Linköping Studies in Science and Technology
Dissertations, No. 1201

Low Power and Low Complexity Shift-and-Add Based Computations

Kenny Johansson

Linköping University
INSTITUTE OF TECHNOLOGY

Department of Electrical Engineering
Linköping University, SE-581 83 Linköping, Sweden

Linköping 2008

Low Power and Low Complexity Shift-and-Add Based Computations

© 2008 Kenny Johansson

Division of Electronics Systems
Department of Electrical Engineering
Linköping University
SE-581 83 Linköping
Sweden

http://www.es.isy.liu.se/

ISBN 978-91-7393-836-5 ISSN 0345-7524

Printed by LiU-Tryck, Linköping 2008

ABSTRACT

The main issue in this thesis is to minimize the energy consumption per operation for the arithmetic parts of DSP circuits, such as digital filters. More specific, the focus is on single- and multiple-constant multiplications, which are realized using shift-and-add based computations. The possibilities to reduce the complexity, i.e., the chip area, and the energy consumption are investigated. Both serial and parallel arithmetic are considered. The main difference, which is of interest here, is that shift operations in serial arithmetic require flip-flops, while shifts can be hardwired in parallel arithmetic.

The possible ways to connect a given number of adders is limited. Thus, for single-constant multiplication, the number of shift-and-add structures is finite. We show that it is possible to save both adders and shifts compared to traditional multipliers. Two algorithms for multiple-constant multiplication using serial arithmetic are proposed. For both algorithms, the total complexity is decreased compared to one of the best-known algorithms designed for parallel arithmetic. Furthermore, the impact of the digit-size, i.e., the number of bits to be processed in parallel, is studied for FIR filters implemented using serial arithmetic. Case studies indicate that the minimum energy consumption per sample is often obtained for a digit-size of around four bits.

The energy consumption is proportional to the switching activity, i.e., the average number of transitions between the two logic levels per clock cycle. To achieve low power designs, it is necessary to develop accurate high-level models that can be used to estimate the switching activity. A method for computing the switching activity in bit-serial constant multipliers is proposed.

For parallel arithmetic, a detailed complexity model for constant multiplication is introduced. The model counts the required number of full and half adder cells. It is shown that the complexity can be significantly reduced by considering the interconnection between the adders. A main factor for energy consumption in constant multipliers is the adder depth, i.e., the number of cascaded adders. The reason for this is that the switch-

ing activity will increase when glitches are propagated to subsequent adders. We propose an algorithm, where all multiplier coefficients are guaranteed to be realized at the theoretically lowest depth possible. Implementation examples show that the energy consumption is significantly reduced using this algorithm compared to solutions with fewer word level adders.

For most applications, the input data are correlated since real world signals are processed. A data dependent switching activity model is derived for ripple-carry adders. Furthermore, a switching activity model for the single adder multiplier is proposed. This is a good starting point for accurate modeling of shift-and-add based computations using more adders.

Finally, a method to rewrite an arbitrary function as a sum of weighted bit-products is presented. It is shown that for many elementary functions, a majority of the bit-products can be neglected while still maintaining reasonable high accuracy, since the weights are significantly smaller than the allowed error. The function approximation algorithms can be implemented using a low complexity architecture, which can easily be pipelined to an arbitrary degree for increased throughput.

ACKNOWLEDGEMENTS

I thank my supervisors, Dr. Oscar Gustafsson, who always takes an active interest in discussing new research ideas, and Professor Lars Wanhammar, for giving me the opportunity to do my Ph.D. studies at Electronics Systems. Furthermore, they both did a great job proofreading the thesis.

I also thank former and current colleagues at Electronics Systems. Dr. Henrik Ohlsson, for considerable support during my first working years, and for the many interesting, not only work related, discussions. Dr. Andrew Dempster for introducing me to the fundamentals in the field of multiple-constant multiplication.

I want to thank all the students that I have taught in various courses within the area of digital circuits. It has been enjoyable teaching you, and I hope that you have learned as much from me as I have learned from you.

All the friends during the years of undergraduate studies, in particular Magnus Karlsson, Joseph Jacobsson, and Ingvar Carlson, thanks to you this was the best time of my life.

Teachers through the years, especially Jan Alvarsson and Arne Karlsson in the upper secondary school. Also, the classmates Martin Källström and Peter Eriksson for all the conversations, about everything but school related subjects, both during and in between lessons.

Above all, I thank my parents, Mona and Nils-Gunnar Johansson, for all the support during my many years of studies. My sisters, Linda Johansson and Tanja Henze, who gave me an early start in mathematics. I remember getting extra homework when playing school because my 3:s looked angry. Considering this method of learning by discipline, it is not surprising that they became a teacher and a police officer.

Finally, I hope that selecting the colors of the club emblem for the front cover will help bringing home many gold medals to Färjestads BK!

This work was financially supported by the Swedish Research Council (Vetenskapsrådet).

The Coffee Room, August 24, 2008
Kenny Johansson

TABLE OF CONTENTS

1 Introduction 1

1.1 Digital Filters 2

1.1.1 IIR Filters 2
1.1.2 FIR Filters 2

1.2 Number Representations 4

1.2.1 Negative Numbers 4
1.2.2 Signed-Digit Numbers 5

1.3 Logarithmic Number Systems 6

1.3.1 Conversions 6
1.3.2 Addition 8

1.4 Constant Multiplication 10

1.4.1 Single-Constant Multiplication 11
1.4.2 Multiple-Constant Multiplication 13
1.4.3 Graph Representation 15
1.4.4 Terms used for Graph Based MCM Algorithms 16

1.5 Computer Arithmetic 18

1.5.1 Parallel Arithmetic 18
1.5.2 Serial Arithmetic 19
1.5.3 Carry-Save Arithmetic 22

1.6 Power and Energy Consumption 23

1.7 Outline and Main Contributions 25

2 Complexity of Serial Constant Multipliers 31

2.1 Graph Multipliers 32

2.1.1 Multiplier Types 32
2.1.2 Graph Elimination 34

2.2 Complexity Comparison – Single Multiplier 35
2.2.1 Comparison of Flip-Flop Cost 36
2.2.2 Comparison of Building Block Cost 39
2.3 Complexity Comparison – RSAG-*n* 42
2.3.1 The Reduced Shift and Adder Graph Algorithm 42
2.3.2 Comparison by Varying the Wordlength 46
2.3.3 Comparison by Varying the Setsize 47
2.4 Digit-Size Trade-Offs 49
2.4.1 Implementation Aspects 51
2.4.2 Specification of the Example Filter 52
2.4.3 Chip Area 53
2.4.4 Sample Rate 54
2.4.5 Energy Consumption 56
2.5 Complexity Comparison – RASG-*n* 58
2.5.1 The Reduced Adder and Shift Graph Algorithm 59
2.5.2 Comparison by Varying the Wordlength 59
2.5.3 Comparison by Varying the Setsize 61
2.5.4 Adder Depth 63
2.6 Implementation Examples 65
2.6.1 Example 1 66
2.6.2 Example 2 73
2.7 Conclusions 77
3 Switching Activity in Bit-Serial Multipliers 79
3.1 Multiplier Stage 80
3.1.1 Preliminaries 80
3.1.2 Sum Output Switching Activity 81
3.1.3 Switching Activity Using STGs 83
3.1.4 Carry Output Switching Activity 87
3.1.5 Input-Output Correlation Probability 88
3.1.6 Glitching Activity 91
3.1.7 Example 91
3.2 Graph Multipliers 93
3.2.1 Correlation Probability Look-Up Tables 93
3.2.2 The Applicability of the Equations 93
3.2.3 Example 95

3.3 Serial/Parallel Multipliers 96
3.3.1 Simplification of the Switching Activity Equation 97
3.3.2 Example 99
3.4 Conclusions 100

4 Complexity of Parallel Constant Multipliers 101

4.1 Bit-Level Optimization 102
4.1.1 Scaling 102
4.1.2 Complexity Model 103
4.1.3 Removing Half Adders 108
4.1.4 Single-Constant Multiplication Example 109
4.2 Low Complexity Algorithm 112
4.2.1 Multiple-Constant Multiplication Example 113
4.2.2 Results for Random Coefficient Sets 113
4.3 Interconnection Algorithms 115
4.3.1 Algorithm Formulations 117
4.3.2 Implementation Examples 119
4.4 Minimum Adder Depth Algorithm 126
4.4.1 Fundamental Pairs 126
4.4.2 MCM Defined as a Covering Problem 130
4.4.3 Optimal Approach 130
4.4.4 Heuristic Approaches 134
4.4.5 Optimal vs. Heuristic 143
4.4.6 Results 148
4.4.7 Implementation Examples 154
4.5 Conclusions 160

5 Energy Estimation for Ripple-Carry Adders 163

5.1 Background 164
5.1.1 Exact Method for Transitions in RCA 164
5.2 Energy Model 165
5.2.1 Timing Issues 169
5.3 Switching Activity 170
5.3.1 Switching due to Change of Input 171
5.3.2 Switching due to Carry Propagation 173
5.3.3 Total Switching Activity 174

5.3.4 Uncorrelated Input Data 177
5.3.5 Summary 179

5.4 Experimental Results 180

5.4.1 Uncorrelated Data 181
5.4.2 Correlated Data 182

5.5 Adopting the Dual Bit Type Method 183

5.5.1 Statistical Definition of Signals 186
5.5.2 The DBT Method 187
5.5.3 DBT Model for Switching Activity in RCA 189
5.5.4 Example.. 191

5.6 Switching Activity in Constant Multipliers 194

5.6.1 Addition with High Correlation 195
5.6.2 Results.. 199
5.6.3 Discussion on Switching Activity in MCM 202
5.6.4 Time Instant Model 204

5.7 Conclusions 205

6 Function Approximation by a Sum of Bit-Products 207

6.1 Background 208

6.1.1 PPA Methods.. 209

6.2 Function Approximation Approach 211

6.2.1 General Formulation... 211
6.2.2 Optimization.. 215
6.2.3 Results for Some Elementary Functions 217

6.3 Architecture 222

6.3.1 Implementation for a Sum of Bit-Products 222
6.3.2 Conditional Blocks 225
6.3.3 Results Using Conditional Blocks 226

6.4 Functions for LNS 232

6.4.1 Sign Transformation 233
6.4.2 Results for the LNS Functions.................................... 237
6.4.3 Comparison with ROM 241

6.5 Sine and Cosine Functions 244

6.5.1 Angle Rotation Based Approach 244
6.5.2 Octant Mapping 246
6.5.3 Comparison with CORDIC.. 248

6.6 Conclusions 249

7 Conclusions 251

7.1 Summary 251

7.2 Future Work 254

References 255

1

INTRODUCTION

There are many hand-held products that include digital signal processing (DSP), for example, cellular phones and hearing aids. For this type of portable equipment, a long battery life time and a low battery weight is desirable. To achieve this, the circuits must have low power consumption.

The main issue in this thesis is to minimize the energy consumed per operation for the arithmetic parts of DSP circuits, such as digital filters. More specific, the focus will be on single- and multiple-constant multiplication, using either serial or parallel arithmetic. Different design algorithms will be compared, not just to determine which one that is the best in terms of complexity, but also to build up an understanding of the connection between algorithm properties and energy consumption. This knowledge is useful when models are derived to be able to estimate the energy consumption. Finally, to close the circle, the energy models can be used to design improved algorithms. However, although most parts are covered in some way, this circle is not completely closed within the content of this thesis.

In this chapter, an elementary background on the design of digital filters using constant multiplication will be presented. The information given here will be assumed familiar in the following chapters. In addition, terms that are used in the rest of the thesis will be introduced.

1.1 Digital Filters

Frequency selective digital filters are used in many DSP systems [149], [150]. The filters studied here are assumed to be causal, linear, and time-invariant systems.

The input-output relation for an Nth-order digital filter is described by the difference equation

$$y(n) = \sum_{k=1}^{N} b_k y(n-k) + \sum_{k=0}^{N} a_k x(n-k) \tag{1.1}$$

where a_k and b_k are constant coefficients while $x(n)$ and $y(n)$ are the input and output sequences. If the input sequence, $x(n)$, is an impulse, the impulse response, $h(n)$, is obtained as output sequence.

1.1.1 IIR Filters

If the impulse response has infinite duration, i.e., theoretically never reaches zero, it is an infinite-length impulse response (IIR) filter. This type of filters can only be realized by recursive algorithms, which means that at least one of the coefficients b_k in (1.1) must be nonzero.

The transfer function, $H(z)$, is obtained by applying the z-transform to (1.1), which gives

$$H(z) = \frac{Y(z)}{X(z)} = \frac{\sum_{k=0}^{N} a_k z^{-k}}{1 - \sum_{k=1}^{N} b_k z^{-k}} \tag{1.2}$$

1.1.2 FIR Filters

If the impulse response becomes zero after a finite number of samples, it is a finite-length impulse response (FIR) filter, which is a common component in many DSP systems. For a given specification, the filter order, N, is higher for an FIR filter than for an IIR filter. However, FIR filters can be guaranteed to be stable and to have a linear phase response, which corresponds to a constant group delay [150].

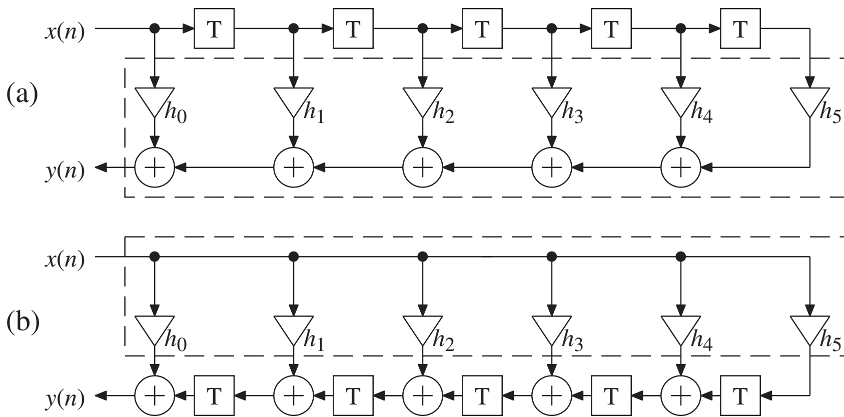


Figure 1.1 Different realizations of a fifth-order (six-tap) FIR filter. (a) Direct form and (b) transposed direct form.

It is not recommended to use recursive algorithms to realize FIR filters, because of stability problems. Hence, here all coefficients b_k in (1.1) are assumed to be zero. If an impulse is applied at the input, each output sample will be equal to the corresponding coefficient a_k, i.e., the impulse response, $h(n)$, is an ordered sequence of the coefficients $a_0, a_1, a_2, ..., a_N$. The transfer function of an Nth-order FIR filter can then be written as

$$H(z) = \sum_{k=0}^{N} a_k z^{-k} \tag{1.3}$$

A realization of (1.3) for $N = 5$ is shown in Fig. 1.1 (a), where $h_k = a_k$. This filter structure is referred to as a direct form FIR filter. If the signal flow graph is transposed the filter structure in Fig. 1.1 (b) is obtained, referred to as transposed direct form [150]. These are the two most common structures for realizing FIR filters. The dashed boxes in Figs. 1.1 (a) and (b) mark a sum-of-products (SOP) and a multiplier block (MB), respectively. In both cases, the part that is not included in the dashed box is referred to as the delay section, and the adders in Fig. 1.1 (b) are called structural adders.

In most practical cases of frequency selective FIR filters, linear-phase filters are used. This means that the phase response, $\Phi(\omega T)$, is proportional to ωT as [150]

$$\Phi(\omega T) \propto -\frac{N\omega T}{2} \tag{1.4}$$

Furthermore, linear-phase FIR filters have a symmetric or antisymmetric impulse response, i.e.,

$$h(n) = \begin{cases} h(N-n) & \text{symmetric} \\ -h(N-n) & \text{antisymmetric} \end{cases} \qquad n = 0, 1, \ldots, N \tag{1.5}$$

This implies that for linear-phase FIR filters, the number of specific multiplier coefficients is at most $N/2 + 1$ and $(N + 1)/2$ for even and odd filter orders, respectively.

1.2 Number Representations

In digital circuits, numbers are represented as a string of bits, using the logic symbols 0 and 1. Normally, the processed data are assumed to take values in the range [–1, 1]. However, as the binary point can be placed arbitrarily by shifting, only integer numbers will be considered here.

The values are represented using n digits x_i with the corresponding weight 2^i. Hence, for positive numbers an ordered sequence $x_{n-1}x_{n-2} \ldots x_1x_0$ where $x_i \in \{0, 1\}$ correspond to the integer value, X, as defined by

$$X = \sum_{i=0}^{n-1} x_i 2^i \tag{1.6}$$

1.2.1 Negative Numbers

There are different ways to represent negative values for fixed-point numbers. One possibility is the signed-magnitude representation, where the sign and the magnitude are represented separately. When this representation is used, simple operations, like addition, become complicated because a sequence of decisions has to be made [75].

Another possible representation is one's-complement, which is the diminished-radix complement in the binary case. Here, the complement is simply obtained by inverting all bits. However, a correction step where a one is added to the least significant bit position is required if a carry-out is obtained in an addition.

For both signed-magnitude and one's-complement, there are two representations of zero, which makes a test for zero operation more complicated.

The most commonly used representation in DSP systems is the two's-complement representation, which is the radix complement in the binary case. Here, there is only one representation of zero and no correction is necessary when addition is performed. For two's-complement representation, an ordered sequence $x_{n-1}x_{n-2} \ldots x_1x_0$ where $x_i \in \{0, 1\}$ correspond to the integer value

$$X = -x_{n-1}2^{n-1} + \sum_{i=0}^{n-2} x_i 2^i \tag{1.7}$$

The range of X is $[-2^{n-1}, 2^{n-1} - 1]$.

1.2.2 Signed-Digit Numbers

In signed-digit (SD) number systems, the digits are allowed to take negative values, i.e., $x_i \in \{\bar{1}, 0, 1\}$ where a bar is used to represent a negative digit. The integer value, X, of an SD coded number can be computed according to (1.6) and the range of X is $[-2^n + 1, 2^n - 1]$. This is a redundant number system, for example, $1\bar{1}$ and 01 both correspond to the integer value one.

An SD representation that has a minimum number of nonzero digits is referred to as a minimum signed-digit (MSD) representation. The most commonly used MSD representation is the canonic signed-digit (CSD) representation [149], where no two consecutive digits are nonzero. Here, each number has a unique representation, i.e., the CSD representation is nonredundant. Consider, for example, the integer value eleven, which has the binary representation 1011 and the unique CSD representation $10\bar{1}0\bar{1}$. Both these representations are also MSD representations, and so is $110\bar{1}$.

The SD numbers are used to avoid carry propagation in additions and to reduce the number of partial products in multiplication algorithms. An algorithm to obtain all SD representations of a given integer value was presented in [30].

1.3 Logarithmic Number Systems

Compared to conventional computer arithmetic, logarithmic number systems (LNS) have some advantages, which are desirable in applications such as adaptive filtering [121]. Logarithmic arithmetic has also been used in various processors [16],[17],[109]. Using LNS, multiplication and division are simplified to addition and subtraction, respectively. Furthermore, powers and roots are in general reduced to multiplication and division, while, in the special case, squaring and square roots are simply obtained by shift operations [75]. On the other hand, addition/subtraction and the conversions to and from LNS are complicated [32],[147]. These operations require that certain functions are computed, which, for example, can be done by using look-up tables.

In the first part of this section, conversions between the conventional fixed-point and the logarithmic number systems are considered. In the second part, addition (subtraction) within the logarithmic domain is discussed. Here, base two is assumed for the logarithms, but note that for a given application it may be advantageous to select another base [3].

1.3.1 Conversions

Assume a fixed-point number, A, in the linear domain. After conversion to the logarithmic domain, A is represented by a logarithm E_A and a sign bit S_A, according to

$$E_A = \log_2|A| \quad \text{and} \quad S_A = \begin{cases} 0 & A > 0 \\ 1 & A < 0 \end{cases} \tag{1.8}$$

Note that $A \neq 0$ is assumed here. Zero can be approximated using the largest negative value of E_A. However, in many applications an extra bit is used as a zero flag.

The conversion back to the linear domain is performed by an antilogarithm operation computed as

$$A = (-1)^{S_A} 2^{E_A} \tag{1.9}$$

The logarithm and antilogarithm functions, which are used in the conversions, are often implemented using look-up tables. These functions are

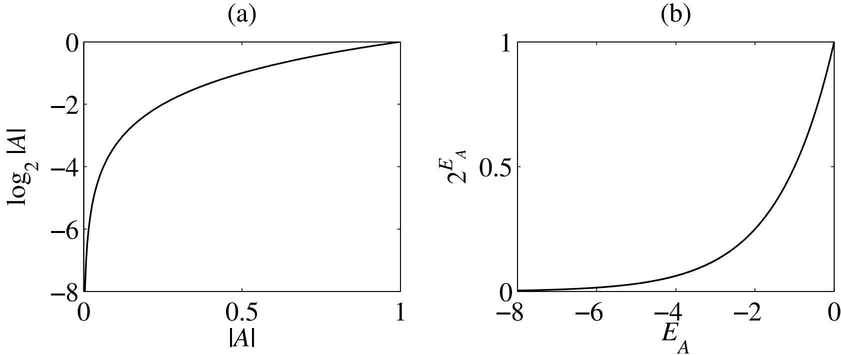


Figure 1.2 Conversions to and from logarithmic number systems. (a) The logarithm and (b) the antilogarithm function.

illustrated in Fig. 1.2 for the range $|A| \le 1$ and $-8 \le E_A < 0$, respectively. Note that the same antilogarithm table can be used for both negative and positive values of E_A by shifting the result.

Since the numeric range of the two domains is different, certain rules can be formulated to preserve the range and resolution. Using two's-complement representation, a fixed-point number, A, is composed of k integer bits and l fractional bits. In the logarithmic domain, E_A includes a K bit integer part and an L bit fractional part, using two's-complement representation.

The maximum value of $|E_A|$ can be limited by either a large value of $|A|$, which gives a large positive value of E_A, or by a small value of $|A|$, which gives a large negative value of E_A. Hence, the number of integer bits, K, is obtained as

$$K = \lceil \log_2(\max(k-1, l)) \rceil + 1 \tag{1.10}$$

Numbers are not equally spaced in the logarithmic domain, i.e., the accuracy depends on E_A. To obtain at least the same accuracy as in the linear domain, the following relation should hold for all values of E_A

$$2^{E_A + 2^{-L}} - 2^{E_A} \le 2^{-l} \quad \Rightarrow \quad L = \lceil -\log_2(\log_2(2^{-l} + 2^{E_A}) - E_A) \rceil \tag{1.11}$$

The required accuracy is higher for large values of $|A|$ as the slope of $|A| = 2^{E_A}$ then is larger, i.e., $E_A = \log_2|2^{k-1} - 2^{-l}| \approx k - 1$ should be used in (1.11) to find the value of L that is required in the worst case.

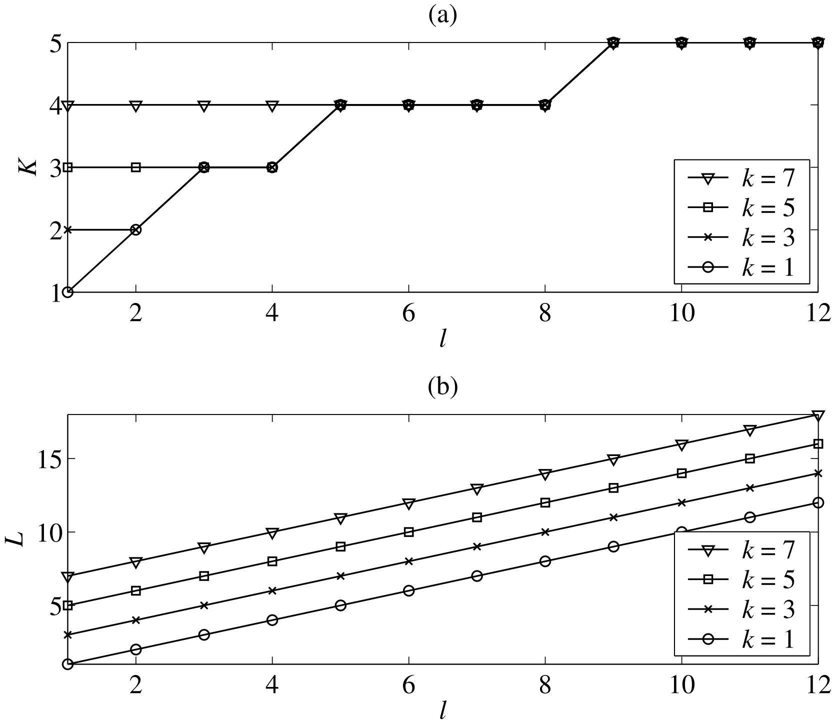


Figure 1.3 Required number of (a) integer and (b) fractional bits using logarithmic number systems.

The relations between the number of required bits in the different domains according to (1.10) and (1.11) are illustrated in Figs. 1.3 (a) and (b), respectively. These rules are of course not suitable for all applications, since it depends on the required accuracy and number range.

1.3.2 Addition

Consider the addition (subtraction) $C = A \pm B$. If $|A| > |B|$ is assumed, the corresponding operation in the logarithmic domain is obtained as

$$\begin{aligned} E_C &= \log_2|A \pm B| = \log_2\left|A\left(1 \pm \frac{B}{A}\right)\right| = \\ &= E_A + \log_2\left|1 + (-1)^{S_\Phi} 2^{E_B - E_A}\right| = E_A + \Phi(E_B - E_A) \end{aligned} \quad (1.12)$$

where Φ is a function of the negative difference $E_B - E_A$. The variable S_Φ is introduced to compensate for the signs of A and B after conversion to

the logarithmic domain, since the absolute values of A and B then are used. Furthermore, S_Φ also include the performed operation. Because $|A| > |B|$, the sign of C will be the same as for A, i.e., $S_C = S_A$.

If instead $|B| > |A|$ the logarithm, E_C, is computed in a similar manner according to

$$E_C = E_B + \Phi(E_A - E_B) \tag{1.13}$$

Since $A - B = -(B - A)$, the sign, S_C, must be inverted if the performed operation is a subtraction, which can be implemented as a logic XOR operation, denoted by $\oplus$, i.e.,

$$S_C = S_B \oplus op \quad \text{where} \quad op = \begin{cases} 0 & \text{addition} \\ 1 & \text{subtraction} \end{cases} \tag{1.14}$$

The sign of the Φ function, S_Φ, must be switched if A and B have different signs, i.e., it should then be opposite to the performed operation. Hence, S_Φ is obtained as

$$S_\Phi = S_A \oplus S_B \oplus op \tag{1.15}$$

and the Φ function is then defined by

$$\Phi(x) = \begin{cases} \Phi^+(x) = \log_2|1 + 2^x| & S_\Phi = 0 \\ \Phi^-(x) = \log_2|1 - 2^x| & S_\Phi = 1 \end{cases} \tag{1.16}$$

As stated before, $x < 0$. In Fig. 1.4, the characteristics of the two $\Phi(x)$ functions are shown. Note that the maximum value of $\Phi^+(x)$ is 1, i.e., no integer part is required for the corresponding look-up table.

The complete schematic for an addition/subtraction, including the conversions to and from the logarithmic domain, is illustrated in Fig. 1.5. This design can be used as a testbench to verify the functionality of the look-up tables. Implementation of these tables will be considered in Section 6.4. Note that the architecture in Fig. 1.5 can be improved from a power consumption point of view by turning off parts that are not used in a specific computation.

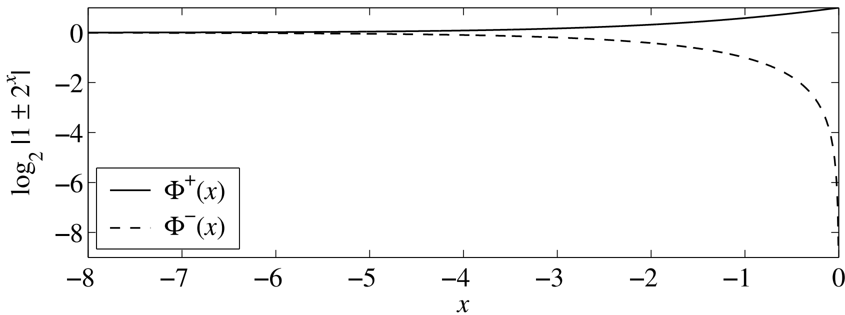


Figure 1.4 The $\Phi(x)$ functions used to perform addition and subtraction in logarithmic number systems.

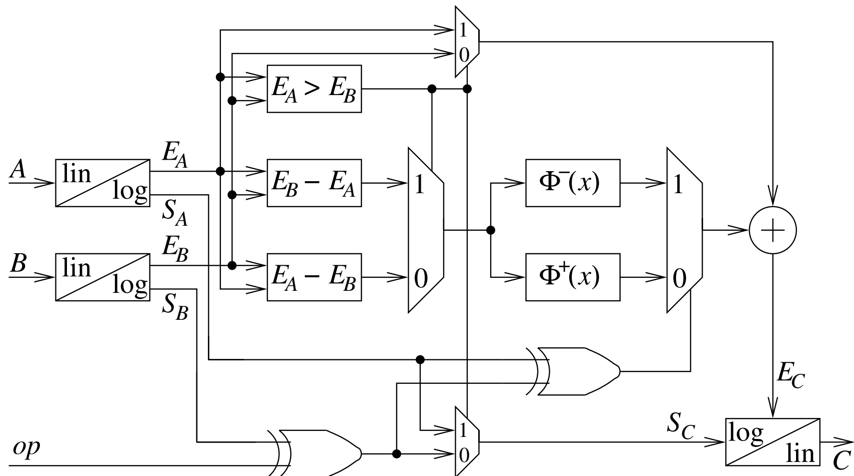


Figure 1.5 Architecture for computing addition and subtraction using logarithmic number systems.

1.4 Constant Multiplication

Multiplication with a constant is commonly used in DSP circuits, such as digital filters [142]. It is possible to use shift-and-add operations [90] to efficiently implement this type of multiplication. Since the general multipliers are replaced by shifts, adders, and subtractors this is sometimes referred to as multiplierless implementation. As the complexity is similar for adders and subtractors we will refer to both as adders, and adder cost will be used to denote the total number of adders/subtractors.

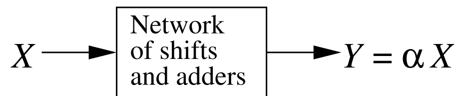


Figure 1.6 The principle of single-constant multiplication.

1.4.1 Single-Constant Multiplication

The general design of a multiplier is shown in Fig. 1.6. The input data, X, is multiplied with a specific coefficient, α, and the output, Y, is the result.

The method based on the CSD representation, which was discussed in Section 1.2.2, is commonly used to implement single-constant multipliers [50]. However, in many cases multipliers can be implemented more efficiently using other structures that require fewer operations [57]. Most work has focused on minimizing the adder cost [24],[37],[42], while the shifts are assumed to be free.

Consider, for example, the coefficient 45, which has the CSD representation $10\bar{1}0\bar{1}01$. The corresponding realization is shown in Fig. 1.7 (a), where 45 is computed as $64 - 16 - 4 + 1$. Note that a left shift correspond to a multiplication by two. If the realization in Fig. 1.7 (b) is used instead, the adder cost is reduced from 3 to 2. Here, the constant 45 is obtained as $5 \cdot 9 = (1 + 4)(1 + 8)$. Furthermore, the number of shift operations is also reduced from 3 to 2, while the number of actual shifts is reduced from 6 to 5. For relatively short coefficient wordlengths, it is possible to find the most beneficial realization of each constant by an exhaustive search. Results on this will be given in Chapter 2.

Subexpression Sharing

Subexpression sharing was introduced in [48] as a method to utilize redundancy between FIR filter coefficients to reduce the number of required adders. However, subexpression sharing is also a commonly applied method in algorithms for design of single-constant multipliers.

In the CSD representation of 45, $10\bar{1}0\bar{1}01$, the patterns $10\bar{1}$ and $\bar{1}01$, which correspond to ± 3, are both included. Hence, the coefficient can be obtained as $(4 - 1)(16 - 1)$ where the first part gives the value of the subexpression and the second part corresponds to the weight and sign difference. This structure is shown in Fig. 1.7 (c). Another set of subexpressions that can be found in the CSD representation of 45 is $1000\bar{1}$ and $\bar{1}0001$, which corresponds to $(16 - 1)(4 - 1)$, i.e., the two stages in Fig. 1.7 (c) are performed in reversed order.

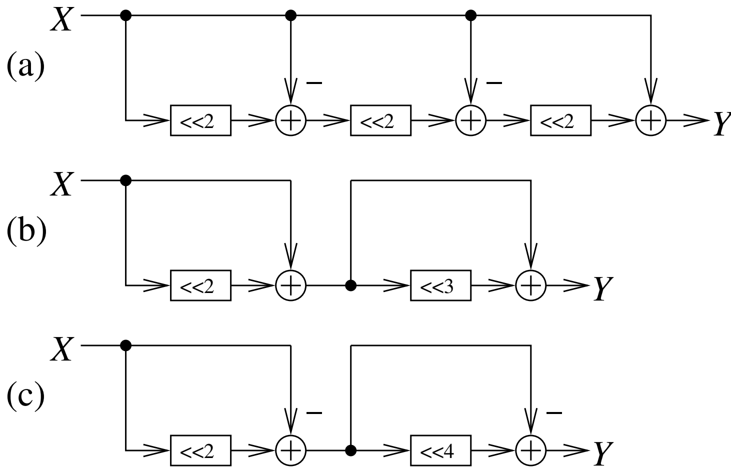


Figure 1.7 Different realizations of multiplication with the coefficient 45. The symbol <<M is used to indicate M left shifts.

It is clear that the results are representation dependent, and in [114] it was shown that better results may be found if other MSD representations than CSD are used.

An algorithm introduced in [30], generates all SD representations of an integer using a specified number k of extra nonzero digits compared with the minimum, i.e., above that used by CSD. So, if $k = 0$, then all MSD representations are produced. For a heuristic approach, it has been shown that single-constant multipliers can be designed using fewer adders when $k > 0$ [29],[42]. In this approach, a subexpression sharing algorithm, for example the one in [48], is applied to all SD representations using at most k extra nonzero digits. The multiplier design with the lowest adder cost is then selected.

The CSD and the best SD representations for three different coefficients are given in Table 1.1, where the resulting number of required adders is also included. The smallest integer for which another MSD representation gives a better result than CSD is 105. The coefficient 363 is the smallest integer where a representation with one extra nonzero digit, i.e., $k = 1$, gives a better result than any MSD representation. The smallest integer where a representation with two extra nonzero digits, i.e., $k = 2$, gives a better result than any representation with up to one extra nonzero digit is 1395. These three coefficients can be realized as shown in Fig. 1.8. Note that the shifts sometimes can be placed in more than one way. For example, the first two adders in Fig. 1.18 (b) are used to com-

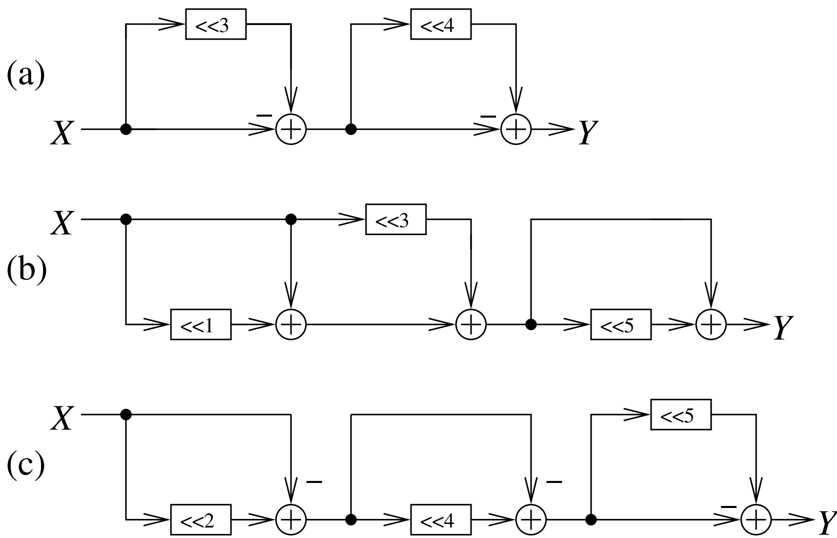


Figure 1.8 Realization of single-constant multiplication using subexpression sharing for the coefficients (a) 105, (b) 363, and (c) 1395.

Coefficient	CSD		Best SD		
	Representation	Adders	Extra digits	Representation	Adders
105	$10\bar{1}01001$	3	0	$100\bar{1}\bar{1}001$	2
363	$10\bar{1}00\bar{1}0\bar{1}0\bar{1}$	4	1	101101011	3
1395	$10\bar{1}0\bar{1}00\bar{1}010\bar{1}$	4	2	$10\bar{1}0\bar{1}\bar{1}11010\bar{1}$	3

Table 1.1 Results for applying subexpression sharing to the CSD representation compared with SD representations possibly using extra digits.

pute 11, which here is done according to the expression $(1 + 2) + 8$. However, 11 can also be obtained as $(1 + 4)\cdot 2 + 1$.

1.4.2 Multiple-Constant Multiplication

In some applications, one signal is to be multiplied with several constant coefficients, as shown in Fig. 1.9. An example of this is the transposed direct form FIR filter where a multiplier block is used, as marked by the dashed box in Fig. 1.1 (b). A simple method to realize multiplier blocks is to implement each multiplier separately, for example, using the methods discussed in the previous section. However, multiplier blocks can be more

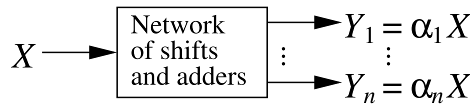


Figure 1.9 The principle of multiple-constant multiplication.

efficiently implemented by using structures that make use of redundant partial results between the coefficients in the shift-and-add network, and thereby reduce the required number of components.

The implementation of FIR filters using shift-and-add based multipliers has received considerable attention during the last decade, and is referred to as the multiple-constant multiplication (MCM) problem. The MCM algorithms can be divided into three groups based on the operation of the algorithm; subexpression sharing [28],[48],[95],[115],[118], difference methods [38],[40],[43],[98],[108], and graph based methods [7],[25],[26],[144]. Heuristics based on subexpression sharing, usually represent each coefficient using the CSD representation, and subexpressions are sought among the coefficients. In difference algorithms, the fact that the successive coefficient values vary slowly in frequency selective FIR filters is exploited by realizing the differences between the coefficients. The graph algorithms are not representation dependent since the nodes simply have integer values. New values are obtained by adding/subtracting existing nodes. Graph based methods will be used in Chapters 2 and 4 to solve MCM problems.

The only considered objective function for most of the MCM algorithms is to minimize the adder cost. However, besides complexity, the adder depth, i.e., the number of cascaded adders, has also been considered [26],[95]. This is partly motivated by the effect on the energy consumption, which in general is lower for a reduced adder depth [21],[22],[23].

Furthermore, the MCM problem has been extended by including the delay elements inherent in FIR filters in the redundancy utilization [48], and to matrix multiplications [27].

By transposing the shift-and-add network, corresponding to a multiplier block, a sum-of-products is obtained. This is illustrated by the dashed box in Fig. 1.1 (a), i.e., a multiplier block together with the structural adders correspond to a sum-of-products. Hence, MCM techniques can be applied to realize both direct form and transposed direct form FIR filters.

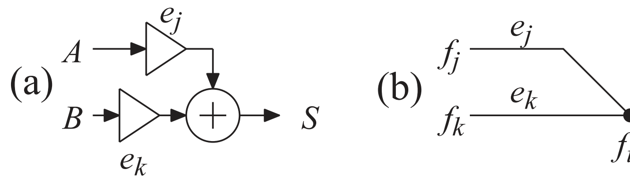


Figure 1.10 Multiplier segment. (a) Arithmetic implementation of a shift-and-add operation and (b) the corresponding graph representation.

1.4.3 Graph Representation

The graph representation of constant multipliers was introduced in [7], and used in, for example, [24], [25], [27], and [37]. As discussed in the two previous sections, single- and multiple-constant multiplications are composed of networks of shifts and adders. These networks can be represented using directed acyclic graphs with the following characteristics [24],[37].

- The input is the vertex that has in-degree zero, and vertices that have out-degree zero are outputs. However, for MCM, vertices with an out-degree larger than zero may also be outputs.
- Each vertex has out-degree larger than or equal to one, except for the output vertices, which may have out-degree zero.
- Each vertex that has an in-degree of two corresponds to an adder (subtractor). Hence, the adder cost is equal to the number of vertices with in-degree two.
- Each edge is assigned a value of $\pm 2^n$, which corresponds to $|n|$ shifts, i.e., a multiplication by a power-of-two, and the sign for any subsequent addition or subtraction.

Although these graphs are directed, this is usually not marked, i.e., it is understood without saying that the leftmost node is the input and that operations are performed from left to right.

The nodes are assigned values, which are referred to as fundamentals. A fundamental, f_i, is computed from two other fundamentals f_j and f_k as

$$f_i = e_j f_j + e_k f_k \tag{1.17}$$

where e_j and e_k are edge values, as illustrated in Fig. 1.10. The obtained signal value in this node will then be f_i times the input signal.

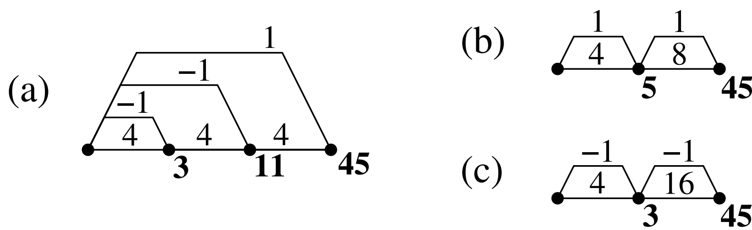


Figure 1.11 Graph representation of the realizations in Fig. 1.7.

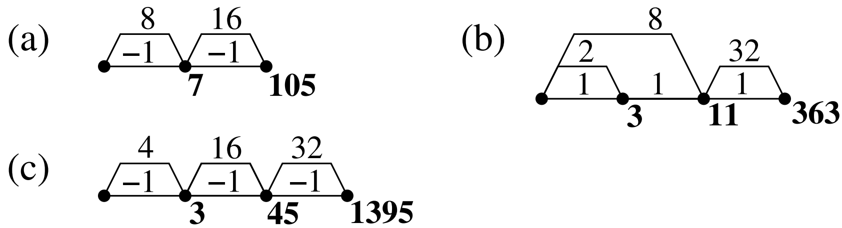


Figure 1.12 Graph representation of the realizations in Fig. 1.8.

Some examples of the graph representation are shown in Figs. 1.11 and 1.12. Note that these illustrations are simpler than Figs. 1.7 and 1.8 although they contain the same information.

In [37], a vertex reduced representation of graph multipliers was introduced. However, since the placement of shift operations sometimes is of importance, the original, fully specified, graph representation will be used throughout this thesis.

1.4.4 Terms used for Graph Based MCM Algorithms

Here, terms that will be used in descriptions of MCM algorithms are introduced.

As discussed earlier, the MCM problem is to determine a shift-and-add network that can realize multiplications of a single input with a set of coefficients, H. It is usually sufficient to only consider odd integer coefficients, since even and fractional coefficients can be obtained by an appropriate shift operation at the output of the multiplier. In most cases, the sign of the coefficient can also be compensated for in other parts of the implementation. Hence, the coefficient set, C, which is the input to the MCM algorithm as illustrated in Fig. 1.13, is often assumed to only contain unique positive odd integers.

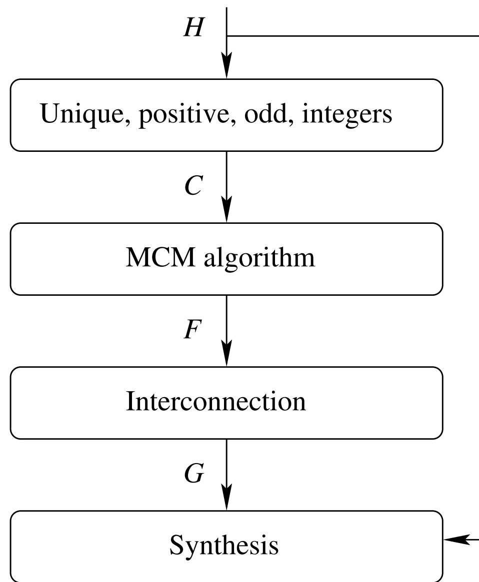


Figure 1.13 Design path for MCM blocks.

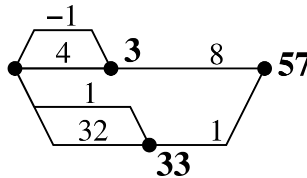


Figure 1.14 Graph representation for a shift-and-add MCM block.

If $|H| = 1$, i.e., there is only one coefficient in the set, the MCM problem is transformed into a single-constant problem. The design path in Fig. 1.13 may still be used, although there exist better strategies than to use general MCM algorithms for this special case, as discussed in Section 1.4.1. Also for the case when $|H| = 2$, there exist more specialized design techniques [31].

As an example, consider the coefficient set $C = \{33, 57\}$. The coefficient 33 can be obtained directly from the input as $1 + 2^5$. However, the coefficient 57 can not be realized and a new node must therefore be included. For example, the value 3 solves the problem by computing the constant 57 as $33 + 3 \cdot 2^3$, which is illustrated by the graph in Fig. 1.14. Hence, the set of extra fundamentals is $E = \{3\}$, and the total fundamental set $F = C \cup E = \{3, 33, 57\}$ is the output of the MCM algorithm. The fundamental set include all vertex values. However, the input vertex value, 1, is usually excluded, since it does not require any adder.

The interconnection graph, G, can be given in a table format. For the graph in Fig. 1.14, the corresponding table is

$$G = \begin{bmatrix} 3 & 1 & 1 & -1 & 4 & 1 \\ 33 & 1 & 1 & 1 & 32 & 1 \\ 57 & 3 & 33 & 8 & 1 & 2 \end{bmatrix} \quad (1.18)$$

In [23] such an interconnection table was referred to as the Dempster format. Here, the first column is the fundamental value, f_i, columns 2 and 3 are the values of the input vertices, f_j and f_k, and columns 4 and 5 are the values of the input edges, e_j and e_k. Finally, the sixth column includes the adder depth, D_i, defined as one more than the largest depth of the input nodes according to

$$D_i = \max(D_j, D_k) + 1 \quad (1.19)$$

The theoretical minimum depth, $D_{i,\,min}$, for a fundamental, f_i, is well defined and computed as [45]

$$D_{i,\,min} = \lceil \log_2 S(f_i) \rceil \quad (1.20)$$

where $S(f_i)$ denotes the number of nonzero digits in a minimum signed-digit representation of f_i.

