Problem, *Lecture Notes in Computer Science*, **3949**, 185-193.
- **Aksoy, L.**, Costa, E., Flores, P. and Monteiro J., 2006. Optimization of Area under a Delay Constraint in Digital Filter Synthesis Using SAT-Based Integer Linear Programming, *Proceedings of Design Automation Conference*, pp. 669-674.
- **Aksoy, L.**, Costa, E., Flores, P. and Monteiro J., 2006. ASSUMEs: Heuristic Algorithms for Optimization of Area and Delay in Digital Filter Synthesis, *Proceedings of International Conference on Electronics, Circuits and Systems*, pp. 748-751.

- **Aksoy, L.**, Costa, E., Flores, P. and Monteiro J., 2007. Optimization of Area in Digital FIR Filters Using Gate-Level Metrics, *Proceedings of Design Automation Conference*, pp. 420-423.

- **Aksoy, L.**, Costa, E., Flores, P. and Monteiro J., 2007. Minimum Number of Operations under a General Number Representation for Digital Filter Synthesis, *Proceedings of European Conference on Circuit Theory and Design*, pp. 252-255.

- **Aksoy, L.**, Gunes, E.O., Costa, E., Flores, P. and Monteiro J., 2007. Effect of Number Representation on the Achievable Minimum Number of Operations in Multiple Constant Multiplications, *Proceedings of IEEE Workshop on Signal Processing Systems*, pp. 424-429.

- **Aksoy, L.**, Costa, E., Flores, P. and Monteiro J., 2008. Exact and Approximate Algorithms for the Optimization of Area and Delay in Multiple Constant Multiplications, *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, **27(6)**, 1013-1026.

- **Aksoy, L.** and Gunes, E.O., 2008. An Approximate Algorithm for the Multiple Constant Multiplications Problem, *Proceedings of Symposium on Integrated Circuits and Systems Design*, pp. 58-63.

- **Aksoy, L.** and Gunes, E.O., 2008. Area Optimization Algorithms in High-Speed Digital FIR Filter Synthesis, *Proceedings of Symposium on Integrated Circuits and Systems Design*, pp. 64-69.

- **Aksoy, L.**, Gunes, E.O. and Flores P., 2008. An Exact Breadth-First Search Algorithm for the Multiple Constant Multiplications Problem, *Proceedings of IEEE Norchip Conference*, pp. 41-46.