1.5 Computer Arithmetic

The operations used in constant multiplication, i.e., additions and shifts, can be implemented using either parallel or serial arithmetic. Here, these two different approaches are discussed. Furthermore, the redundant carry-save arithmetic is also briefly introduced.

1.5.1 Parallel Arithmetic

Using parallel arithmetic, all bits of the input data word are processed concurrently. This makes it possible to achieve a high throughput, at a relatively low clock frequency. In Fig. 1.15, the straightforward architecture for adding two signals, A and B, is given. A drawback with the ripple-carry adder (RCA) is that the carry propagation path is linearly proportional to the input wordlength. Several other carry propagation adders

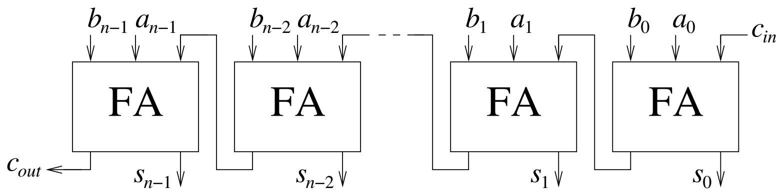


Figure 1.15 The structure for a ripple-carry adder (RCA).

(CPA) have been proposed, where different types of carry acceleration are used [8],[75],[149]. However, these techniques always come with a cost of increased area.

The carry propagation can be avoided by using a redundant number system, as mentioned in Section 1.2.2. For example, the carry-save representation, which will be presented in Section 1.5.3.

1.5.2 Serial Arithmetic

In digit-serial arithmetic, each data word is divided into digits that are processed one digit at a time [47],[130]. The number of bits in each digit is usually denoted the digit-size, *d*. This provides a trade-off between area, speed, and energy consumption [47],[134]. For the special case where *d* equals the data wordlength we have bit-parallel processing, and when *d* equals one we have bit-serial processing.

Digit-serial processing elements can be derived either by unfolding bit-serial processing elements [111] or by folding bit-parallel processing elements [112]. In Fig. 1.16, a digit-serial adder, subtractor, and shift are shown. These are the operations that are required to implement constant multiplication, as discussed in Section 1.4. Note that the carry feedback register should be set to one at the beginning of a subtract operation. Each left shift, corresponding to a multiplication by two, requires one flip-flop, as illustrated by Fig. 1.16 (d). Digit-serial arithmetic operations can be performed by either processing the most significant bit (MSB) first or the least significant bit (LSB) first. If the MSB is processed first, redundant arithmetic is required [149]. In this thesis, we will only consider the case with LSB first since this is less complicated.

It is clear that serial architectures with a small digit-size have the advantage of area efficient processing elements. Furthermore, the routing complexity for communication between operators, including the required number of input and output pads, is also reduced. Another benefit is that the carry propagation path is short, since it increases linearly with the

digit-size. On the other hand, only a few bits of the input data word are processed during each clock cycle for a small digit-size. Hence, a high clock frequency is required to obtain a reasonable throughput. How the energy consumption depend on the digit-size is difficult to predict, and this will be investigated in Sections 2.4.5 and 2.6.

One main difference compared to parallel arithmetic is that the shift operations can be hardwired, i.e., realized without using any flip-flops, in a bit-parallel architecture. However, the flip-flops included in serial shifts have the benefit of reducing the glitch propagation between subsequent adders/subtractors. To prevent glitches further, pipelining can be introduced, which also increases the throughput. Note that less hardware is needed for pipelining in serial arithmetic compared to the parallel case. For example, in bit-serial arithmetic a single flip-flop is required in the pipelining register between two operations. In addition, the available shifts can be rearranged to obtain an improved design, i.e., with a shorter critical path.

Complexity of Serial Constant Multiplication

As mentioned before, shifts are normally assumed to be free as they can be hardwired in a bit-parallel implementation. However, since a serial shift requires a flip-flop, as seen in Fig. 1.16 (c), it must be taken into account when the overall complexity is considered. The total number of shifts, n_{SH}, will be referred to as the flip-flop cost.

From G and F, as defined in Section 1.4.4, the flip-flop cost can be computed as

$$n_{SH} = \sum_{i=1}^{M+1} \log_2(e(i)) \tag{1.21}$$

where M is the length of F. The vector e contains the largest absolute edge value at the output of each vertex, including the input. Hence, $e(i)$ is computed for each node value in the set $F_1 = 1 \cup F$ as

$$e(i) = \max(1, |G_{4,5}(k)|) \qquad \begin{cases} \forall k \mid G_{2,3}(k) = F_1\{i\} \\ k \in \{1, 2, \ldots, 2M\} \\ i \in \{1, 2, \ldots, M+1\} \end{cases} \tag{1.22}$$

where $G_{i,j}$ is a vector containing the elements in columns i and j of G.

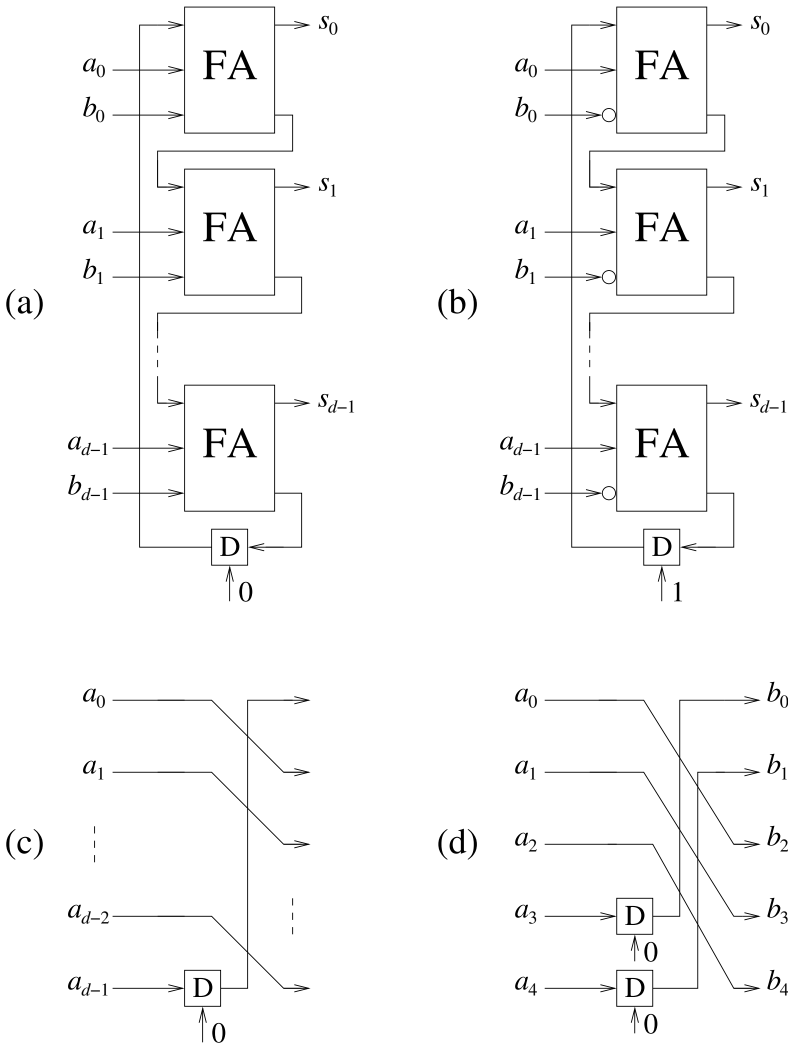


Figure 1.16 Digit-serial (a) adder, (b) subtractor, and (c) left shift. (d) Two cascaded left shifts with digit-size five, i.e., $B = 4A$ and $d = 5$.

Using (1.21) and (1.22), the flip-flop cost for the graph in Fig. 1.14 is obtained as

$$F_1 = \{1, 3, 33, 57\} \Rightarrow e = \begin{bmatrix}32 & 8 & 1 & 1\end{bmatrix} \Rightarrow n_{SH} = 5 + 3 = 8 \quad (1.23)$$

A digit-serial implementation with $d = 1$ for this MCM example design is shown in Fig. 1.17. Note that the first two shifts are shared, which illus-

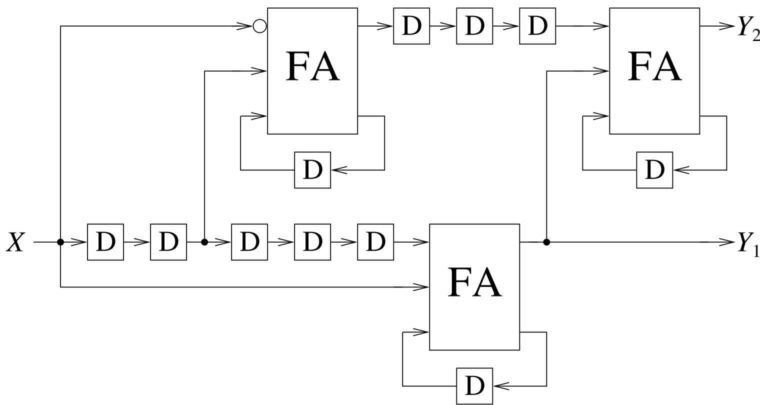


Figure 1.17 Bit-serial implementation of the multiplier coefficients $\alpha_1 = 33$ and $\alpha_2 = 57$, using one subtractor, two adders, and eight shifts.

trates why it is enough to only consider the maximum outgoing edge weight from each node, as defined by (1.22).

From this, it is obvious that an MCM algorithm that considers the number of shifts may yield digit-serial filter implementations with smaller overall complexity. As discussed in Section 1.4.4, it is normally sufficient to only realize odd coefficients. However, since the placement of shifts is important when serial arithmetic is used, it might be preferable to keep the even coefficients within the MCM algorithm. This will be further investigated in Chapter 2.

1.5.3 Carry-Save Arithmetic

As mentioned earlier, the redundant carry-save representation can be used to avoid carry propagation. The architecture of a carry-save adder (CSA) is shown in Fig. 1.18. Note that the critical path is a single full adder. As can be seen, the adder has three inputs and two outputs. Hence, a number is here represented by two data, one sum and one carry vector. Conversion to the nonredundant two's-complement representation can be performed by simply adding the two vectors. The adder used for this is often referred to as a vector merging adder (VMA), and can be realized by any CPA.

An area where carry-save arithmetic is commonly used is for adding several operands in an adder tree architecture. An example of this, using a Wallace tree [145], is illustrated in Fig. 1.19. As can be seen, each CSA reduces the number of operands by one. For this tree, the critical path is only three full adders.

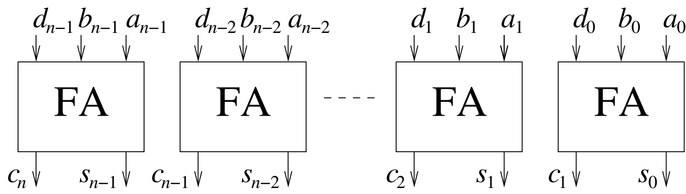


Figure 1.18 The structure for a carry-save adder (CSA).

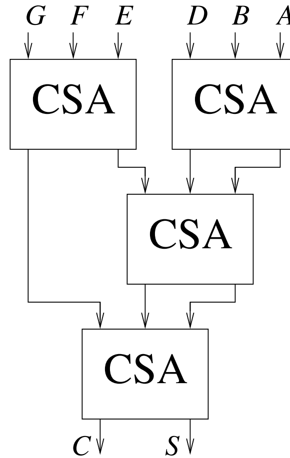


Figure 1.19 The structure for an adder tree with six inputs using carry-save arithmetic, where $S + C = A + B + D + E + F + G$.

The carry-save arithmetic has been used for high-speed DSP algorithms [107] and constant multiplication [36],[39]. However, in this thesis, it will only be used for the adder trees in Chapter 6.

1.6 Power and Energy Consumption

Low power design is always desirable in integrated circuits. To obtain this, it is necessary to find accurate and efficient methods that can be used to estimate the power consumption. In digital CMOS circuits, the dominating part of the total power consumption is the dynamic part. Although the relation between static and dynamic power becomes more equal because of scaling. Note that serial arithmetic can be used to decrease the leakage power compared to parallel implementations due to the reduced number of devices [104],[105]. However, since the static part mainly depends on the process and transistor level circuitry rather than the design on the algorithm and arithmetic levels, the focus throughout this thesis

will be on the dynamic part. Furthermore, the power figure of interest is the average power since this, opposed to peak power, determines the battery life time.

The average dynamic power can be approximated by

$$P_{dyn} = \frac{1}{2}\alpha f_c C_L V_{DD}^2 \tag{1.24}$$

where α is the switching activity, f_c is the clock frequency, C_L is the load capacitance, and V_{DD} is the supply voltage. All these parameters, except α, are normally defined by the layout and specification of the circuit. Hence, an important task is to develop accurate models for estimation of the switching activity, i.e., the average number of transitions between the two logic levels per clock cycle. For example, the switching activity is 2 for a clock signal and 0.5 for a random signal.

In this thesis, several examples are presented where we will see that implementations with the same functionality may differ significantly in power consumption. Hence, an algorithm, used to realize for example a digital filter, can often be modified to achieve lower power [13].

There exist many general methods for reducing the power consumption. An efficient method is to use power down techniques such as clock gating, which according to (1.24) result in zero dynamic power while in idle mode. Another method to reduce the switching activity is to limit the affect of glitches, i.e., the unwanted transitions. This can be done by equalizing the propagation delay between logical paths, either by introducing buffers [122] or by transistor sizing [152]. The power due to glitch spreading can also be reduced by pipelining [84],[125], i.e., by propagating delay elements into nonrecursive parts of the design, since shorter paths then are obtained. Furthermore, pipelining increase the throughput [12],[53], which can be traded for power by scaling the supply voltage [14]. However, the margin for power supply voltage scaling is naturally restricted in new technologies. In multiple-threshold voltage CMOS processes [100], high-speed low-threshold transistors can be used in time critical parts, while low-leakage high-threshold transistors are used in the rest of the circuit.

When different implementations are to be compared, a measure that does not depend on the clock frequency, f_c, used in the simulation, is often preferable. Hence, the energy consumption, E, can be used instead of the power, P, by computing

$$E = \frac{P}{f_c} \tag{1.25}$$

Since the number of clock cycles required to perform one computation varies with the digit-size in serial arithmetic, we will in this work use energy per computation, or energy per sample, as comparison measure for such implementations.

1.7 Outline and Main Contributions

Here, the outline of the rest of this thesis is given. In addition, related publications are specified. Papers that are not included in the thesis, although related, are marked with the symbol †. Parts of this work was earlier presented in

- K. Johansson, *Low Power and Low Complexity Constant Multiplication Using Serial Arithmetic*, Linköping Studies in Science and Technology, Thesis No. 1249, ISBN 91-85523-76-3, Linköping, Sweden, Apr. 2006.

Chapter 2

The complexity of constant multipliers using serial arithmetic is discussed in Chapter 2. In the first part, all possible graph topologies containing up to four adders are considered for single-constant multipliers. In the second part, two new algorithms for multiple-constant multiplication using serial arithmetic are presented and compared to an algorithm designed for parallel arithmetic. This chapter is based on the following publications

- K. Johansson, O. Gustafsson, A. G. Dempster, and L. Wanhammar, "Algorithm to reduce the number of shifts and additions in multiplier blocks using serial arithmetic," in *Proc. IEEE Mediterranean Electrotechnical Conf.*, Dubrovnik, Croatia, May 12–15, 2004, vol. 1, pp. 197–200.
- K. Johansson, O. Gustafsson, and L. Wanhammar, "Low-complexity bit-serial constant-coefficient multipliers," in *Proc. IEEE Int. Symp. Circuits Syst.*, Vancouver, Canada, May 23–26, 2004, vol. 3, pp. 649–652.

- K. Johansson, O. Gustafsson, and L. Wanhammar, "Implementation of low-complexity FIR filters using serial arithmetic," in *Proc. IEEE Int. Symp. Circuits Syst.*, Kobe, Japan, May 23–26, 2005, vol. 2, pp. 1449–1452.
- K. Johansson, O. Gustafsson, A. G. Dempster, and L. Wanhammar, "Trade-offs in low power multiplier blocks using serial arithmetic," in *Proc. National Conf. Radio Science (RVK)*, Linköping, Sweden, June 14–16, 2005, pp. 271–274.
- K. Johansson, O. Gustafsson, and L. Wanhammar, "Trade-offs in multiplier block algorithms for low power digit-serial FIR filters," in *Proc. WSEAS Int. Conf. Circuits*, Vouliagmeni, Greece, July 10–12, 2006.
- K. Johansson, O. Gustafsson, and L. Wanhammar, "Multiple constant multiplication for digit-serial implementation of low power FIR filters," *WSEAS Trans. Circuits Syst.*, vol. 5, no. 7, pp. 1001–1008, July 2006.

Chapter 3

Here, a novel method to compute the switching activities in bit-serial constant multipliers is presented. All possible graph topologies containing up to four adders are considered. The switching activities for most graph topologies can be obtained by the derived equations. However, look-up tables are required for some graphs. Related publications are

- K. Johansson, O. Gustafsson, and L. Wanhammar, "Switching activity in bit-serial constant coefficient serial/parallel multipliers," in *Proc. IEEE NorChip Conf.*, Riga, Latvia, Nov. 10–11, 2003, pp. 260–263.
- K. Johansson, O. Gustafsson, and L. Wanhammar, "Power estimation for bit-serial constant coefficient multipliers," in *Proc. Swedish System-on-Chip Conf.*, Båstad, Sweden, Apr. 13–14, 2004.
- K. Johansson, O. Gustafsson, and L. Wanhammar, "Switching activity in bit-serial constant coefficient multipliers," in *Proc. IEEE Int. Symp. Circuits Syst.*, Vancouver, Canada, May 23–26, 2004, vol. 2, pp. 469–472.

Chapter 4

The complexity of constant multiplication using parallel arithmetic is investigated and optimized at the bit-level. A complexity model is developed, and an MCM algorithm based on this model is shown to reduce the number of full adders. Different strategies to obtain an improved interconnection graph, given the fundamental set, are formulated. Finally, an MCM algorithm where all coefficients are realized at a minimum adder depth is presented. These ideas have been discussed in

- K. Johansson, O. Gustafsson, and L. Wanhammar, "A detailed complexity model for multiple constant multiplication and an algorithm to minimize the complexity," in *Proc. European Conf. Circuit Theory Design*, Cork, Ireland, Aug. 28–Sept. 2, 2005, vol. 3, pp. 465–468.
- O. Gustafsson, A. G. Dempster, K. Johansson, M. D. Macleod, and L. Wanhammar, "Simplified design of constant coefficient multipliers," *Circuits Syst. Signal Processing*, vol. 25, no. 2, pp. 225–251, Apr. 2006. †
- O. Gustafsson, K. Johansson, H. Johansson, and L. Wanhammar, "Implementation of polyphase decomposed FIR filters for interpolation and decimation using multiple constant multiplication techniques," in *Proc. Asia-Pacific Conf. Circuits Syst.*, Singapore, Dec. 4–7, 2006, pp. 924–927. †
- H. Johansson, O. Gustafsson, K. Johansson, and L. Wanhammar, "Adjustable fractional-delay FIR filters using the Farrow structure and multirate techniques," in *Proc. Asia-Pacific Conf. Circuits Syst.*, Singapore, Dec. 4–7, 2006, pp. 1055–1058. †
- K. Johansson, O. Gustafsson, and L. Wanhammar, "Bit-level optimization of shift-and-add based FIR filters," in *Proc. IEEE Int. Conf. Electronics Circuits Syst.*, Marrakech, Morocco, Dec. 11–14, 2007, pp. 713–716.
- M. Abbas, F. Qureshi, Z. Sheikh, O. Gustafsson, H. Johansson, and K. Johansson, "Comparison of multiplierless implementation of nonlinear-phase versus linear-phase FIR filters," in *Proc. Asilomar Conf. Signals Syst. Comp.*, Pacific Grove, CA, Oct. 26–29, 2008. †
- K. Johansson, O. Gustafsson, L. S. DeBrunner, and L. Wanhammar, "Low power multiplierless FIR filters implemented using a minimum depth algorithm," in preparation.

Chapter 5

In this chapter, an approach to derive a detailed estimation of the energy consumption for ripple-carry adders is presented. The model includes both computation of the theoretic switching activity and the simulated energy consumption for each possible transition. Furthermore, the model can be used for any given correlation of the input data. The method is also simplified by adopting the dual bit type method [78]. Finally, the switching activity in constant multiplication is studied. An accurate model for single adder multipliers is presented. Parts of this work was previously published in

- K. Johansson, O. Gustafsson, and L. Wanhammar, "Power estimation for ripple-carry adders with correlated input data," in *Proc. Int. Workshop Power Timing Modeling Optimization Simulation*, Santorini, Greece, Sept. 15–17, 2004, pp. 662–674.
- K. Johansson, O. Gustafsson, and L. Wanhammar, "Estimation of switching activity for ripple-carry adders adopting the dual bit type method," in *Proc. Swedish System-on-Chip Conf.*, Tammsvik, Sweden, Apr. 18–19, 2005.
- O. Gustafsson, S. T. Oskuii, K. Johansson, and P. G. Kjeldsberg, "Switching activity reduction of MAC-based FIR filters with correlated input data," in *Proc. Int. Workshop Power Timing Modeling Optimization Simulation*, Gothenburg, Sweden, Sept. 3–5, 2007, pp. 526–535. †
- K. Johansson, O. Gustafsson, and L. Wanhammar, "Switching activity estimation for shift-and-add based constant multipliers," in *Proc. IEEE Int. Symp. Circuits Syst.*, Seattle, WA, May 18–21, 2008, pp. 676–679.

Chapter 6

Function approximation is discussed in Chapter 6. A general method to rewrite any function as a sum of weighted bit-products is presented. It is shown that a majority of the bit-products can be neglected for many elementary functions, while still maintaining high accuracy. The proposed method can be implemented using a low complexity hardware architecture. In addition, it is possible to divide the architecture into sub-blocks, which may be turned off to reduce the energy consumption. To evaluate the presented approximation approach, functions that are required for conversions and addition in LNS are implemented. Furthermore, sine and cosine can be simultaneously computed. This work is covered in the listed publications

- L. Wanhammar, K. Johansson, and O. Gustafsson, “Efficient sine and cosine computation using a weighted sum of bit-products,” in *Proc. European Conf. Circuit Theory Design*, Cork, Ireland, Aug. 28–Sept. 2, 2005, vol. 1, pp. 139–142.
- O. Gustafsson, K. Johansson, and L. Wanhammar, “Optimization and quantization effects for sine and cosine computation using a sum of bit-products,” in *Proc. Asilomar Conf. Signals Syst. Comp.*, Pacific Grove, CA, Oct. 30–Nov. 2, 2005, pp. 1347–1351. †
- K. Johansson, O. Gustafsson, and L. Wanhammar, “Low power architectures for sine and cosine computation using a sum of bit-products,” in *Proc. IEEE NorChip Conf.*, Oulu, Finland, Nov. 21–22, 2005, pp. 161–164.
- K. Johansson, O. Gustafsson, and L. Wanhammar, “Approximation of elementary functions using a weighted sum of bit-products,” in *Proc. IEEE Int. Symp. Circuits Syst.*, Kos Island, Greece, May 21–24, 2006, pp. 795–798.
- O. Gustafsson and K. Johansson, “Multiplierless piecewise linear approximation of elementary functions,” in *Proc. Asilomar Conf. Signals Syst. Comp.*, Pacific Grove, CA, Oct. 29–Nov. 1, 2006, pp. 1678–1681. †
- K. Johansson, O. Gustafsson, and L. Wanhammar, “Conversion and addition in logarithmic number systems using a sum of bit-products,” in *Proc. IEEE NorChip Conf.*, Linköping, Sweden, Nov. 20–21, 2006, pp. 39–42.
- S. Tahmasbi Oskuii, K. Johansson, O. Gustafsson, and P. G. Kjeldsberg, “Power optimization of weighted bit-product summation tree for elementary function generator,” in *Proc. IEEE Int. Symp. Circuits Syst.*, Seattle, WA, May 18–21, 2008, pp. 1240–1243. †
- K. Johansson, O. Gustafsson, and L. Wanhammar, “Implementation of elementary functions for logarithmic number systems,” *IET Computers Digital Techniques (Selected Papers NorChip 2006)*, vol. 2, no. 4, pp. 295–304, July 2008.
- O. Gustafsson and K. Johansson, “An empirical study on standard cell synthesis of elementary function look-up tables,” in *Proc. Asilomar Conf. Signals Syst. Comp.*, Pacific Grove, CA, Oct. 26–29, 2008. †

Publications by the Author that are not Related to the Thesis

- T. Lindkvist, J. Löfvenberg, H. Ohlsson, K. Johansson, and L. Wanhammar, “A power-efficient, low-complexity, memoryless coding scheme for buses with dominating inter-wire capacitance,” in *Proc. IEEE Int. Workshop System-on-Chip Real-Time Appl.*, Banff, Canada, July 19–21, 2004, pp. 257–262.
- H. Ohlsson, B Mesgarzadeh, K. Johansson, O. Gustafsson, P. Löwenborg, H. Johansson, and A. Alvandpour, “A 16 GSPS 0.18 μm CMOS decimator for single-bit ΣΔ-modulation,” in *Proc. IEEE NorChip Conf.*, Oslo, Norway, Nov. 8–9, 2004, pp. 175–178.
- J. Löfvenberg, O. Gustafsson, K. Johansson, T. Lindkvist, H. Ohlsson, and L. Wanhammar, “New applications for coding theory in low-power electronic circuits,” in *Proc. Swedish System-on-Chip Conf.*, Tammsvik, Sweden, Apr. 18–19, 2005.
- J. Löfvenberg, O. Gustafsson, K. Johansson, T. Lindkvist, H. Ohlsson, and L. Wanhammar, “Coding schemes for deep sub-micron data buses,” *National Conf. Radio Science (RVK)*, Linköping, Sweden, June 14–16, 2005, pp. 257–260.
- L. Wanhammar, B. Soltanian, K. Johansson, and O. Gustafsson, “Synthesis of circulator-tree wave digital filters,” in *Proc. Int. Symp. Image Signal Processing Analysis*, Istanbul, Turkey, Sept. 27–29, 2007, pp. 206–211.
- L. Wanhammar, B. Soltanian, O. Gustafsson, and K. Johansson, “Synthesis of bandpass circulator-tree wave digital filters,” in *Proc. IEEE Int. Conf. Electronics Circuits Syst.*, Malta, Aug. 31–Sept. 3, 2008.

2

COMPLEXITY OF SERIAL CONSTANT MULTIPLIERS

In this chapter, the possibilities to minimize the complexity of bit-serial single-constant multipliers are investigated [57]. This is done in terms of the required number of building blocks, which includes adders and shifts. The multipliers are described using a graph representation. It is shown that a minimum set of graphs, required to obtain optimal results given certain restrictions, can be found.

In the case of single-constant multipliers, the number of possible solutions can be limited because of the finite number of graph topologies. However, if a shift-and-add network realizing several coefficients is required, a multiple-constant multiplication (MCM) problem is obtained. Different heuristic algorithms can then be used to reduce the complexity, by utilizing the redundancy between the coefficients. Two algorithms suitable to achieve efficient realization of MCM using serial arithmetic are presented [56],[62],[66]. It is shown that the new algorithms reduce the total complexity significantly.

Furthermore, we study the trade-offs in implementations of FIR filters using MCM and digit-serial arithmetic. Comparisons considering area, speed, and energy consumption, with respect to the digit-size, are performed [61],[67].

2.1 Graph Multipliers

In this section, different types of single-constant graph multipliers will be defined, with respect to constraints on adder cost and throughput. Furthermore, the possibilities to exclude some graphs from the search space are examined.

The investigation covers all coefficients up to 4095 and all types of graph multipliers containing up to four adders. All possible graphs, using the representation discussed in Section 1.4.3, for adder costs from 1 to 4 are presented in Fig. 2.1 [24].

Note that although bit-serial arithmetic will be assumed for the multipliers, results considering adder and flip-flop costs are generally also valid for any digit-serial implementation. However, the number of registers that are required to perform pipelining depend on the digit-size. Furthermore, the cost difference between adders and shifts becomes higher for larger digit-sizes, since the number of full adders increases linearly while the number of flip-flops is constant. Hence, such trade-offs are mainly of interest for small digit-sizes.

2.1.1 Multiplier Types

Different multiplier types can be defined based on the requirements considering adder cost, flip-flop cost, and pipelining. The types that will be discussed here are described in the following.

- **CSD – Canonic Signed-Digit multiplier**
  Multiplier based on the CSD representation, as discussed in Section 1.4.1, with an adder cost equal to one less than the number of nonzero digits.
- **MSD – Minimum Signed-Digit multiplier**
  Similar to the CSD multiplier and requires the same number of adders, but can in some cases decrease the flip-flop cost by using other MSD representations, which were discussed in Section 1.2.2.
- **MAG – Minimum Adder Graph multiplier**
  Graph multiplier that is based on any of the topologies in Fig. 2.1 and, for any given coefficient, has the lowest possible adder cost.
- **CSDAG – CSD Adder Graph multiplier**
  Similar to the MAG multiplier, but may use the same number of

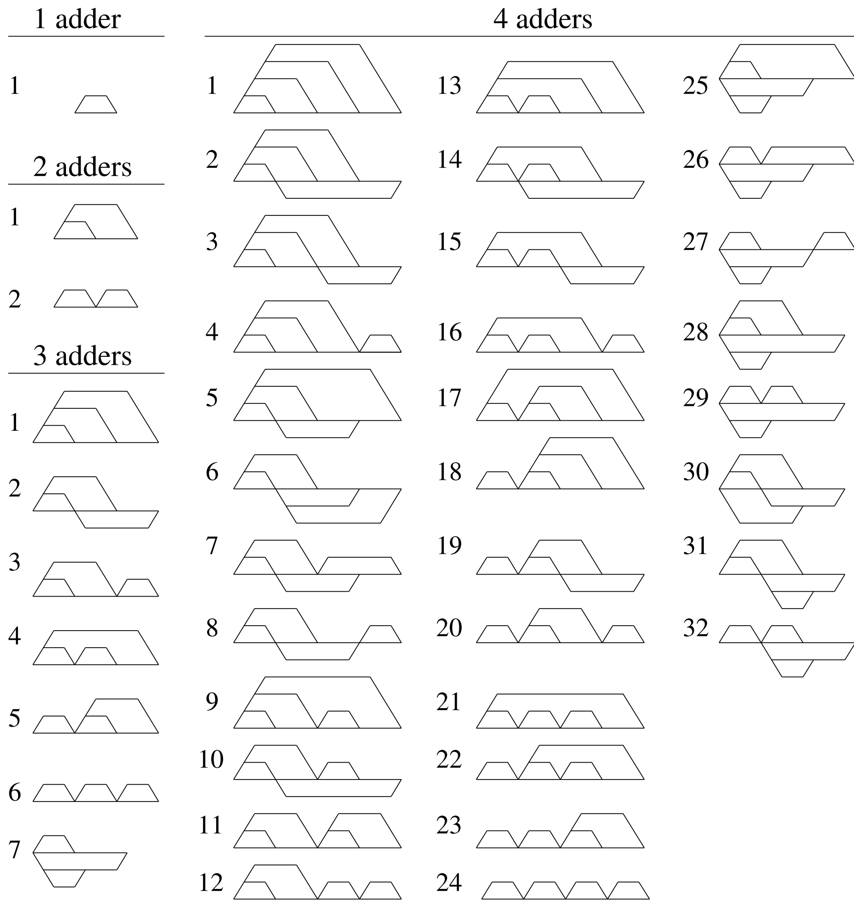


Figure 2.1 Possible graph topologies for an adder cost up to four.

adders as the corresponding CSD/MSD multiplier, and can by that reduce the flip-flop cost.

- **PL MAG/PL CSDAG – Pipelined graph multiplier**
  In a pipelined bit-serial graph multiplier, there is at least one intermediate flip-flop between the full adders. This property, which is always obtained for the CSD/MSD multipliers, result in higher throughput.

Example

To describe the difference between the defined multiplier types, corresponding realizations of the coefficient 2813, which has the CSD representation $10\bar{1}0\bar{1}00000\bar{1}01$, are shown in Fig. 2.2. There are other possible solutions for all types except the CSD multiplier. However, note that the values corresponding to the nonzero digits in the CSD representation can

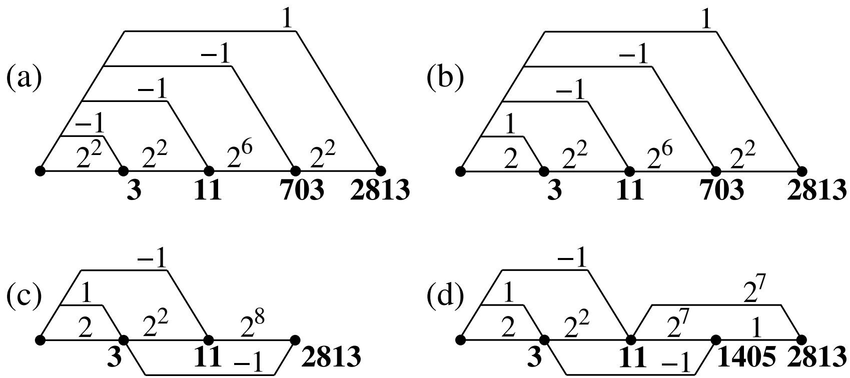


Figure 2.2 Different realizations of the coefficient 2813. (a) CSD, (b) MSD, (c) MAG, and (d) CSDAG.

be added in different orders, resulting in other structures. Since this may eliminate the pipeline feature, the basic structure used in Fig. 2.2 (a) will be assumed for CSD multipliers.

The adder costs for the multipliers in Figs. 2.2 (a), (b), (c), and (d) are 4, 4, 3, and 4, respectively. The flip-flop costs are 12, 11, 11, and 10. This implies that it is possible to save either two shifts, or one adder and one shift compared to the CSD multiplier. Note that shifts may be shared as discussed in Section 1.5.2, for example, the two 2^7-edges in Fig. 2.2 (d).

Pipelined CSDAG and MAG can be obtained from the multipliers in Figs. 2.2 (b) and (c) with an extra cost of 0 and 1 register, respectively. Note that the flip-flop cost will include both shifts and pipelining registers, since both correspond to a single flip-flop in bit-serial arithmetic.

2.1.2 Graph Elimination

To make the search for the best solutions less extensive, it is desirable to find a minimum set of graphs that is sufficient to obtain optimal results. If, for example, we consider the two graphs shown in Fig. 2.3, we will see that they can realize the same set of coefficients. For the graph in Fig. 2.3 (a) we get the coefficient set [37] expression

$$1 + 2^{b} + 2^{c} + 2^{a+b} + 2^{a+c} \tag{2.1}$$

and the corresponding formula for the graph in Fig. 2.3 (b) is

$$1 + 2^{y+z} + 2^{z} + 2^{x+y+z} + 2^{x+z} \tag{2.2}$$

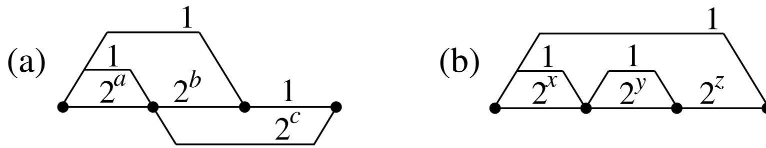


Figure 2.3 Different graphs with the same coefficient set.

The substitutions $x = a$, $y = b - c$, and $z = c$ in (2.2) gives the same coefficient set expression as in (2.1). A simplification, in this example, was that all edge signs were assumed to be positive, but the graphs have the same coefficient set even if signs are considered.

It is also possible to set up expressions for the flip-flop cost. For the graph in Fig. 2.3 (a) the flip-flop cost is $a + \max(b, c)$. The additional cost with pipelining is 1 if $b > c$ and 2 otherwise. The corresponding expression for the graph in Fig. 2.3 (b) is $x + y + z$, with the extra pipelining cost 0 if $z > 1$ and 1 otherwise.

From the coefficient set and flip-flop cost expressions, it is possible to eliminate graphs that are not required to obtain optimal results. This covering problem [101] has different solutions, and one minimal graph set for each multiplier type is given in Fig. 2.4. Note that some graphs occur more than once, but with different positions of the shift operations. There are in total 147 different graph types that can be obtained from the 42 graphs shown in Fig. 2.1. Out of these 147 graph types, only 16 and 18 are required to always obtain an optimal realization for MAG and CSDAG, respectively. Corresponding numbers for PL MAG and PL CSDAG are 18 and 13. Note that the graph structures in Fig. 2.4 (e) generally require fewer additional registers when pipelining is introduced than the ones in Fig. 2.4 (b).

2.2 Complexity Comparison – Single Multiplier

Here, we will compare the complexity of different multiplier types. Due to the fact that adder cost has been discussed before [24],[37], the focus is on the flip-flop cost. Since the CSD representation is more commonly used than other MSD representations, most comparisons will be between CSD and graph multipliers. As a rule of thumb, it can be said that the average flip-flop cost for MSD multipliers is about 1/3 lower than for CSD multipliers, i.e., one shift can be saved in every third case.

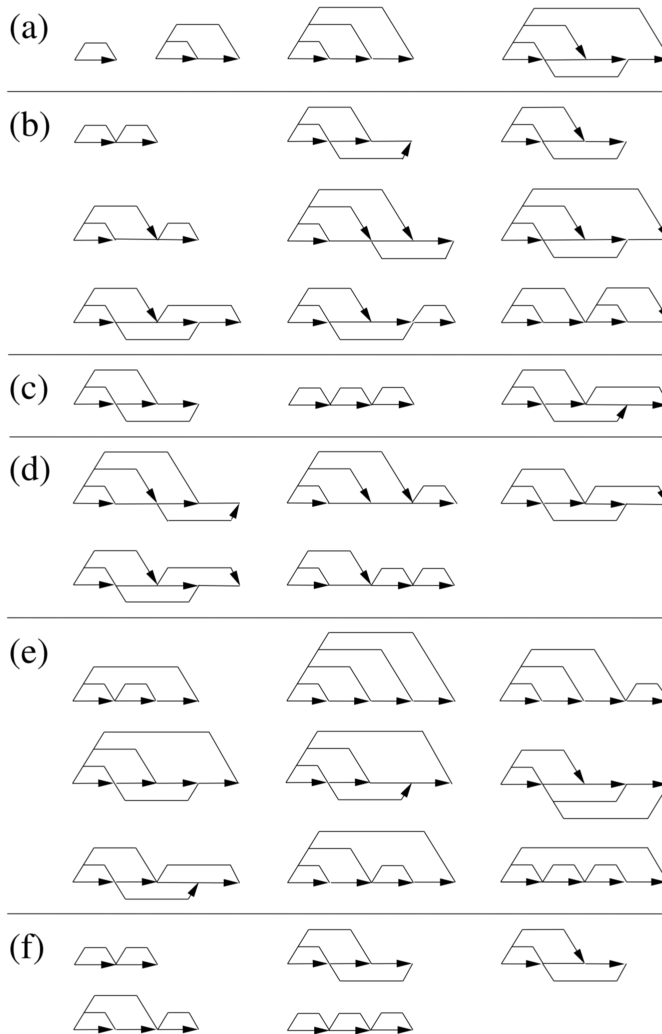


Figure 2.4 (a) Graphs required for all multiplier types. Additional graphs for (b) both MAG and CSDAG, (c) MAG, (d) CSDAG, (e) both PL MAG and PL CSDAG, and (f) PL MAG. Here, edges with shifts are marked by arrows.

2.2.1 Comparison of Flip-Flop Cost

The multiplier types are here compared in terms of the average flip-flop cost that is required to realize all coefficients of a given wordlength, i.e., for wordlength B all integer values from 1 to $2^B - 1$ are considered. Note

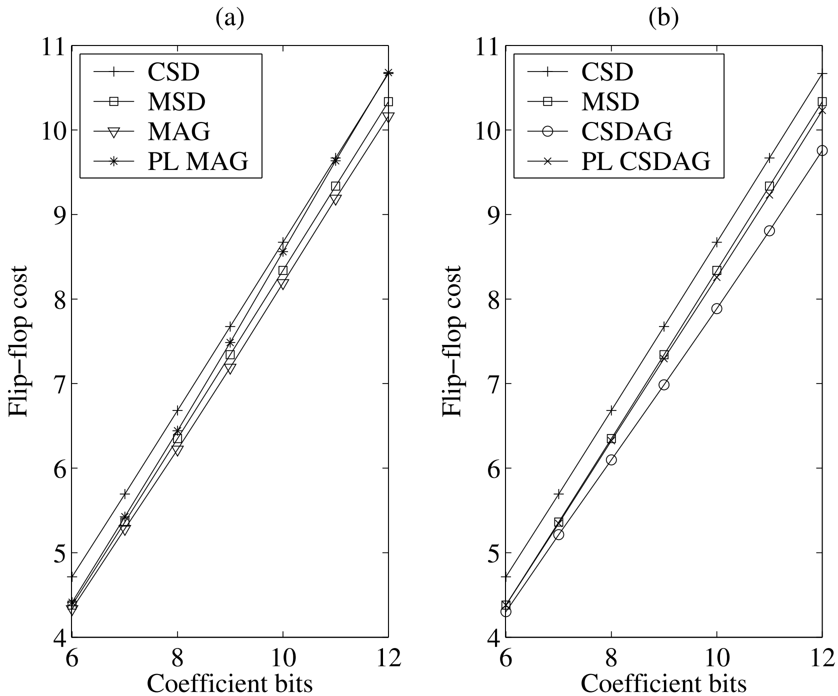


Figure 2.5 Average flip-flop cost compared with CSD/MSD multipliers. (a) MAG and (b) CSDAG multipliers.

that the flip-flop cost for a CSD multiplier is directly defined by the position of the most significant nonzero bit in the CSD representation.

The results for MAG multipliers are shown in Fig. 2.5 (a). It can be seen that not only adders are saved [24], but also shifts by using the graph multipliers instead of CSD/MSD multipliers. This is true as long as the multipliers do not need to be pipelined. In Fig. 2.5 (b), we do not have the minimum adder cost requirement, but no more adder than for the corresponding CSD/MSD multiplier is allowed. Since it here is possible to select the same structure as CSD/MSD for all coefficients, also the pipelined graph multiplier has a lower flip-flop cost (this is not completely true as we will see soon). The savings in shifts is higher than in the previous case. The conclusion is that a trade-off between adder cost and flip-flop cost is possible.

In Fig. 2.6, it can be seen that the percentage improvement in flip-flop cost for the CSDAG multiplier is almost constant around 9%, i.e., independent of the number of coefficient bits. For the MAG and PL CSDAG multipliers, the savings does not increase as fast as the average flip-flop

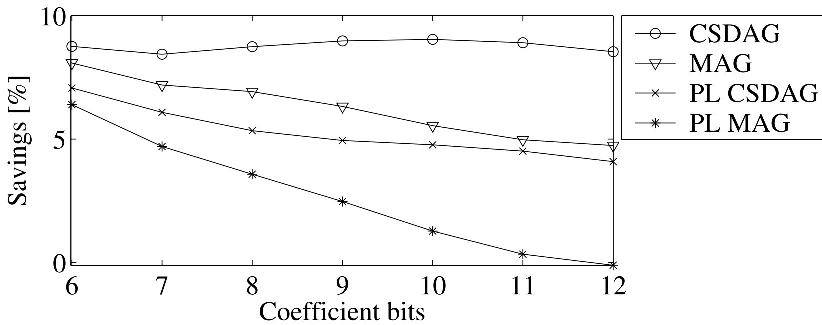


Figure 2.6 Average percentage improvement in flip-flop cost for different types of graph multipliers over CSD multipliers.

cost, which result in a decreasing percentage improvement for larger number of coefficient bits. The flip-flop cost for the PL MAG multipliers is increasing faster than for the CSD multipliers, and for 12 bit coefficients they have approximately the same average flip-flop cost, i.e., the only improvement is a reduced adder cost.

The average cost does not show how often shifts are saved. To visualize this, we can study histograms where the frequency of a certain number of shifts saved is presented. In Fig. 2.7, the four different graph multiplier types are compared to the CSD multiplier, considering all coefficients with 12 bits. In Fig. 2.7 (a), we can see that one shift is saved for 52% of the coefficients in the CSDAG case, and that two shifts are saved for 19% of the coefficients. The corresponding histogram for MAG is shown in Fig. 2.7 (b), where the savings in flip-flop cost are significantly lower because of the minimum adder cost requirement. Here, one shift is saved for 46% and two shifts for 2% of the coefficients.

The savings becomes smaller if a pipelined multiplier is required, since this is inherent for the CSD multipliers, as shown in Figs. 2.7 (c) and (d). One result that might seem strange is that the savings are negative in a few cases for the PL CSDAG multiplier in Fig. 2.7 (c). The explanation to this is that the CSD multipliers for some coefficients have to use more than four adders, which is not possible for the studied graph multipliers since only the structures in Fig. 2.1 are considered. Hence, in these cases where the PL CSDAG multiplier has a higher flip-flop cost, a lower adder cost is guaranteed.

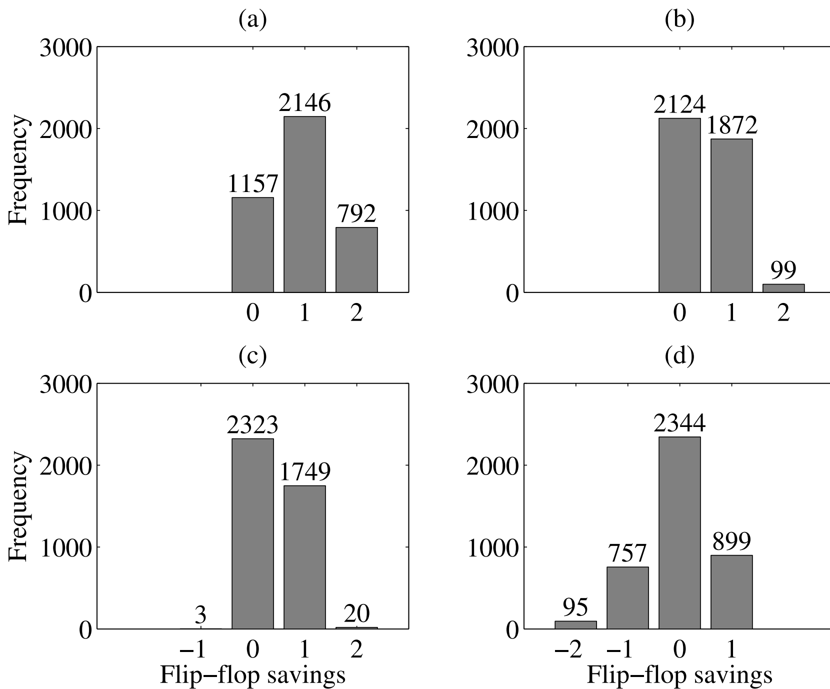


Figure 2.7 Different types of graph multipliers compared to CSD multipliers in terms of flip-flop cost, considering all coefficients with up to 12 bits. (a) CSDAG, (b) MAG, (c) PL CSDAG, and (d) PL MAG.

2.2.2 Comparison of Building Block Cost

In the previous section, we have only discussed the flip-flop cost, under the condition that the adder cost is either minimized or at least not higher than for the corresponding CSD multiplier. To obtain a complete complexity measure, we need to consider both adders and shifts. The cost difference between adders and shifts in terms of chip area and energy consumption depend on the implementation technology. A general rule can be formulated from the results in [55], stating that an adder in terms of energy consumption is more expensive than a shift, but less expensive than two shifts. Note that this relation only is valid for the bit-serial case, and that it may depend on the process technology. However, we will basically assume an equal cost for adders and shifts in the following comparison. Hence, we study the savings in number of building blocks, which is shown in Fig. 2.8. The histograms in Figs. 2.8 (a) and (b) are almost iden-

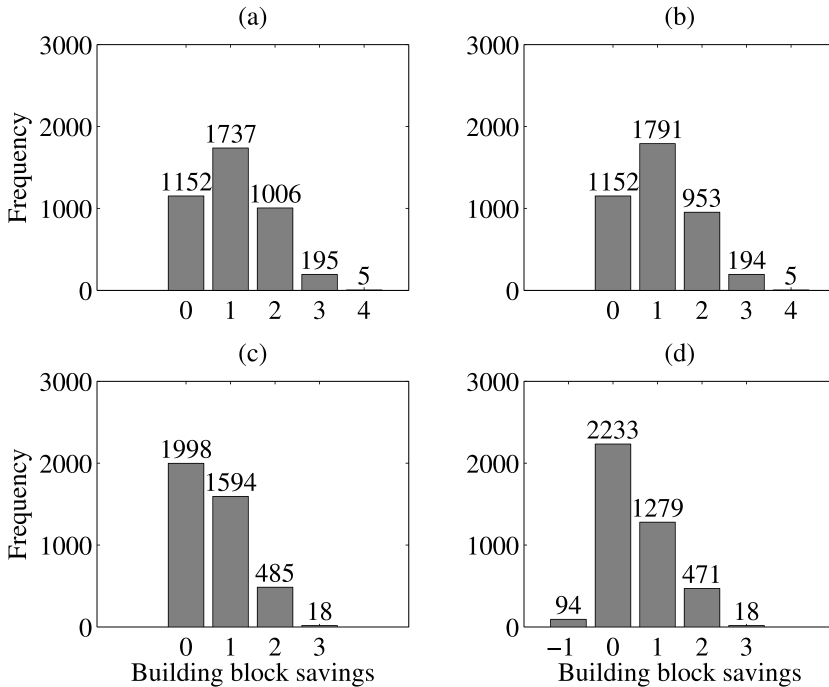


Figure 2.8 Different types of graph multipliers compared to CSD multipliers in terms of building block cost, considering all coefficients with up to 12 bits. (a) CSDAG, (b) MAG, (c) PL CSDAG, and (d) PL MAG.

tical. From this we can conclude that the extra savings in shifts for CSDAG multipliers, is approximately as large as the extra savings in adders for MAG multipliers. The savings for the pipelined graph multipliers, corresponding to the histograms in Figs. 2.8 (c) and (d), are similar to each other for the same reason.

In a few cases, it is possible to save four building blocks compared to the CSD multiplier. An example of this is the coefficient 2739, with the CSD representation $10\bar{1}0\bar{1}0\bar{1}0\bar{1}010\bar{1}$. In Fig. 2.9 (a) it can be seen that the CSD realization for this coefficient requires 6 adders and 12 shifts, while only 4 adders and 10 shifts are needed in the MAG design illustrated in Fig. 2.9 (b).

As was shown in Fig. 2.8, the savings in building blocks are similar for MAG and CSDAG multipliers. The difference in adder cost and flip-flop cost is given in Table 2.1. It can be seen that MAG and CSDAG multipliers have the same number of adders and shifts for 2490 coefficients, while the case for 55 coefficients is that the CSDAG multiplier

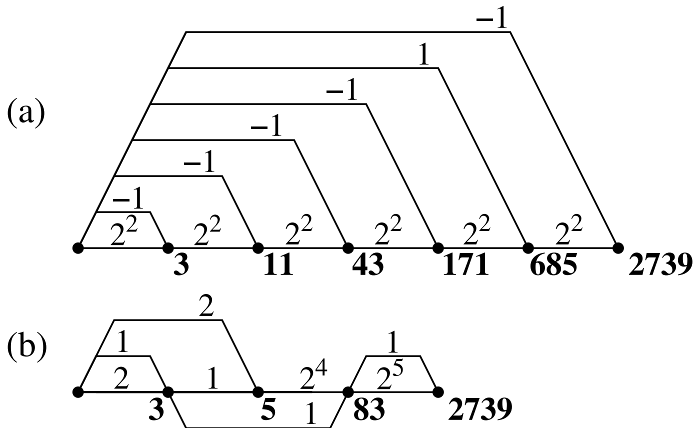


Figure 2.9 Different realizations of the coefficient 2739. (a) CSD using 18 building blocks and (b) MAG using 14 building blocks.

MAG vs. CSDAG		
Extra shifts for MAG	Number of cases	
0	2490	0
1	0	1550
2	0	55
3	0	0
Extra adders for CSDAG	0	1

PL MAG vs. PL CSDAG		
Extra shifts for PL MAG	Number of cases	
0	2698	0
1	0	1001
2	0	355
3	0	41
Extra adders for PL CSDAG	0	1

Table 2.1 Difference in adder and flip-flop costs for graph multipliers considering all coefficients with 12 bits.

require one adder more than the MAG multiplier but in return saves two shifts. The average building block cost for CSDAG/PL CSDAG is lower than for MAG/PL MAG, especially for the pipelined graph multipliers. As stated in Table 2.1, 2 and 3 shifts can be saved in 355 and 41 cases, respectively, to a cost of one extra adder. This shows that a minimum number of adders does not necessarily result in an optimal solution.

2.3 Complexity Comparison – RSAG-*n*

An MCM algorithm suitable for serial arithmetic will be presented and compared to a well-known algorithm, referred to as RAG-*n* [25], in terms of adder and flip-flop costs.

2.3.1 The Reduced Shift and Adder Graph Algorithm

In [25], the *n*-dimensional Reduced Adder Graph (RAG-*n*) algorithm was introduced. This algorithm is known to be one of the best MCM algorithms in terms of adder cost. Based on this algorithm, an *n*-dimensional Reduced Shift and Adder Graph (RSAG-*n*) algorithm has been developed [56]. Hence, RSAG-*n* is also a graph based algorithm. The new algorithm not only tries to minimize the adder cost, but also the sum of the maximum number of shifts of all fundamentals, i.e., the flip-flop cost computed according to (1.21).

The steps of the RSAG-*n* algorithm are in the following described using the terms introduced in Sections 1.4.3 and 1.4.4. Realized coefficients are removed from the coefficient set and added to the interconnection table, which specifies how the value is obtained. The termination condition of the algorithm is that the coefficient set is empty.

1. Check the input vector, i.e., the required coefficient set, H. Remove zeros, ones, and repeated coefficients and then copy the rest of the elements to the coefficient set, C.
2. For each coefficient, c_i, with adder cost zero, i.e., c_i is a power-of-two, add the row $[c_i\ 1\ 0\ c_i\ 0\ 0]$ to the interconnection table, G, and remove c_i from the coefficient set, C.
3. Compute a sum matrix based on power-of-two multiples of the values that are available in the fundamental set. At start this matrix is

$$
e_k f_k \;
\begin{array}{c} \\ 1 \\ -1 \\ 2 \\ -2 \\ 4 \\ \dots \end{array}
\quad
\begin{array}{c}
e_j f_j \\
\begin{array}{cccccc} 1 & -1 & 2 & -2 & 4 & \dots \end{array} \\
\begin{bmatrix}
2 & 0 & 3 & -1 & 5 & \dots \\
0 & -2 & 1 & -3 & 3 & \dots \\
3 & 1 & 4 & 0 & 6 & \dots \\
-1 & -3 & 0 & -4 & 2 & \dots \\
5 & 3 & 6 & 2 & 8 & \dots \\
\dots & \dots & \dots & \dots & \dots & \dots
\end{bmatrix}
\end{array}
\tag{2.3}
$$

and it is extended when new fundamentals are included. The cost zero coefficients found in step 2 can be ignored since they are powers-of-two, and therefore already available. Note that only half the matrix is essentially needed due to symmetry. Fundamentals, f_i, that appear in the sum matrix can be computed according to (1.17) and therefore realized with one extra adder. If any required coefficients are found in the matrix, compute the flip-flop costs according to (1.21). Find the coefficients that require the lowest number of additional shifts, and select the smallest of those. Add this coefficient to the fundamental set, F, and the interconnection table, G, and remove it from the coefficient set, C.

4. Repeat step 3 until no new coefficient is found in the sum matrix.
5. For each remaining coefficient, c_i, check if it can be obtained by the strategies illustrated in Fig. 2.10. For both these cases, two new adders are required. If any coefficients are found, select the smallest coefficient, c_i, of those that require the lowest number of additional shifts. Add this coefficient and the extra fundamental, f_{new}, to the fundamental set, F, and the interconnection table, G. Remove the coefficient from the coefficient set, C.
6. Repeat steps 4 and 5 until no new coefficient is found.
7. Choose the smallest coefficient among the ones with lowest single-coefficient adder cost. Different sets of fundamentals that can be used to realize the coefficient are obtained from a look-up table. Both the single-coefficient costs and the fundamental sets are generated by the MAG algorithm [24]. For each set, remove values that are already included in the fundamental set and compute the flip-flop cost. Find the sets that require the lowest number of additional shifts, and of

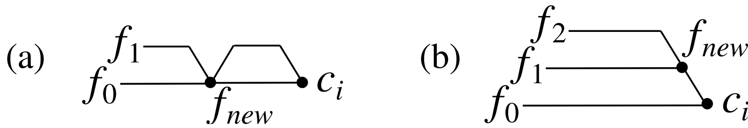


Figure 2.10 The coefficient, c_i, is obtained from (a) two existing fundamentals or (b) three existing fundamentals. Note that two (or more) f_i may be the same fundamental. All edge values are arbitrary powers-of-two.

those, select the set with smallest sum. Add this set and the coefficient to the fundamental set and the interconnection table. Remove the coefficient from the coefficient set.

8. Repeat steps 4, 5, 6, and 7 until the coefficient set is empty.

The basic ideas for the RAG-*n* [25] and RSAG-*n* algorithms are similar, but the resulting solutions differ significantly. The main difference is that RAG-*n* chooses to realize coefficients by using extra fundamentals of minimum value, while RSAG-*n* chooses fundamentals that require a minimum number of additional shifts. The result of these two different strategies is that RAG-*n* is more likely to reuse fundamentals due to the selection of smaller values (see Section 4.4.1) and by that reduce the adder cost, while RSAG-*n* is more likely to reduce the flip-flop cost.

Because RAG-*n* assumes shifts to be free, it only considers odd coefficients. Hence, it divides all even coefficients in the input set by two until they become odd. RSAG-*n* on the other hand preserves the even coefficients, so that all shifts remain inside the shift-and-add network, which enable an overall optimization.

Another difference is that RAG-*n* concurrently adds all possible coefficients that can be realized with one additional adder each, while RSAG-*n* only adds one coefficient at a time to be able to minimize the number of shifts in an efficient way. This often results in a larger adder depth for the RSAG-*n* algorithm. Furthermore, RSAG-*n* is slower due to more iterations to include the same number of coefficients. Another contribution to the execution time is the repeated counting of shifts. This is performed according to (1.21) and requires that the interconnection table is computed during the search, which is not necessary for the RAG-*n* algorithm. Hence, the RAG-*n* algorithm agrees with the flow graph in Fig. 1.13, while the RSAG-*n* algorithm differ both regarding the retained even coefficients and the simultaneous interconnection creation. These

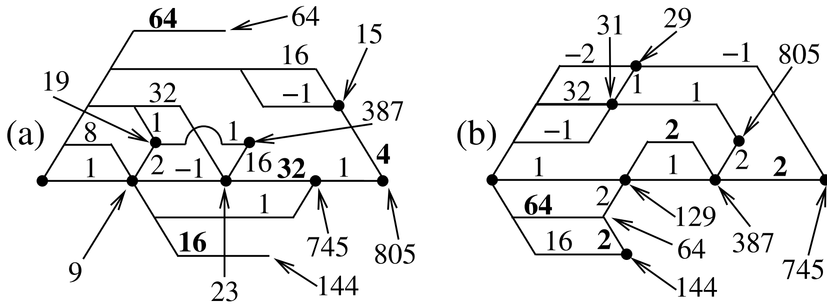


Figure 2.11 Realizations of the same coefficient set using different algorithms, (a) RAG-n and (b) RSAG-n. The largest absolute edge values (except ones) for each vertex are in bold.

changes have been done to enable control over the shifts, but may cause an increase in adder cost and adder depth.

It is worth noting that if all coefficients are realized before step 5 of the algorithm, the corresponding realization is guaranteed to be optimal in terms of adder cost [25],[45].

Example

To illustrate some of the differences between the two algorithms, we consider an example. The coefficient set, C, contains five random coefficients of wordlength 10 (the current limit of the table used in step 7 is 12 bits) and is given by

$$C = \{387, 144, 64, 745, 805\} \tag{2.4}$$

The resulting fundamental sets are

$$\begin{aligned} F_{\text{RAG}-n} &= \{9, 23, 745, 15, 805, 19, 387, 64, 144\} \\ F_{\text{RSAG}-n} &= \{64, 144, 129, 387, 31, 805, 29, 745\} \end{aligned} \tag{2.5}$$

where the different order in which the coefficients are added to the fundamental sets can be seen. For example, RAG-n first divides all even coefficients by two until they are odd (144 to 9 and 64 to 1) and then has to compensate for this at the end, while RSAG-n in this case starts with the easily realized even coefficients.

In Fig. 2.11 (a), the shift-and-add network using the RAG-n algorithm is shown. The realization requires 7 adders and 17 shifts. If the RSAG-n algorithm is used, the realization illustrated in Fig. 2.11 (b) is obtained. Here, the number of adders is the same, while the number of shifts is

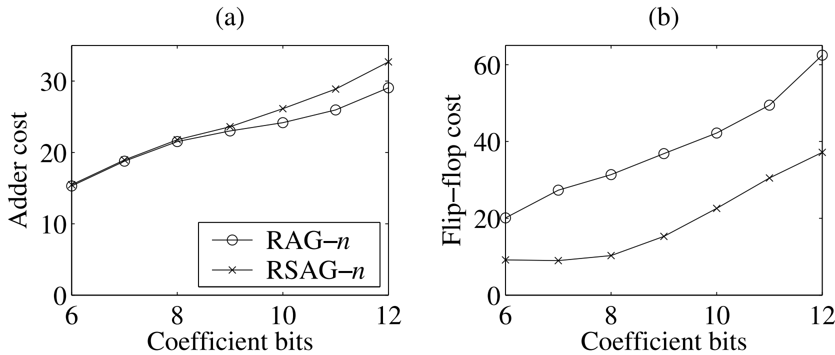


Figure 2.12 Average number of (a) adders and (b) shifts. Sets of 25 coefficients with wordlengths from 6 to 12 bits are used.

reduced to nine. In Fig. 2.11, it can be seen that RSAG-*n* only has edge values larger than two at the input vertex, while RAG-*n* has large edge values also at some other vertices, which will increase the flip-flop cost.

2.3.2 Comparison by Varying the Wordlength

In the following, the presented algorithm is compared to the RAG-*n* algorithm. Average results are for 100 random coefficient sets, containing a certain number of coefficients of a certain wordlength. The maximum coefficient wordlength is restricted to 12 bits due to the size of the look-up table used by both algorithms.

In Fig. 2.12, the average adder and flip-flop costs for the two algorithms are shown for varying number of coefficient bits, when sets of 25 coefficients are used (the same coefficient sets are of course used for both algorithms). It is clear that the average flip-flop cost is lower for RSAG-*n*, while the adder cost is lower for RAG-*n*. This relation was predicted in the previous section. For the worst case wordlength in Fig. 2.12, on average more than six shifts are saved for every extra adder. Such a trade-off should be advantageous in most implementations.

In Figs. 2.13 and 2.14, histograms of the savings in adder and flip-flop costs using 7 and 10 bit coefficients, respectively, are shown. For 7 bit coefficients, the adder cost for 85% of the coefficient sets are the same for both algorithms, while the adder cost is significantly smaller for RAG-*n* when 10 bit coefficients are used. The savings in shifts are large for almost all coefficient sets, but does not differ significantly depending on the number of coefficient bits.

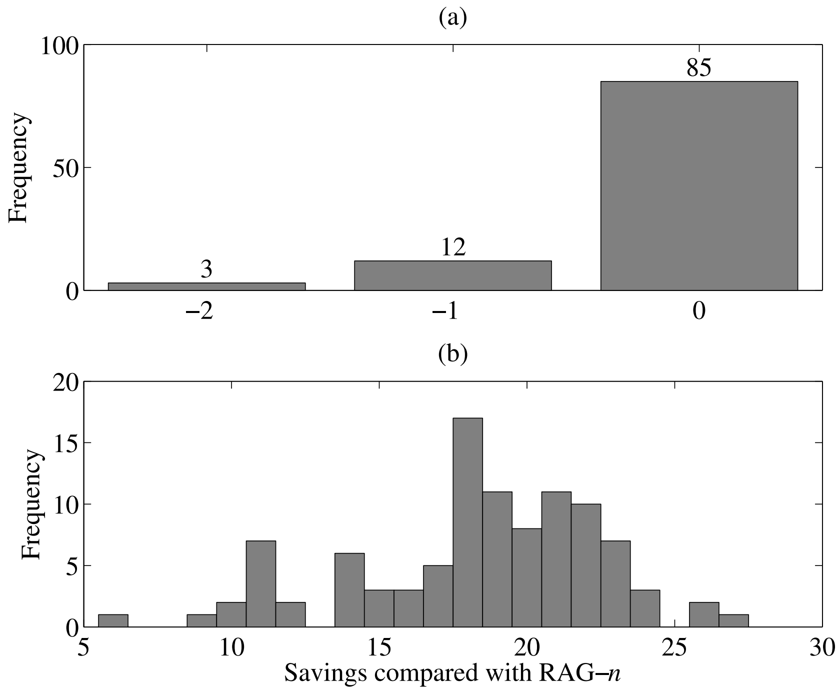


Figure 2.13 Frequency of savings in (a) adder and (b) flip-flop costs using RSAG-n compared with RAG-n. 100 sets of 25 coefficients with a wordlength of 7 bits are used.

2.3.3 Comparison by Varying the Setsize

In Fig. 2.15, the average adder and flip-flop costs for the two algorithms are shown for varying number of coefficients, when coefficients of word-length 10 are used. The difference in adder cost has a maximum when the coefficient setsize is 20. For large coefficient sets, both algorithms are likely to have an optimal adder cost. This is due to the fact that more coefficients give more possibilities in step 3 of the algorithm.

The flip-flop cost for RSAG-n has a maximum for setsize 20. When more fundamentals are available, the flexibility is increased and, if it is desired, coefficients are more likely to be obtained without any additional shifts. The RAG-n algorithm does not take advantage of this, and therefore has an increasing flip-flop cost for larger sets. Hence, for large coefficient sets the number of shifts is drastically reduced at a low extra adder cost. For the worst case coefficient setsize, an average of six shifts are saved for each extra adder.

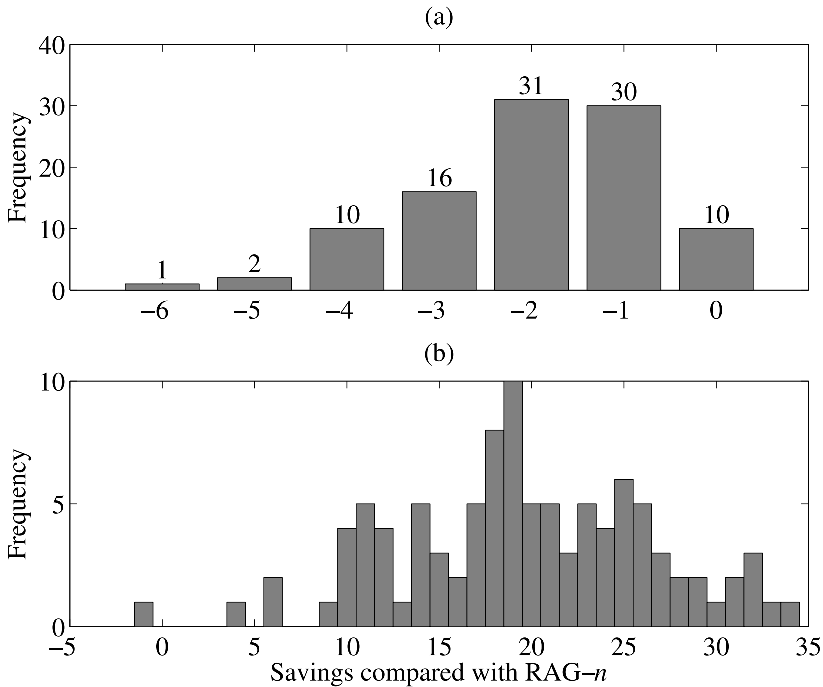


Figure 2.14 Frequency of savings in (a) adder and (b) flip-flop costs using RSAG-*n* compared with RAG-*n*. 100 sets of 25 coefficients with a wordlength of 10 bits are used.

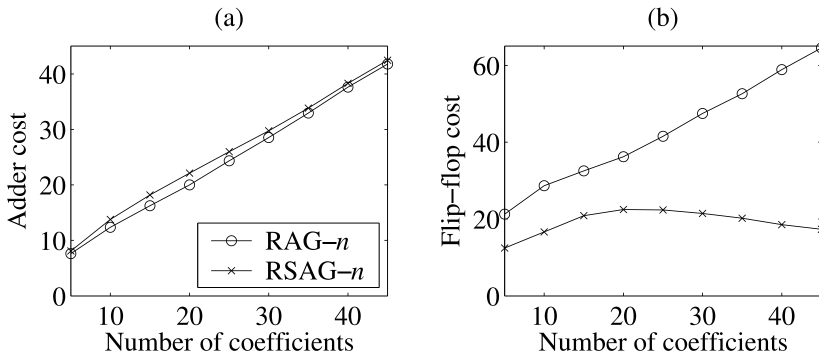


Figure 2.15 Average number of (a) adders and (b) shifts. Sets containing from 5 to 45 coefficients with a wordlength of 10 bits are used.

In Figs. 2.16 and 2.17, histograms of the savings in adder and flip-flop costs using sets of 10 and 40 coefficients, respectively, are shown. In Fig. 2.16 it can be seen that RAG-*n* has a higher adder cost in one out of 100 cases and a lower flip-flop cost in three out of 100 cases, compared

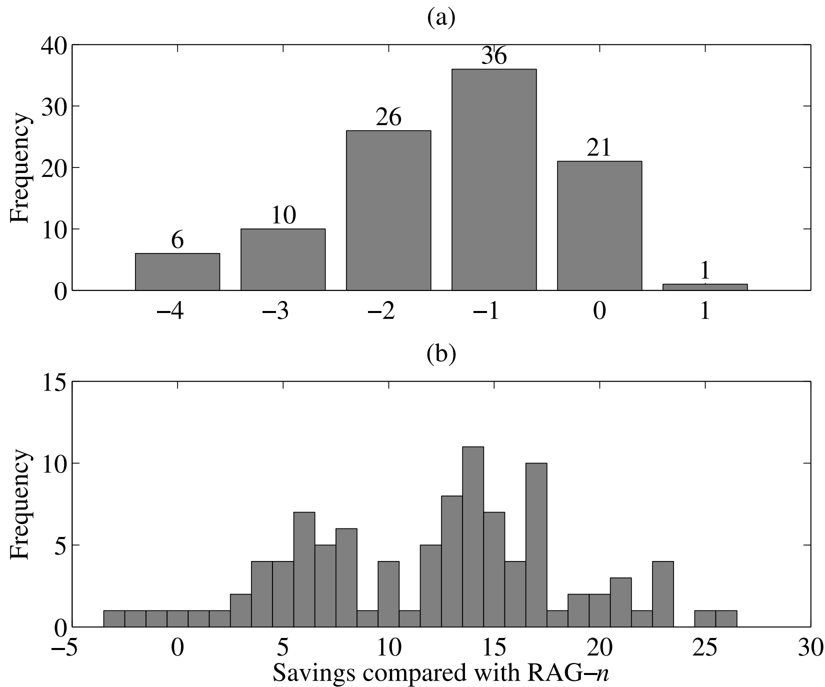


Figure 2.16 Frequency of savings in (a) adder and (b) flip-flop costs using RSAG-n compared with RAG-n. 100 sets of 10 coefficients with a wordlength of 10 bits are used.

with RSAG-n. The reason for these unexpected results is that both algorithms are greedy, i.e., they make decisions based on what seems to be best at the moment, without considering the future. For sets of 40 coefficients, the average adder cost for RSAG-n is only 0.67 higher than for RAG-n, while it is significantly lower for RAG-n when only 10 coefficients are used. The savings in shifts for RSAG-n compared to RAG-n are significantly larger for sets of 40 coefficients than for sets of 10 coefficients.

2.4 Digit-Size Trade-Offs

Implementation of FIR filters using digit-serial arithmetic has been studied in [50], [82], [105], [134], and [140]. For most cases, the focus has been on mapping the filters to field programmable gate arrays (FPGA). Furthermore, most of the work has considered generally programmable FIR filters. Digit-serial implementation of FIR filters using MCM algorithms has not been studied.

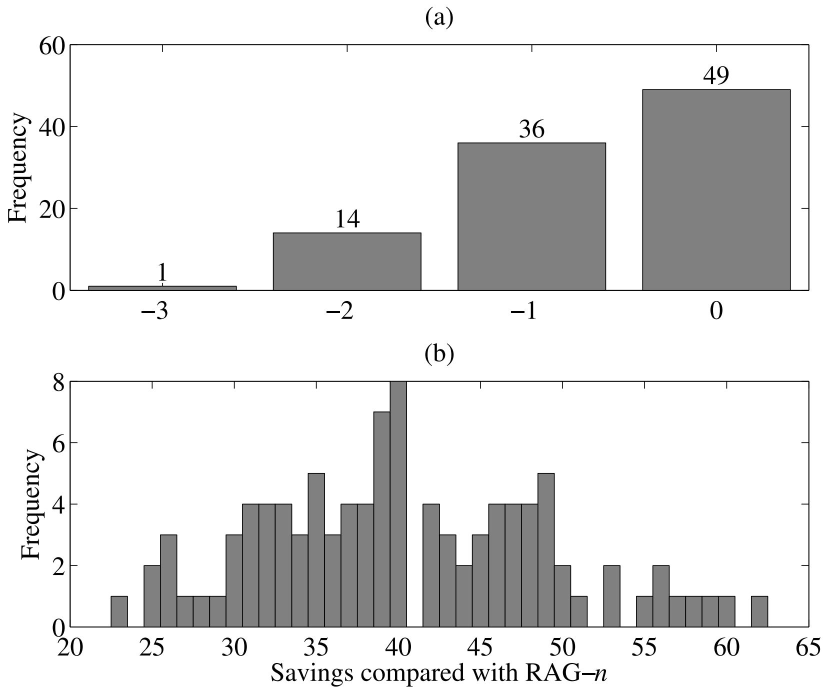


Figure 2.17 Frequency of savings in (a) adder and (b) flip-flop costs using RSAG-*n* compared with RAG-*n*. 100 sets of 40 coefficients with a wordlength of 10 bits are used.

In digit-serial adders, the number of full adders is proportional to the digit-size while exactly one flip-flop is used in digit-serial shifts independent of the digit-size, as can be seen in Fig. 1.16. Hence, the number of adders will have larger effects on the complexity compared to shifts for increasing digit-size.

In the remaining part of this section, the effect of digit-size on implementations of digit-serial transposed direct form FIR filters, as shown in Fig. 1.1 (b), using multiplier block techniques is studied. The two previously discussed MCM algorithms, i.e., RAG-*n* [25] and the modified version of this algorithm that drastically reduces the number of shifts, referred to as RSAG-*n* [56], are used in the comparison. Results, obtained by the use of an example filter, on area, sample rate, and energy consumption are presented. The focus is on the arithmetic parts of the FIR filter, i.e., the multiplier block and the structural adders, which refers to the adders that are not included in the dashed box in Fig. 1.1 (b).

2.4.1 Implementation Aspects

The transposed direct form FIR filter is mapped to a hardware structure using a direct mapping. The wordlength is here selected as an integer multiple of the digit-size, d. It is possible to use an arbitrary wordlength, but this requires a more complex structure of each processing element [113]. Furthermore, the partial results are not quantized, as this would lead to higher complexity of the processing elements. On the other hand, it may lead to delay elements with shorter wordlength.

Assuming an input data wordlength of W_0 bits and that the maximum number of fractional bits of the filter coefficients is W_f, the total wordlength, W_T, is

$$W_T = \left\lceil \frac{W_0 + W_f}{d} \right\rceil d \tag{2.6}$$

This leads to that in some cases, W_e extra bits are required, where

$$W_e = W_T - W_0 - W_f \tag{2.7}$$

These extra bits are used as guard bits to further reduce the risk of overflow. However, the filter coefficients are assumed to be properly scaled. The number of clock cycles between each input sample is W_T/d. Hence, the input word should be sign-extended with $W_T - W_0$ bits.

Delay Elements

Each delay element is implemented as W_T cascaded digit-serial shifts. This implies that a delay element will contain W_T flip-flops and have the structure illustrated in Fig. 2.18. For larger digit-size, d, the delay elements will have a more parallel structure, resulting in fewer switches per sample of the flip-flops. Therefore, the energy consumption for the delay elements is reduced with increasing digit-size.

With a different implementation of the delay elements, using interleaving of the flip-flops or RAMs, the energy consumption may be decreased. Furthermore, the digit-size effect on the energy consumption of the delay elements is likely to decrease since the total number of read and written bits then is independent of the digit-size.

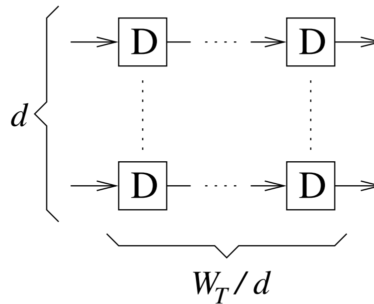


Figure 2.18 A digit-serial delay element realized using W_T flip-flops.

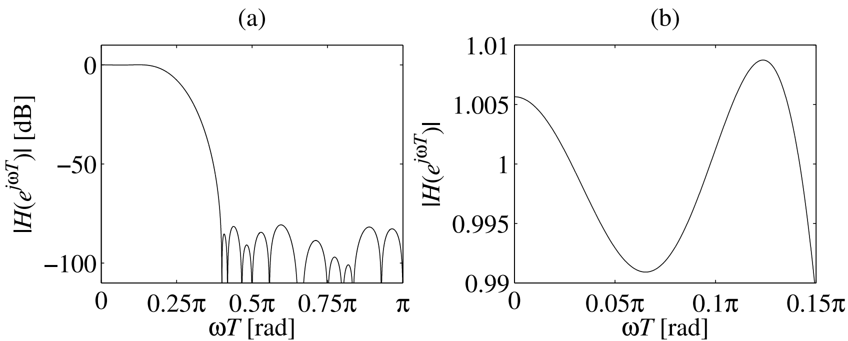


Figure 2.19 (a) Magnitude response for the example filter. (b) Passband.

2.4.2 Specification of the Example Filter

A 27th-order lowpass linear-phase FIR filter with passband edge 0.15π rad and stopband edge 0.4π rad is used for the comparison. The maximum passband ripple is 0.01, while the minimum stopband attenuation is 80 dB. The 28-tap filter has the symmetric coefficients $H = \{4, 18, 45, 73, 72, 6, -132, -286, -334, -139, 363, 1092, 1824, 2284, 2284, 1824, \ldots\}/2^{13}$. The coefficients have been optimized for a minimum number of signed-powers-of-two (SPT) terms. The magnitude response of the filter is shown in Fig. 2.19.

The input data wordlength, W_0, is selected to 16 bits. The number of fractional bits, W_f, of the coefficients is 13. Nine different values of the digit-size, $d = \{1, 2, 3, 4, 5, 6, 8, 10, 15\}$, are considered. The total wordlength, W_T, is computed for each digit-size from (2.6) as $W_T = \{29, 30, 30, 32, 30, 30, 32, 30, 30\}$.

Algorithm	n_{ADD}	n_{SH}
RAG-*n* [25]	12	30
RSAG-*n*	14	19
CSD	28	98

Table 2.2 Complexity of the multiplier block for the example filter.

The multiplier block is designed using the two MCM algorithms RAG-*n* [25] and RSAG-*n* [56]. For comparison, an approach using the CSD representation with serial/parallel multipliers [141],[149] is also included. Here, each multiplier is realized independent of other coefficients.

The required number of adders, n_{ADD}, and shifts, n_{SH} computed according to (1.21), for the different approaches is shown in Table 2.2. As expected, the RAG-*n* algorithm requires the lowest number of adders and is optimal for this coefficient set, while the RSAG-*n* algorithm requires the lowest number of shifts. There are 27 structural adders. Hence, the adders in the multiplier block can be expected to have a lower energy consumption compared with the adders in the delay section.

2.4.3 Chip Area

The chip area depends on the number of components. Since the only components used in a digit-serial FIR filter are full adders (*FA*), where one of the inputs may be inverted, and flip-flops (*FF*), the complexity can be described by simple expressions. The number of components in a multiplier block (*MB*) implemented with digit-size *d*, are computed as

$$\begin{aligned} n_{FA,\,MB} &= dn_{ADD} \\ n_{FF,\,MB} &= n_{ADD} + n_{SH} \end{aligned} \tag{2.8}$$

This implies that the number of full adders, $n_{FA,\ MB}$, increase with *d*, while the number of flip-flops, $n_{FF,\ MB}$, is constant and independent of *d*. For the total filter (*FIR*), the number of components are computed as

$$\begin{aligned} n_{FA,\,FIR} &= n_{FA,\,MB} + dN = d(n_{ADD} + N) \\ n_{FF,\,FIR} &= n_{FF,\,MB} + N + W_T N = n_{ADD} + n_{SH} + N(1 + W_T) \end{aligned} \tag{2.9}$$

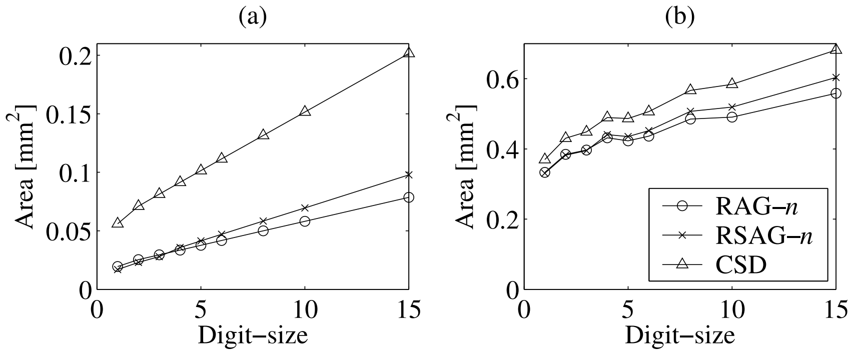


Figure 2.20 Chip area for (a) the multiplier block and (b) the total FIR filter.

where N is the filter order. From (2.9) it is clear that the area is more affected by the number of adders for a larger digit-size. Note that the delay elements and the structural adders are independent of the number of adders, n_{ADD}, and shifts, n_{SH}, in the multiplier block, i.e., the complexity of the delay section is the same for all algorithms. The control unit, which is implemented as a circular shift register, is also the same for all algorithms and was not considered here.

The FIR filter implementations are obtained by synthesis of VHDL code using a 0.35 μm standard cell library. In Fig. 2.20 (a), the area for the multiplier block reported by the synthesis tool is shown. RSAG-*n* has a smaller area than RAG-*n* for $d \leq 3$, i.e., for digit-sizes up to three. This is also true for the total FIR filter area, which is given in Fig. 2.20 (b). The fact that the total area is smaller for digit-size five than for digit-size four is explained by the difference in total wordlength, W_T, as discussed in Sections 2.4.1 and 2.4.2.

2.4.4 Sample Rate

In Fig. 2.21 (a), the maximum clock frequency, f_{clk}, reported by the synthesis tool is given. The clock frequency decrease for larger digit-size, as the critical path includes more and more full adders. For the CSD implementations there are at most d full adders in the critical path within the multiplier block, since no subsequent adders are connected without at least two shifts between them. This result in a higher maximum clock frequency compared to the other two algorithms. Furthermore, the maximum clock frequency is lower for RSAG-*n* than for RAG-*n*, since the adder depth is higher. This will be studied in detail in Section 2.6.1.

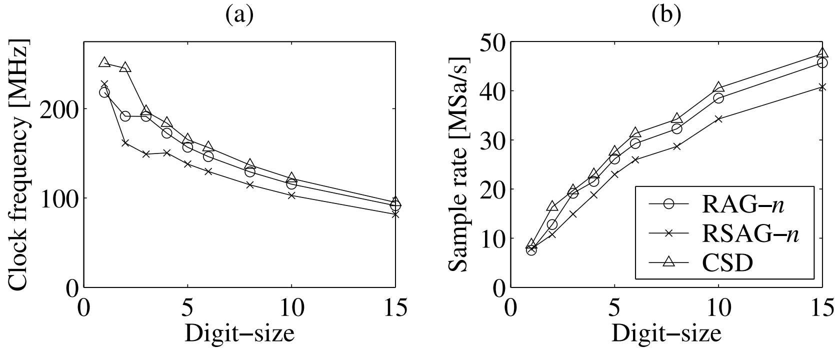


Figure 2.21 Maximum (a) clock and (b) sample frequency.

The maximum sample frequency, f_s, is shown in Fig. 2.21 (b), and can be computed as

$$f_s = \frac{f_{clk}}{W_T/d} \tag{2.10}$$

Ideally, the sample frequency should be constant, at least for the CSD case, since the critical path, t_{cp}, of the multiplier block increases linearly with the digit-size, which then would give

$$f_s = \frac{1/t_{cp}}{W_T/d} = \frac{1/dt_{FA}}{W_T/d} = \frac{1}{W_T t_{FA}} = \text{constant} \tag{2.11}$$

However, the set-up time, t_{su}, and the propagation delay, t_p, of the edge-triggered flip-flops must also be included according to [119]

$$t_p + t_{cp} + t_{su} \leq \frac{1}{f_{clk}} \tag{2.12}$$

Hence, the sample frequency increases for a larger digit-size as can be seen in Fig. 2.21 (b), since the overhead in delay caused by the flip-flops then is relatively smaller. Furthermore, the structural adders, which also give a relatively larger contribution to the delay for a small digit-size, must also be considered when studying the critical path.

2.4.5 Energy Consumption

As standard cell flip-flops are characterized by robustness, rather than low energy, the energy consumption will be dominated by the delay elements. Instead, low-power flip-flops [73],[153] or any of the different implementation strategies discussed in Section 2.4.1 should be used. Hence, the focus of the energy analysis is instead on the arithmetic parts.

The power consumption was obtained using NanoSim™ with 100 random input samples. The obtained energy per sample is shown in Fig. 2.22 for different parts of the design. The implementations here include a clock tree.

For the shifts in the multiplier block, the average energy per sample is shown in Fig. 2.22 (a). The large number of shifts used in the CSD implementation result in significantly higher energy consumption than for the other two algorithms. The energy consumed by the adders is given in Fig. 2.22 (b). For increasing digit-size, more glitches are generated since the carry-propagation paths in the adders become longer. Furthermore, the leakage increases with the digit-size, because more full adders are used. However, the number of added bits is constant, resulting in relatively small variations in energy consumption for different digit-size. Note that the adders in the multiplier block consume less energy for RAG-*n* than RSAG-*n*, while the situation is the opposite for the shifts.

By adding the energy for the adders and the shifts, the energy for the multiplier block is obtained, which is shown in Fig. 2.22 (c). There is a minimum for all three approaches; RSAG-*n* for $d = 3$, and both RAG-*n* and CSD for $d = 6$. For $d \leq 2$ the multiplier block consumes less energy using the RSAG-*n* algorithm, while RAG-*n* is preferable for a larger digit-size. The RSAG-*n* algorithm is aimed at reducing the number of shifts. Since the complexity of the shifts is constant, the gain is more prominent for a small digit-size, which is illustrated by Fig. 2.22 (d).

The structural adders, i.e., the adders in the delay section, consume the most energy for RSAG-*n*, as shown in Fig. 2.22 (e). This is because the outputs from the multiplier block are more likely to be directly connected from an adder without any intermediate shifts, compared to the RAG-*n* and CSD implementations. Thus, more glitches are propagated to the structural adders. Furthermore, the adder depth is larger for the RSAG-*n* algorithm, which result in that more glitches are generated. Hence, it is clear that not only the complexity is important, but also the glitch propagation.

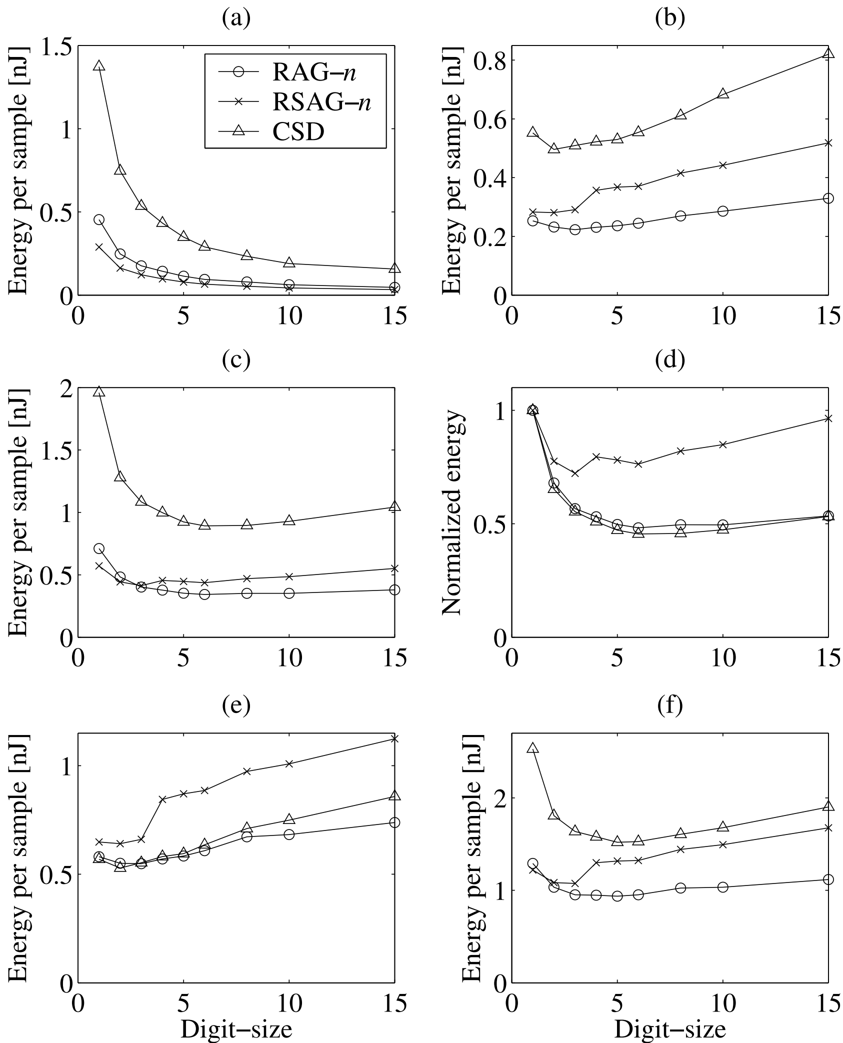


Figure 2.22 Consumed energy per sample for (a) shifts in the multiplier block, (b) adders in the multiplier block, (c) the total multiplier block, (d) the total multiplier block (normalized), (e) structural adders, and (f) all arithmetic parts.

Combining the energy consumption for the multiplier block and the structural adders result in the energy consumption for all arithmetic parts, which is illustrated in Fig. 2.22 (f). Again, an optimum digit-size in terms of minimum energy consumption is obtained. For RSAG-*n* it is still $d = 3$, while for RAG-*n* and CSD it has decreased to $d = 5$. Taking all arithmetic parts into account, the multiplier block designed using RSAG-*n* only has

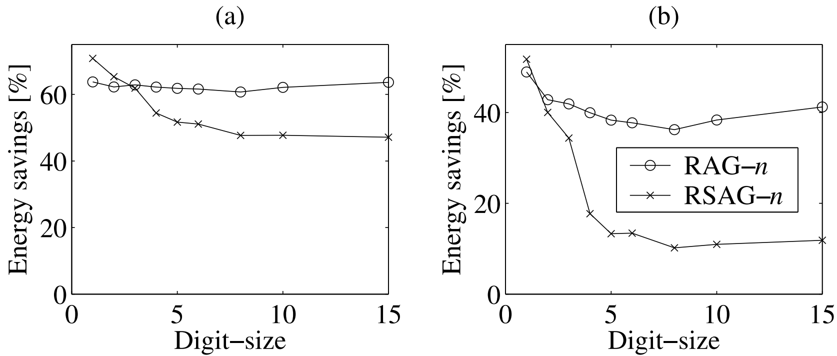


Figure 2.23 Energy savings for RSAG-n and RAG-n compared to CSD. (a) For the multiplier block and (b) for the total FIR filter (except the delay elements).

a lower energy consumption than the one designed by RAG-n for $d = 1$. Hence, because adders and shifts have a linearly increasing and constant complexity, respectively, the RSAG-n algorithm is in most practical cases probably only appropriate for bit-serial implementations.

The savings in energy for RSAG-n and RAG-n compared to CSD are illustrated in Fig. 2.23. From Fig. 2.23 (a) it is clear that the savings obtained in the multiplier block are significant, ranging from 47 to 71 percent for RSAG-n. The savings for RAG-n are in the range 60 to 64 percent. The energy savings for all arithmetic parts are shown in Fig. 2.23 (b). For RSAG-n the range of energy savings are 10 to 52 percent, while for RAG-n they are 36 to 49 percent.

2.5 Complexity Comparison – RASG-*n*

Considering the results in the previous section, it seems reasonable that an algorithm that firstly aim at minimizing the number of adders, while keeping the number of shifts low, should be preferable in most cases.

Here, a new algorithm that reduces the number of shifts while the number of adders is on average the same as for RAG-n [25] is presented. Hence, the total complexity is reduced for multiplier blocks implemented using serial arithmetic, where shift operations have a cost.

It is investigated how large savings that can be achieved compared with RAG-n and RSAG-n [56], respectively. The three algorithms are compared in terms of complexity, including both adders and shifts. Fur-

thermore, the adder depth is also studied. Average results are shown for 100 random coefficient sets.

2.5.1 The Reduced Adder and Shift Graph Algorithm

The new algorithm is a hybrid of the RAG-*n* [25] and RSAG-*n* [56] algorithms, and is referred to as the *n*-dimensional Reduced Adder and Shift Graph (RASG-*n*) algorithm [62]. Since the RSAG-*n* algorithm was described in detail in Section 2.3.1, only the main differences will be discussed here.

The RASG-*n* algorithm works with odd integer coefficients, like RAG-*n*. Hence, even coefficients are divided by two until odd at start, as suggested in Section 1.4.4. RASG-*n* store the number of times each coefficient is divided. These shifts at the outputs can be considered to be free when other coefficients are realized. RASG-*n* only adds one coefficient in each iteration, like RSAG-*n*. When it is possible to realize more than one coefficient, RASG-*n* selects the one that require the lowest number of additional shifts. This makes it possible for RASG-*n* to minimize both the number of adders and shifts in an efficient way, as will be shown in the following.

2.5.2 Comparison by Varying the Wordlength

The different algorithms are here used to design multiplier blocks with coefficient sets of varying wordlength. The setsize is fixed to 25 coefficients.

The average number of additional adders for each realized coefficient using the RASG-*n* algorithm is shown in Fig. 2.24 (a). Coefficients that can be realized with no adders include zeros, power-of-two's, and repeated coefficients. Naturally, all these types of coefficients are less common for larger wordlengths. Most coefficients can be realized with only one additional adder, corresponding to step 3 of the RSAG-*n* algorithm presented in Section 2.3.1. As long as only one adder is required for each coefficient, the adder cost is assured to be optimal. This is true for all coefficient sets of wordlengths up to 8 bits, as illustrated in Fig. 2.24 (b). However, when larger coefficients are included in the sets, it is more likely that some coefficients require two or more additional adders, as can be seen in Fig. 2.24 (a), and the optimality then becomes unknown. Corresponding statistics for the other two algorithms would look similar.

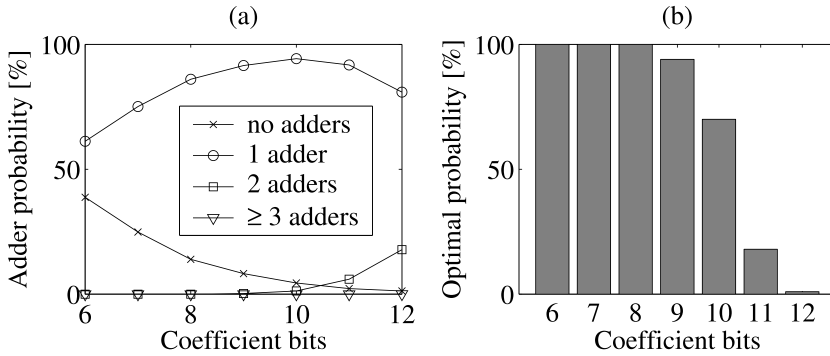


Figure 2.24 Statistics from realization of multiplier blocks for sets of 25 coefficients using the RASG-n algorithm. (a) Average number of additional adders for each coefficient. (b) The probability that the total number of adders for a set is guaranteed to be optimal.

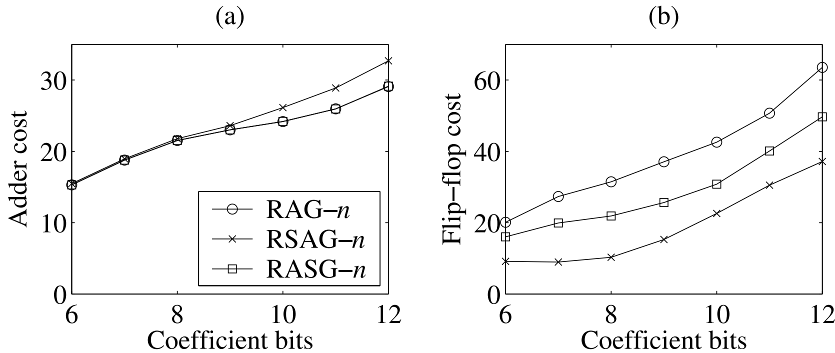


Figure 2.25 Average number of (a) adders and (b) shifts for sets of 25 coefficients. Wordlengths from 6 to 12 bits are used.

In Fig. 2.25 (a), the average number of adders for the three algorithms is shown. It is clear that the number of adders is higher for RSAG-n. The average number of shifts is lower for RASG-n than for RAG-n, while RSAG-n has the lowest number of shifts, as illustrated in Fig. 2.25 (b). The same coefficient sets as in Section 2.3 were also used here, and, hence, the lines for RAG-n and RSAG-n are identical to the ones in Fig. 2.12.

The required number of adders using 12 bit coefficients is illustrated by the histogram in Fig. 2.26 (a). The RASG-n and RAG-n algorithms have a different number of adders in 55 out of the 100 cases. However, the average adder costs are almost the same; 29.08 for RAG-n and 29.15 for RASG-n. From Fig. 2.26 (b) it can be seen that RASG-n have on average

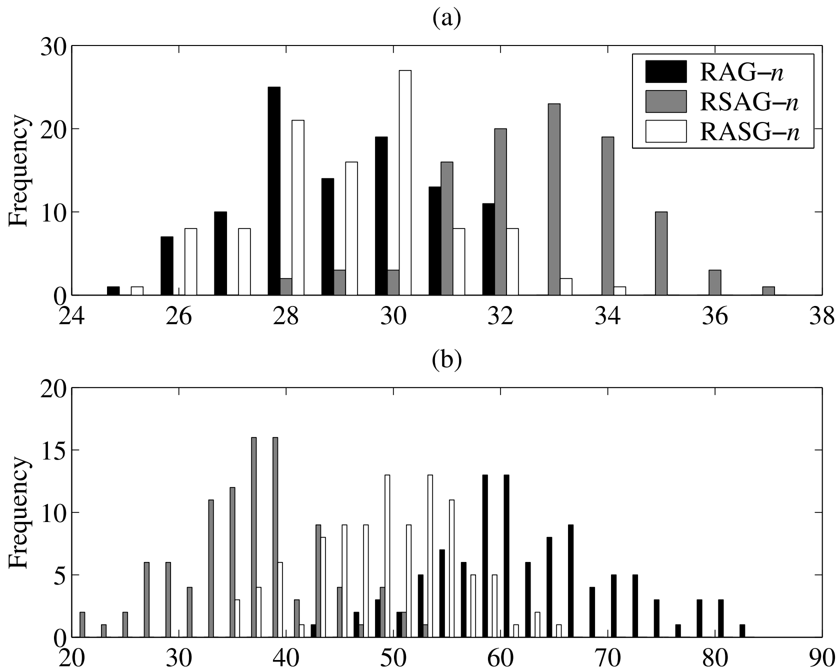


Figure 2.26 Frequency of the number of (a) adders and (b) shifts for the different algorithms using sets of 25 coefficients of wordlength 12.

13.85 shifts less than RAG-*n*. As expected, RSAG-*n* has the highest number of adders and the lowest number of shifts.

2.5.3 Comparison by Varying the Setsize

With the coefficient wordlength fixed to 10 bits, the different algorithms are here used to design multiplier blocks of varying setsize.

The average number of additional adders is shown in Fig. 2.27 (a) for the RASG-*n* algorithm. For a small setsize, many of the coefficients will require two additional adders, which result in a low probability of guaranteed optimality, as illustrated in Fig. 2.27 (b). For a large setsize, most coefficients can be realized with only one additional adder, and, hence, the probability that the total number of adders is optimal becomes high. The algorithm performs better for sets containing many coefficients since it is then more likely to find required coefficients in the sum matrix, which is used in step 3 of the MCM algorithm described in Section 2.3.1.

In Fig. 2.28 (a), the average number of adders for the three algorithms are shown (the results for RAG-*n* and RSAG-*n* are the same as in

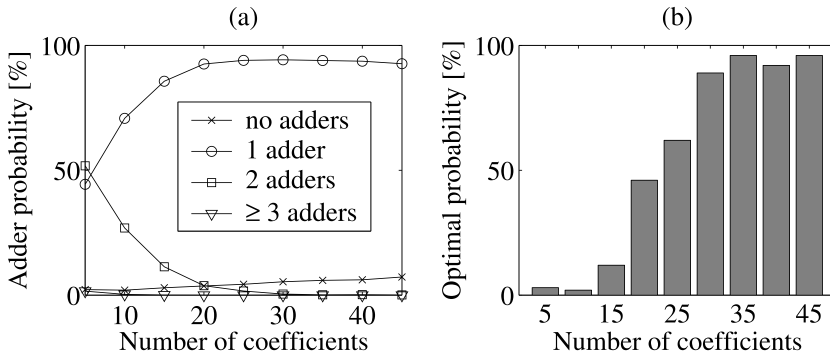


Figure 2.27 Statistics from realization of multiplier blocks for 10 bit coefficients using the RASG-*n* algorithm. (a) Average number of additional adders for each coefficient. (b) The probability that the total number of adders for a set is guaranteed to be optimal.

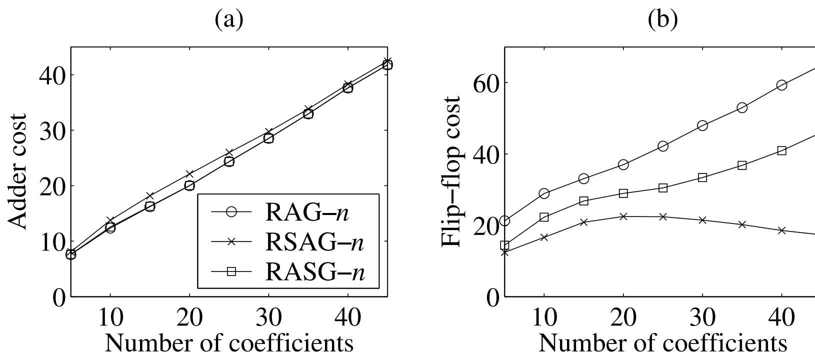


Figure 2.28 Average number of (a) adders and (b) shifts for 10 bit coefficients. Sets of 5 to 45 coefficients are used.

Fig. 2.15). Again, the number of adders for RAG-*n* and RASG-*n* are similar. All algorithms are likely to have an optimal number of adders for a large setsize, and the difference is naturally reduced when a small setsize is used. Hence, the difference between RSAG-*n* and the other two algorithms has a maximum, which occur for a setsize of 20 coefficients.

The difference in number of shifts is increasing for a larger setsize, as illustrated in Fig. 2.28 (b). The RSAG-*n* algorithm takes full advantage of the fact that coefficients may be realized with fewer additional shifts when more values are available, and of course has the lowest number of shifts. The average number of shifts is lower for RASG-*n* than for RAG-*n*.

In Fig. 2.29 (a), a histogram for the required number of adders using sets of five coefficients is shown. RASG-*n* and RAG-*n* have the same

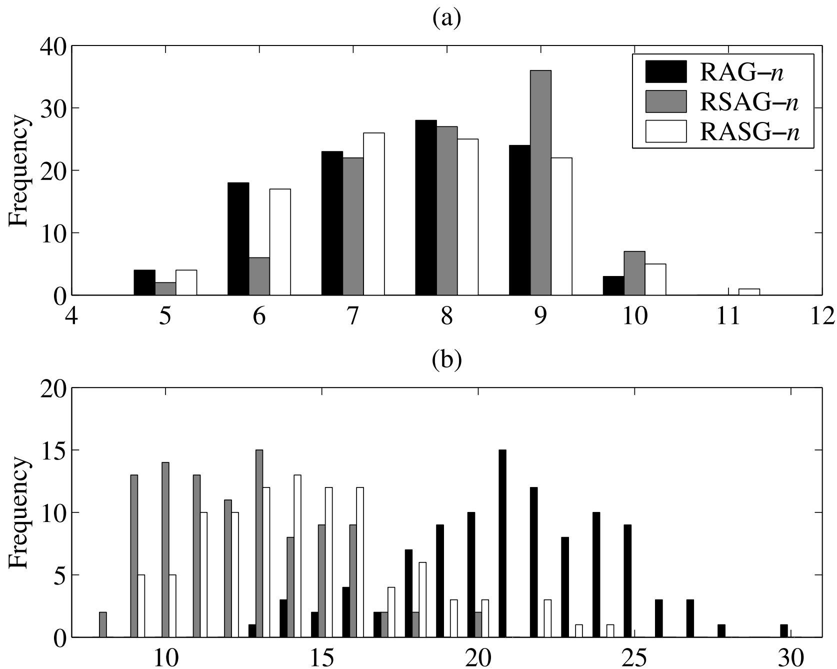


Figure 2.29 Frequency of the number of (a) adders and (b) shifts for the different algorithms using sets of 5 coefficients of wordlength 10.

number of adders in 70 out of the 100 cases. For the remaining 30 cases the differences evens out, resulting in almost the same number of adders on average. As can be seen in Fig. 2.29 (b), RASG-*n* have on average almost seven shifts less than RAG-*n*.

In Fig. 2.30, sets of 40 coefficients have been used. RASG-*n* and RAG-*n* have the same number of adders in all 100 cases. Furthermore, RASG-*n* has on average almost 18 shifts less than RAG-*n*.

2.5.4 Adder Depth

In [21] and [22], methods to predict the number of transitions in multiplier blocks were introduced. These methods are based on the fact that high adder depth results in more transitions, and consequently increased energy consumption. Here, the RAG-*n* algorithm is compared to the two previously presented algorithms, which are suited for serial arithmetic, in terms of adder depth.

The characteristics for the three algorithms when varying the coefficient wordlength, considering average and maximum adder depth are

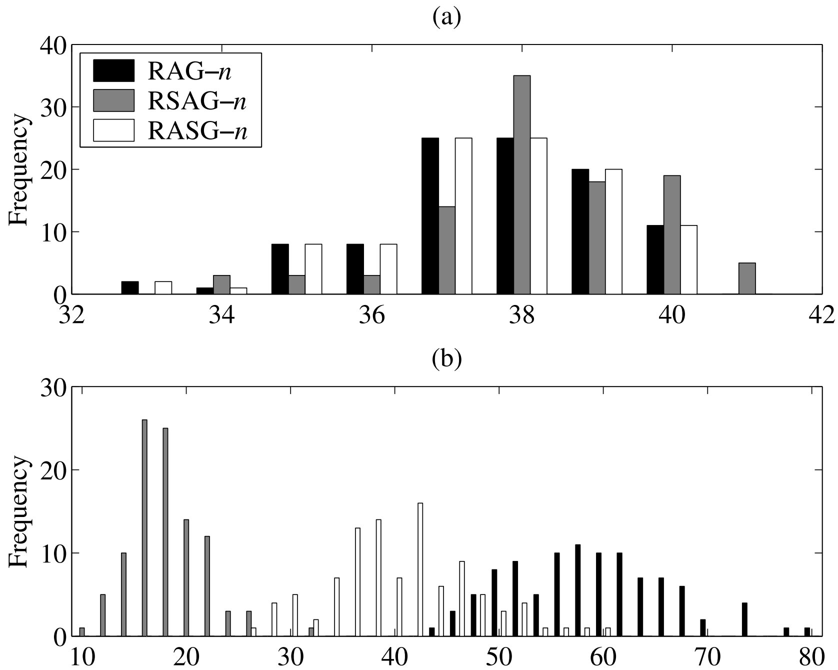


Figure 2.30 Frequency of the number of (a) adders and (b) shifts for the different algorithms using sets of 40 coefficients of wordlength 10.

shown in Figs. 2.31 (a) and (b), respectively. The same coefficient sets as before are used. It is clear that RAG-*n* has the lowest adder depth among the three algorithms. However, for larger wordlengths the difference becomes smaller. For comparison, included are also the results obtained when all coefficients are implemented separately using the CSD representation and are realized in the structure of a binary adder tree, which according to (1.20) gives a lower bound (LB) [42],[45]. It can be seen that RAG-*n* is close to the lower bound for small coefficients. Note the similarity between the curves for average and maximum adder depth for all algorithms.

The adder depth decrease for larger coefficient sets for the RAG-*n* algorithm, as can be seen in Figs. 2.31 (c) and (d). This is not surprising because, as was mentioned in Section 2.3.3, more coefficients give more flexibility. In the original algorithm, this flexibility is used to obtain a low adder depth, while it is used to reduce the number of shifts in the two serial algorithms. Hence, the RAG-*n* algorithm approaches the lower bound for large coefficient sets.

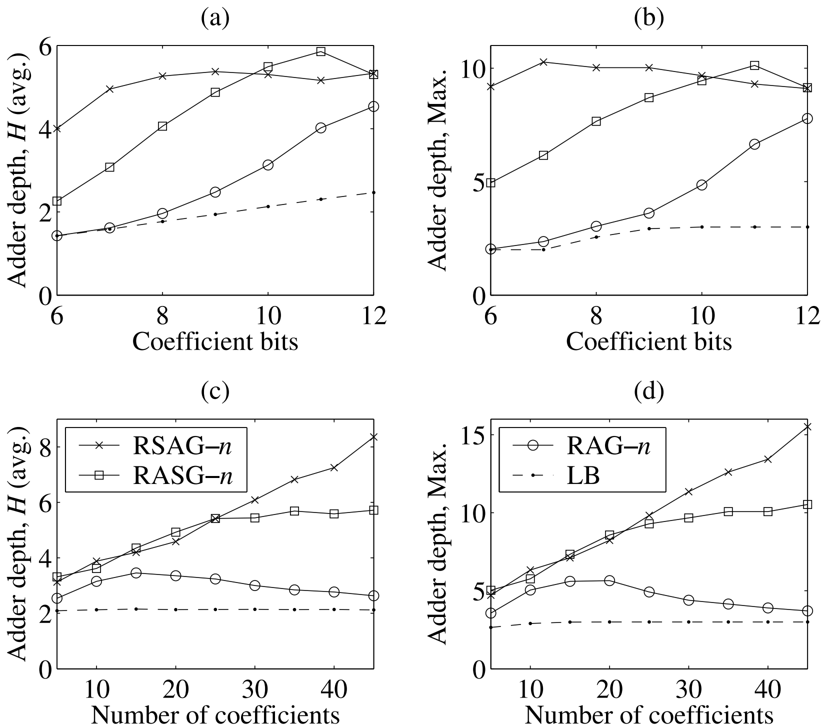


Figure 2.31 Average and maximum adder depth. (a), (b) Sets of 25 coefficients and wordlengths from 6 to 12 bits. (c), (d) 10 bit coefficients and sets of 5 to 45 coefficients.

2.6 Implementation Examples

The energy consumption is here studied by the use of two example filters, implemented by logic synthesis of VHDL code using a 0.35 µm CMOS standard cell library. The power consumption results are obtained using NanoSim™ with 100 random input samples. The filters are implemented using the transposed direct form structure shown in Fig. 1.1 (b). In the same way as in Section 2.4.5, only the arithmetic parts are considered. Furthermore, the relation between adder depth and energy consumption is extensively discussed.

2.6.1 Example 1

Again, the 27th-order FIR filter defined in Section 2.4.2 is used in the first evaluation. In Fig. 2.32, the multiplier block realizations for the three different algorithms are given. The realizations are illustrated using the graph representation discussed in Section 1.4.3. If all integer coefficients are divided by two until odd and the absolute value is taken, the set will be {1, 9, 45, 73, 9, 3, 33, 143, 167, 139, 363, 273, 57, 571, 571, 57, ...}, which is exactly what is realized by RAG-*n* and RASG-*n*. The fact that both 3 and 6 are included in the realization shown in Fig. 2.32 (b) originate from the greedy property of the algorithm, when first 6 is added to the realization it can not be removed when it later is found that 3 is also required. Note that this behavior will be avoided if the interconnection is constructed after that all fundamentals have been found, according to the design flow illustrated in Fig. 1.13.

The required number of adders and shifts for four different algorithms are given in Table 2.3, where the RASG-*n* and Pasko [115] algorithms are added compared to Table 2.2. The Pasko algorithm is based on subexpression sharing, and normally achieves a relatively low adder depth. The RAG-*n* and RASG-*n* algorithms require 12 adders, which is optimal for this coefficient set. The number of shifts has been divided into an internal part, corresponding to shifts within the multiplier block, and an external part, referring to additional shifts between the multiplier block and the structural adders. The number of internal and output shifts, as stated in Table 2.3, can be found from the realizations. For example, for the RSAG-*n* realization, shown in Fig. 2.32 (b), all coefficients except 1824 are obtained directly without any output shifts. Hence, only one output shift is required to obtain the coefficient 1824 from 912. Note that not all coefficients, for example 72, are explicitly shown in Fig. 2.32 (b). However, 72 is obtained by shifting of 18 and these shifts are already included in the shift-and-add network. Hence, no extra shifts are required at the output of node 18. This way to obtain free shifts at the outputs also explains the difference in external shifts given in Table 2.3 for the RAG-*n* and RASG-*n* algorithms, although realizing the same fundamental set. Since the even coefficient values are maintained in the RSAG-*n* design algorithm, few external shifts are required.

The Pasko algorithm has the largest number of both adders and shifts. Hence, the area for the implementation of the corresponding multiplier block will be large, as can be seen in Fig. 2.33 (a). For a digit-size up to

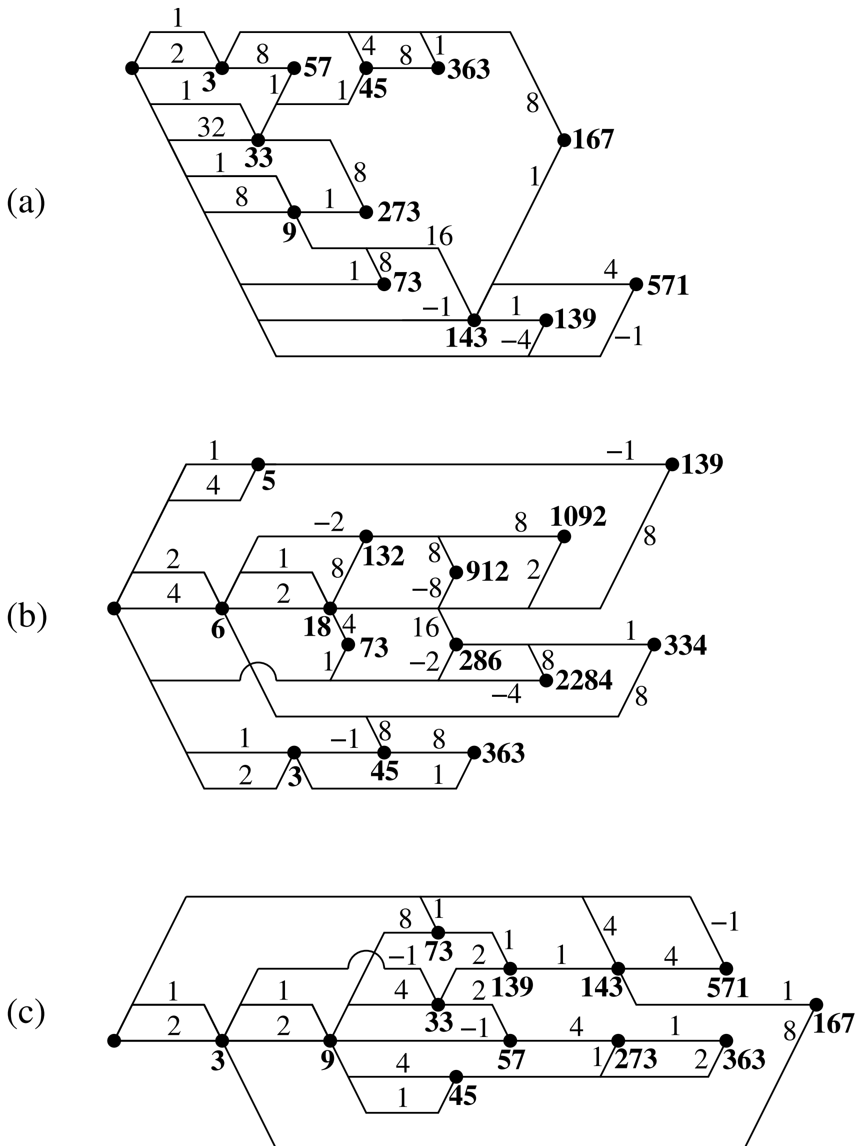


Figure 2.32 Multiplier block realizations using the (a) RAG-n, (b) RSAG-n, and (c) RASG-n algorithm. Fundamental values are in bold.

three, i.e., $d \leq 3$, the smallest area is obtained for the RASG-n algorithm, while RAG-n is the best for a larger digit-size.

Results for the adder depth, i.e., the number of cascaded adders, are given in Table 2.4. Average depths are given both for the nodes in the fundamental set F, i.e., all the adders in the multiplier block, and for the nodes in the coefficient set H, i.e., the 28 taps that are connected to the

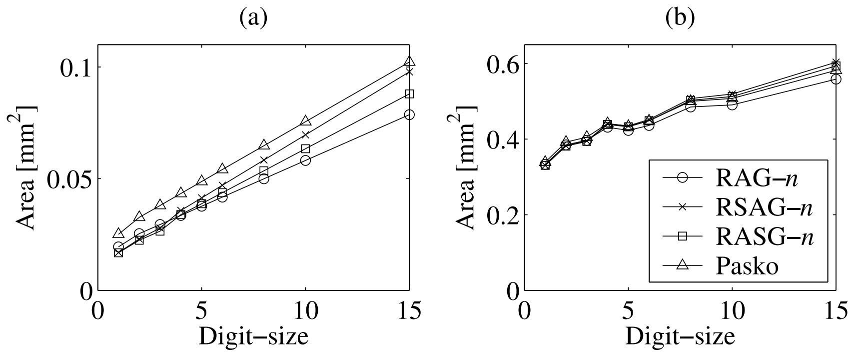


Figure 2.33 Chip area for (a) the multiplier block and (b) the total FIR filter.

Algorithm	Adders	Shifts		
		Internal	External	Total
RAG-*n* [25]	12	20	10	30
RSAG-*n*	14	18	1	19
RASG-*n*	12	14	9	23
Pasko [115]	15	27	12	39

Table 2.3 Complexity results for the MCM blocks in Example 1.

Algorithm	Adder depth			Directly cascaded adders		
	F (avg.)	H (avg.)	Max.	F (avg.)	H (avg.)	Max.
RAG-*n* [25]	2.0833	1.8571	3	1.5000	1.3571	2
RSAG-*n*	2.7143	2.7143	4	1.3571	1.3571	2
RASG-*n*	4.0000	3.5714	6	2.5833	2.3571	5
Pasko [115]	1.9333	1.8571	3	1.4667	1.3571	3

Table 2.4 Critical path measures for the MCM blocks in Example 1.

structural adders. It is clear that the adder depth is lower for RAG-*n* and Pasko than for the serial algorithms.

As long as there is at least one shift between cascaded adders, the number of full adders in the critical path will not exceed the digit-size, d. Furthermore, for bit-serial implementations there will be no direct paths

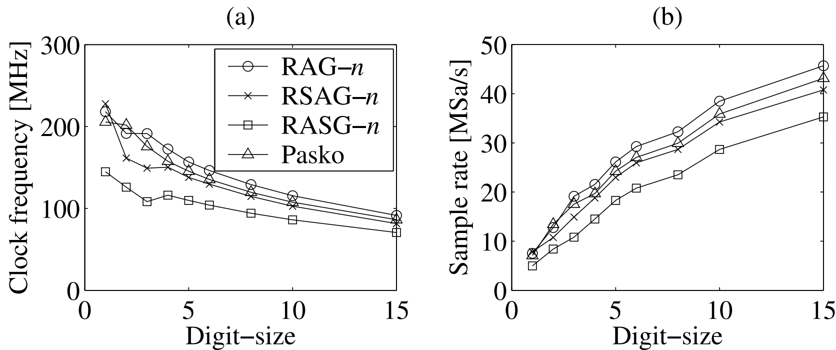


Figure 2.34 Maximum (a) clock and (b) sample frequency.

at all between adders, i.e., the critical path will be a single full adder. Hence, it is of interest to study the number of cascaded adders without any intermediate shifts, which is given in Table 2.4 in the same format as the adder depth. Here, RSAG-*n* has an advantage due to the fact that for five adders at depth two, or higher, there are shifts at both input edges, which then only gives the same contribution as a depth one adder. There are also some similar cases present in the other realizations, for example, the node with the fundamental value 571 for RAG-*n* in Fig. 2.32 (a) where one edge has shifts and the other edge is connected to the multiplier block input. Note that the input of the multiplier block is assumed to be glitch free, since an input register is normally used.

The maximum clock frequency and corresponding maximum sample frequency are shown in Fig. 2.34. The benefit of few cascaded adders in RSAG-*n* is clear for the bit-serial implementation. The slowest implementations are the ones based on the RASG-*n* algorithm, which is not surprising considering the results in Table 2.4.

Energy Consumption

Although the energy consumption is also related to the results given in Tables 2.3 and 2.4, it is more difficult to predict than area and sample rate. The results are given in Fig. 2.35, where the energy for the RAG-*n* and RSAG-*n* algorithms are the same as previously given in Fig. 2.22. As can be seen in Fig. 2.35 (a), the energy per sample for the shifts in the multiplier block is lowest for RSAG-*n* and highest for the Pasko algorithm, which agrees with the complexity stated in Table 2.3. The energy per sample for the adders in the multiplier block is shown in Fig. 2.35 (b). RAG-*n* consumes less energy for any digit-size. By adding the energy for

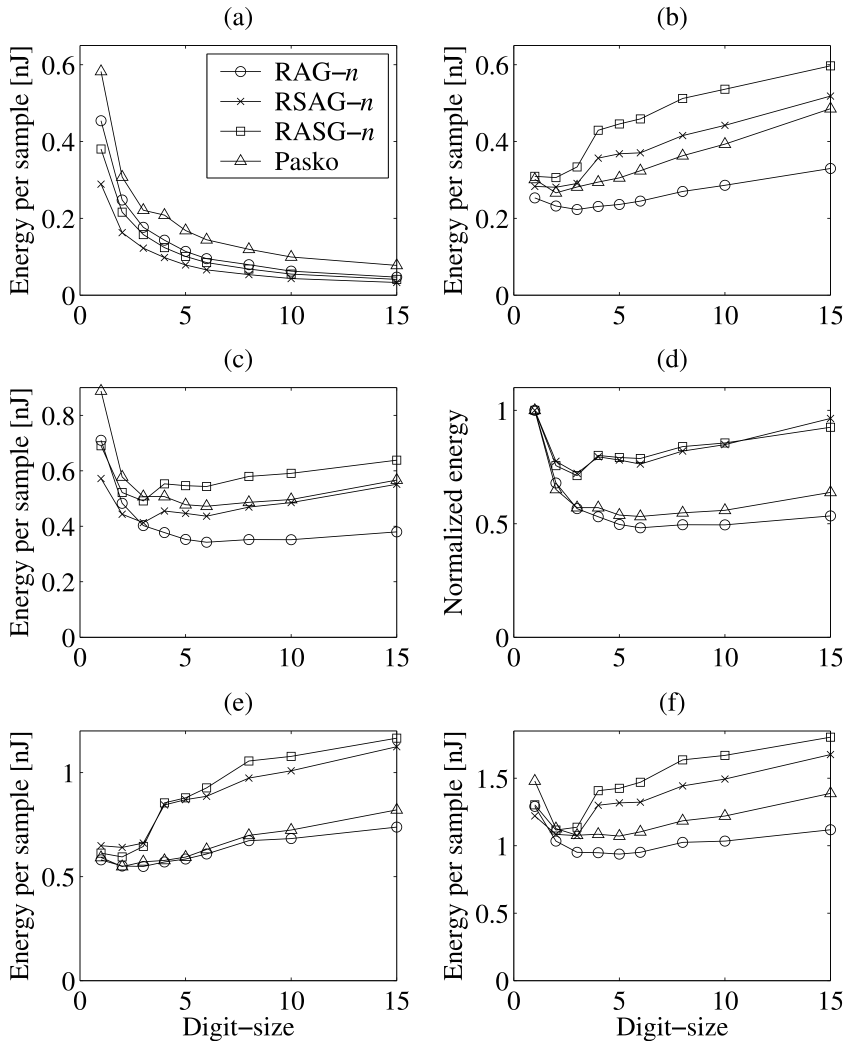


Figure 2.35 Consumed energy per sample for (a) shifts in the multiplier block, (b) adders in the multiplier block, (c) the total multiplier block, (d) the total multiplier block (normalized), (e) structural adders, and (f) all arithmetic parts.

the shifts and the adders, the energy for the total multiplier block is obtained, which is illustrated in Fig. 2.35 (c). RSAG-*n* consumes the least energy for digit-sizes one and two, while RAG-*n* is the best for a larger digit-size. Note that the energy consumption corresponding to shifts and adders dominates for small and large values of the digit-size, respectively. In Fig. 2.35 (d), the normalized energy per sample is shown. From this it

can be seen that three is the optimal digit-size for the serial algorithms, while the other two algorithms have a minimum at six.

The energy per sample consumed for the structural adders and for all arithmetic operations are shown in Figs. 2.35 (e) and (f), respectively. The energy for the structural adders is only affected by the switching activity at the connections to the multiplier block. For a large digit-size, it is clear that more glitches are received for the RASG-*n* and RSAG-*n* implementations. For RSAG-*n*, the reason is that there are few external shifts, which would provide glitch reduction between the multiplier block and the structural adders. For RASG-*n*, the large number of glitches propagated to the structural adders is due to high adder depth in the multiplier block.

A surprising result is that the energy consumed by the multiplier block adders is larger for RASG-*n* than RSAG-*n*, as can be seen in Fig. 2.35 (b), although the adder cost is lower. The reason for this will be discussed in the following.

In Fig. 2.36, the adder depth is illustrated for each coefficient in the implementations of the example filter using the three different realizations in Fig. 2.32. For example, the longest path from the input to the node corresponding to coefficient 363 only passes three nodes for the realizations in Figs. 2.32 (a) and (b), but as many as six in (c). As also stated by Table 2.4, it is clear that RASG-*n* has a larger adder depth than RSAG-*n*, which explains the higher energy consumption.

The fact that adder depth is highly correlated with energy consumption is established when the energy consumed in each adder is investigated. This is shown in Figs. 2.37 (a) and (b) for digit-size one and five, respectively. Note that the RSAG-*n* implementation includes two extra adders, i.e., the total energy is larger than illustrated by Fig. 2.37.

The energy consumption also depends on other factors. Consider, for example, the coefficients 363 and 2284 ($4 \cdot 571$). Although both nodes have adder depth six in the RASG-*n* realization, the adder that generates the output corresponding to the coefficient 363 consumes more than two times as much energy for the implementation with digit-size five. The explanation can be found in Fig. 2.32 (c). One of the inputs to the 571 adder is directly connected to the input, i.e., it is glitch free, and the glitches at the other input are reduced by two shifts. For the 363 adder, there is a path from the input through all adders in the critical path without any shifts at all, i.e., generated glitches are propagated without any reduction. Hence, to use complexity and adder depth as cost measures is not enough to develop an MCM algorithm optimized for low energy consumption

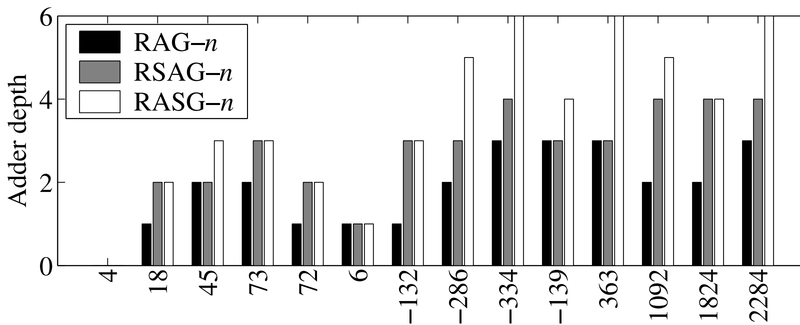


Figure 2.36 Adder depth for each coefficient in the multiplier block realizations of Example 1 using three different algorithms.

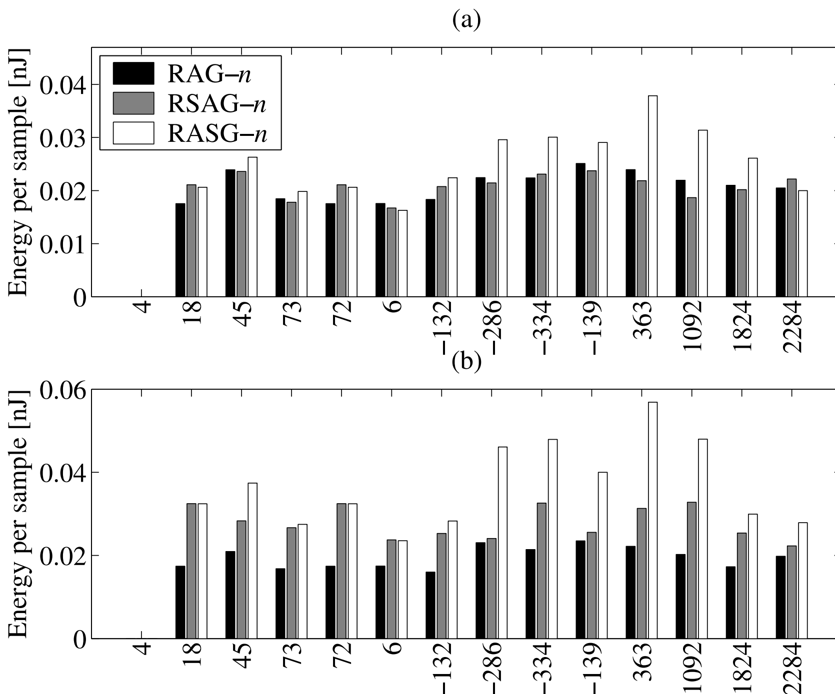


Figure 2.37 Energy per sample for the adders corresponding to each coefficient of the example filter. (a) Digit-size one and (b) digit-size five.

implementations when using serial arithmetic. Furthermore, the number of directly cascaded adders, as stated in Table 2.4, might also be a part of the solution.

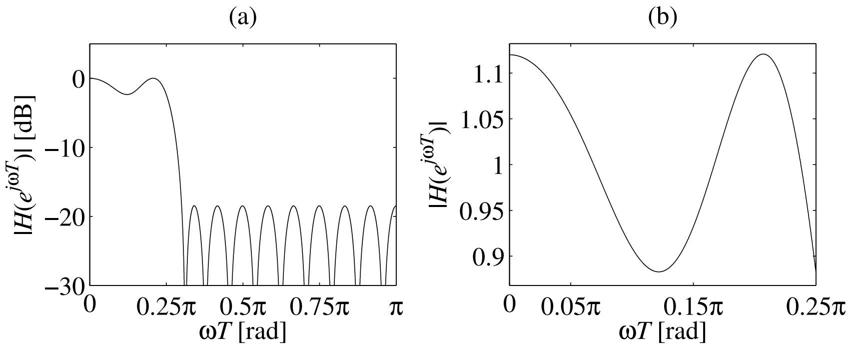


Figure 2.38 (a) Magnitude response for the filter from [26]. (b) Passband.

2.6.2 Example 2

Since adder depth was shown to highly affect the energy consumption, an example, where a large difference in adder depth can be expected for various algorithms, is considered. The three previously studied algorithms are here compared to the C1 algorithm [26], which aims at minimizing the adder depth. For the simple coefficient set used in Example 1, the C1 algorithm gives exactly the same solution to the MCM problem as RAG-*n*, and was therefore not included.

The 24th-order linear-phase FIR filter used for the example in [26] is considered. The filter has symmetric coefficients, $H = \{-710, 327, 505, 582, 398, -35, -499, -662, -266, 699, 1943, 2987, 3395, 2987, ...\}/2^{14}$. The magnitude response is shown in Fig. 2.38.

The input data wordlength, W_0, is 20 bits and the considered digit-sizes are $d = \{1, 2, 3, 4, 5, 6, 7, 9, 12, 17\}$. Furthermore, the different total wordlengths, as computed according to (2.6), are $W_T = \{34, 34, 36, 36, 35, 36, 35, 36, 36, 34\}$.

The obtained number of adders and shifts for the different algorithms are presented in Table 2.5. As expected, RAG-*n* has the lowest number of adders and RSAG-*n* has the lowest number of shifts.

The average and maximum adder depths are given in Table 2.6 and illustrated in Fig. 2.39 for each filter coefficient. The C1 algorithm has a significantly lower adder depth than any of the other three algorithms. Naturally, the number of cascaded adders without intermediate shifts is also low for C1, as can be seen in Table 2.6. Furthermore, the RSAG-*n* algorithm has relatively few directly connected adders, which also was noted for the first example filter.

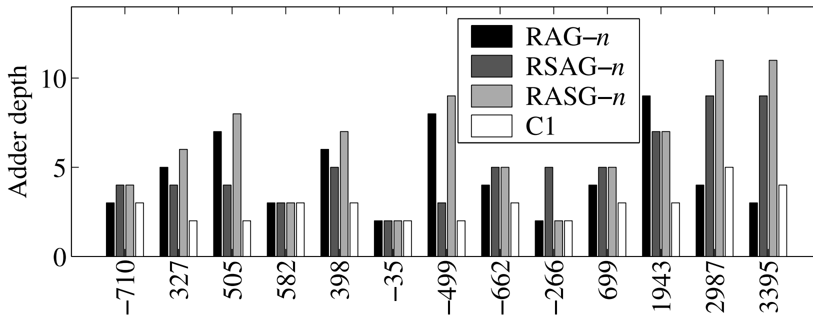


Figure 2.39 Adder depth for each coefficient in the multiplier block realizations of Example 2 using four different algorithms.

Algorithm	Adders	Shifts		
		Internal	External	Total
RAG-*n* [25]	17	31	4	35
RSAG-*n*	20	18	0	18
RASG-*n*	19	18	1	19
C1 [26]	19	28	5	33

Table 2.5 Complexity results for the MCM blocks in Example 2.

Algorithm	Adder depth			Directly cascaded adders		
	F (avg.)	*H* (avg.)	Max.	*F* (avg.)	*H* (avg.)	Max.
RAG-*n* [25]	3.8824	4.6800	9	3.0588	3.6000	6
RSAG-*n*	4.7500	4.8400	9	2.9500	3.0400	6
RASG-*n*	5.7895	5.9600	11	4.0000	4.3600	7
C1 [26]	2.3158	2.8000	5	2.2632	2.7200	5

Table 2.6 Critical path measures for the MCM blocks in Example 2.

The synthesis results are reported in Figs. 2.40 and 2.41. The implementations based on C1 require the largest area for any digit-size. The RASG-*n* algorithm is preferable for a small digit-size, while RAG-*n* has the smallest area for a digit-size $d \geq 7$. This is true for both the multiplier block and the total FIR filter. When considering the maximum sample

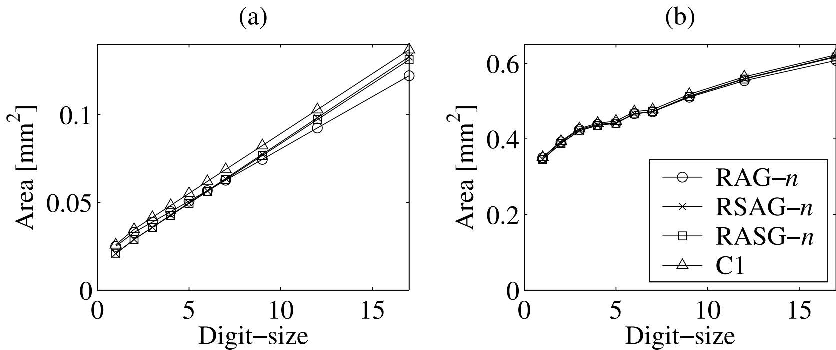


Figure 2.40 Chip area for (a) the multiplier block and (b) the total FIR filter.

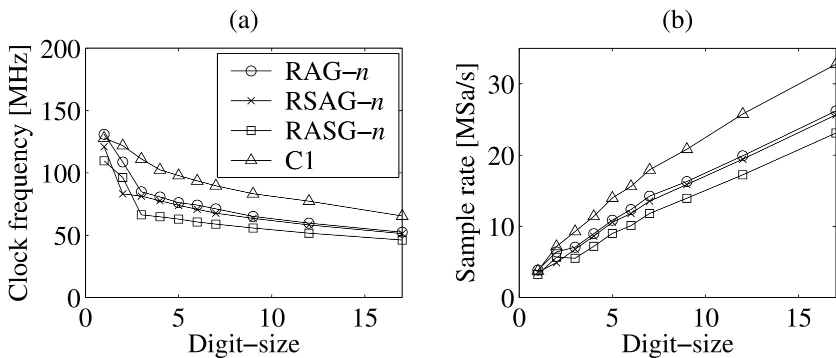


Figure 2.41 Maximum (a) clock and (b) sample frequency.

rate, it is clear that C1 is the fastest and RASG-*n* is the slowest. This agrees well with the figures given in Table 2.6.

Energy Consumption

The energy consumption for shifts and adders are given in Figs. 2.42 (a) and (b), respectively. As expected, the serial algorithms have an advantage for the shifts, while the adders have a lower consumption when using C1 and RAG-*n*. For bit-serial arithmetic the multiplier block consumes the least energy using the RSAG-*n* algorithm, as shown in Fig. 2.42 (c). For a digit-size larger than one, C1 performs better than the other algorithms. Furthermore, the structural adders consume less energy using C1 for any digit-size, which is illustrated in Fig. 2.42 (e). In Fig. 2.42 (f), all arithmetic parts are considered. It can be seen that the RASG-*n* and RSAG-*n* algorithms both have a minimum for digit-size two. RAG-*n* and C1 also have a minimum, but for $d = 4$.

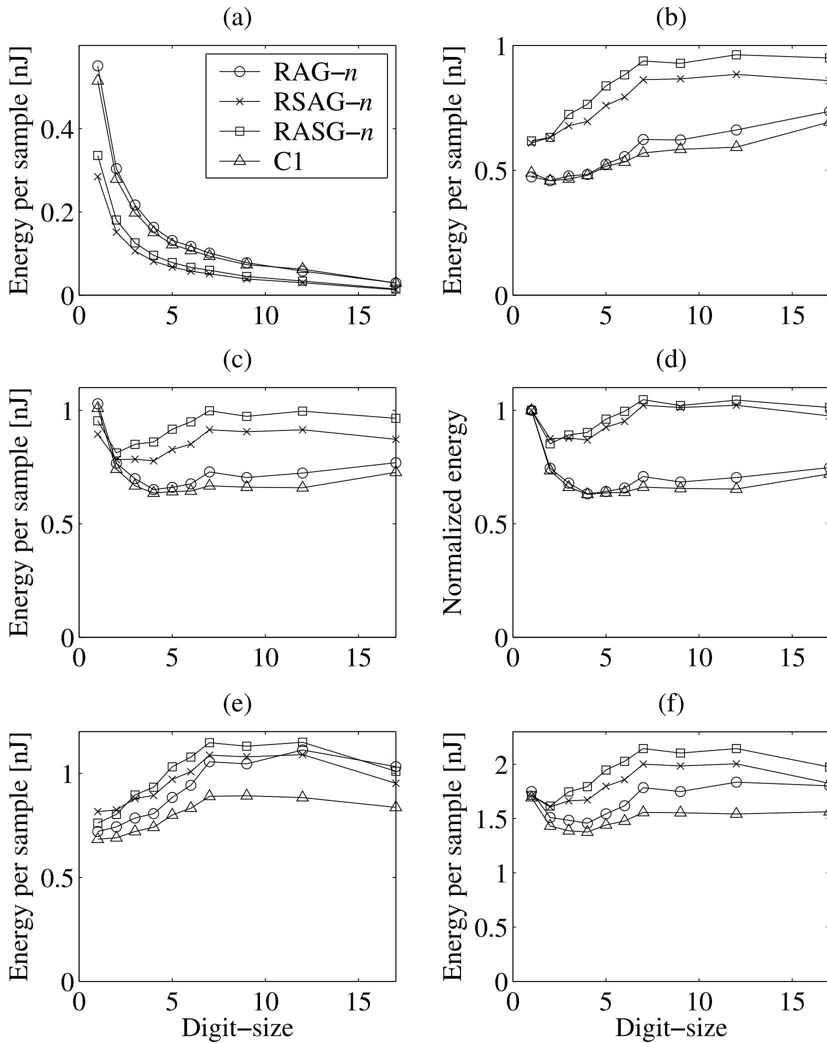


Figure 2.42 Consumed energy per sample for (a) shifts in the multiplier block, (b) adders in the multiplier block, (c) the total multiplier block, (d) the total multiplier block (normalized), (e) structural adders, and (f) all arithmetic parts.

Despite an increased complexity, it is clear that the C1 algorithm have low energy consumption due to low adder depth. However, it should be possible to obtain improved results using an algorithm that combines the good qualities of different algorithms.

2.7 Conclusions

In this chapter, we have investigated the possibilities to minimize the complexity of single-constant multipliers using serial arithmetic. Minimum sets of graphs that are required to obtain optimal results for different multiplier types were found. The results show that it is possible to save both adders and shifts compared to the CSD multipliers. Since shift operations in serial arithmetic require flip-flops, the total complexity was decreased. It was stated that there is a clear trade-off between adder cost and flip-flop cost.

The trade-offs in serial multiplier blocks were also studied in terms of complexity and adder depth. Two new MCM algorithms adapted for serial arithmetic were presented, and compared to an algorithm aimed at bit-parallel arithmetic. For the first algorithm, referred to as RSAG-*n*, it was shown that the number of shifts can be significantly reduced, while the number of adders is only slightly increased. The second algorithm, RASG-*n*, has on average the same adder cost as the parallel algorithm, while the flip-flop cost is kept low. Hence, for both algorithms, the total complexity is reduced for multiplier blocks implemented using serial arithmetic with a small digit-size. However, the proposed algorithms were shown to perform poorly when adder depth was considered.

Two example implementations of digit-serial FIR filters with varying digit-size were considered. The presented MCM algorithms were compared to several different approaches, including separate realization of the multipliers using CSD serial/parallel multipliers. The focus was on the arithmetic parts, i.e., the multiplier block and the adders in the delay section. From the results, it is evident that the multiplier blocks of each approach have an optimal digit-size, for which the energy per sample is minimized.

The design of multiplier blocks with low energy consumption turned out to be a more complicated problem than to just decrease the complexity. The results provide some guidelines for FIR filters implemented using digit-serial arithmetic. First and foremost, adder depth is a main factor that needs to be considered. Regarding the complexity, the number of adders is more important than the number of shifts. However, the shifts should also be considered, especially for bit-serial processing. It is advantageous to have serial shifts between subsequent adders, both within the multiplier block and before the delay section, since this reduces the glitch propagation. Furthermore, this will also increase the maximum sample

rate. For even coefficients, the shifts at the outputs may instead be placed before the preceding additions. In a similar way, shifts inside the multiplier block can be propagated through adders. Hence, a heuristic for placing of the shifts would be useful.

3

SWITCHING ACTIVITY IN BIT-SERIAL MULTIPLIERS

In this chapter, a method for computing the switching activity in bit-serial constant multipliers is presented [58].

The multipliers are described using the graph representation discussed in Section 1.4.3. Included in the investigation are all graph multipliers containing up to four adders, which is enough to realize all coefficients in the range [–4096, 4096], and in addition many coefficients with larger magnitude.

Theoretical functions to compute the switching activity are derived by solving the Chapman-Kolmogorov equations for discrete-time Markov chains. It is shown that the average switching activity can be determined for all graph multipliers. Most of the switching activities can be computed directly from the derived equations. The remaining cases are obtained by using look-up tables, which also can be generated using the Chapman-Kolmogorov equations. The switching activities are useful to estimate the energy consumption, and enable selection of the most energy efficient multiplier structure. Furthermore, it is noted that glitches also can be modeled, which is carried out for one specific type of full adder cell [55].

Finally, the method is simplified for bit-serial constant serial/parallel multipliers [54]. Here, the method makes it possible to choose a coefficient representation that result in reduced energy consumption.

3.1 Multiplier Stage

In this section, equations that describe the switching activities in bit-serial constant multipliers will be derived.

Bit-serial constant multipliers are based on additions/subtractions of shifted signals, and the generic building block that is required is the multiplier stage shown in Fig. 3.1. The variable d is the number of shifts at the B input. The boxes b_A and b_B correspond to a wire if the value is 1 and an inverter if the value is –1. These sign variables should be selected so that

$$(b_A, b_B) \in \{(1, 1), (1, -1), (-1, 1)\} \tag{3.1}$$

If b_A or b_B is negative, a subtraction is performed (given that the corresponding carry register then is initialized with a one). In an implementation, it is preferable to combine the b_A and b_B boxes with the full adder to obtain the classical definition of an adder and a subtractor, respectively. However, there would be three different cases to be studied separately if this merging was done already here. The signals A_0 and B_0 are therefore used instead of A and B to eliminate the effect of b_A and b_B. By doing this, there is only one case to consider, and the sign variables can then be compensated for in a final step.

3.1.1 Preliminaries

The input data is assumed to be a random sequence containing an infinite number of bits, which imply that the sign extension phase is not considered in the computation of the switching activity.

Any effect of glitches, which can be reduced by introducing pipelining, depends on the implementation of the full adders. For a commonly used full adder circuit called mirror adder [119], the glitching activity will be derived in Section 3.1.6 [55]. However, apart from that, glitches will not be considered. Thus, a model that assumes at most one logic change per clock cycle is used.

The probability function, *prob*, can be used to describe the probability that two signals have the same logic value, which for the signals X and Y is expressed as $prob(X = Y)$. In the following this is referred to as the correlation probability, and will be denoted $p_{X,Y}$. The switching activity at a node u is then defined as

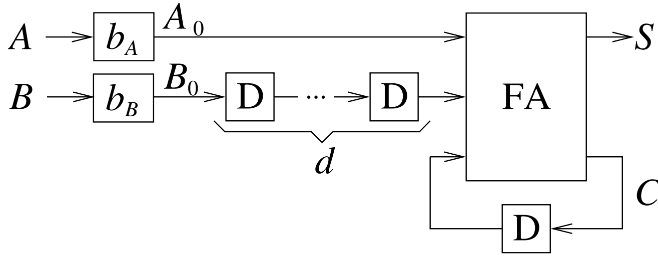


Figure 3.1 General stage in a bit-serial multiplier.

$$\alpha(u) = 1 - p_{u(n),\, u(n-1)} \tag{3.2}$$

which gives the probability that the logic value at the node changes between two following clock cycles. If the switching activity for each individual node is considered, the equation for average dynamic power given in (1.24) can be rewritten as [103]

$$P_{dyn} = \frac{1}{2} f_c V_{DD}^2 \sum_{i=1}^{N} \alpha(u_i) C_i \tag{3.3}$$

where C_i is the load capacitance at node u_i and N is the number of outputs of logic cells in the circuit. Given that no glitches occur, the switching activity at the output of a flip-flop is the same as at the input. In [55] it was shown that the energy consumption for the carry feedback flip-flop in the structure illustrated in Fig. 3.1 is linearly proportional to the switching activity of the C signal. This is illustrated in Fig. 3.2 for eight test cases implemented in a 0.35 μm CMOS process, and simulated using NanoSim™ with 1000 random input bits and a clock frequency of 4 MHz. Thus, the only switching activities that need to be computed in a multiplier are the sum and carry outputs for the full adders.

3.1.2 Sum Output Switching Activity

A multiplier stage, as illustrated in Fig. 3.1, is said to be in a certain state depending on the content of its flip-flops. From Fig. 3.1 it is clear that there are 2^{d+1} different state vectors $V = [v_d \ldots v_1\ v_0]$, where v_1 and v_0 refer to the values stored in the two flip-flops connected to the full adder. For each of these states we have that

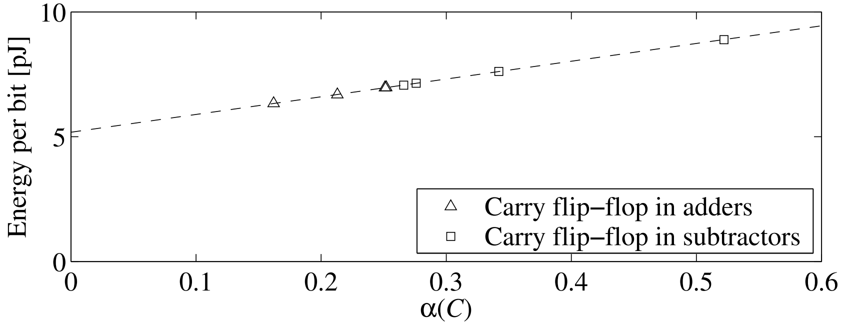


Figure 3.2 Relation between the carry switching activity and the energy consumption for the carry feedback flip-flop in bit-serial adders and subtractors. A straight line is adjusted to the simulation results.

$$S = A_0 \oplus f(V) = \begin{cases} f(V) & A_0 = 0 \\ \overline{f(V)} & A_0 = 1 \end{cases} \tag{3.4}$$

where the symbol ⊕ denotes a logic XOR operation, and a bar is used to give the inverted value. This means that for each specific state V, $prob(S = 0) = prob(S = 1)$ if $prob(A_0 = 0) = prob(A_0 = 1)$. Hence, if the input sequence A_0 is random, the sum output, S, will also be random since $f(V)$ and the inverted value of $f(V)$ then are selected in a random manner. The relation in (3.4) is explained in Table 3.1, where it can be seen that $f(V) = v_1 \oplus v_0$. However, note that the distribution of $f(V)$ will not affect the switching activity of the S signal.

According to the discussion above it is clear that S is random for the first stage, since A then is equal to the multiplier input, X, which is assumed to be a random sequence. For the next stage, A is either equal to X or to the sum output, S, of the first stage, which both are random. Thus, the relation is also true for the second stage, and so on. The conclusion is that for all multiplier stages in a bit-serial constant multiplier, with a random infinite input sequence X, we have

$$\alpha(S) = \alpha(A_0) = \alpha(A) = \alpha(X) = \frac{1}{2} \tag{3.5}$$

Hence, the carry outputs of the full adders are the only switching activities that remain to be determined.

A_0	v_1	v_0	$f(V)$	S	C
0	0	0	0	0	0
0	0	1	1	1	0
0	1	0	1	1	0
0	1	1	0	0	1
1	0	0	0	1	0
1	0	1	1	0	1
1	1	0	1	0	1
1	1	1	0	1	1

Table 3.1 Truth table for the multiplier stage in Fig. 3.1.

3.1.3 Switching Activity Using STGs

How the transitions between the states depend on the input signals can be visualized in a state transition graph (STG). STGs are commonly used to describe switching activities both in combinational [88] and sequential [138] circuits.

To determine the carry switching activities, the steady-state probability q_i for each state i in the STG is required. This can be obtained by solving the Chapman-Kolmogorov equations for discrete-time Markov chains defined as [110]

$$Q\Pi = Q \quad \text{where} \quad \sum_i q_i = 1 \tag{3.6}$$

where Π is the transition matrix and Q is the steady-state probability vector. The transition matrix is defined such that each entry, π_{ij}, is the probability to make a transition from state i to state j given that i is the current state. The system of equations can be solved by replacing one equation with the condition that the total probability should be equal to one [138], by iteration [72], from the eigenvalues associated with the transition matrix [124], or with an algorithm based on algebraic decision diagrams (ADD) [46]. Here, the first mentioned method has been used.

Example 1

Consider the first stage of a multiplier where the input signal, X, is delayed one clock period and then added to the nonshifted input, i.e., a multiplication by three according to Fig. 3.3. Note that A is always equal to B for the first stage, and, hence, $p_{A,B} = 1$. There are two flip-flops in this circuit, and therefore four possible states. The corresponding STG is shown in Fig. 3.4, where also the output signals, S and C, are indicated (v_1 and v_0 refer to the values stored in the flip-flops).

The system of equations corresponding to the STG in Fig. 3.4 is

$$Q\Pi = Q \quad \text{where} \quad \Pi = \frac{1}{2}\begin{bmatrix} 1 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 \end{bmatrix} \quad \text{and} \quad \sum_{i=0}^{3} q_i = 1 \tag{3.7}$$

which have the solution

$$Q = \begin{bmatrix} q_0 & q_1 & q_2 & q_3 \end{bmatrix} = \begin{bmatrix} \frac{1}{3} & \frac{1}{6} & \frac{1}{6} & \frac{1}{3} \end{bmatrix} \tag{3.8}$$

Finally, when the steady-state probabilities are known, it is straightforward to compute the carry switching activity from the STG. As marked in Fig. 3.4, the carry output will change value when state transitions are performed between the upper and lower half of the STG, which gives the carry switching activity

$$\alpha(C) = \frac{1}{2}(q_1 + q_2) = \frac{1}{6} \tag{3.9}$$

Example 2

In Fig. 3.5, a multiplier stage with $d = 1$ and $(b_A, b_B) = (1, 1)$ is shown. The corresponding STG is given in Fig. 3.6 (v_1 and v_0 refer to the values stored in the flip-flop at the B input and the C output, respectively).

In the same way as before, a system of equations for the steady-state probabilities is stated according to (3.6). The transition matrix corresponding to the STG in Fig. 3.6 is

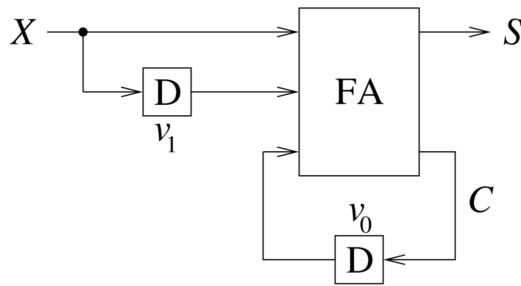
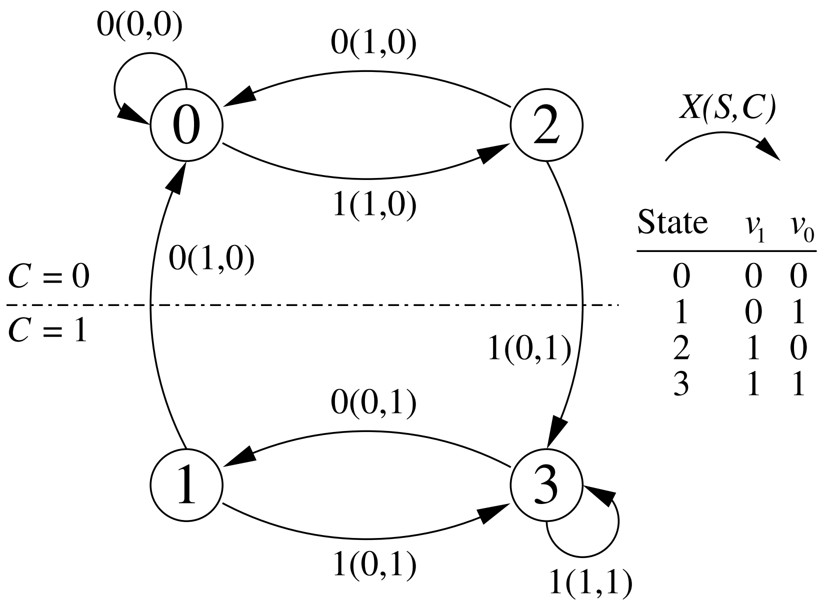


Figure 3.3 The first stage of a multiplier with one shift.



State	v_1	v_0
0	0	0
1	0	1
2	1	0
3	1	1

Figure 3.4 State transition graph for the circuit in Fig. 3.3.

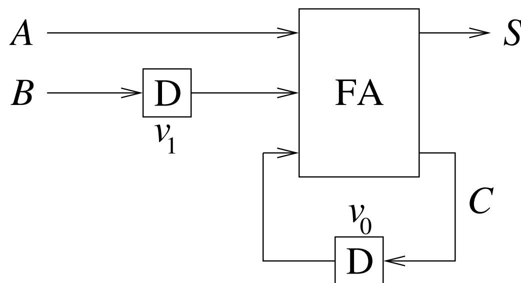


Figure 3.5 An intermediate stage of a multiplier with one shift at the *B* input.

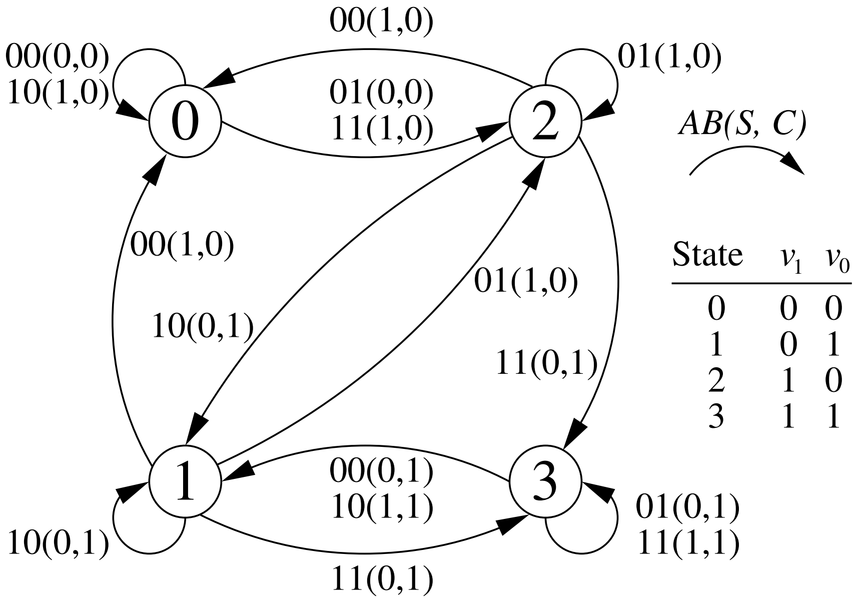


Figure 3.6 State transition graph for the circuit in Fig. 3.5.

$$\Pi = \frac{1}{2}\begin{bmatrix} 1 & 0 & 1 & 0 \\ p_{A,B} & (1-p_{A,B}) & (1-p_{A,B}) & p_{A,B} \\ p_{A,B} & (1-p_{A,B}) & (1-p_{A,B}) & p_{A,B} \\ 0 & 1 & 0 & 1 \end{bmatrix} \tag{3.10}$$

Note that this transition matrix is equal to the one in (3.7) if $p_{A,B} = 1$.

The solution to the Chapman-Kolmogorov equations is

$$\begin{cases} q_0 = q_3 = \dfrac{p_{A,B}}{2p_{A,B}+1} \\ q_1 = q_2 = \dfrac{1}{2(2p_{A,B}+1)} \end{cases} \tag{3.11}$$

and the carry switching activity can then be derived from Fig. 3.6 as

$$\alpha(C) = \frac{1}{2}(q_1 + q_2) = \frac{1}{2(2p_{A,B}+1)} \tag{3.12}$$

3.1.4 Carry Output Switching Activity

The number of shifts, i.e., the number of clock periods, d, that the B_0 signal is delayed, is different from case to case. Furthermore, the correlation probability associated with the stage input signals, A_0 and B_0, may also vary. For each specific case, it is possible to set up and solve the corresponding system of equations. Some of the results are given in Table 3.2. Note that the row for which $d = 1$ agrees with (3.12). A general expression for the carry switching activity can be derived from this according to

$$\alpha(C) = \frac{2^d}{4(2^d - 1 + 2p_{A_0, B_0})} \tag{3.13}$$

The correlation probability depends on the performed operation, i.e., if it is an addition or a subtraction, as follows

$$p_{A_0, B_0} = \begin{cases} p_{A, B} & b_A b_B = 1 \\ 1 - p_{A, B} & b_A b_B = -1 \end{cases} \tag{3.14}$$

For simplicity, the variable, λ, which is a function of the correlation probability associated with the stage input signals, is introduced. This variable is defined as

$$\lambda = 2p_{A, B} - 1 \tag{3.15}$$

which, for example, implies that λ is 1 if A and B are the same signal, 0 for uncorrelated signals, and –1 if A and B are complementary signals. The corresponding variable for the signals A_0 and B_0 is consequently defined by

$$\lambda_0 = 2p_{A_0, B_0} - 1 = b_A b_B \lambda \tag{3.16}$$

where (3.14) is used to obtain the final expression.

By combining (3.13) and (3.16), the carry switching activity for a multiplier stage with various signs can then be computed as

$$\alpha(C) = \frac{2^d}{4(2^d + b_A b_B \lambda)} \tag{3.17}$$

d	p_{A_0, B_0}				
	0	$\frac{1}{3}$	$\frac{1}{2}$	$\frac{2}{3}$	1
1	$\frac{1}{2}$	$\frac{3}{10}$	$\frac{1}{4}$	$\frac{3}{14}$	$\frac{1}{6}$
2	$\frac{1}{3}$	$\frac{3}{11}$	$\frac{1}{4}$	$\frac{3}{13}$	$\frac{1}{5}$
3	$\frac{2}{7}$	$\frac{6}{23}$	$\frac{1}{4}$	$\frac{6}{25}$	$\frac{2}{9}$
4	$\frac{4}{15}$	$\frac{12}{47}$	$\frac{1}{4}$	$\frac{12}{49}$	$\frac{4}{17}$
5	$\frac{8}{31}$	$\frac{24}{95}$	$\frac{1}{4}$	$\frac{24}{97}$	$\frac{8}{33}$

Table 3.2 Carry switching activity, $\alpha(C)$, for various correlation probability and number of shifts at the input.

This relation is illustrated in Fig. 3.7, for both addition and subtraction operations. Some conclusions that can be drawn from this figure are that $\alpha(C)$ is 1/4 when A and B are uncorrelated, and that $\alpha(C)$ can be far from 1/4 when few shifts are used, i.e., for a small d value.

According to (3.17), the only values that are required to compute all carry switching activities in a bit-serial multiplier are the correlation probabilities between all pairs of stage inputs. Thus, the correlation probability is what will be discussed next.

3.1.5 Input-Output Correlation Probability

Here, the correlation probability between the inputs, A and B, and the output, S, of a multiplier stage will be derived.

First, consider Example 1 above, where $d = 1$ and $\lambda = 1$. From the STG, which is shown in Fig. 3.4, and the steady-state probabilities given in (3.8), the correlation probability is obtained as

$$p_{X,S} = q_0 + q_3 = \frac{2}{3} \quad (3.18)$$

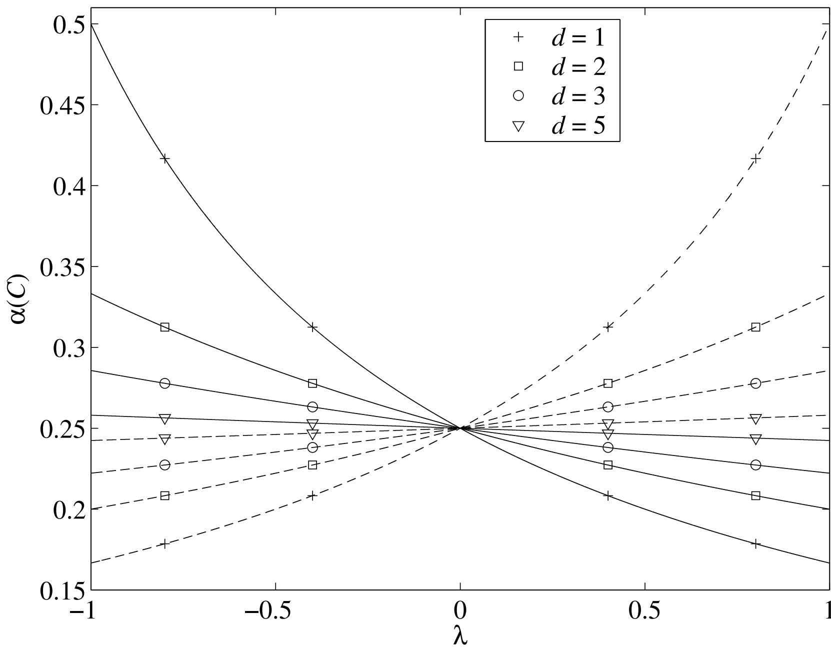


Figure 3.7 Relation between the correlation variable, λ, and the carry switching activity, $\alpha(C)$, for various number of shifts, d. Solid and dashed lines correspond to addition and subtraction, respectively.

For Example 2 with arbitrary input correlation, i.e., $\lambda = 2p_{A,B} - 1$, the STG in Fig. 3.6 and the steady-state probabilities in (3.11) gives

$$\begin{cases} p_{A,S} = q_0 + q_3 = \dfrac{2p_{A,B}}{2p_{A,B}+1} = \dfrac{1+\lambda}{2+\lambda} \\ p_{B,S} = p_{A,B}(q_0+q_3) + (1-p_{A,B})(q_1+q_2) = \\ \quad = \dfrac{2p_{A,B}^2 - p_{A,B} + 1}{2p_{A,B}+1} = \dfrac{2+\lambda+\lambda^2}{2(2+\lambda)} \end{cases} \tag{3.19}$$

By computing the correlation probabilities under different circumstances, in the same manner as was done for the carry switching activity, it can be shown that

$$\begin{cases} p_{A_0,S} = \dfrac{2^d + 2\lambda_0}{2(2^d + \lambda_0)} \\ p_{B_0,S} = \dfrac{2^d + \lambda_0 + \lambda_0^2}{2(2^d + \lambda_0)} \end{cases} \qquad (3.20)$$

Finally, the sign variables, b_A and b_B, are considered. When the operation is an addition, (3.20) is also valid for the A and B signals, and with $\lambda_0 = \lambda$ according to (3.16) we have

$$\begin{cases} p_{A,S} = p_{A_0,S} = \dfrac{2^d + 2\lambda}{2(2^d + \lambda)} \\ p_{B,S} = p_{B_0,S} = \dfrac{2^d + \lambda + \lambda^2}{2(2^d + \lambda)} \end{cases} \quad (b_A, b_B) = (1, 1) \qquad (3.21)$$

However, if a subtraction is performed, (3.16) gives that $\lambda_0 = -\lambda$ since either b_A or b_B is negative. If $b_A = -1$ then A and A_0 are complementary signals, and hence

$$\begin{cases} p_{A,S} = 1 - p_{A_0,S} = \dfrac{2^d}{2(2^d - \lambda)} \\ p_{B,S} = p_{B_0,S} = \dfrac{2^d - \lambda + \lambda^2}{2(2^d - \lambda)} \end{cases} \quad (b_A, b_B) = (-1, 1) \qquad (3.22)$$

In the same way, if $b_B = -1$, (3.20) is rewritten as

$$\begin{cases} p_{A,S} = p_{A_0,S} = \dfrac{2^d - 2\lambda}{2(2^d - \lambda)} \\ p_{B,S} = 1 - p_{B_0,S} = \dfrac{2^d - \lambda - \lambda^2}{2(2^d - \lambda)} \end{cases} \quad (b_A, b_B) = (1, -1) \qquad (3.23)$$

Merging the three different cases, i.e., (3.21) for addition and the two versions of subtraction given in (3.22) and (3.23), results in

$$\begin{cases} p_{A,S} = \dfrac{2^d + (1+b_A)b_B\lambda}{2(2^d + b_A b_B \lambda)} \\ p_{B,S} = \dfrac{2^d + b_A b_B \lambda + b_B \lambda^2}{2(2^d + b_A b_B \lambda)} \end{cases} \tag{3.24}$$

To summarize, the equations that are of interest when the carry switching activity in a bit-serial multiplier is to be computed are (3.15), (3.17), and (3.24).

3.1.6 Glitching Activity

Unwanted switches between the two logic levels, referred to as glitches, are common in synchronous digital circuits such as bit-serial adders. When studying the transistor schematic for a mirror adder [119], it is possible to establish that a glitch will occur if the inputs change so that the carry output, C, switches but not the sum output, S. This is illustrated in Fig. 3.8, where A, B, and C_{in} are the input signals to the full adder circuit. Here, two glitches occur on the S signal, one positive and one negative. Note that each glitch corresponds to two logic changes.

In the same way as before, also the glitching activity can be computed using a state transition graph [138] and solving the corresponding Chapman-Kolmogorov equations for discrete-time Markov chains [110]. It is found that the most complicated case is when only one shift is used. Some results of such computations are given in Table 3.3, where the glitching activity is denoted $\alpha_2(S)$. It can be shown that

$$\alpha_2(S) = \begin{cases} \dfrac{1-\lambda^2}{8(2+b_A b_B \lambda)} & d = 1 \\ \dfrac{\alpha(C)}{4} & d \neq 1 \end{cases} \tag{3.25}$$

3.1.7 Example

The equations derived in the previous sections are here verified by an example. Consider the realization of a multiplier with the coefficient 347

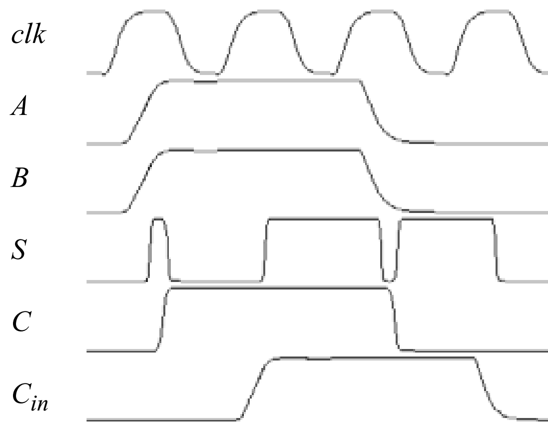


Figure 3.8 Glitches that may occur in a mirror adder. The clock frequency used in the visualized simulation was 500 MHz.

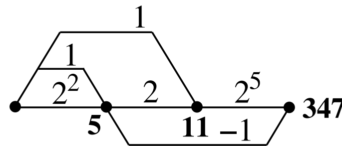


Figure 3.9 Graph representation for a realization of the coefficient 347.

$p_{A,B}$	λ	$\alpha_2(S)$
1	1	0
2/3	1/3	1/21
3/5	1/5	3/55
1/2	0	1/16
2/5	–1/5	1/15
1/10	–4/5	3/80

Table 3.3 Glitching activity for $d = 1$ and different correlation probabilities.

shown in Fig. 3.9. The multiplier is composed of two adders and one subtractor.

The switching and glitching activities are computed using (3.5), (3.15), (3.17), (3.24), and (3.25). The theoretical results are compared with a high-level simulation using 1 000 000 random input bits. As can be seen in Table 3.4, the model agrees well with the simulated results.

Adder number	Derived equations			Simulation		
	$\alpha(C)$	$\alpha(S)$	$\alpha_2(S)$	$\alpha(C)$	$\alpha(S)$	$\alpha_2(S)$
1	0.2000	0.5000	0.0500	0.2003	0.5008	0.0501
2	0.2273	0.5000	0.0545	0.2272	0.4996	0.0546
3	0.2501	0.5000	0.0625	0.2501	0.4991	0.0625

Table 3.4 Computed and simulated switching and glitching activities.

3.2 Graph Multipliers

The possibility to compute the switching activity, using the equations derived in the previous section, is here investigated for all graph multipliers with up to four adders, which are shown in Fig. 2.1.

3.2.1 Correlation Probability Look-Up Tables

To be able to derive the carry switching activity in a specific multiplier stage, the correlation probability associated with the stage inputs has to be known. This is not always possible to derive from the equations. It is, however, possible to determine the correlation probability between the inputs to the stage by solving the Chapman-Kolmogorov equations for each specific case and store the results in look-up tables.

One stage that can not be solved using the equations is, for example, the last one in the graph illustrated in Fig. 3.10. An STG corresponding to the first two stages in this graph has $2^{d_1 + d_2 + 2}$ states, since there are $d_1 + d_2$ shifts and one carry feedback flip-flop in each adder. Besides the large number of possible shift combinations, there are 3^2 different sign combinations of the first two stages that must be considered. Some of the resulting values, assuming that the first two stages correspond to additions, are given in Table 3.5. Note that the table is symmetric and that the values converge towards 1/2.

3.2.2 The Applicability of the Equations

As stated in the previous section, it is not always possible to derive the switching activities from the equations. In Table 3.6, statistics on the equations applicability with respect to all multiplier stages are presented.

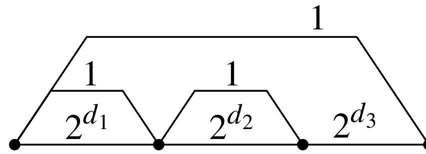


Figure 3.10 Graph number 4 with three adders.

d_1	d_2				
	1	2	3	4	5
1	0.5556	0.5333	0.5185	0.5098	0.5051
2	0.5333	0.5200	0.5111	0.5059	0.5030
3	0.5185	0.5111	0.5062	0.5033	0.5017
4	0.5098	0.5059	0.5033	0.5017	0.5009
5	0.5051	0.5030	0.5017	0.5009	0.5005

Table 3.5 Input correlation probability, $p_{A,\ B}$, for the last stage of the graph shown in Fig. 3.10.

Definition of the stage types, and the symbols used in Table 3.7		Adders				Total
		1	2	3	4	
|	Same input signals, i.e., $\lambda = 1$	1	3	13	68	85
x	The derived equations applies	0	1	6	36	43
o_4	Same type as graph 4 with 3 adders	0	0	1	5	6
o_7	Same type as graph 7 with 3 adders	0	0	1	4	5
o	Unsolvable with equations	0	0	0	12	12
⊗	Consequence error occur	0	0	0	3	3
Total number of stages		1	4	21	128	154

Table 3.6 Statistics on the classification of stages.

For example, in Fig. 2.1 there are 32 different structures with four adders, which result in 32·4 = 128 stages. It is shown that the equations can be used to derive the carry switching activity in more than 83% of the multiplier stages. If look-up tables for graph number 4 and 7 with three adders are produced, the carry switching activity in more than 92% of the multiplier stages can be derived. Using the symbols defined in Table 3.6, the

Gr. no.	Stage no.			Gr. no.	Stage no.				Gr. no.	Stage no.			
	1	2	3		1	2	3	4		1	2	3	4
1	$\vert$			1	$\vert$	x	x	x	17	$\vert$	$\vert$	x	o
				2	$\vert$	x	x	o	18	$\vert$	$\vert$	x	x
1	$\vert$	x		3	$\vert$	x	x	x	19	$\vert$	$\vert$	x	x
2	$\vert$	$\vert$		4	$\vert$	x	x	$\vert$	20	$\vert$	$\vert$	x	$\vert$
				5	$\vert$	x	x	o	21	$\vert$	$\vert$	$\vert$	o
1	$\vert$	x	x	6	$\vert$	x	x	x	22	$\vert$	$\vert$	$\vert$	o_4
2	$\vert$	x	x	7	$\vert$	x	x	x	23	$\vert$	$\vert$	$\vert$	x
3	$\vert$	x	$\vert$	8	$\vert$	x	x	$\vert$	24	$\vert$	$\vert$	$\vert$	$\vert$
4	$\vert$	$\vert$	o_4	9	$\vert$	x	$\vert$	o	25	$\vert$	$\vert$	o_7	o
5	$\vert$	$\vert$	x	10	$\vert$	x	$\vert$	o	26	$\vert$	$\vert$	o_7	$\otimes$
6	$\vert$	$\vert$	$\vert$	11	$\vert$	x	$\vert$	x	27	$\vert$	$\vert$	o_7	$\vert$
7	$\vert$	$\vert$	o_7	12	$\vert$	x	$\vert$	$\vert$	28	$\vert$	$\vert$	x	o
				13	$\vert$	$\vert$	o_4	$\otimes$	29	$\vert$	$\vert$	$\vert$	o
				14	$\vert$	$\vert$	o_4	o	30	$\vert$	x	x	o
				15	$\vert$	$\vert$	o_4	$\otimes$	31	$\vert$	x	$\vert$	o
				16	$\vert$	$\vert$	o_4	$\vert$	32	$\vert$	$\vert$	$\vert$	o_7

Table 3.7 Classification of multiplier stages in all graph topologies with up to four adders. The graphs are shown in Fig. 2.1.

type of each multiplier stage for all the structures in Fig. 2.1 is given in Table 3.7.

3.2.3 Example

How the switching activity can be significantly reduced by selecting the best graph structure will be shown in an example. Two different graphs, both implementing the coefficient 87, are shown in Fig. 3.11. Note that the graph in Fig. 3.11 (a) corresponds to the commonly used CSD representation, which is $10\bar{1}0\bar{1}00\bar{1}$. The multiplier in Fig. 3.11 (b) must be

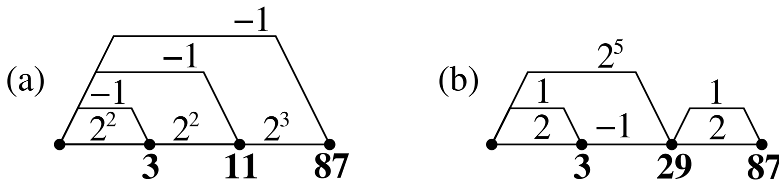


Figure 3.11 Graph representations for the coefficient 87.

	$\alpha(C_1)$	$\alpha(C_2)$	$\alpha(C_3)$	Average
Multiplier (a)	0.3333	0.2727	0.2529	0.2863
Multiplier (b)	0.1667	0.2526	0.1667	0.1953

Table 3.8 Switching activities for the graphs in Fig. 3.11.

pipelined to prevent propagation of glitches, but pipelining will not affect the carry switching activities.

The switching activities can be computed by the derived equations, and the results are presented in Table 3.8. The savings in average carry switching activity is more than 30%.

3.3 Serial/Parallel Multipliers

The design of a constant serial/parallel multiplier [141] is based on shifted sums of the input data. Only the structures corresponding to graph multiplier number 1 (see Fig. 2.1) with different number of adders are used here. Furthermore, the shifts, i.e., the flip-flops, are placed so that no adders are directly connected without any intermediate shifts, as illustrated in Fig. 3.12. Hence, any glitches caused by the adders will not propagate to subsequent stages.

The number of adders (stages) required to implement a serial/parallel multiplier is one less than the number of nonzero digits in the coefficient representation. A common method to implement the multipliers is based on the CSD representation, which requires a minimum number of adders for this specific structure. However, other MSD representations can sometimes result in more energy efficient implementations.

As indicated in Fig. 3.12, some of the multiplier stages may perform a subtraction instead of an addition. A subtractor is implemented by adding an inverter at the input of the full adder and initiate the carry flip-flop with a one, which was mentioned in Section 3.1.

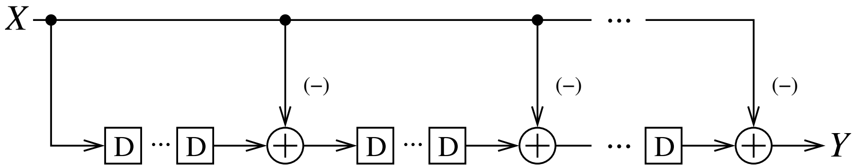


Figure 3.12 The structure of a bit-serial, constant serial/parallel multiplier.

All stages, except the first one, of a bit-serial constant serial/parallel multiplier have two different input signals; the multiplier input (i.e. the signal X in Fig. 3.12) and the sum output from the previous stage. In the following, these signals are referred to as signal A and B, in the same way as for the general multiplier stage shown in Fig. 3.1.

3.3.1 Simplification of the Switching Activity Equation

In Fig. 3.13, two intermediate stages of a bit-serial constant serial/parallel multiplier are shown. The main difference compared to the general stage given in Fig. 3.1 is that only one sign variable, b_i, is required for each stage i. This variable is defined so that $b_i = 1$ and $b_i = -1$ correspond to an addition and a subtraction, respectively. Hence, it is straightforward to deduce an expression for the carry switching activity by taking the general equation, where the sign variables are excluded, as starting point. Rewriting (3.13) using λ_0 as defined in (3.16) result in

$$\alpha(C) = \frac{2^d}{4(2^d + \lambda_0)} \tag{3.26}$$

To simplify the computations, it is desirable to eliminate the term λ_0 in (3.26). From Fig. 3.13, it is clear that the A_0 signals are equal in two subsequent stages if the corresponding sign variables are equal. On the contrary, if the sign variables are different in two following stages, the A_0 signals are complementary. Furthermore, the signals S_{i-1} and $B_{0,\,i}$ are always identical. Hence, the equation for λ_0 given in (3.16) can then be rewritten for a specific stage i as

$$\lambda_{0,\,i} = 2p_{A_{0,\,i},\,B_{0,\,i}} - 1 = \begin{cases} 2p_{A_{0,\,i-1},\,S_{i-1}} - 1 & b_i = b_{i-1} \\ 2(1 - p_{A_{0,\,i-1},\,S_{i-1}}) - 1 & b_i \neq b_{i-1} \end{cases} \tag{3.27}$$

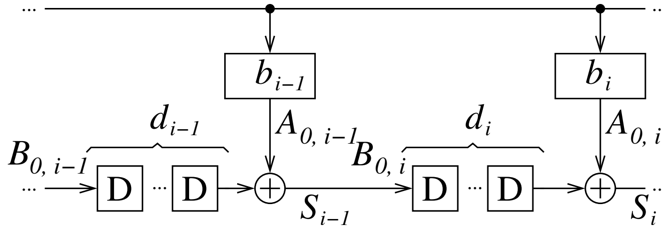


Figure 3.13 Stages $i-1$ and i of a serial/parallel multiplier.

which can be combined into the single expression

$$\lambda_{0,\, i} = b_i b_{i-1}(2p_{A_{0,\, i-1},\, S_{i-1}} - 1) \tag{3.28}$$

Using (3.26), the equation for $p_{A_0,\, S}$ given in (3.20) can be rewritten according to

$$p_{A_0,\, S} = \frac{2^d + 2\lambda_0}{2(2^d + \lambda_0)} = \frac{2(2^d + \lambda_0) - 2^d}{2(2^d + \lambda_0)} = 1 - 2\alpha(C) \tag{3.29}$$

Substituting (3.29) in (3.28) gives

$$\lambda_{0,\, i} = b_i b_{i-1}(1 - 4\alpha(C_{i-1})) \tag{3.30}$$

It is now possible to derive a function for the carry switching activity that only depends on the carry switching activity in the preceding stage and the coefficient representation, i.e., the performed operation and the number of shifts. An equation that can be used to compute the carry switching activity, for all adders in multipliers that comply with the structure shown in Fig. 3.12, is obtained by introducing conditions so that also the first stage is covered. Hence, this is expressed as

$$\alpha(C_i) = \frac{2^{d_i}}{4(2^{d_i} + b_i b_{i-1}(1-4\alpha(C_{i-1})))} \qquad \begin{cases} 1 \le i \le N \\ b_0 = 1 \\ \alpha(C_0) = 0 \end{cases} \tag{3.31}$$

where N is the number of stages.

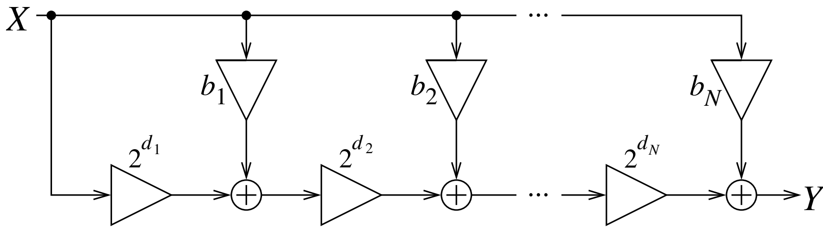


Figure 3.14 Variables in a serial/parallel multiplier.

Representation	Sign variables			Shift variables		
	b_1	b_2	b_3	d_1	d_2	d_3
MSD1	1	1	1	2	1	4
MSD2	–1	–1	1	2	2	4
MSD3	1	–1	1	1	2	4

Table 3.9 Design variables for the coefficient representations in (3.32).

3.3.2 Example

How the switching activity can be reduced by selecting the best coefficient representation will be shown in this example. The coefficient 177 can be represented in signed-digit code using four or more nonzero digits. Three different MSD representations are

$$177_{10} = 10110001_{MSD1} = 10\bar{1}0\bar{1}0001_{MSD2} = 110\bar{1}0001_{MSD3} \quad (3.32)$$

where a bar is used to denote negative digits. Note that MSD1 and MSD2 correspond to the binary and CSD representations, respectively.

The odd and positive coefficient value, c, for a serial/parallel multiplier, with the variables b_i and d_i as illustrated in Fig. 3.14, can be computed as

$$c = 2^{d_1 + \ldots + d_N} + \sum_{i=2}^{N} (b_{i-1} 2^{d_i + \ldots + d_N}) + b_N \quad (3.33)$$

The variables corresponding to the three coefficient representations are given in Table 3.9. The switching activities are computed according to (3.31), and the results are presented in Table 3.10. For this example, the

Representation	$\alpha(C_1)$	$\alpha(C_2)$	$\alpha(C_3)$	Average
MSD1	0.2000	0.2273	0.2486	0.2253
MSD2	0.3333	0.2727	0.2486	0.2849
MSD3	0.1667	0.2727	0.2486	0.2293

Table 3.10 Carry switching activities for different coefficient representations.

average carry switching activity can be decreased with more than 20% by using binary instead of CSD representation of the coefficient.

3.4 Conclusions

A method for computing the switching activity in bit-serial constant multipliers has been presented. A key factor is to determine the correlation between signals that are inputs to the same adder. It was shown that the average switching activity in all multipliers with up to four adders can be found. The carry switching activity in more than 83% of the adders in all possible graph topologies can be obtained directly from the derived formulas, and the remaining by using look-up tables. Hence, it is possible to reduce the switching activity by selecting the best graph structure of the multiplier to be implemented, as was illustrated in an example.

Furthermore, it was noted that glitches might occur in various full adder circuits. For the mirror adder this happens when the carry output switches while the sum output is supposed to be stable. The unwanted glitching activity obtained at the sum output was also computed theoretically.

For serial/parallel multipliers, a general function that directly gives the carry switching activities was derived. It was shown that the energy consumption can be decreased by choosing the best representation of the coefficient to be implemented.

4

COMPLEXITY OF PARALLEL CONSTANT MULTIPLIERS

Implementation of FIR filters using shift-and-add multipliers has been an active research area for the last decade. However, almost all algorithms so far have only been focused on reducing the adder cost, i.e., the number of adders and subtractors, while little effort was put on the bit-level implementation.

Here, a complexity measure that can be used for all types of constant operations based on shifts, additions, and subtractions is presented [63]. A multiple-constant multiplication (MCM) algorithm based on this complexity model is shown to have a similar adder cost compared with previous algorithms, while the number of full adder cells is significantly reduced.

In general, it is possible to compute a given multiplier coefficient in several ways, and even though the total number of adders/subtractors is the same, the number of full and half adder cells may vary [42]. Methods to optimize the number of full and half adders required to realize a given set of additions are presented [69]. Detailed implementation results for two case studies show that the area can be reduced.

Furthermore, a graph based minimum adder depth MCM algorithm is presented. All multiplier coefficients are here guaranteed to be realized at the theoretically lowest depth possible. The motivation for low adder depth is that this has been shown to be a main factor for the energy consumption.

4.1 Bit-Level Optimization

To differentiate multiple-constant multiplication designs of equal adder cost, a more detailed cost measure can be used. It has been shown that the number of full adder cells can be reduced for implementations of constant multipliers [96]. In [24] a cost corresponding to the required number of full adders was introduced. This was referred to as adder-bit cost, and an improved approach was used in [22]. Here, this method is refined further and an MCM algorithm that minimizes this complexity measure is presented. Furthermore, it is shown that the use of half adders can be removed if the sign of the coefficient may be changed.

It is assumed that ripple-carry adders are used. To realize constant multiplication using other, more complex, adder types is usually not beneficial since a large part of the gain in speed obtained for a single addition then is lost. The reason for this is that the total critical path normally only increases with one full adder when cascading two ripple-carry adders. However, if an accelerated adder structure is used, the results can still be used as a start to form suitable complexity measures. Furthermore, the transformation that will be presented in Section 4.1.3 to eliminate half adders, should be advantageous for most adder structures since the corresponding part of the adder will be removed, effectively reducing the wordlength.

Terms and concepts introduced in Section 1.4 are assumed to be familiar in the rest of this chapter.

4.1.1 Scaling

Here, all nodes in the MCM blocks are explicitly scaled using safe scaling [149], i.e., there will never be an overflow and the output wordlength is exactly enough to represent all possible outputs. Quantization is not considered, and, hence, full precision is kept throughout the MCM blocks. However, it should be noted that internal rounding may be used to reduce the complexity with the cost of an introduced quantization error [20],[93].

Using safe scaling, the number of output bits, W_i, from the add operation associated with the fundamental f_i is

$$W_i = W_0 + \lceil \log_2(|f_i|) \rceil \tag{4.1}$$

where W_0 is the wordlength at the input of the MCM block.

If the fundamental is a function of several input signals, as in a matrix multiplication, the output wordlength is

$$W_i = W_0 + \lceil \log_2(\sum |f_i|) \rceil \tag{4.2}$$

where in this case there are several f_i:s, each one being the coefficient corresponding to one input.

In a similar way, it is possible to derive the output wordlengths in FIR MCM blocks where the delays are taken into the redundancy utilization (sometimes referred to as vertical subexpressions), as in [48]. The output wordlength for each node, W_i, is also identical to the required number of register bits if that node is to be delayed. Hence, it is possible to use this model to compare the number of register bits for FIR filter realizations.

For the FIR filters shown in Fig. 1.1, (4.2) can be used as

$$W_{h_i} = W_0 + \left\lceil \log_2\left(\sum_{k=i}^{N} |h_k|\right) \right\rceil \qquad 0 \le i \le N-1 \tag{4.3}$$

where W_{h_i} is the required wordlength at the output of the structural adder with a multiplication by h_i at one of the inputs. The expression in (4.3) is in practice only relevant for the transposed direct form, as the additions would be performed in a tree structure for the direct form.

4.1.2 Complexity Model

Here, the idea is to count the number of full adder cells required to realize a wordlevel adder.

For each fundamental, f_i, obtained according to (1.17), there are two possibilities associated with the magnitude of the edge values, e_j and e_k. In the first case, the value at one of the input nodes is left shifted at least once while the significance of the other value is unchanged (otherwise the result would not be odd). The shift operation is, for simplicity, always associated with the input node f_j. The second case occurs when the magnitude of both edge values is less than one, i.e., two right shifts are performed. In order to obtain an odd integer fundamental, the edge values must then be of equal significance. Hence, we have

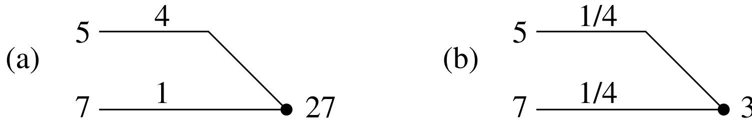


Figure 4.1 Example of the case with (a) one left shift and (b) two right shifts.

$$\begin{aligned} &|e_j| > 1, |e_k| = 1 \quad \text{(one left shift) or} \\ &|e_j| = |e_k| < 1 \quad \text{(two right shifts)} \end{aligned} \tag{4.4}$$

An example of each of the two cases is shown in Fig. 4.1.

Both these operations should correspond to an addition/subtraction, and, hence, at most one of the edge values can be negative. The requirement for the signs of the edge values is therefore stated as

$$(sign\{e_j\}, sign\{e_k\}) \in \{(1, 1), (-1, 1), (1, -1)\} \tag{4.5}$$

When signs are also considered, this leads to the five different cases illustrated in Fig. 4.2. The hardware complexity for these cases can be determined by counting the number of full adder (FA) cells required to realize each wordlevel (WL) adder.

To be an actual add operation, the number of left shifts must be less than the wordlength of the edge without any shift operation. Hence, for the left shift case, we have

$$\log_2(|e_j|) < W_0 + \lceil \log_2(|f_k|) \rceil = W_k \tag{4.6}$$

The required number of full adders, $n_{i,\ FA}$, to perform the add operation associated with the fundamental f_i can then be defined as

$$n_{FA,\,i} = W_i - \log_2(|e_j|) \qquad |e_j| > 1 \tag{4.7}$$

This means that the required number of full adders is W_i, as defined in (4.1), minus the number of shifts associated with the edge value e_j, as illustrated in Fig. 4.3 for the three sign combinations.

In the right shift case, on the other hand, the magnitude of the edge values are less than one, which gives more full adders than W_i as shown in Fig. 4.4. However, the sum bits corresponding to the fractional bits are known to be zero, and, hence, the carry in bit to the full adder correspond-

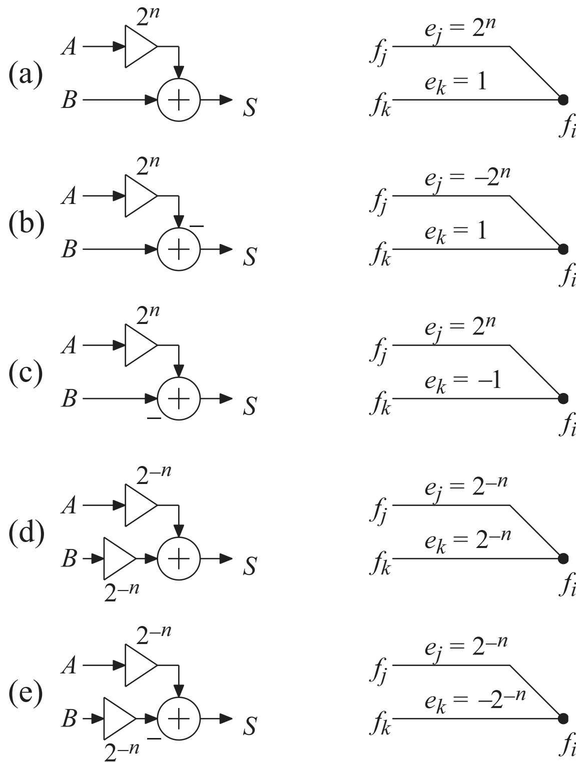


Figure 4.2 Alternatives using (a)–(c) one left shift and (d)–(e) two right shifts with corresponding graphs. Here, n is a positive integer, equal to the number of shifts.

ing to the least significant output bit, s_0, can be computed without using full adders as explained in the following.

Consider the case shown in Fig. 4.4 (a), where the right shifts are followed by an addition. The first full adder can be removed as a_0 is always equal to b_0 (otherwise the sum output would not be zero), and any of them can therefore be used as carry input to the next full adder. If a_0 is 0, then the situation is the same as for the first full adder, i.e., a_1 and b_1 must be equal, and if a_0 is 1 then a_1 and b_1 must be different. What this means in practice is that once a carry is generated, it is always propagated to the subsequent full adder in order to obtain zeros at the sum outputs. Hence, the carry input, c_n, to the full adder with output s_0, is one if any of the inputs to the preceding full adder is one and can therefore be computed using a single logic OR operation as

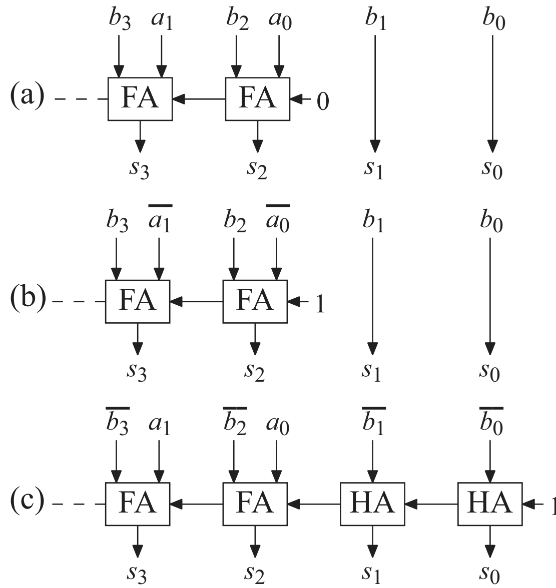


Figure 4.3 Adding the least significant bits for the adders in Figs. 4.2 (a)–(c), respectively. The A signal is left shifted two times, i.e., $|e_j| = 4$.

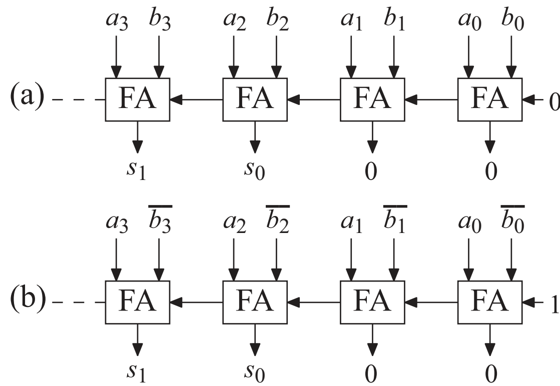


Figure 4.4 Adding the least significant bits for the adders in Figs. 4.2 (d) and (e), respectively. Here, $n = 2$, which gives $|e_j| = |e_k| = 1/4$.

$$c_n = a_{n-1} \vee b_{n-1} \tag{4.8}$$

The reduced architecture is given in Fig. 4.5 (a). Furthermore, Table 4.1 shows an example of this, corresponding to the operation in Fig. 4.1 (b), for a 3 bit input, X.

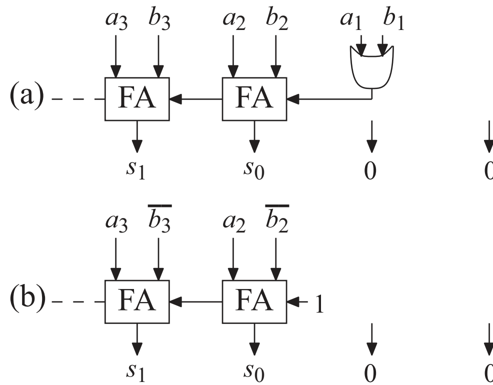


Figure 4.5 Simplified structures for the adders in Fig. 4.4, obtained by removing the full adders with zero outputs.

Carries		x_2	x_2	x_2			
			x_1	x_1	x_1		
				x_0	x_0	x_0	
$\frac{5X}{4}$			x_2	x_1	x_0	$.0$	0
			0	0	x_2	$.x_1$	x_0
$\frac{7X}{4}$			x_2	x_1	x_0	$.0$	0
			0	x_2	x_1	$.x_0$	0
	+		0	0	x_2	$.x_1$	x_0
$3X$			x_2	x_1	x_0	$.0$	0
		x_2	x_1	x_0			

Table 4.1 Addition of two right shifted values.

In the case of a subtraction, as in Fig. 4.4 (b), the n first full adders can be removed since a_i is always equal to b_i for $i < n$. Hence, the carry is propagated so that the carry input, c_n, to the full adder with output s_0 should be set to one, as illustrated in Fig. 4.5 (b).

If the cost for the OR-gate in the addition case is ignored, the complexity for the two cases illustrated in Fig. 4.4 is

$$n_{FA,i} = W_i \qquad |e_j| < 1 \tag{4.9}$$

The number of overhead full adders is defined as the difference between the total number of full adders, $n_{FA,\ i}$, according to (4.7) and (4.9), respectively, and the input wordlength, W_0. For the fundamental f_i this is computed as

$$n^{OH}_{FA,\,i} = n_{FA,\,i} - W_0 = \begin{cases} \lceil \log_2(|f_i|) \rceil - \log_2(|e_j|) & |e_j| > 1 \\ \lceil \log_2(|f_i|) \rceil & |e_j| < 1 \end{cases} \tag{4.10}$$

If the edge without any shift operation is negative, i.e., $e_k = -1$, overhead half adders are required to compute the least significant output bits, as illustrated in Fig. 4.3 (c). Hence, we have

$$n^{OH}_{HA,\,i} = \begin{cases} \log_2(e_j) & e_k = -1 \\ 0 & \text{otherwise} \end{cases} \tag{4.11}$$

Note that there are some cases where a complete full adder is not required, for example, in Fig. 4.3 (a) where the full adder of least significance has a carry input equal to constant zero, which means that a half adder would be sufficient. However, all full adders are here assumed to be complete. Furthermore, it would be possible to remove the full adder corresponding to the most significant output bit in nodes that gives positive fundamentals, as the sign then will be equal to the sign of the input data. However, this simplification has not been used due to the resulting large load of the input sign bit. The reason that a corresponding method can not be used for negative fundamentals is that a zero valued input data then would give an incorrect result.

4.1.3 Removing Half Adders

As illustrated in Fig. 4.3, half adders are only required for the operation $f_i = 2^n f_j - f_k$. However, this type of operation can be avoided by the transformation shown in Fig. 4.6 (a), i.e., by changing the sign of the edge weights. The required fundamental is then obtained as $-f_i = -2^n f_j + f_k$, for which no half adders are needed. The negation is compensated for by changing the sign of outgoing edges. This might result in two negated incoming edges to some of the following nodes, for which the negation then is propagated through the node, as illustrated in Fig. 4.6 (b). The

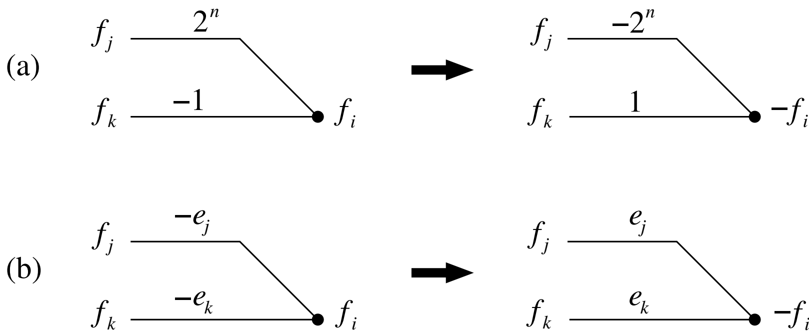


Figure 4.6 Changing the sign of the edge weights. (a) Transformation to eliminate the use of half adders. (b) Transformation for the case when both edge values are negative.

graph is searched from the input node throughout the MCM block, and the sign transformation is applied to remove all overhead half adders.

This sign transformation may result in that some coefficients are realized with a sign opposite to the specification. However, the sign of the coefficients that are realized in a shift-and-add network is usually not of interest. The reason is that an incorrect sign of the coefficient in most DSP applications can be compensated for in the subsequent operations. This is certainly the case for FIR filters, as shown in Fig. 1.1, where the following structural addition simply can be replaced by a subtraction. Therefore, MCM algorithms normally only consider positive coefficients.

4.1.4 Single-Constant Multiplication Example

As discussed in Chapter 2, the number of possible solutions is limited for single-constant multiplication. However, the number of realizations grows fast with the coefficient wordlength. In [37] a simplified graph representation, referred to as vertex reduced representation, of the constant coefficient multipliers introduced in [24] was proposed. It was shown that several of the different graphs could be considered as the same case during the design process. In [42] methods to reduce the search time by exploring redundancies in the exhaustive search were presented. However, when realizing the multiplier one of the several possible cases must be selected. The presented model can then be used to determine which of the realizations that has the lowest hardware complexity.

Consider the coefficient 1717, which can be computed as

$$1717 = 11 \cdot 63 + 1024 = (1 + 2 + 8) \cdot (64 - 1) + 1024 \qquad (4.12)$$

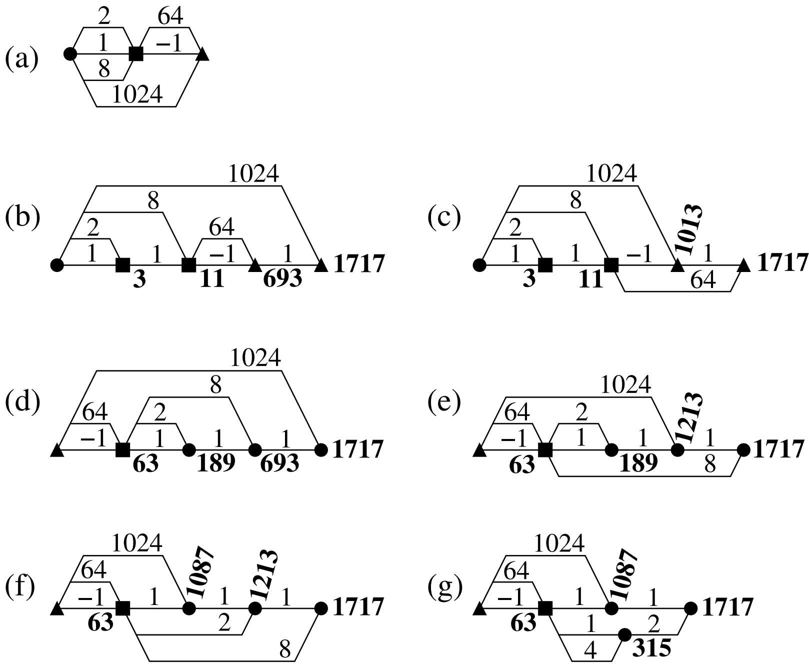


Figure 4.7 (a) Vertex reduced graph for the coefficient 1717. (b)–(g) Fully specified graphs for the coefficient 1717. The node symbols (● ■ ▲) indicate the relation between nodes in the vertex reduced graph and nodes in the fully specified graphs.

The corresponding simplified, i.e., vertex reduced, graph is shown in Fig. 4.7 (a). Note that there exist other vertex reduced graphs that can be used to realize the coefficient 1717, but none with lower adder cost than four. There also exist other possibilities using the same vertex reduced graph with different edge weights, for example

$$1717 = 13 \cdot 33 \cdot 4 + 1 = (1 + 4 + 8) \cdot (32 + 1) \cdot 4 + 1 \qquad (4.13)$$

However, in this example we will use the design first suggested. When realizing this multiplication, any of the six associated fully specified graphs shown in Figs. 4.7 (b)–(g) may be used. Note that the realizations in Figs. 4.7 (d)–(g) correspond to the transposed vertex reduced graph.

For the graphs in Figs. 4.7 (b), (c), (d), and (f) it is possible to reorder the additions corresponding to the weights 2 and 8. However, this will lead to an intermediate result with longer wordlength, and is therefore not considered here.

Realization	Adder depth	Input load	Maximum internal fan-out	GP count
Fig. 4.7 (b)	4	4	2	14
Fig. 4.7 (c)	4	4	2	12
Fig. 4.7 (d)	4	3	3	15
Fig. 4.7 (e)	4	3	3	14
Fig. 4.7 (f)	4	3	3	13
Fig. 4.7 (g)	3	3	3	12

Table 4.2 Properties for the multiplier graphs in Figs. 4.7 (b)–(g).

The adder depth, input load, and maximum internal fan-out for the different realizations are given in Table 4.2. The realization in Fig. 4.7 (g) has the lowest adder depth. The input load is higher for the nontransposed realizations, whereas the maximum internal fan-out is higher for the transposed graphs. Also given in Table 4.2, is the glitch path (GP) count [21]. This is a high-level estimation of the energy consumption, based on the fact that the switching activity will increase with the adder depth. The difference in GP count is too small to give a reliable estimate about the relative energy consumption. However, it indicates which realizations that are more likely to be a better choice. From the results in Table 4.2, it is not obvious which of the realizations that should be selected.

The final cost measure to be studied is the adder-bit cost, i.e., the complexity. The results in terms of overhead full adders are shown in Table 4.3, where the associated extra fundamental values are also given. If the sign transformation in Fig. 4.6 (a) is applied, the use of half adders can be eliminated, resulting in the coefficient –1717 for all realizations. Hence, the cost in half adders will be ignored.

To obtain the total number of full adders, the input wordlength, W_0, should be included for each adder. For example, the total number of full adders, $n_{tot,\,FA}$, for the realization in Fig. 4.7 (d) is

$$n_{tot,\,FA} = W_0 + (W_0 + 7) + (W_0 + 7) + (W_0 + 1) = 4W_0 + 15 \quad (4.14)$$

Assume that the input wordlength, W_0, is 16 bits. If the realizations in Figs. 4.7 (b) or (c) are used, $4 \cdot 16 + 7 = 71$ full adders are required. Using the realization in Fig. 4.7 (g), which has the lowest adder depth, 82

Realization	Fundamental 1		Fundamental 2		Fundamental 3		Output	Total
	Value	FA (HA)	Value	FA	Value	FA (HA)	FA	FA (HA)
(b)	3	1	11	1	693	4 (6)	1	7 (6)
(c)	3	1	11	1	1013	0 (10)	5	7 (10)
(d)	63	0 (6)	189	7	693	7	1	15 (6)
(e)	63	0 (6)	189	7	1213	1	8	16 (6)
(f)	63	0 (6)	1087	1	1213	10	8	19 (6)
(g)	63	0 (6)	1087	1	315	7	10	18 (6)

Table 4.3 Extra fundamental values and the associated number of overhead full adders, according to (4.10), for the multiplier graphs in Figs. 4.7 (b)–(g). The number of overhead half adders, according to (4.11), are in brackets.

$(4{\cdot}16 + 18)$ full adders are required. Hence, in this case the total number of full adders is decreased by more than 13% if the realization with lowest complexity is selected.

When the results from both Table 4.2 and Table 4.3 are considered, the realization in Fig. 4.7 (c) seems to be the best one, as it has the lowest GP count as well as a minimum number of full adder cells. As illustrated by this example, there are usually several possible realizations to choose from when implementing a single-constant multiplier, and, hence, the presented complexity model can be used to differentiate them.

4.2 Low Complexity Algorithm

In the same way as the algorithms discussed in Chapter 2, the algorithm presented here is also based on the RAG-n algorithm from [25]. The input to this algorithm is a set of coefficients, C, and the output is the total fundamental set, F, as illustrated in Fig. 1.13. Initially, the only available value in F is 1. All values f_i that can be obtained according to (1.17), i.e., from any pair, f_j and f_k, which are available in F, are removed from C and added to F. This procedure is iterated until either C is empty, or it is not possible to realize any more required coefficients. For the latter case, one or more extra fundamentals, that make it possible to realize one of the coefficients in C, are added to F. Then we start combining all available values according to (1.17) again.

The presented algorithm, referred to as the *n*-dimensional Reduced Full Adder Graph (RFAG-*n*) algorithm, only adds one coefficient at a time, the one which require the smallest number of overhead full adders. Furthermore, RFAG-*n* selects extra fundamentals leading to the smallest number of overhead full adders, instead of the smallest fundamental values.

The result is that RAG-*n* is more likely to reuse extra fundamentals, due to the selection of smaller values and by that reduce the number of wordlevel adders, while RFAG-*n* is more likely to reduce the number of overhead full adders.

4.2.1 Multiple-Constant Multiplication Example

Here, the 24th-order linear-phase FIR filter used for the example in [26] is considered, i.e., the same filter as was implemented in Section 2.6.2 and with the magnitude response illustrated in Fig. 2.38. The symmetric impulse response is $H = \{-710, 327, 505, 582, 398, -35, -499, -662, -266, 699, 1943, 2987, 3395, 2987, ...\}/2^{14}$. The set of unique positive odd integer coefficients to be realized contain 13 coefficients, and is $C = \{355, 327, 505, 291, 199, 35, 499, 331, 133, 699, 1943, 2987, 3395\}$. This is a well-known set of coefficients for which it is difficult to find a good solution. It has been shown that using three extra fundamentals is optimal in terms of adders. Such solutions have been found using, for example, the set $E = \{5, 83, 105\}$ obtained from H_{cub} [144] or the set $E = \{33, 51, 311\}$ found by DiffAG [43].

In Table 4.4, the results in terms of adders, and overhead full and half adder cells for various algorithms are given. The total number of full adders is computed using an input wordlength, W_0, of 16 bits. From this, it is clear that the presented algorithm provides the minimum number of full adder cells, although two of the other algorithms result in solutions with fewer wordlevel adders. The advantage of RFAG-*n* is the reduced number of overhead full adders, while the number of adders is kept low. Here, overhead full adders corresponding to almost two wordlevel adders are saved, compared to the second best algorithm, H_{cub}.

4.2.2 Results for Random Coefficient Sets

To study the properties of the proposed MCM algorithm, tests with random coefficients have been performed. For each combination of coeffi-

Algorithm	Wordlevel adders	Overhead full adders	Overhead half adders	Total full adders
Pasko [115]	23	68	68	436
BHM [25]	20	99	19	419
DA-MST [40]	19	88	22	392
C1 [26]	19	70	34	374
RAG-*n* [25]	17	74	7	346
DiffAG [43]	16	89	5	345
H_{cub} [144]	16	77	18	333
RFAG-*n*	17	48	57	320

Table 4.4 Complexity results for the FIR filter in [26].

cient wordlength and number of coefficients, 100 random coefficient sets have been used. The presented algorithm is compared with the RAG-*n* algorithm in [25]. This algorithm was during one decade, generally known for obtaining the best results in terms of adders. However, recently published algorithms have shown even better results [43],[144].

Considering sets of 25 coefficients and a varying coefficient wordlength, the results in Fig. 4.8 is obtained. From Fig. 4.8 (a) it is clear that the number of overhead full adders is reduced for RFAG-*n* compared to RAG-*n*. The difference in terms of adders is shown in Fig. 4.8 (b), where it can be seen that a small increase is obtained for longer coefficient wordlengths. This is because of the different strategies when extra fundamentals must be included, as discussed in Section 4.2. However, a more detailed analysis shows that for some cases RFAG-*n* actually requires fewer adders than RAG-*n*. Transforming the overhead full adders into wordlevel adders, it is clear from Fig. 4.8 (c) that the savings in overhead full adders more than enough compensates for the slight increase in wordlevel adders. The relative savings are shown in Fig. 4.8 (d).

Varying the coefficient setsize with a fixed coefficient wordlength of 10 bits, the results in Fig. 4.9 are obtained. From Fig. 4.9 (a) it can be seen that significant savings in overhead full adders are obtained for large coefficient sets. The reason for this is that the flexibility increases when more coefficients to choose from are available. This also result in that both algorithms are likely to be optimal in terms of adders for large sets,

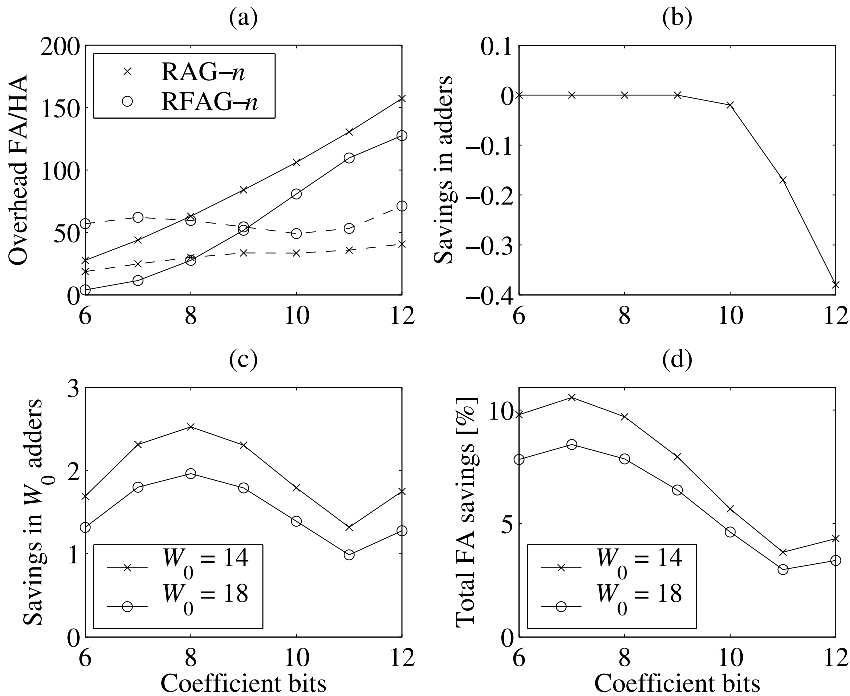


Figure 4.8 Comparison results for sets of 25 coefficients. (a) Number of overhead full (solid) and half (dashed) adders. (b) Average savings in adders using RFAG-*n* over RAG-*n*. (c) Corresponding savings in wordlevel adders considering both adders and overhead full adders. (d) Relative savings in full adder cells for RFAG-*n* over RAG-*n*.

which is clear from Fig. 4.9 (b). Finally, the possibility to reduce the complexity for large coefficient sets, by using the RFAG-*n* algorithm, is illustrated in Figs. 4.9 (c) and (d).

4.3 Interconnection Algorithms

Many proposed MCM algorithms only aim at minimizing the adder cost, i.e., the number of extra fundamentals. This corresponds to the second step of the design path illustrated in Fig. 1.13. However, little effort is put into the next step, i.e., to find a suitable interconnection graph, *G*, from the total set of fundamentals, *F*. Normally, a straightforward interconnection, i.e., directly corresponding to the way in which the MCM algorithm determine the solution, is selected without considering different possibilities. Furthermore, in many MCM algorithms the interconnection graph is

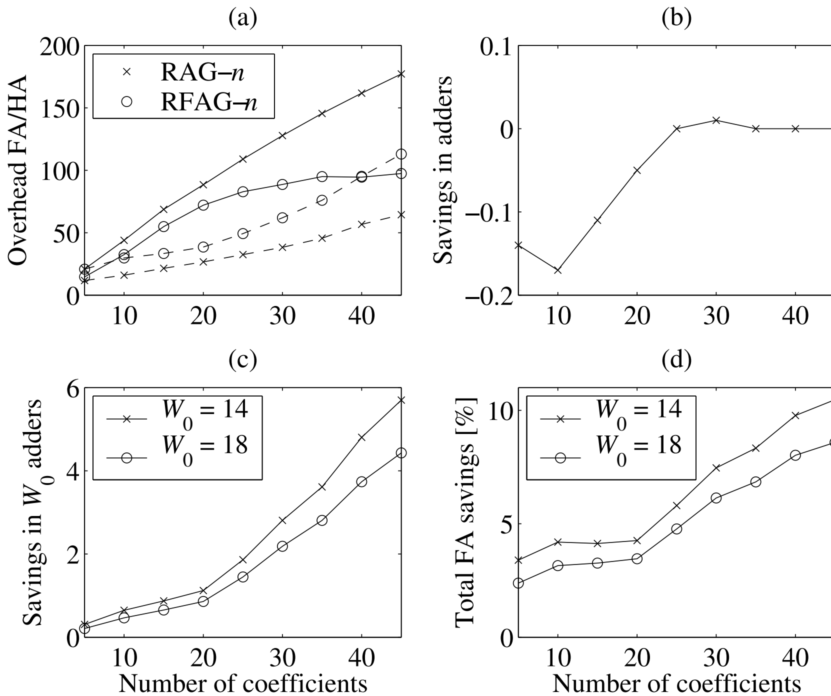


Figure 4.9 Comparison results for 10 bit coefficients. (a) Number of overhead full (solid) and half (dashed) adders. (b) Average savings in adders using RFAG-*n* over RAG-*n*. (c) Corresponding savings in wordlevel adders considering both adders and overhead full adders. (d) Relative savings in full adder cells for RFAG-*n* over RAG-*n*.

created during the search for extra fundamentals. However, it can be shown that it is preferable to perform the design steps in sequence as illustrated in Fig. 1.13, i.e., to separate the tasks of finding the fundamental set, *F*, and to find the interconnection graph, *G*. The reason is simply that it is easier to create a good interconnection when all information is available, i.e., after the complete MCM problem has been solved. Once the set of extra fundamentals, *E*, has been found, it is possible to obtain significant complexity improvements by selecting a beneficial interconnection. Here, the focus is on the second problem, i.e., how to build up the graph given all extra fundamentals. Hence, the presented methods can be used independently of which MCM algorithm that is used to solve the first problem, i.e., to find the set of extra fundamentals.

In the case of FIR filters, only a few extra fundamentals are often required due to the coefficients distribution where several small values are

Expression	e_j	f_j	e_k	f_k	Overhead	
					FA	HA
16 – 5	16	1	–1	5	0	4
8 + 3	8	1	1	3	1	0
4 + 7	4	1	1	7	2	0
4·3 – 1	4	3	–1	1	2	2
2·3 + 5	2	3	1	5	3	0
2·5 + 1	2	5	1	1	3	0
2·7 – 3	2	7	–1	3	3	1

Table 4.5 Complexity for different realizations of the coefficient 11.

normally included, which then can be used to realize coefficients that are more complicated. Many MCM algorithms are therefore likely to find an optimal solution in terms of adder cost. Hence, it is instead of interest to find the best interconnection since different realizations result in different area and energy consumption.

4.3.1 Algorithm Formulations

Given the set of coefficients, C, and extra fundamentals, E, there exist several possible interconnections with different costs in terms of overhead full and half adders. This is illustrated in the following example, where the coefficient 11 is to be realized in a graph that already includes the fundamentals 3, 5, and 7. The different solutions, all with the same adder cost, are given in Table 4.5. As can be seen, the number of overhead full adders, computed according to (4.10), varies from 0 to 3 depending on which interconnection that is selected. Hence, assuming that the sign can be compensated for, the coefficient should be implemented as 5 – 16, for which no overhead full adders are required and the half adders are eliminated.

Besides the complexity, adder depth has been shown to strongly affect the energy consumption [21]. The adder depth of a node is defined as the longest path from the input to that node, i.e., the number of cascaded adders. It is of interest to find an interconnection graph, G, where all

nodes have minimum depth. Given the total set of fundamentals, F, this is a straightforward task according to the following algorithm.

1. Initialize a graph G that only contains the input node.
2. Realize, i.e., add to G, all fundamentals $f_i \in F$ that can be obtained directly from the input using one addition (these are often referred to as cost-1's and includes all $2^n \pm 1$ values, for example, 3 and 129).
3. Add to G all fundamentals, $f_i \in F$, which can be obtained from the existing nodes in G, i.e., by adding/subtracting, according to (1.17), any two fundamentals (including the input) realized in preceding steps.
4. Repeat step 3 until all fundamentals in F have been realized.

This algorithm corresponds to the optimal part of the RAG-*n* algorithm introduced in [25], for which it is clear that an optimal adder cost is obtained if no extra fundamentals are needed.

By applying this algorithm, one level at a time is added to the graph, i.e., in step 2 all nodes will have depth 1, the first time step 3 is executed all realized nodes will have depth 2, the second time depth 3 and so on. This assures that all fundamentals are realized at a minimum depth, given F. Hence, the adder depth can, if possible, only be reduced by adding or changing extra fundamentals, i.e., by using a different MCM algorithm in the preceding design step of Fig. 1.13.

By also considering the complexity, i.e., to make sure that each fundamental is realized using a minimum number of overhead full adders computed according to (4.10), the complexity is optimized while the adder depth is still minimal. This is done for each fundamental by selecting the most beneficial among all possible interconnections, which is a limited number of combinations to evaluate since the available fundamentals at lower depths are fixed. Hence, the execution time of this algorithm is neglectable.

Another possible solution is to change the priorities between adder depth and complexity. This would result in the following algorithm.

1. Among the fundamentals which can be obtained from the previously realized nodes (just the input at the first iteration), only realize the fundamental f_i that require the fewest number of overhead full adders, according to (4.10). If no unique, select the smallest of the ones with lowest full adder cost.
2. Repeat step 1 until all fundamentals have been realized.

Strategy	Description
Original	The interconnection found by a certain MCM algorithm is used directly without any improvement.
No HA	All half adders are eliminated using the sign transformation.
Min. depth	All nodes in the graph, G, are assured to have minimum depth, given F. For each fundamental, the interconnection requiring the lowest number of overhead full adders is selected.
Opt. FA	Only one fundamental at a time is realized in this algorithm, aiming at minimizing the number of overhead full adders.

Table 4.6 Description of the different interconnection strategies.

The idea used here is the same as for the RFAG-n algorithm presented in Section 4.2, where it was shown that significant complexity savings can be obtained, especially for large coefficient sets. Hence, using this algorithm to obtain the interconnection for a given set of fundamentals, F, will decrease the number of overhead full adders. However, the adder depth might increase significantly.

The different interconnection strategies discussed in this section are summarized in Table 4.6.

4.3.2 Implementation Examples

In this section, two FIR filters are implemented using the previously discussed interconnection strategies. The filters are realized using the transposed direct form structure shown in Fig. 1.1 (b), and implemented by logic synthesis of VHDL code. A Xilinx Virtex-II and a 0.35 µm CMOS standard cell library are used for FPGA and ASIC implementations, respectively.

Area and speed results are given for both the FPGA and ASIC implementations. Furthermore, for the ASIC designs, power consumption results, obtained using NanoSim™ with 1000 random input samples and a clock frequency of 10 MHz, are also provided.

Synthesis and energy figures are given both for the complete FIR filter, as well as specifically for the multiplier block, which is the most interesting part in this study. However, the structural adders, i.e., the adders in the delay chain, are also slightly affected by the interconnection algorithm,

since the signs may change. Furthermore, if the adder depth is changed, this will effect the energy consumption for the structural adders.

Note that the only decision made by the interconnection algorithm is how to connect the nodes, i.e., the set, F, of fundamental values is exactly the same as was found by the used MCM algorithm.

Example 1

Here, the same 24th-order FIR filter as was used for the example in Section 4.2.1 is considered. The input data wordlength, W_0, is selected to be 16 bits.

In Table 4.7, the results in terms of adders, and overhead full and half adder cells for the H_{cub} [144] and Pasko [115] algorithms are shown. From this, it is clear that the proposed interconnection strategies provide improvements. However, for the solution found by H_{cub}, only the elimination of half-adders can be done as the different interconnection possibilities are extremely limited due to the highly optimized number of adders. This is also the reason for the high adder depth. In fact, 5 is the only fundamental that is realized at depth one. For the Pasko algorithm, on the other hand, except to remove the half adders, it is also possible to reduce both the number of overhead full adders and the adder depth. Two of the coefficients, 199 and 699, could here be realized with a lower depth. The average depth is given both for all MCM adders, F, and for all filter coefficients, H. The former is related to the energy consumed within the MCM block, while the latter will affect the energy consumed in the structural adders.

The high-level estimation of energy consumption, GP count [21], is also given in Table 4.7. According to this measure, it is clear that the designs with high depth are expected to consume more energy. The reason that the GP count is not reduced for the minimum depth strategy, is that the two coefficients 199 and 699 now are realized from two low depth fundamentals while before they were realized from one high depth fundamental and the input, which result in the same GP value.

For ASIC implementations, the area, throughput, and energy consumption results for the different multiplier block designs are given in Table 4.8. The area is reduced for both MCM algorithms if the half adders are eliminated. The area can be further reduced for the Pasko algorithm by taking the number of overhead full adders into consideration when the interconnection is selected.

Algorithm	Strategy	Adder cost	Overhead		Adder depth			GP
			FA	HA	F (avg.)	H (avg.)	Max.	
H_{cub}	Original	16	77	18	4.1875	4.4800	7	119
H_{cub}	No HA	16	77	(18)	4.1875	4.4800	7	119
Pasko	Original	23	68	68	2.3043	2.8000	4	60
Pasko	No HA	23	68	(68)	2.3043	2.8000	4	60
Pasko	Min. depth	23	50	(76)	2.2174	2.6400	4	60
Pasko	Opt. FA	23	48	(79)	2.6957	3.1600	5	71

Table 4.7 Complexity results for the MCM blocks in Example 1.

Algorithm	Strategy	Area [mm^2]	Sample rate [MSa/s]	Energy [nJ]
H_{cub}	Original	0.1048	49.75	0.4576
H_{cub}	No HA	0.1026	49.60	0.4495
Pasko	Original	0.1315	59.74	0.3829
Pasko	No HA	0.1226	64.06	0.3578
Pasko	Min. depth	0.1208	59.52	0.3492
Pasko	Opt. FA	0.1255	51.10	0.4189

Table 4.8 Implementation results for the MCM blocks in Example 1.

Results for both FPGA and ASIC implementations of the total FIR filters are given in Table 4.9. Similar to what was noted for the ASIC area of the MCM blocks, the number of CLB slices is reduced when the half adders are removed. There are more structural adders, i.e., adders in the delay chain, than MCM adders. Furthermore, the delay elements also occupy a large part of the area. Hence, the total area is significantly larger than the MCM area. Note that the absolute value of the area improvements are almost the same for the total FIR filters as for the MCM parts, i.e., the area of the structural adders is not significantly affected.

The H_{cub} implementations have a lower sample rate due to higher adder depth. However, the critical path is usually only increased by one

Algorithm	Strategy	FPGA		ASIC		
		CLB Slices	Sample rate [MSa/s]	Area [mm^2]	Sample rate [MSa/s]	Energy [nJ]
H_{cub}	Original	969	36.06	0.5561	35.74	1.8799
H_{cub}	No HA	963	37.22	0.5532	34.75	1.8321
Pasko	Original	1074	40.57	0.5836	40.45	1.6580
Pasko	No HA	1050	42.03	0.5773	44.35	1.6470
Pasko	Min. depth	1036	39.79	0.5721	43.92	1.5989
Pasko	Opt. FA	1037	39.45	0.5773	41.98	1.7220

Table 4.9 Implementation results for the total FIR filters in Example 1.

full adder for each extra cascaded wordlevel adder, and, hence, the difference in sample rate is not as significant as in adder depth.

Although the H_{cub} algorithm results in more than 30% fewer MCM adders than the Pasko algorithm, the energy consumption of the MCM part is almost 20% higher for the original designs. This further illustrates the importance of interconnection, i.e., from an energy consumption point of view it may be beneficial to have more adders if the interconnection then can be done more efficient. According to Table 4.8, the lowest energy consumption is obtained for the Pasko algorithm using the minimum depth strategy, for which the energy of the MCM part is reduced by 8.8% compared to the original Pasko design. Again, note that this saving is obtained just by changing the interconnection, i.e., both designs realize exactly the same fundamental set, *F*.

Considering the difference in energy consumption for the MCM part and the complete FIR filter, given in Tables 4.8 and 4.9, respectively, it is clear that the structural adders consume more energy for the optimized FA case, while the energy is reduced for the minimum depth implementation. For the former, the adder depth is increased, while for the latter, it is decreased. Hence, the adder depth also effects the energy consumption of the structural adders.

When comparing the four different strategies used for the Pasko algorithm, it is clear that adder depth affects the energy consumption more than full and half adder overhead. However, reducing the complexity overhead also decreases the area and energy consumption.

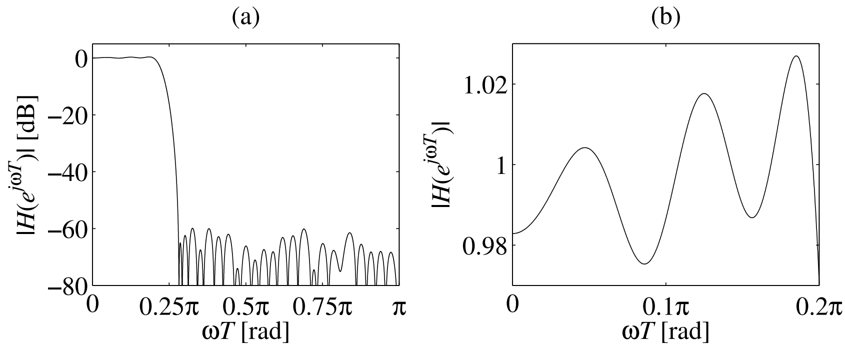


Figure 4.10 (a) Magnitude response for the filter from [87]. (b) Passband.

Example 2

In this example, an FIR filter from [87] of order 62 is used. The magnitude response is shown in Fig. 4.10. The data wordlength, W_0, is here set to 12 bits. The symmetric impulse response is $H = \{3, 6, 8, 7, 1, –9, –19, –24, –20, –5, 15, 31, 33, 16, –15, –46, –59, –42, 4, 61, 99, 92, 29, –71, –164, –195, –119, 74, 351, 642, 862, 944, 862, ...\}/2^{12}$, which gives the set of unique positive odd integer coefficients as $C = \{3, 7, 9, 19, 5, 15, 31, 33, 23, 59, 21, 61, 99, 29, 71, 41, 195, 119, 37, 351, 321, 431\}$. Opposite to the previous example, this is a simple set and most algorithms would find a solution without requiring any extra fundamentals, i.e., only one adder is used for each coefficient. Since E is an empty set, no MCM algorithm is actually needed for this filter, and, hence, the only problem is to find a suitable interconnection.

In Table 4.10, the complexity and adder depth results are given using the different strategies. Here, the RAG-*n* algorithm [25], which for this simple case gives a minimum depth solution, has been used as the original strategy. Hence, the first three strategies result in exactly the same adder depth. For these solutions that are optimal in terms of adders, the maximum adder depth is three. However, the only depth 3 coefficient, 431, can be realized at depth 2 by increasing the adder cost and including any of the extra fundamentals 17, 65, or 511, as will be seen in Section 4.4.4. For the optimized FA case, where the complexity is given priority, the adder depth is increased.

Still with minimum adder depth, the number of overhead full adders can be reduced by more than 50%. If a higher depth is allowed, almost 70% of the overhead full adders can be eliminated. From Tables 4.10 and 4.11 it is clear that the number of CLB slices is closely related to the total

Strategy	Adder cost	Overhead		Total (adders + FA + HA)	Adder depth			GP
		FA	HA		F (avg.)	H (avg.)	Max.	
Original	22	46	36	346	1.7273	1.4286	3	42
No HA	22	46	(36)	310	1.7273	1.4286	3	42
Min. depth	22	22	(54)	286	1.7273	1.4286	3	42
Opt. FA	22	14	(84)	278	2.4091	1.9841	4	57

Table 4.10 Complexity results for the MCM blocks in Example 2.

Strategy	FPGA		ASIC		
	CLB Slices	Sample rate [MSa/s]	Area [mm^2]	Sample rate [MSa/s]	Energy [nJ]
Original	301	47.87	0.0841	97.94	0.1658
No HA	266	63.13	0.0802	98.33	0.1637
Min. depth	244	59.84	0.0733	93.20	0.1633
Opt. FA	241	55.22	0.0834	77.04	0.2232

Table 4.11 Implementation results for the MCM blocks in Example 2.

complexity. Hence, the number of CLB slices is decreased when the complexity overhead is reduced. At most, a 20% reduction of CLB slices is obtained for the MCM part.

Area results for the complete FIR filters are presented in Table 4.12. Due to the large filter order, the structural adders and the delay elements will be the dominating part of the implementation, corresponding to more than 90% of the area. Again, the difference in ASIC area is similar for the MCM blocks and the total FIR filters, i.e., the area of the structural adders does not depend on the interconnection.

In most cases, the area required for the ASIC implementations agree with the FPGA results. However, the ASIC area is more difficult to predict as it depends on the selection of standard cell driving strengths. For example, the ASIC implementation of the design with the smallest complexity, i.e., the optimized FA case, has a larger area than expected.

Strategy	FPGA		ASIC		
	CLB Slices	Sample rate [MSa/s]	Area [mm^2]	Sample rate [MSa/s]	Energy [nJ]
Original	1519	46.13	0.9557	46.51	2.2225
No HA	1495	47.30	0.9531	49.48	2.2455
Min. depth	1484	46.95	0.9453	49.48	2.2503
Opt. FA	1479	46.95	0.9558	41.08	2.4411

Table 4.12 Implementation results for the total FIR filters in Example 2.

There are no significant differences in sample rate, except that the ASIC implementation of the optimized FA case, which has a larger adder depth, is slightly slower.

The energy consumption results for the MCM part, as given in Table 4.11, might seem reasonable at a first glance, i.e., the energy is reduced for decreased overhead and increased with adder depth. However, when comparing this with the previous Pasko example, the gain obtained by removing the half adders is about ten times smaller. The suspicion that something is wrong, becomes stronger when considering the energy consumption for the complete FIR filter given in Table 4.12. The original design actually gives the best result, despite larger complexity. To be able to draw, and prove, any general conclusions, this should be investigated in more detail. However, some likely reasons are discussed in the following.

When the number of overhead full and half adders is decreased, some adders are replaced by subtractors in the MCM block. For the three modified implementations, half of the 22 operations in the MCM block are subtractions, while only 1/3 for the original design. Hence, one reason for the higher energy consumption is that subtractors reduce the correlation between signals, which in turn give rise to increased switching activity. Note that the increased number of subtractions can not be generalized as a common effect of the sign transformation used to remove half adders, i.e., the probability for increasing or decreasing the number of subtractions is the same. However, in this specific case, the number of subtractors was significantly increased.

Another part of the explanation, is that when half adders are removed, the timing of signals will be skewed, i.e., the least significant bits arrives

earlier to subsequent additions. This will also increase the switching activity, and, hence, also the energy consumption.

Hence, these small modifications in the MCM block, which in fact decrease the energy consumed in that part exactly as desired, will have a significant impact on the large number of structural adders. The fact that the MCM part strongly affects the structural adders, is clear when considering the optimized FA case, for which the structural adders consume much more energy due to the increased adder depth in the MCM block.

The main conclusion is that it again is shown that the energy consumption strongly depends on the adder depth. The design with higher depth, clearly has the largest energy consumption although the number of overhead full adders is low.

4.4 Minimum Adder Depth Algorithm

As discussed in Section 1.4.2, all algorithms for the multiple-constant multiplication (MCM) problem are based on either subexpression sharing, adder graphs, or difference methods. It is sometimes claimed that graph based MCM algorithms result in high adder depth, i.e., more cascaded adders compared to other types of algorithms. However, here a graph based minimum depth MCM algorithm is presented.

As mentioned before, adder depth is probably the most important factor when designing low energy multiplier blocks. Therefore, the main requirement for this MCM algorithm is that all coefficients must be realized at minimum depth. Hence, only fundamental pairs $\{f_j, f_k\}$, as defined by (1.17), from which f_i can be obtained at the absolute minimum depth are considered. Note the difference in finding a minimum depth graph, G, given the fundamental set F, as discussed in Section 4.3.1, and finding a fundamental set, F, given the coefficient set C, such that all coefficients are realized at its theoretically minimum depth, which is considered here.

As defined by (1.20), the minimum depth for a fundamental, f_i, is directly related to the number of nonzero digits, $S(f_i)$, in any MSD representation of f_i [45].

4.4.1 Fundamental Pairs

Considering the five different realizations illustrated in Fig. 4.2 to obtain a fundamental, f_i, a table containing all possible pairs of fundamentals from which each coefficient can be realized at minimum depth can be cre-

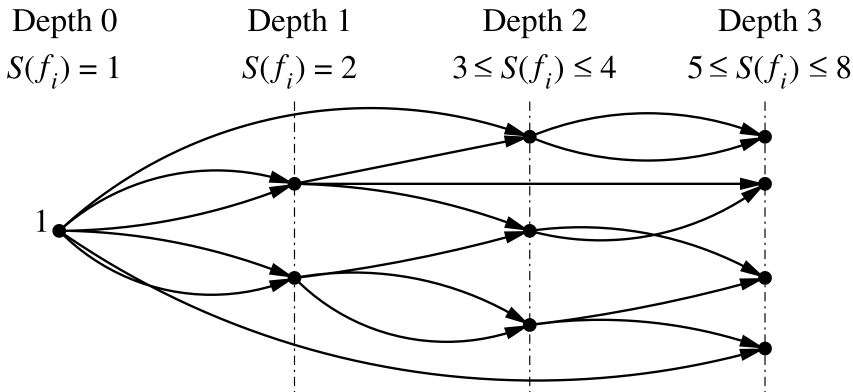


Figure 4.11 Each fundamental is obtained from two lower depth values.

Coefficient										
3	5	7	9	...	215	217	219	221	...	255
1 1	1 1	1 1	1 1		5 255	7 7	5 7	3 7		1 1
					7 9	31 31	9 255	17 255		
					31 33		33 63	31 63		
								33 127		

Table 4.13 Look-up table with fundamental pairs for $N = 8$.

ated. The straightforward way to do this is illustrated in Fig. 4.11, where all depth 1 coefficients are first obtained directly from the input node, then all depth 2 coefficients uses at least one depth 1 fundamental and so on. During this search, all fundamental pairs that result in a minimum depth realization of a certain coefficient are stored. Hence, all combinations are tested to obtain a complete set of pairs, which is required to enable selection of the most suitable pair depending on available fundamentals. A part of such a look-up table is given in Table 4.13. For example, the coefficient 221 can be computed from the pair {31, 63} as $4 \cdot 63 - 31$. How the coefficient is realized from a specific pair does not need to be stored since this is straightforward according to the algorithm in Section 4.3.1, i.e., it is enough to know that the coefficient can be obtained from that pair in some way. Again, the final interconnection will not be decided until all extra fundamentals have been found.

Depth	Number of coefficients realized at different depths ($N = 6, ..., 14$)					Average number of possible fundamental pairs ($N = 6, ..., 14$)				
	6	8	10	12	14	6	8	10	12	14
1	9	13	17	21	25	1.0	1.0	1.0	1.0	1.0
2	22	106	318	722	1382	11.4	8.6	6.9	5.9	5.3
3	–	8	176	1304	6784	–	255.8	628.8	949.5	1029.0

Table 4.14 Fundamental pairs statistics.

Depth	Pairs in total	Interconnection graph as defined in Fig. 4.2					Percentage sum
		(a)	(b)	(c)	(d)	(e)	
1	21	52.38%	0	52.38%	0	0	104.76%
2	4 277	47.35%	15.01%	34.98%	2.08%	1.68%	101.10%
3	1 238 161	35.49%	16.79%	34.37%	8.38%	5.06%	100.10%

Table 4.15 Distribution of the fundamental pairs, among the five realization cases on the form $f_i = e_j f_j + e_k f_k$ illustrated in Fig. 4.2, for $N = 12$.

Some statistics of the obtained look-up tables are presented in Table 4.14. Here, N is the number of coefficient bits, i.e., the largest odd coefficient value is $2^N - 1$. For example, for 6 bit coefficients, 9 out of the 31 odd values 3, 5, ..., 63 can be obtained from {1, 1}, while there are between 8 and 17, with an average of 11.4, different fundamental pairs from which the other 22 coefficients can be obtained at depth 2.

Table 4.15 gives the number of fundamental pairs realized using each of the different interconnections, as illustrated in Fig. 4.2. The number of coefficient bits is here set to twelve, but the percentage distribution is similar for other wordlengths. Case (b) and (c) can in practice be interchanged resulting in the opposite sign at the adder output, which was the property that enabled removal of half adders as discussed in Section 4.1.3. However, only realizations that give a positive coefficient have been considered here. Note that the right shift cases are becoming more useful at higher depth. The depth 1 coefficient 3 can be realized either as 1 + 2 or 4 – 1, corresponding to case (a) and (c), respectively. Hence, the same fundamental pair, in this case {1, 1}, can sometimes be used in different realization cases to obtain the same coefficient, leading to over 100% cov-

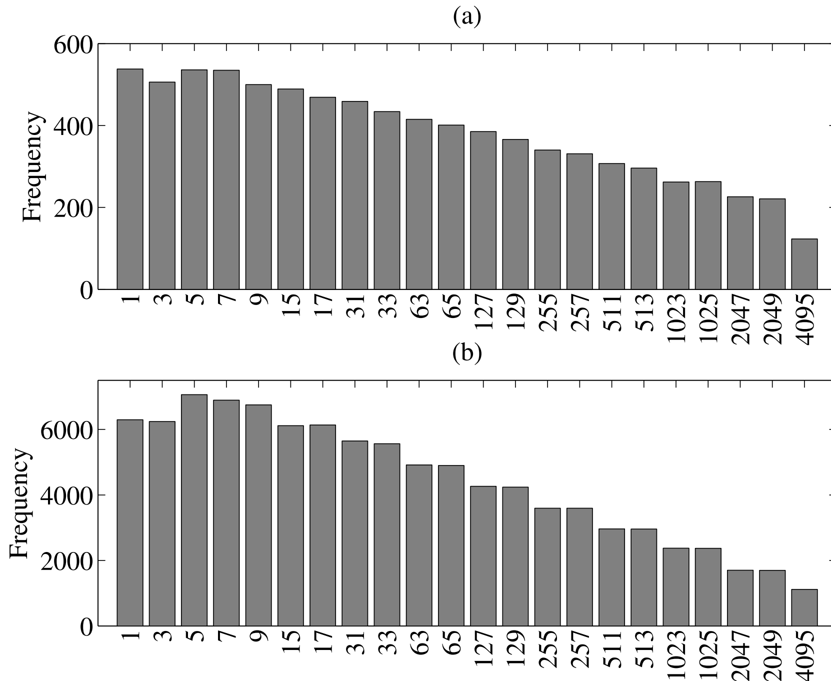


Figure 4.12 Number of pairs that include each depth 1 fundamental. (a) Depth 2 coefficients, and (b) depth 3 coefficients, for $N = 12$.

erage as shown in Table 4.15. However, this kind of overlap is relatively rare for depth 2 and 3 coefficients. For $N = 12$, there are 47 such cases at depth 2 and 1244 cases at depth 3. For example, the depth 2 coefficient 11 can be obtained from {1, 3} both as $8 + 3$ and $4 \cdot 3 - 1$, and also from {1, 5} as $1 + 2 \cdot 5$ or $16 - 5$. The smallest depth 3 coefficient is 171, which can be realized in three different ways from {57, 285}, either as $285 - 2 \cdot 57$, $8 \cdot 57 - 285$, or $2^{-1} \cdot 57 + 2^{-1} \cdot 285$.

Naturally, small values are more likely to be included in fundamental pairs, which is used in many MCM algorithms when selecting fundamentals. How common the different depth 1 fundamentals are when realizing depth 2 and 3 coefficients is illustrated in Fig. 4.12. Note that the fundamentals 5, 7, and 9 are included in more pairs than 3, when creating depth 3 coefficients.

$\{f_j, f_k\}$		3 9	5 5	9 33	7 17	15 15	5 65	3 63	31 31	33 63	3 513
c_1	75	1	1	1	1	1	1	1	–	–	–
c_2	465	–	–	–	1	1	–	–	1	1	1

Table 4.16 Covering problem for the coefficients 75 and 465.

4.4.2 MCM Defined as a Covering Problem

The MCM problem can be described as a covering problem (CP) where each row of the covering matrix [18] corresponds to one coefficient and each column corresponds to a fundamental pair. The idea is then to cover all rows by selecting as few columns as possible. An example of this is shown in Table 4.16 for the coefficient set $C = \{75, 465\}$. For this case, it is clear that either of the columns {7, 17} or {15, 15} will solve the CP matrix. However, the cost for these two solutions is different since {7, 17} would require two additional adders while {15, 15} only requires one. Hence, the best choice is to realize 75 as 15·(4 + 1) and 465 as 15·(32 – 1), with a total cost of 3 adders.

In the previous example, both coefficients can be realized at depth 2. If a higher depth realization is required, the fundamental pairs will include values that can not be realized directly from the input. A simple way to handle this would be to solve one depth at a time, starting at the highest. However, this would not be efficient as not all information about lower depth requirements is then available. Instead, implication constraints can be used. For example, consider the coefficient 4875, which can be obtained from 75 as 75·(64 + 1). The constraint that 75 must be available to realize 4875 in this way can be described by

$$\{75, 75\} \rightarrow \{3, 9\} \vee \{5, 5\} \vee \{9, 33\} \vee \{7, 17\} \vee \ldots \vee \{3, 63\} \quad (4.15)$$

which in the CP matrix correspond to that {75, 75} is not selected (marked by a zero) or one of the possible fundamental pairs for 75 is selected, as illustrated in Table 4.17.

4.4.3 Optimal Approach

If the cost for each fundamental pair is accurately defined and the CP is solved with a minimum cost, an optimal solution is obtained. However,

$\{f_j, f_k\}$		...	75 75	75 4575	3 9	5 5	9 33	7 17	15 15	5 65	3 63	...
c_1	4875	...	1	1	–	–	–	–	–	–	–	...
f_1	75	...	0	–	1	1	1	1	1	1	1	...
f_1	75	...	–	0	1	1	1	1	1	1	1	...

Table 4.17 A part of the CP matrix for the coefficient 4875, using implication constraints.

this is a difficult task, especially since the columns depend on each other, i.e., by selecting one column the cost in terms of adders for other columns might change. Furthermore, for certain coefficient sets there will be many implication constraints. To solve such a complicated CP would be very time consuming, and in many cases probably not even possible.

The idea here is to use a modified CP matrix, instead of having each column representing a fundamental pair, it represent a single fundamental. This will simplify the cost definitions, since setting a column to one, i.e., to include a new fundamental, always will require exactly one more adder. However, this requires a dynamic CP matrix as a new row, and possibly some columns, might be needed if a fundamental at depth two or higher is selected. Hence, the actual adder cost for a single fundamental can still turn out to be larger than one. Using this modified CP matrix together with a standard branch-and-bound algorithm [52], an MCM algorithm that gives optimal solutions in terms of adders, given that all coefficients are realized at minimum depth, can be constructed.

The matrix corresponding to the previous small example is given in Table 4.18. Again, it can be seen that the matrix is solved by selecting the fundamental 15. As these single fundamental matrices will be rather sparse, and might even contain rows without any ones, a good second choice method to select a column to branch on is required. Therefore, another matrix is created where the occurrence of each fundamental is registered, i.e., the number of pairs in which the fundamental is included, see Table 4.19 for the example matrix. There will be more possibilities in the CP matrix for larger coefficient sets as all coefficients are available, i.e., all fundamental pairs that include a coefficient will at most only require one extra fundamental. Note that the matrices are complemented in the sense that the included matrix is empty where the CP matrix has ones, since the number of pairs in which a certain fundamental is included

f_i		3	5	7	9	15	17	31	33	63	65	513
c_1	75	–	1	–	–	1	–	–	–	–	–	–
c_2	465	–	–	–	–	1	–	1	–	–	–	–

Table 4.18 Single fundamental covering problem matrix.

f_i		3	5	7	9	15	17	31	33	63	65	513
c_1	75	2	–	1	2	–	1	0	1	1	1	0
c_2	465	1	0	1	0	–	1	–	1	1	0	1

Table 4.19 Fundamental included matrix.

is not of interest if it solves that coefficient. Hence, using a suitable format, the memory required to store these matrices may be halved.

Using these two matrices, the proposed optimal algorithm is as follows:

1. Initialize the fundamental set, F, to be equal to the coefficient set, C.
2. Remove depth 1 fundamentals from the coefficient set, C.
3. For all remaining coefficients, c_i, obtain the list of fundamental pairs, $f_{list}(c_i)$, from the look-up table created as described in Section 4.4.1, and set all occurrences of values available in F to be free in $f_{list}(c_i)$.
4. For each coefficient c_i, check if any pair in $f_{list}(c_i)$ is free, if so remove c_i from C.
5. For each remaining coefficient c_i, add one row both in the single fundamental CP matrix and the fundamental included matrix, using the information in $f_{list}(c_i)$.
6. Compute a lower bound of the cost, given the current fundamental set, F. Check if the lower bound is not smaller than the cost of the best solution found so far. Otherwise, check if a terminal case is obtained, i.e., if the problem is solved or infeasible. If any of these conditions are true, this path of the binary search tree can be cut.
7. Find the column with most ones, i.e., corresponding to the fundamental f_i that solves most coefficients. If more than one column with maximum number of ones or no ones at all in the CP matrix, use the included matrix by computing the sum of these columns and select the

one with largest value, i.e., the column that will add most ones to the updated CP matrix. If still more than one candidate, select the smallest fundamental f_i.

8. Branch on the column corresponding to f_i. Add f_i to F and remove any solved coefficients c_i from C. If f_i can not be realized at minimum depth from the values available in F, then f_i must be added to C. Update the fundamental lists and the two matrices, then make a recursive call to step 6. If the cost of the obtained solution is smaller than the best so far, then make this the new best cost. If also equal to the lower bound, this path can be cut. Otherwise, exclude f_i as a possible fundamental. Update the fundamental lists and the two matrices, then make a recursive call to step 6. If the cost of the obtained solution is smaller than the best so far, then make this the new best cost.

9. The algorithm will exit when all relevant paths of the binary search tree have been explored, and return an optimal fundamental set, F.

This algorithm will be referred to as the minimum adder depth branch-and-bound (MADbb) algorithm.

Size of the Look-Up Table

Since, the MADbb algorithm is optimal, it will theoretically give better results if larger look-up tables are used, i.e., tables that include fundamentals of larger wordlength than the largest coefficient. However, as can be seen in Tables 4.20 and 4.21, the gain in terms of adders is small when using tables with one extra bit. Note that a reduction of 0.01 in average adder cost means that one adder is saved in one out of the 100 coefficient sets. Furthermore, no additional reduction is obtained when using tables including fundamentals with a wordlength of two more bits than the coefficients.

Execution Time

The MATLAB code description of the algorithm is executed on an Intel® Core™2 Quad (Q6600) processor with a clock speed of 2.40 GHz.

From Fig. 4.13, it is clear that the execution time increases when tables including fundamentals of larger wordlength are used. Furthermore, as always in MCM problems, it is in most cases simple to find an optimal solution for large coefficient sets, due to the number of available node values that often eliminate the use of extra fundamentals. The execution time increases drastically with the coefficient wordlength. In fact, for

Extra bits	Coefficient bits					
	6	7	8	9	10	11
0	15.33	19.16	23.14	26.12	28.27	29.83
1	0	0	0	–0.01	–0.01	0
2	0	0	0	–0.01	–0.01	0

Table 4.20 Change in average adder cost for MADbb, when larger tables are used, for different coefficient wordlengths and sets of 25 coefficients.

Extra bits	Number of coefficients								
	5	10	15	20	25	30	35	40	45
0	7.82	13.67	18.90	23.40	28.28	32.59	37.14	41.78	45.95
1	–0.01	–0.01	–0.01	0	–0.01	0	–0.01	–0.01	0
2	–0.01	–0.01	–0.01	0	–0.01	0	–0.01	–0.01	0

Table 4.21 Change in average adder cost for MADbb, when larger tables are used, for different coefficient setsize and 10 bit coefficients.

many cases using 12 bit coefficients, no guaranteed optimal solution was found in reasonable time. Therefore, a simple heuristic is presented in the next section.

4.4.4 Heuristic Approaches

The algorithm presented here gives the solution that is first found by the optimal approach, i.e., a sequence of columns is selected without any recursion. Step 1 to 5 are exactly the same as for the MADbb algorithm, then the proposed heuristic algorithm is as follows.

6. Select a column, f_i, based on the CP and included matrices, in the same way as in step 7 of the MADbb algorithm.
7. Add f_i to F and remove any solved coefficients c_i from C. If f_i can not be realized at minimum depth from the values available in F, then f_i must be added to C.
8. Exit if C is empty and return the found set F. Otherwise, update the fundamental lists and the two matrices, then go to step 6.

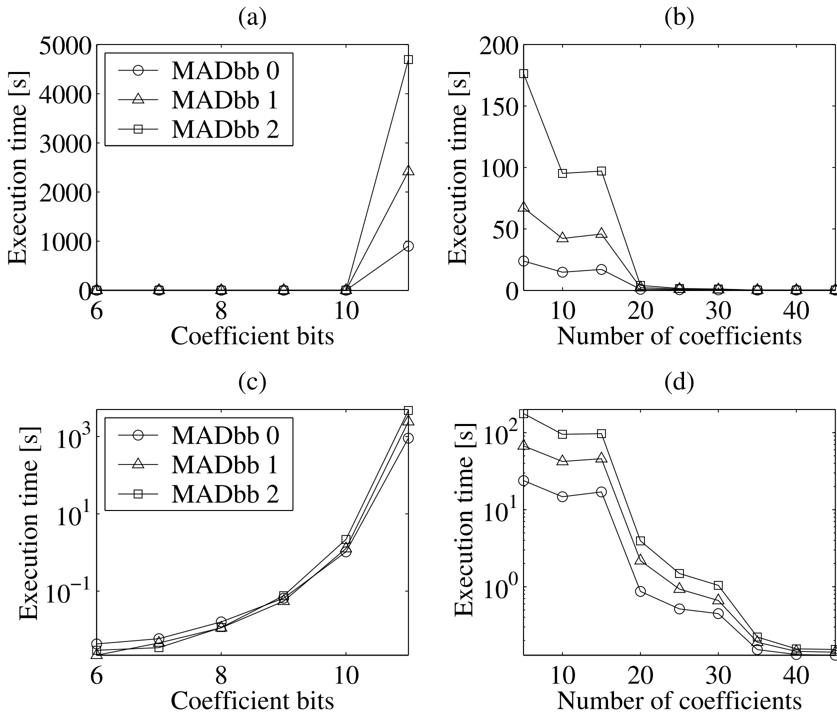


Figure 4.13 Execution time using look-up tables of various size. (a) Different wordlength using sets of 25 coefficients, and (b) different setsize using 10 bit coefficients. (c) Same as (a) in logarithmic scale. (d) Same as (b) in logarithmic scale.

In this algorithm, the best column is selected first. Another possible alternative is to find the extra fundamentals, f_i, based on selecting the most difficult row first. Step 6 would then be described as follows:

6. Find the row with fewest ones, i.e., corresponding to the coefficient, c_i, which is solved by fewest fundamentals. If it is enough to add one fundamental, select the one that solves most other rows as well. Otherwise, i.e., there are no ones on the c_i row of the CP matrix, consider all fundamental pairs in $f_{list}(c_i)$, and select the pair that solves most other coefficients as well. If in any case more than one candidate when selecting c_i or f_i, first make a decision based on the included matrix, and then based on the smallest values.

Note that f_i here can be a set of two fundamentals. It will be shown that these two strategies give the best solution in a varied way, which is not surprising since both are heuristics. Note however that they are still opti-

Coefficient						
57			155	189		431
1 7	3 63	9 129	5 5	3 3	33 255	5 511
1 65	5 17	15 63	7 127	15 129	63 63	7 17
3 9	5 31	15 129	31 31	17 257	65 127	31 65
3 15	9 15			31 65		
3 33	9 33			31 127		

Table 4.22 Pairs found in the look-up table for the example coefficients.

mal in the sense that all coefficients are guaranteed to be realized at a minimum depth. The former, where the best column is considered, is referred to as MADc, while the latter, considering the most difficult row, is referred to as MADr.

Example

The difference between MADc and MADr is explained in the following example. Consider the set $H = \{155, 756, 862, 611, 912\}$, which gives the odd coefficient set $C = \{155, 189, 431, 611, 57\}$.

In the initialization phase it is found that the depth 3 coefficient, 611, can be realized from 630 different pairs, of which one pair is composed by the two depth 2 coefficients, 57 and 155, since $8{\cdot}57 + 155 = 611$. Furthermore, 189 and 431 are also depth 2 coefficients, as illustrated in Fig. 4.14 (a). All possible pairs for the four depth 2 coefficients are given in Table 4.22.

From the list of fundamental pairs, the CP matrix can be created as shown in Table 4.23. First, considering the MADc algorithm, there are six fundamentals that may be included to solve one of the coefficients. From the included matrix in Table 4.24, it is clear that both 3 and 31 are part of four pairs. Since 3 is smaller it is selected. Note that this result in four new ones on the row for 57 in the updated CP matrix given in Table 4.25. Here, the row for the solved coefficient 189 is removed, as well as the columns for the included fundamental 3 and the fundamentals 255 and 257, which were only included in pairs for the solved coefficient. Again, no single fundamental will solve more than one coefficient, and by studying the included matrix in Table 4.26, the fundamental 5 is selected. This will result in three new ones in the updated CP matrix, shown in Table 4.27.

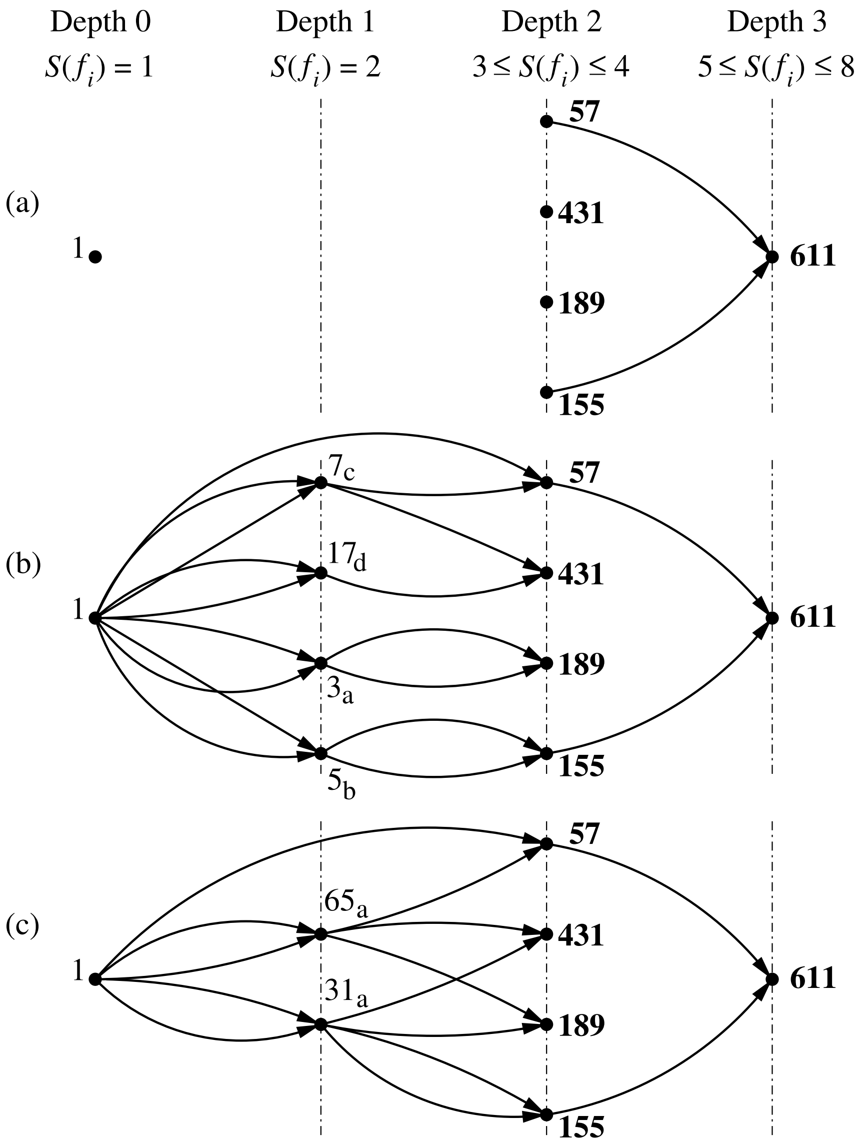


Figure 4.14 Graphs for the example. (a) Initialization, (b) MADc solution, and (c) MADr solution. The letter subscripts indicate the order in which extra fundamentals are added.

Among the four most promising fundamentals given in Table 4.28, 7 is selected since it is the smallest. The final CP and included matrices are shown in Tables 4.29 and 4.30, respectively. To solve the last coefficient, 431, the fundamental 17 is included. The complete solution found by the MADc algorithm is shown in Fig. 4.14 (b), where the extra fundamentals

f_i	3	5	7	9	15	17	31	33	63	65	127	129	255	257	511
57	–	–	1	–	–	–	–	–	–	1	–	–	–	–	–
155	–	1	–	–	–	–	1	–	–	–	–	–	–	–	–
189	1	–	–	–	–	–	–	–	1	–	–	–	–	–	–
431	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–

Table 4.23 Single fundamental covering problem matrix. (Iteration a)

f_i	3	5	7	9	15	17	31	33	63	65	127	129	255	257	511
57	4	2	–	4	4	1	1	2	2	–	0	2	0	0	0
155	0	–	1	0	0	0	–	0	0	0	1	0	0	0	0
189	–	0	0	0	1	1	2	1	–	2	2	1	1	1	0
431	0	1	1	0	0	1	1	0	0	1	0	0	0	0	1

Table 4.24 Included matrix. The fundamental 3 is selected. (Iteration a)

f_i	**3**	5	7	9	15	17	31	33	63	65	127	129	255	257	511
57		–	1	**1**	**1**	–	–	**1**	**1**	1	–	–			–
155		1	–	–	–	–	1	–	–	–	–	–			–
431		–	–	–	–	–	–	–	–	–	–	–			–

Table 4.25 Single fundamental covering problem matrix. (Iteration b)

f_i	**3**	5	7	9	15	17	31	33	63	65	127	129	255	257	511
57		2	–	–	–	1	1	–	–	–	0	2			0
155		–	1	0	0	0	–	0	0	0	1	0			0
431		1	1	0	0	1	1	0	0	1	0	0			1

Table 4.26 Included matrix. The fundamental 5 is selected. (Iteration b)

have been marked with the same letter as given in the tables used to select that specific fundamental. Note that this graph is not constructed until after all extra fundamentals have been found. When each coefficient is solved, it is known that there exist at least one way to realize that coeffi-

f_i	**3**	**5**	7	9	15	17	31	33	63	65	127	129	255	257	511
57			1	1	1	**1**	**1**	1	1	1		–			–
431			–	–	–	–	–	–	–	–		–			**1**

Table 4.27 Single fundamental covering problem matrix. (Iteration c)

f_i	**3**	**5**	7	9	15	17	31	33	63	65	127	129	255	257	511
57			–	–	–	–	–	–	–	–		2			0
431			1	0	0	1	1	0	0	1		0			–

Table 4.28 Included matrix. The fundamental 7 is selected. (Iteration c)

f_i	**3**	**5**	**7**	9	15	17	31	33	63	65	127	129	255	257	511
431						**1**	–			–					1

Table 4.29 Single fundamental covering problem matrix. (Iteration d)

f_i	**3**	**5**	**7**	9	15	17	31	33	63	65	127	129	255	257	511
431						–	1			1					–

Table 4.30 Included matrix. The fundamental 17 is selected. (Iteration d)

cient. However, afterwards it is possible to select the best interconnection for a certain coefficient, for example, requiring fewer overhead full adders than was possible to achieve using the current fundamental set at the time the coefficient was solved.

For the MADr approach, it is clear from the initial CP matrix in Table 4.23 that 431 is the most complicated coefficient, since it can not be solved by any single fundamental. When considering the three possible pairs for 431 given in Table 4.22, it is found that $\{31, 65\}$ is the most beneficial one, which actually solves the complete coefficient set as illustrated in Fig. 4.14 (c). For this example, the MADr solution is the same as obtained by the optimal MADbb algorithm. From this example, the difference between MADc and MADr is obvious, i.e., while MADc includes the most promising single fundamental first, MADr make sure that the most difficult coefficient is solved first. Here, this result in that the coefficient 431 is solved last of all using MADc, while it is considered first of all by MADr. The solutions for the MADc and MADr algorithms, includ-

MADc				MADr			
f_i	Depth	Pair	Expression	f_i	Depth	Pair	Expression
3	1	{1, 1}	4·1 – 1				
5	1	{1, 1}	4·1 + 1				
7	1	{1, 1}	8·1 – 1	31	1	{1, 1}	32·1 – 1
17	1	{1, 1}	16·1 + 1	65	1	{1, 1}	64·1 + 1
57	2	{1, 7}	64·1 – 7	57	2	{1, 65}	65 – 8·1
155	2	{5, 5}	32·5 – 5	155	2	{31, 31}	4·31 + 31
189	2	{3, 3}	64·3 – 3	189	2	{31, 65}	4·31 + 65
431	2	{7, 17}	64·7 – 17	431	2	{31, 65}	16·31 – 65
611	3	{57, 155}	8·57 + 155	611	3	{57, 155}	8·57 + 155

Table 4.31 Solution graphs for MADc and MADr in table format.

ing all expressions with appropriate edge weights and signs, are given in Table 4.31.

Note that it is possible to formulate other rules on which fundamental that should be selected in a certain situation. For example, it can be argued that 31 should be selected instead of 3 as the first extra fundamental for the MADc algorithm. Both will give four new ones in the updated CP matrix, for 3 all will be on the same row while for 31 they will be distributed among three different rows. In this case, 31 would be advantageous, resulting in the same solution as MADr. However, in many other cases this strategy would give worse result. Hence, as for all heuristics it is not possible to formulate a general rule that always will give the best solution. Furthermore, a combination of the MADc and MADr algorithms could be constructed by giving weights to the different rows according to how difficult they are to solve, and select the fundamental that gives the largest contribution. Also, heuristics that consider a few steps ahead, similar to the optimal branch-and-bound algorithm, before making a decision could be designed. Another idea could be to try all combinations of different selection strategies, to end up with a tree of several different solutions, from which the best one can be chosen. Note that most of these ideas would significantly increase the execution time. Another simple, and probably efficient, heuristic is to use MADbb with a limited number of

Algorithm	Extra bits	Coefficient bits						
		6	7	8	9	10	11	12
MADc	0	15.33	19.16	23.20	26.31	28.50	30.32	32.94
	1	0	0	0	–0.02	0	0	0.08
	2	0	0	0	–0.02	0	0.03	0.10
MADr	0	15.33	19.17	23.19	26.26	28.47	30.22	32.86
	1	0	–0.01	–0.01	0.03	0.03	0.05	0
	2	0	–0.01	–0.01	0.03	0.03	0.05	–0.01

Table 4.32 Change in average adder cost for the heuristic algorithms when larger tables are used. Sets of 25 coefficients and different coefficient wordlengths.

branch-and-bound calls, corresponding to a reasonable execution time. From this discussion, it is clear that the possibilities to formulate MCM heuristics based on the proposed minimum depth look-up table are almost infinite. However, only the MADc and MADr heuristic algorithms will be considered in the following.

Size of the Look-Up Table

Adder cost results, using 100 coefficient sets, for look-up tables with up to two extra bits compared to the coefficient wordlength are given in Tables 4.32 and 4.33, for various coefficient wordlength and setsize, respectively. In contrast to the optimal MADbb algorithm, the results using larger tables here almost seems to have a random behavior. Larger tables give worse results both for 12 bit coefficients using MADc and for large coefficient sets using MADr. No gain using larger tables can be guaranteed, possibly except for large coefficient sets using the MADc algorithm. Furthermore, it is only for 12 bit coefficients using the MADr algorithm, where the result using two extra bits is better than with one extra bit. Hence, just because there are more possible fundamentals to choose from, does not mean that a heuristic MCM algorithm will find a better solution.

Execution Time

The execution time for the heuristic algorithms, using look-up tables with larger wordlength than the coefficient sets, is illustrated in Fig. 4.15. Sim-

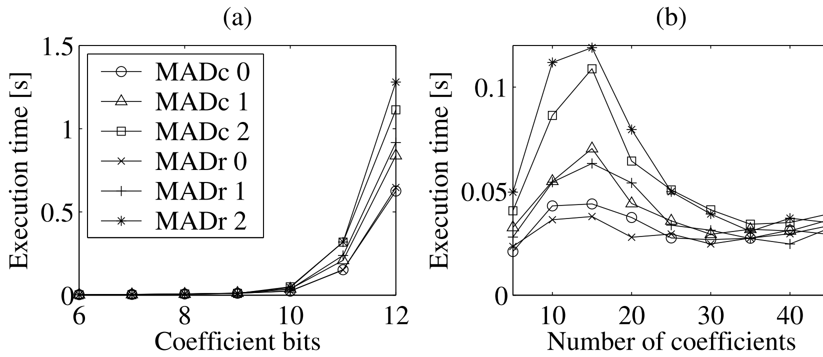


Figure 4.15 Execution time using look-up tables of various size. (a) Different wordlength using sets of 25 coefficients, and (b) different setsize using 10 bit coefficients.

Algorithm	Extra bits	Number of coefficients						
		5	10	15	20	25	35	45
MADc	0	8.15	14.05	19.20	23.63	28.57	37.40	46.20
	1	–0.02	0.02	–0.01	–0.01	–0.02	0	–0.04
	2	–0.01	0.03	0	–0.01	–0.02	0	–0.04
MADr	0	8.00	13.92	19.18	23.56	28.61	37.39	46.13
	1	0	0	–0.01	0.07	0	0.02	0.06
	2	0	0	–0.01	0.07	0	0.02	0.06

Table 4.33 Change in average adder cost for the heuristic algorithms when larger tables are used. Different setsize and 10 bit coefficients.

ilar conclusions as for the optimal MADbb algorithm can be drawn, i.e., the execution time increases with table size and coefficient wordlength while it is decreased for large coefficient sets.

The solutions for sets with 12 bit coefficients, which could not be carried out by the MADbb algorithm, are here in average obtained in less than one second. The difference between the two heuristic algorithms is neglectable, although MADc seems to be slightly faster for larger coefficient wordlength while MADr is faster for small coefficient sets.

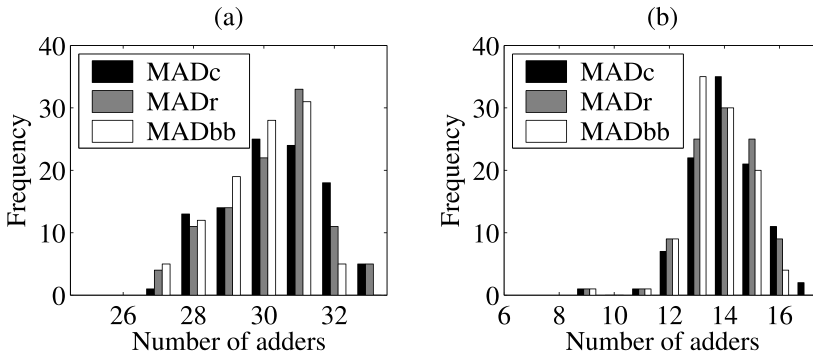


Figure 4.16 Adder cost for 100 cases using the different algorithms. (a) Sets of 25 coefficients with a wordlength of 11 bits, and (b) sets of 10 coefficients with a wordlength of 10 bits.

4.4.5 Optimal vs. Heuristic

Here, the optimal and the heuristic approaches will be compared. When considering both complexity and execution time for the heuristic algorithms in the previous section, no motivation for using look-up tables with extra bits can be found. Also, for the MADbb algorithm, the significantly longer execution time is not worth paying for the small gain in adder cost. Hence, in the rest of this chapter, look-up tables with zero extra bits, i.e., the same wordlength as the coefficient sets, will be used for both the optimal and the heuristic algorithms.

Complexity

The distribution of the adder cost for two different cases is shown in Fig. 4.16. The average number of adders is around 5 and 4 more than the number of coefficients for the case in Fig. 4.16 (a) and (b), respectively, which indicate that these cases are rather complicated. It is clear that MADbb result in a significantly lower cost than the heuristic algorithms, while MADr is slightly better than MADc.

In Table 4.34, the average adder cost, for sets of 25 coefficients with different wordlength, using the optimal MADbb algorithm is given. For short coefficient wordlengths, the increase in average adder cost using the heuristic algorithms is small. However, for eleven coefficient bits the increase is significant. No results could be obtained for the optimal algorithm when 12 bit coefficients were used. Also included in Table 4.34 is an algorithm referred to as MADcr, which for each case simply selects the

Algorithm	Coefficient bits						
	6	7	8	9	10	11	12
MADbb	15.33	19.16	23.14	26.12	28.27	29.83	–
MADc	0	0	0.06	0.19	0.23	0.49	32.94
MADr	0	0.01	0.05	0.14	0.20	0.39	32.86
MADcr	0	0	0.03	0.06	0.06	0.24	32.56

Table 4.34 Increase in average adder cost for the heuristics compared to the optimal MADbb algorithm. Sets of 25 coefficients are used.

Algorithm	Extra adders	Coefficient bits					
		6	7	8	9	10	11
MADc	0	100	100	94	82	78	59
	1			6	17	21	33
	2				1	1	8
MADr	0	100	99	95	86	82	63
	1		1	5	14	16	35
	2					2	2
MADcr	0	100	100	97	94	94	78
	1			3	6	6	20
	2						2

Table 4.35 Number of cases for which an increase in adder cost is obtained compared to MADbb using sets of 25 coefficients.

best solution obtained by MADc and MADr. It is clear that significant improvements are achieved by combining the two heuristic algorithms in this manner. The reason for this is that MADc and MADr use strategies that are more or less opposite to each other, which result in that they often are preferable in different cases.

A more detailed picture over the adder cost is shown in Table 4.35. Here the number of cases, out of the 100 in total, that require 0, 1, or 2 more adders than the optimal algorithm is given. For example, using 11

Algorithm	Number of coefficients							
	5	10	15	20	25	35	40	45
MADbb	7.82	13.67	18.90	23.40	28.28	37.14	41.78	45.95
MADc	0.33	0.38	0.30	0.23	0.29	0.26	0.21	0.25
MADr	0.18	0.25	0.28	0.16	0.33	0.25	0.29	0.18
MADcr	0.13	0.16	0.14	0.07	0.16	0.11	0.11	0.10

Table 4.36 Increase in average adder cost for the different heuristics compared to the optimal MADbb algorithm, using 10 bit coefficients.

Algorithm	Extra adders	Number of coefficients								
		5	10	15	20	25	30	35	40	45
MADc	0	69	64	73	79	72	75	78	79	78
	1	29	34	24	19	27	23	18	21	19
	2	2	2	3	2	1	2	4		3
MADr	0	82	76	73	84	69	77	79	75	82
	1	18	23	26	16	29	18	18	21	18
	2		1	1		2	5	2	4	
	3							1		
MADcr	0	87	84	86	93	84	85	90	89	90
	1	13	16	14	7	16	14	9	11	10
	2						1	1		

Table 4.37 Number of cases for which an increase in adder cost is obtained compared to MADbb using 10 bit coefficients.

bit coefficients, the MADc algorithm have one more adder in 33 cases and two more adders in 8 cases, which result in an increase of the average adder cost by $(33 \cdot 1 + 8 \cdot 2)/100 = 0.49$, as given in Table 4.34. Using the MADcr approach result in the same adder cost as MADbb for at least 94% of the sets using coefficients with up to 10 bits.

Considering sets of different size, the results in Table 4.36 are obtained. The adder cost is naturally increasing for larger sets. However,

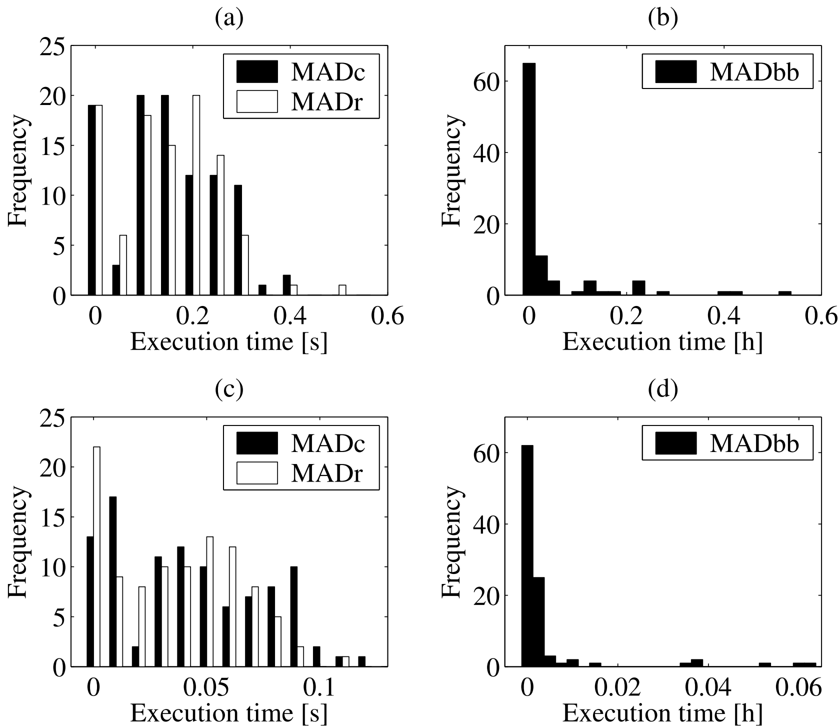


Figure 4.17 Execution time for 100 cases using the different algorithms. (a), (b) Sets of 25 coefficients with a wordlength of 11 bits, and (c), (d) sets of 10 coefficients with a wordlength of 10 bits.

the difference between the number of adders and the number of coefficients is decreasing, and for sets of 45 coefficients the average overhead is less than one adder for the MADbb algorithm. The increase in adder cost for the heuristic algorithms compared to MADbb is almost constant, i.e., independent of the setsize. Again, the combined MADcr algorithm offers a clear reduction in adder cost compared to using MADc and MADr individually.

The specific number of cases with a certain number of extra adders compared to the optimal algorithm is given in Table 4.37. The heuristic algorithms MADc, MADr, and MADcr result in the same number of adders as MADbb in 74%, 77%, and 88% of the cases, respectively.

Execution Time

In Fig. 4.17, the distribution of the execution time for two different cases is shown. The execution times for the heuristic MADc and MADr algorithms are very short, and only have small variations. In the first test case,

Algorithm	Coefficient bits						
	6	7	8	9	10	11	12
MADbb	0.0043	0.0059	0.0160	0.066	1.038	900.14	–
MADc	0.0028	0.0042	0.0070	0.012	0.024	0.15	0.63
MADr	0.0023	0.0039	0.0068	0.011	0.026	0.15	0.65
Factor	1.7	1.5	2.3	5.8	42.2	5927.8	–

Table 4.38 Average execution time in seconds for the optimal and heuristic algorithms, respectively. Using different coefficient wordlengths and sets of 25 coefficients. Also the execution time increase factor when using MADbb instead of the heuristic algorithms is given.

all solutions are obtained within 0.55 seconds, and in the second case within 0.13 seconds, as illustrated in Figs. 4.17 (a) and (c), respectively. For the optimal MADbb algorithm, the distribution is extremely scattered due to the fact that the execution time depend on how fast the binary tree can be reduced during the branch-and-bound search. In the first case, 65% of the solutions were obtained within 45 seconds, while the five most complicated sets took 0.9, 1.9, 3.1, 3.9, and 11.2 hours, respectively, and are out of the range in Fig. 4.17 (b). In the second test case, 87% of the solutions were obtained within 15 seconds, while the longest execution time was 227 seconds, as illustrated in Fig. 4.17 (d).

The average execution time for all algorithms, using sets of 25 coefficients, is given in Table 4.38. Up to seven coefficient bits, the optimal algorithm is less than two times slower, i.e., faster than running both MADc and MADr to select the best solution. Note however that the execution time for MADcr would be slightly less than the sum of the execution time for MADc and MADr, since parts of the initialization may be shared. For long coefficient wordlengths, the MADbb algorithm becomes impractical. For 12 bit coefficients, the execution time is in many cases too long to be completed, while for the heuristic algorithms the execution time is only around 0.65 seconds.

The variation in execution time is small for the heuristic algorithms when considering coefficient sets of different size, as can be seen in Table 4.39. For the optimal MADbb algorithm, on the other hand, the execution time is significantly longer for small coefficient sets.

Algorithm	Number of coefficients							
	5	10	15	20	25	35	40	45
MADbb	23.833	14.706	16.995	0.867	0.515	0.152	0.132	0.129
MADc	0.021	0.043	0.044	0.037	0.028	0.027	0.031	0.029
MADr	0.023	0.036	0.038	0.028	0.029	0.027	0.030	0.033
Factor	1073.6	371.4	416.6	26.6	18.1	5.6	4.3	4.1

Table 4.39 Average execution time in seconds for the optimal and heuristic algorithms, respectively. Using different coefficient setsize and 10 bit coefficients. Also the execution time increase factor when using MADbb instead of the heuristic algorithms is given.

4.4.6 Results

In this section, the proposed algorithms, MADc and MADr, are compared to the RAG-*n* algorithm [25], which for a long time was considered to be the best MCM algorithm, and to the DiffAG algorithm [43], which together with H_{cub} [144] are the two current algorithms resulting in lowest adder cost. Finally, results are also included for the C1 algorithm [26], which is based on RAG-*n* and is one of the few MCM algorithms that have been designed to give low depth solutions. The algorithms are compared in terms of complexity, adder depth, and execution time. Each marker in the graphs corresponds to the average result obtained from 100 random coefficient sets. The same sets have been used for the different algorithms. For all algorithms, the interconnection has been obtained using the minimum depth algorithm described in Section 4.3. Also, half adders are eliminated and the FA overhead is optimized by selecting the best interconnection using (4.10).

Complexity

To minimize the number of adders and subtractors, i.e., the adder cost, has been the only objective in most previous work. Comparing DiffAG and RAG-*n*, it can be seen in Fig. 4.18 that the results are similar except for long coefficient wordlengths and small coefficient sets. The reason is that for both these cases the algorithms are more likely to use a heuristic, which is not required as long as each coefficient in C can be realized with only one additional adder. Clearly, DiffAG has a better heuristic. The behaviors of the proposed algorithms are similar, with a small advantage

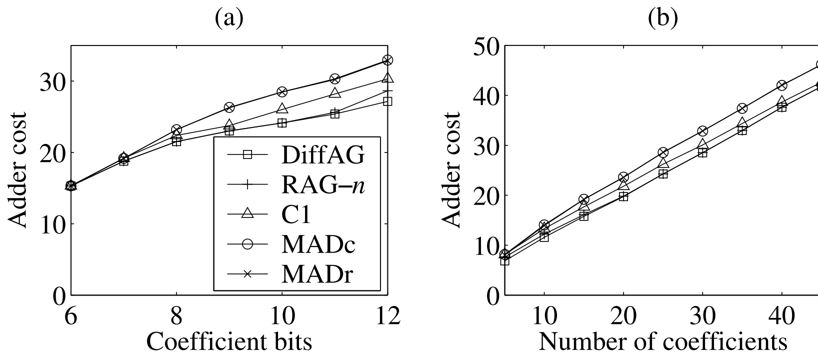


Figure 4.18 Adder cost (a) for different wordlength using sets of 25 coefficients, and (b) for different setsize using 10 bit coefficients.

for MADr for long coefficient wordlengths and small coefficient sets. It is not surprising that it for these cases is beneficial to start with the more difficult coefficients, as the first added fundamentals then are more likely to be usable for the rest of the coefficients. On the other hand, MADc is slightly better for large coefficient sets as the more difficult coefficients then are likely to be realized anyway due to more available fundamentals in F. The proposed algorithms require on average almost 6 more adders than DiffAG for 12 bit coefficients, and just over 4 more adders for large coefficient sets. The C1 algorithm has a larger adder cost than DiffAG and RAG-*n*, but lower than the MAD algorithms. This is exactly as expected since the idea behind C1 is to minimize the adder cost, while keeping the adder depth low, i.e., a trade-off between the ideas used for the other algorithms.

To get the full picture of complexity, it is not enough to only consider wordlevel adders, but also the actual number of required full adder cells. In Fig. 4.19, it can be seen that the proposed algorithms use around 26 overhead full adders less than DiffAG for 12 bit coefficients. For sets of 45 coefficients, around 35 less full adders are used, which correspond to more than two wordlevel adders if the data wordlength, W_0, is 16 bits. Thus, the total complexity is not increased as much as indicated in Fig. 4.18. When coefficients are realized at a low depth, the number of shifts is likely to be large, which will decrease the number of full adders according to (4.10). At a high depth, on the other hand, a coefficient is more likely to be realized by adding two fundamentals of large magnitude, using few shifts. Hence, the reduction in FA overhead is a natural consequence of the reduced adder depth, which will be investigated next.

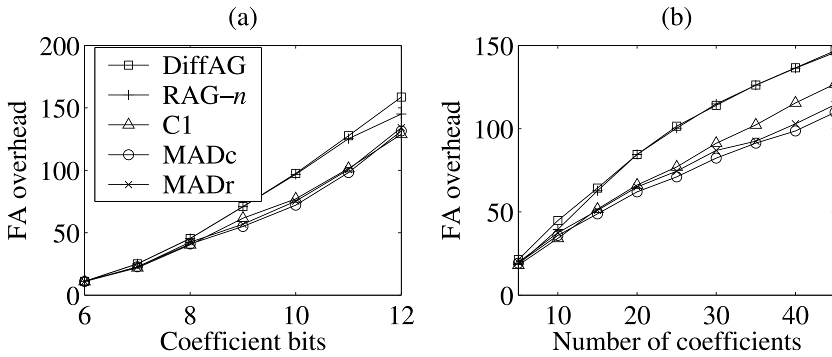


Figure 4.19 Full adder overhead (a) for different wordlength using sets of 25 coefficients, and (b) for different setsize using 10 bit coefficients.

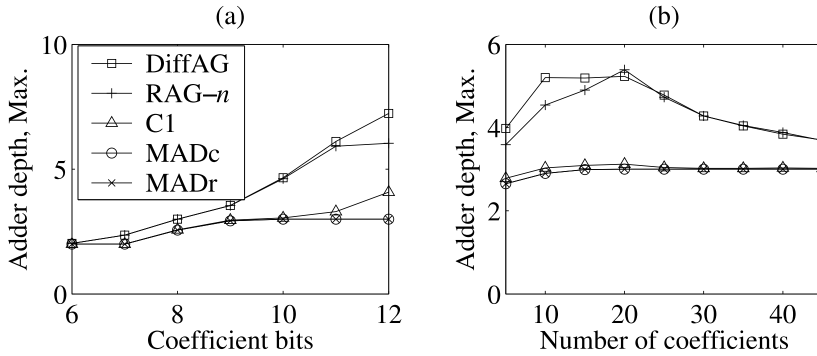


Figure 4.20 Maximum adder depth (a) for different wordlength using sets of 25 coefficients, and (b) for different setsize using 10 bit coefficients.

Adder Depth

In Fig. 4.20, the maximum adder depth using the different algorithms is shown. Since all coefficients up to a wordlength of 12 can be realized at depth 3 or lower, the proposed algorithms will not exceed this. DiffAG, RAG-*n*, and C1 have on average a maximum adder depth of about 7.2, 6.0, and 4.1, respectively, for 12 bit coefficients. When sets of different size are considered, the DiffAG and RAG-*n* algorithms has a peak for sets of 20 coefficients, for which the maximum adder depth is on average about 5.3, i.e., 2.3 higher than for the proposed algorithms. Small coefficient sets are naturally less likely to contain high depth coefficients, while for large coefficient sets there are more options, which make it easier to

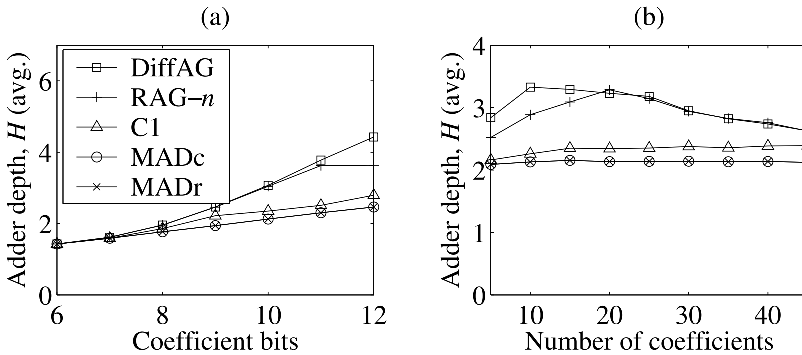


Figure 4.21 Average adder depth (a) for different wordlength using sets of 25 coefficients, and (b) for different setsize using 10 bit coefficients.

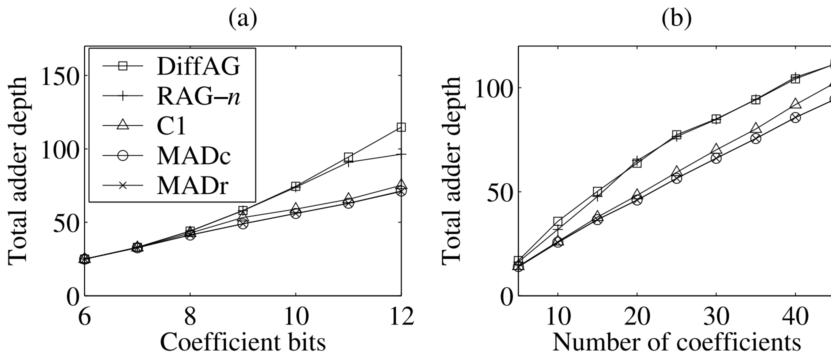


Figure 4.22 Total depth for all adders in the MCM block. (a) Different wordlength using sets of 25 coefficients, and (b) different setsize using 10 bit coefficients.

keep the adder depth low. The maximum depth for the C1 algorithm is close to the MAD algorithms.

The same characteristics can be observed when the average adder depth, for all the coefficients in H, is considered, as illustrated in Fig. 4.21. However, it is here clear that the average adder depth is lower for the MAD algorithms than for C1.

Complexity vs. Adder Depth

It is a well-known fact that there exist a trade-off between the complexity in terms of adders and the adder depth, this trade-off is investigated here.

In Fig. 4.22, the total adder depth, i.e., the sum of the depth for all fundamentals in F, is shown. Since this measure includes both the adder cost, i.e., the number of adders, and the depth of each adder, it can be seen as a

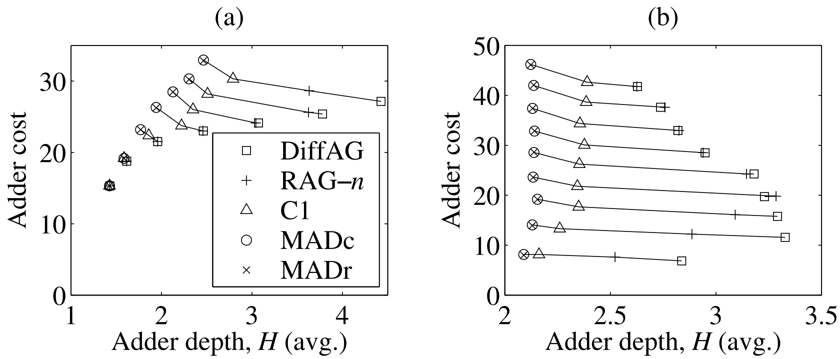


Figure 4.23 Adder cost as a function of average adder depth. (a) Coefficient wordlengths from 6 to 12 bits using sets of 25 coefficients. (b) Sets of size 5, 10, ..., 45 using 10 bit coefficients.

high-level estimation of energy consumption, similar to the GP count [21]. Figure 4.22 (a) shows the same behavior as Fig. 4.21 (a). Even though the C1 algorithm results in significantly fewer adders than the MAD algorithms, the total adder depth is still higher. For large coefficient sets, the difference in total adder depth between the MAD algorithms and C1 becomes larger, as can be seen in Fig. 4.22 (b).

Figure 4.23 clearly shows the trade-off between adder depth and complexity. The lower left gathering of markers in Fig. 4.23 (a) correspond to the obtained results using coefficients of 6 bits. Hence, for short coefficient wordlengths, all algorithms give similar results. When using coefficients of more bits, on the other hand, it is possible to choose between a low adder cost, corresponding to the DiffAG solution, a low adder depth, obtained by the MAD algorithms, or somewhere in between using, for example, the C1 algorithm. Note that the MAD algorithms are optimal in terms of adder depth, i.e., it is impossible to find solutions with a marker further to the left. Furthermore, the MADbb algorithm would be placed just below the heuristic MAD algorithms, corresponding to solutions with the exact same depth but slightly lower adder cost. Similar conclusions can be drawn from Fig. 4.23 (b), where the lowest and highest line correspond to sets of 5 and 45 coefficients, respectively. The difference between C1 and the MAD algorithms, both in adder cost and depth, slightly increases for a larger coefficient setsize.

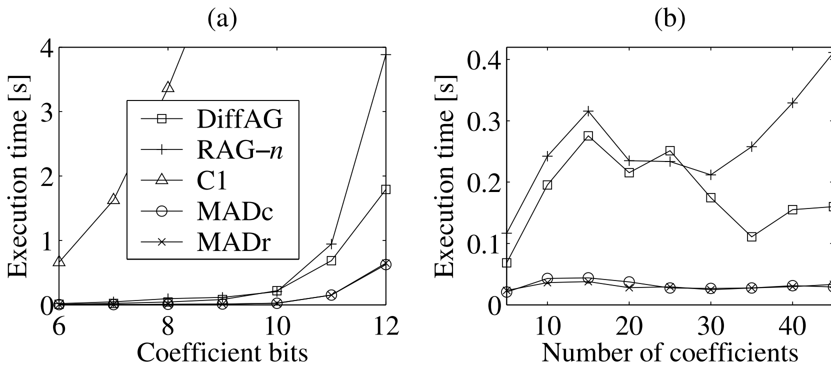


Figure 4.24 Execution time (a) for different wordlength using sets of 25 coefficients, and (b) for different setsize using 10 bit coefficients.

Execution Time

Finally, the algorithms are compared in terms of execution time. The results are shown in Fig. 4.24. Naturally, the execution time may vary slightly from time to time. However, all simulations have of course been performed on the same type of computer without any other major processes running at the same time. Hence, the difference is large enough to establish that the MAD algorithms are significantly faster than any of the other algorithms, while C1 is definitely the slowest. For the C1 algorithm, most of the markers are not visible in Fig. 4.24. However, for 6 to 12 bit coefficients, the execution time increases from 0.7 to 24.1 seconds, and for sets of 5 to 45 coefficients, the execution time increases from 0.8 to 30.1 seconds. Hence, opposite to the other algorithms, the C1 algorithm does not have any maxima or minima for a certain setsize.

The relation between the coefficient setsize and the degree of difficulty of MCM problems, may be compared to the popular Sudoku puzzles. Lets say that a small setsize, with for example five coefficients, correspond to a puzzle with six regions using numbers from 1 to 6. There is not much information available, but on the other hand the puzzle is so small that it can be solved in short time anyway. For a large setsize of around 45 coefficients, the puzzle is huge, say 16 regions with numbers from 1 to 16. However, most of the squares are already filled in with the correct value, and we only need to add a few numbers to solve the complete problem, which do not take much time. Finally, the most difficult problem occur with the standard size of nine regions with numbers from 1 to 9, corresponding to a setsize of about 15 coefficients. There are a lot of empty

squares and you do not really believe that the available information is enough to find a solution. Here, only a small part at a time can be solved, i.e., the puzzle does not collapse like the big one did. This discussion agrees closely to the behavior of many MCM algorithms, for example, the DiffAG and MAD algorithms.

4.4.7 Implementation Examples

In this section, two FIR filters are implemented using different algorithms to design the MCM blocks. The filters are realized using the transposed direct form structure shown in Fig. 1.1 (b), and implemented by logic synthesis of VHDL code. The input data wordlength, W_0, is selected to be 16 bits. The filters are implemented using a 0.35 μm CMOS standard cell library. Area and sample rate results are given as reported by the synthesis tool, while power consumption results have been obtained using NanoSim™ with 1000 random input samples and a clock frequency of 10 MHz. All figures are given both for the multiplier block, which is the most interesting part in this study, and for the overall FIR filter.

Example 1

Here, one of the random sets used in the previous section will be implemented as an FIR filter with 30 taps. The selected coefficient set is $H = \{975, 283, 424, 994, 716, 752, 441, 133, 844, 253, 370, 372, 409, 915, 324, 532, 330, 424, 990, 963, 647, 296, 57, 522, 894, 268, 223, 961, 375, 734\}$. Of course, this will not correspond to a frequency selective filter in the normal sense, since it has not been designed for any specific requirements. However, the implementation is done in the same way. The main purpose of this implementation is to investigate the relation between adder cost and adder depth, from an energy consumption point of view. This is done by comparing the different MAD algorithms with the C1 algorithm, which normally has a relatively low maximum adder depth and fewer adders than the proposed MAD algorithms.

The set of unique odd coefficients, $C = \{37, 47, 53, 57, 67, 81, 93, 133, 165, 179, 185, 211, 223, 253, 261, 283, 367, 375, 409, 441, 447, 495, 497, 647, 915, 961, 963, 975\}$, includes 28 values. Note that two of the coefficients in H are equal to 424, and that $532 = 4 \cdot 133$, which explains why C contains two values less than H. The C1 algorithm finds a solution with only two extra fundamentals, $E = \{3, 7\}$. Out of the 28 realized coefficients in C, 10 are realized at depth two and 18 at depth three, resulting

Algorithm	Adder cost		Adder depth				GP count
	WL	FA	F (avg.)	C (avg.)	H (avg.)	Max.	
C1 [26]	**30**	110	2.5333	2.6429	2.6333	**3**	104
MADbb	33	81	1.9697	**2.1429**	**2.1333**	**3**	**85**
MADc	34	**79**	1.9412	**2.1429**	**2.1333**	**3**	86
MADr	35	102	**1.9143**	**2.1429**	**2.1333**	**3**	92

Table 4.40 Complexity and adder depth results for Example 1.

Algorithm	Area [mm^2]		Rate [MSa/s]		Energy [nJ]	
	MB	FIR	MB	FIR	MB	FIR
C1 [26]	0.1816	0.7184	55.80	43.55	0.5148	2.0224
MADbb	**0.1777**	**0.7140**	61.27	43.16	**0.4543**	1.9014
MADc	0.1807	0.7169	**62.50**	43.20	0.4668	**1.8850**
MADr	0.1900	0.7271	61.88	**45.35**	0.5013	1.9395

Table 4.41 Implementation results for Example 1.

in an average depth of 2.64. The MADbb, MADc, and MADr algorithms, require 5, 6, and 7 extra fundamentals, respectively, all of them at depth one. The MAD algorithms guarantee that all coefficients are realized at the lowest adder depth that is theoretically possible, which in this case result in 24 coefficients at depth two and 4 at depth three with an average depth of 2.14. Hence, the solution obtained by the C1 algorithm has fewer adders and the same maximum adder depth, but higher average adder depth, when compared to the MAD algorithms. These results are summarized in Table 4.40, where the best result in each column is marked in bold. Also included in Table 4.40 is the GP count [21], indicating that the C1 solution have more occurrences of energy consuming glitch propagation than the MAD solutions.

The implementation results are given in Table 4.41. Even though the C1 implementation has three fewer wordlevel adders than MADbb, the area is still larger. This is mainly explained by the increased number of overhead full adders, corresponding to almost two wordlevel adders.

Only small differences in maximum sample rate can be observed. C1 has more long paths in the multiplier block due to more depth three coefficients, and is therefore more likely to have a longer critical path. However, when connected to the structural adders, the critical paths are increased by more full adders for the MAD algorithms, resulting in basically the same throughput for all FIR filter implementations.

Using MADc instead of C1, the energy consumption is reduced by 9.3% and 6.8% for the multiplier block and the complete FIR filter, respectively. Note that the corresponding numbers for the area are only 0.5% and 0.2%, respectively. Hence, a significant reduction in energy consumption is obtained, although the two implementations have the same maximum adder depth and basically the same area. This example shows that the average adder depth plays an important role when minimizing the energy consumption.

Example 2

Again, the same FIR filter as was used in Section 4.2.1 and for Example 1 in Section 4.3.2 is considered.

In Table 4.42, the complexity in terms of wordlevel adders and overhead full adder cells, the maximum and average adder depth, and the GP count for the different algorithms are given. For all implementations, the minimum depth interconnection algorithm described in Section 4.3, has been used to optimize the adder depth and the overhead full adder cost. Thus, the best Pasko and H_{cub} designs have been selected among the ones in Table 4.7. Also, note that the number of overhead full adders has been slightly reduced for the DiffAG, RAG-*n*, and C1 algorithms compared to the values given in Table 4.4. Furthermore, the adder depth stated in Table 2.6 has been significantly decreased for the RAG-*n* algorithm, while no improvement was possible for C1.

The area, throughput, and energy consumption results are given in Table 4.43. Naturally, the three algorithms with the lowest number of wordlevel adders have smallest area. For MADc, the area of the MCM part and the total FIR filter is increased by 6.1% and 1.0%, respectively, compared to H_{cub}, which has the smallest area.

Although the H_{cub} and DiffAG algorithms result in 20% fewer wordlevel adders than the MAD algorithms, the energy consumption is significantly higher. For example, using the MADc algorithm, the energy consumption in the multiplier block is reduced by 27.0% and 23.5% compared to the H_{cub} and DiffAG algorithms, respectively. For the total FIR

Algorithm	Adder cost		Adder depth			GP count
	WL	FA	F (avg.)	H (avg.)	Max.	
Pasko [115]	23	**50**	2.2174	2.6400	4	**60**
H_{cub} [144]	**16**	77	4.1875	4.4800	7	119
DiffAG [43]	**16**	83	3.8125	3.8800	7	91
RAG-*n* [25]	17	67	3.0588	3.5600	6	91
C1 [26]	19	69	2.3158	2.8000	5	69
MADbb	19	83	2.1053	**2.4400**	**3**	**60**
MADc	20	53	**2.0500**	**2.4400**	**3**	62
MADr	20	78	**2.0500**	**2.4400**	**3**	61

Table 4.42 Complexity and adder depth results for Example 2.

Algorithm	Area [mm^2]		Rate [MSa/s]		Energy [nJ]	
	MB	FIR	MB	FIR	MB	FIR
Pasko [115]	0.1208	0.5721	59.52	43.92	0.3492	1.5989
H_{cub} [144]	**0.1026**	**0.5532**	49.60	34.75	0.4495	1.8321
DiffAG [43]	0.1080	0.5559	59.31	44.39	0.4291	1.7183
RAG-*n* [25]	0.1045	0.5535	56.79	38.40	0.3784	1.7003
C1 [26]	0.1119	0.5600	54.64	41.24	0.3595	1.6254
MADbb	0.1164	0.5658	58.38	**45.60**	0.3287	**1.5213**
MADc	0.1089	0.5589	**65.36**	45.52	**0.3283**	1.5463
MADr	0.1178	0.5683	53.94	43.78	0.3422	1.5362
Direct Imp.	0.4982	0.9597	61.65	37.91	1.3744	2.9703

Table 4.43 Implementation results for Example 2.

filter, the corresponding reduction is 15.6% and 10.0%, respectively. This clearly shows the importance of adder depth, i.e., from an energy consumption point of view it is often beneficial to have more adders if the adder depth can be decreased. This is further illustrated in Fig. 4.25. It is

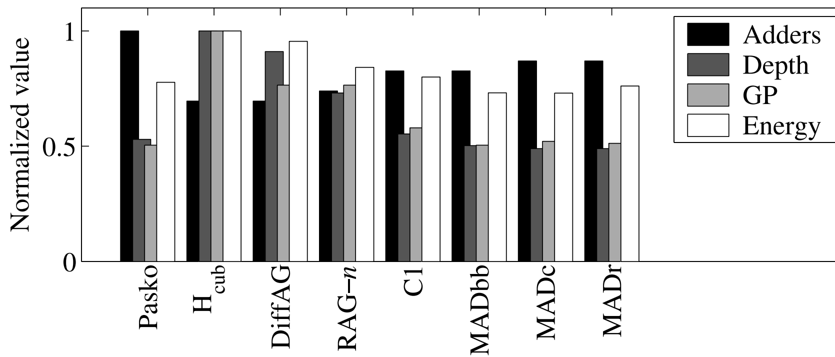


Figure 4.25 Normalized values of the adder cost, adder depth, and GP count, which all can be seen as high-level measures for the energy consumption, obtained for the different algorithms in Example 2.

clear that the adder cost is not in correlation with the energy consumption of the multiplier block, since the two algorithms with the lowest number of adders have the highest energy. Hence, the average adder depth and the GP count are better, but not perfect, high-level measures of the energy consumption.

Also included in Table 4.43 is a straightforward implementation. The first filter coefficient, –710, is here realized by the VHDL expression `resize(-710*x, 26)`, where 26 is the wordlength of the result according to (4.1). The remaining 12 coefficients are obtained in the same way, while the delay elements and the structural adders are realized exactly as before. Compared with the MADc implementation, the area and power consumption of the multiplier block is increased by a factor 4.6 and 4.2, respectively. Furthermore, it is clear that the power consumption for the structural adders is also increased. However, these results are not really fair since the used synthesis tool does not put much effort into optimize the design, which is the reason for poor results when using such a high-level description. Instead, using Design Compiler together with the same standard cell library as before, the results given in Table 4.44 were obtained. For the H_{cub} and MADc algorithms, the improvement in area is not significant, while the area for the multiplier block using the direct implementation is reduced by 64%. However, this is still 66% larger than the MCM block obtained by MADc, while the total FIR filter is 13% larger. The energy consumption for the MCM part and the total FIR filter is 55% and 7%, respectively, larger for the direct implementation than for MADc. It has been shown that, using this cell library, subtractions are often more expensive than additions. Hence, a likely reason that the struc-

Algorithm	Area [mm^2]		Rate [MSa/s]		Energy [nJ]	
	MB	FIR	MB	FIR	MB	FIR
H_{cub} [144]	**0.0992**	**0.5213**	39.57	30.20	0.4714	2.0809
MADc	0.1088	0.5314	51.49	43.14	**0.3139**	**1.5300**
Direct Imp.	0.1807	0.6016	**78.13**	**52.03**	0.4858	1.6400

Table 4.44 Implementation results using Design Compiler and the same 0.35 μm standard cell library as before.

Algorithm	Area [mm^2]		Rate [MSa/s]		Energy [nJ]	
	MB	FIR	MB	FIR	MB	FIR
H_{cub} [144]	**0.0117**	**0.0577**	97.85	59.56	**0.0188***	**0.0739***
MADc	0.0128	0.0587	**115.87**	80.06	0.0228*	0.0778*
Direct Imp.	0.0234	0.0670	109.29	**83.33**	0.0396*	0.0922*

Table 4.45 Implementation results using a 0.13 μm standard cell library. *The energy numbers given here have been obtained from the synthesis tool by propagating the switching activities in a zero delay model.

tural adders consume less power in the direct implementation is that only additions are performed, while for the MADc implementation there are 9 additions and 15 subtractions. Furthermore, the higher throughput for the direct implementation indicates that the logic paths are shorter, which may reduce the switching activity of the signals connected to the structural adders. Note that the wordlengths of the structural adders are exactly the same for all algorithms.

Finally, the effect of scaling is considered by using a 0.13 μm CMOS standard cell library. The MCM part area is increased by 83% and the total FIR filter area by 14%, when using a direct implementation instead of MADc. This gives an even larger advantage for MADc than when using 0.35 μm, and, hence, efficient MCM algorithms will not become less important in future technologies. Furthermore, the throughput for the direct implementation is not higher than for MADc using this cell library, which was the case before.

The energy results in Table 4.45 are mainly included to motivate why a 0.35 μm standard cell library, for which the transistor models needed to

perform accurate power simulations were available, have been used throughout this thesis. Since the synthesis tool here uses a zero delay model, the effect of glitches is not included, and, therefore, the energy results will be closely related to the area. Hence, these results, which are the only ones that could be obtained for the 0.13 µm standard cell library, are obviously not in correlation with an actual implementation.

4.5 Conclusions

Here, we have considered bit-level optimization of FIR filters implemented using parallel arithmetic and shift-and-add multiplication.

A detailed complexity model for multiple-constant multiplication (MCM) has been presented. The model counts the number of full and half adder cells required to realize an MCM block. Based on this model, a novel algorithm for the MCM problem was formulated. It was shown that this algorithm provides significantly improved results compared with previous algorithms. The complexity model can also be used for single constant coefficient multipliers, constant matrix multipliers, and FIR filters. Furthermore, a transformation that can be used to eliminate the use of half adders, at no extra cost, was introduced.

Interconnection strategies, which can be applied independently of which algorithm that is used to solve the multiple-constant multiplication problem, have been presented. Given the complete fundamental set, a minimum depth realization can be guaranteed. Furthermore, it was shown that the complexity in terms of full and half adders can be significantly reduced. For ASIC implementations this directly correspond to a decreased area, and for the FPGA example design the number of CLB slices used in the multiplier block was reduced by 20%. A main factor for energy consumption in multiplier blocks is the adder depth, i.e., the number of cascaded adders. Hence, the algorithm that result in a minimum depth interconnection, while the complexity is also considered, seems advantageous in most cases. The energy consumption was decreased by 8.8% for the ASIC multiplier block example design.

Finally, we have proposed an algorithm for multiple-constant multiplication problems, where all multiplier coefficients are realized at the theoretically lowest possible adder depth. One optimal version, which result in a minimum adder cost realization given the minimum depth requirement, as well as two fast heuristic versions were presented. An FIR filter was implemented using different MCM algorithms, and it was shown that

the proposed algorithm result in multiplier blocks with around 25% lower energy consumption compared to algorithms with solutions using fewer wordlevel adders.

5

ENERGY ESTIMATION FOR RIPPLE-CARRY ADDERS

In this chapter, modeling of the energy consumption for ripple-carry adders implemented in CMOS, is considered [59]. Based on the switching activity of each input bit, two switching models, one full and one simplified, are derived. Combined with transition energies for a full adder obtained by simulation, these switching models can be used to derive the average energy consumed for one operation. Examples show that the model is accurate if all switches in the ripple-carry adder are rail-to-rail (full swing), but in the actual implementation this is not always the case. Hence, our model overestimates the energy consumption.

In [78], the dual bit type method was presented, which can be used to obtain certain region properties, i.e., determine the switching activity for different bit positions, from word-level statistics. Hence, this method offers an efficient model for two's-complement numbers used in many real world DSP applications. Here, the proposed switching model for ripple-carry adders is modified by adopting the dual bit type method [60].

In the last part, an important step towards a switching activity model for multiplier blocks is taken by presenting an accurate model for single adder multipliers [70]. This corresponds to the case where a signal is added to a shifted version of itself, which is a common operation in constant multiplication. Hence, the model is suitable to be used in energy consumption aware MCM algorithms. Finally, an event-based model for estimation of the switching activity in MCM implementations is outlined.

5.1 Background

Energy modeling and estimation of digital circuits have received considerable interest during the last decade [91],[103]. Especially for DSP systems, energy modeling with correlated data has been considered [79], [89].

A key component in almost all DSP systems is the binary adder. A basic adder structure is the ripple-carry adder (RCA), as shown in Fig. 5.1. The ripple-carry adder is based on full adder cells which adds two input bits and the incoming carry bit to yield a result in the form of a sum bit and an outgoing carry bit. Numerous full adder cells aiming at low energy consumption have been presented and recent comparisons are found in [1] and [129].

There are other adder structures providing higher speeds than the ripple-carry adder, for which the execution time is proportional to the data wordlength. Comparisons in terms of area, time, and power consumption for different adder structures are found in [9] and [102]. However, only numerical results are presented for the power consumption.

In [97], the average energy consumption for the ripple-carry adder based on the average length of the carry chain was derived assuming random inputs. The method in [97] was extended in [35], where a better approximation was derived based on the average energy for all possible lengths of the carry chains. It was concluded that this model agrees well with the results of the power simulation tool HEAT [124]. Again, random inputs were assumed.

This work extends previous results by introducing a data dependent energy model for ripple-carry adders, i.e., correlated input data are considered. In real world signals, the switching activity is different for different bit positions [79],[89]. Hence, the proposed model provides a more relevant base for higher-level energy modeling and estimation of energy consumption in ripple-carry adders.

Note that although a parallel architecture of the ripple-carry adder is assumed in this chapter, the energy models can also be used for serial arithmetic if the switching activity of the carry feedback is derived.

5.1.1 Exact Method for Transitions in RCA

In the following, a theoretically exact model for computation of the switching activity in ripple-carry adders with correlated input data is

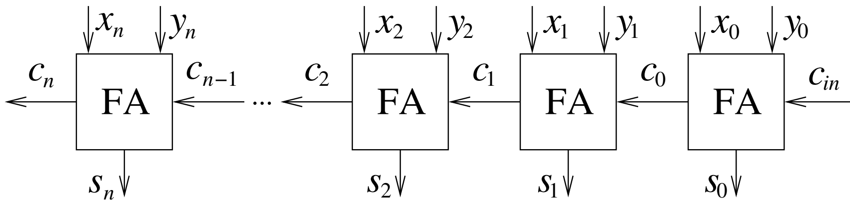


Figure 5.1 Ripple-carry adder of $n + 1$ bits.

derived. This is done by computing the probability for all possible transitions for each full adder of the RCA.

Two different energy models are proposed. One is based on a probability matrix, where each transition probability is multiplied with the corresponding switching energy. The other is using a simplified representation, where the average energy consumed by switching of the carry, sum, or both outputs is used. Assuming random inputs, the second model will produce identical results to the model in [35], while further simplifications will lead to the model in [97]. However, the proposed model is not based on carry-chains so, apart from supporting correlated data, a different method is used for the derivation.

To validate the models, some experimental results are presented and compared with [35] and [97], where applicable. It is shown that the proposed models give accurate results in terms of switches, while the energy consumption is overestimated due to the fact that not all switches are rail-to-rail. However, this is a problem for the methods in [35] and [97] as well.

5.2 Energy Model

A ripple-carry adder is composed of a number of cascaded full adders, as shown in Fig. 5.1. When at least one of the three inputs of a full adder changes value, the outputs perform a transition according to Table 5.1. The five different transition types are defined so that the symbol *S* means that only the sum output switches, *C* means that only the carry output switches, *CS* means that both the sum and the carry outputs switch, *none* means that the outputs do not switch, and *static* means that not even the inputs switch.

The state of a full adder, as shown in Fig. 5.2, is here defined as the logic value at its three inputs, in the order x_i, y_i, and c_{i-1}, which is

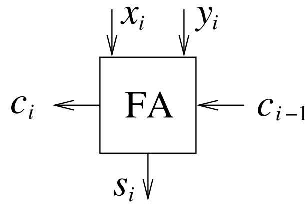


Figure 5.2 Full adder used in a ripple-carry adder.

Change of input ones	Transition type
$0 \leftrightarrow 2$, $1 \leftrightarrow 3$	C
$0 \leftrightarrow 1$, $2 \leftrightarrow 3$	S
$0 \leftrightarrow 3$, $1 \leftrightarrow 2$	CS
$1 \rightarrow 1$, $2 \rightarrow 2$	*none* or *static*
$0 \rightarrow 0$, $3 \rightarrow 3$	*static*

Table 5.1 Type of transitions at the two outputs of a full adder for different changes of the number of ones at the three inputs.

described in Table 5.2. In Table 5.3, the switching activity at the outputs is defined for every state transition.

Since the carry propagate through the adder, each full adder can make several transitions for each pair of applied input signals, X and Y. The switching activities will be computed for a generate phase and a propagate phase, respectively. The data inputs are changed in the generate phase, and the carry outputs are stabilized in the chain of full adders in the propagate phase. This implies that transitions where both the data inputs and the carry input change are not possible. These transitions are in brackets in Table 5.3. One exception from this is the first full adder, corresponding to the least significant bit (LSB), for which also the carry input, c_{in}, may change in the generate phase.

The energy corresponding to every state transition combination at the inputs of a full adder can be simulated and stored in a matrix, as illustrated by Table 5.4. The energy consumption will be estimated in two different ways. One method is to compute the probability for each specific state transition, and a simplified method is to compute the probabilities for each kind of transition. In the simplified case, the average transition energies are used as given in Table 5.5.

State	x_i	y_i	c_{i-1}	c_i	s_i
0	0	0	0	0	0
1	0	0	1	0	1
2	0	1	0	0	1
3	0	1	1	1	0
4	1	0	0	0	1
5	1	0	1	1	0
6	1	1	0	1	0
7	1	1	1	1	1

Table 5.2 Truth table for the full adder in Fig. 5.2.

State	0	1	2	3	4	5	6	7
0	*static*	*S*	*S*	(*C*)	*S*	(*C*)	*C*	(*CS*)
1	*S*	*static*	(*none*)	*CS*	(*none*)	*CS*	(*CS*)	*C*
2	*S*	(*none*)	*static*	*CS*	*none*	(*CS*)	*CS*	(*C*)
3	(*C*)	*CS*	*CS*	*static*	(*CS*)	*none*	(*none*)	*S*
4	*S*	(*none*)	*none*	(*CS*)	*static*	*CS*	*CS*	(*C*)
5	(*C*)	*CS*	(*CS*)	*none*	*CS*	*static*	(*none*)	*S*
6	*C*	(*CS*)	*CS*	(*none*)	*CS*	(*none*)	*static*	*S*
7	(*CS*)	*C*	(*C*)	*S*	(*C*)	*S*	*S*	*static*

Table 5.3 Resulting switching of the output signals for different transitions of the input signals. Transitions are performed from the state in the left column to the state in the heading row.

State	0	1	2	3	4	5	6	7
0	$E_{0\to0}$	$E_{0\to1}$	$E_{0\to2}$	$E_{0\to3}$	$E_{0\to4}$	$E_{0\to5}$	$E_{0\to6}$	$E_{0\to7}$
1	$E_{1\to0}$	$E_{1\to1}$	$E_{1\to2}$	$E_{1\to3}$	$E_{1\to4}$	$E_{1\to5}$	$E_{1\to6}$	$E_{1\to7}$
2	$E_{2\to0}$	$E_{2\to1}$	$E_{2\to2}$	$E_{2\to3}$	$E_{2\to4}$	$E_{2\to5}$	$E_{2\to6}$	$E_{2\to7}$
3	$E_{3\to0}$	$E_{3\to1}$	$E_{3\to2}$	$E_{3\to3}$	$E_{3\to4}$	$E_{3\to5}$	$E_{3\to6}$	$E_{3\to7}$
4	$E_{4\to0}$	$E_{4\to1}$	$E_{4\to2}$	$E_{4\to3}$	$E_{4\to4}$	$E_{4\to5}$	$E_{4\to6}$	$E_{4\to7}$
5	$E_{5\to0}$	$E_{5\to1}$	$E_{5\to2}$	$E_{5\to3}$	$E_{5\to4}$	$E_{5\to5}$	$E_{5\to6}$	$E_{5\to7}$
6	$E_{6\to0}$	$E_{6\to1}$	$E_{6\to2}$	$E_{6\to3}$	$E_{6\to4}$	$E_{6\to5}$	$E_{6\to6}$	$E_{6\to7}$
7	$E_{7\to0}$	$E_{7\to1}$	$E_{7\to2}$	$E_{7\to3}$	$E_{7\to4}$	$E_{7\to5}$	$E_{7\to6}$	$E_{7\to7}$

Table 5.4 Energy matrix.

Transition	*CS*	*C*	*S*	*none*	*static*
Energy	E_{CS}	E_C	E_S	E_{none}	E_{static}

Table 5.5 Average energy table for the different types of transitions.

The average transition energies are computed as

$$\begin{cases} E_{CS} = \dfrac{1}{12}\sum_{j,k} E_{j\to k} & (j,k) \in CS \text{ and } j+k \neq 7 \\ E_C = \dfrac{1}{4}\sum_{j,k} E_{j\to k} & (j,k) \in \{(0,6),(1,7),(6,0),(7,1)\} \\ E_S = \dfrac{1}{12}\sum_{j,k} E_{j\to k} & (j,k) \in S \\ E_{none} = \dfrac{1}{4}\sum_{j,k} E_{j\to k} & (j,k) \in \{(2,4),(3,5),(4,2),(5,3)\} \\ E_{static} = \dfrac{1}{8}\sum_{j,k} E_{j\to k} & (j,k) \in static \end{cases} \tag{5.1}$$

where, for example, the *CS* set includes all pairs of (j, k) corresponding to a *CS* transition according to Table 5.3.

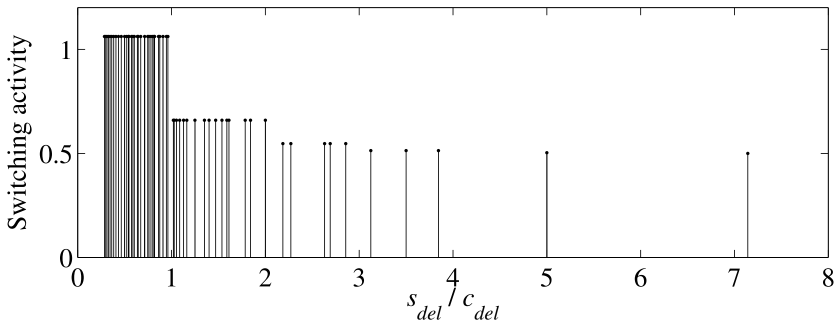


Figure 5.3 Average switching activity at the sum output of an eight-bit ripple-carry adder for simulations using different timing relations. s_{del} is the delay corresponding to a sum computation and c_{del} is the delay corresponding to a carry computation.

5.2.1 Timing Issues

In a full adder, there are two computations performed, and the results are given at the carry and sum outputs, respectively. Both these outputs depend on the carry input. In a ripple-carry adder, the carry output from each full adder (except the last full adder) is input to a subsequent full adder. The effect of this is that a fast computation of the carry result in low switching activity at the sum outputs. This relation is shown in Fig. 5.3, where s_{del} is the delay corresponding to a sum computation and c_{del} is the delay corresponding to a carry computation. ModelSim™ was used for this high-level simulation. The switching activity in Fig. 5.3 is quantified because a different value is obtained depending on the number of carry computations that are performed during one sum computation. Note that the relation of the delays does not affect the carry switching activities, since all carry bits will reach their final value in a certain order, i.e., from LSB to MSB.

All carry switching is assumed to effect the sum output, which means that $0 < s_{del}/c_{del} \leq 1$ and implies that the energy will be overestimated. However, this is still a realistic model since every change at the carry input will in fact affect the sum output, although not always resulting in full swing switches. An example of the difference in high-level simulation between $s_{del}/c_{del} = 1$ and $s_{del}/c_{del} = 1.5$ is shown in Fig. 5.4. In this figure, it can be seen that the carry outputs are not effected while the switching activities at the sum outputs are significant larger for $s_{del}/c_{del} = 1$.

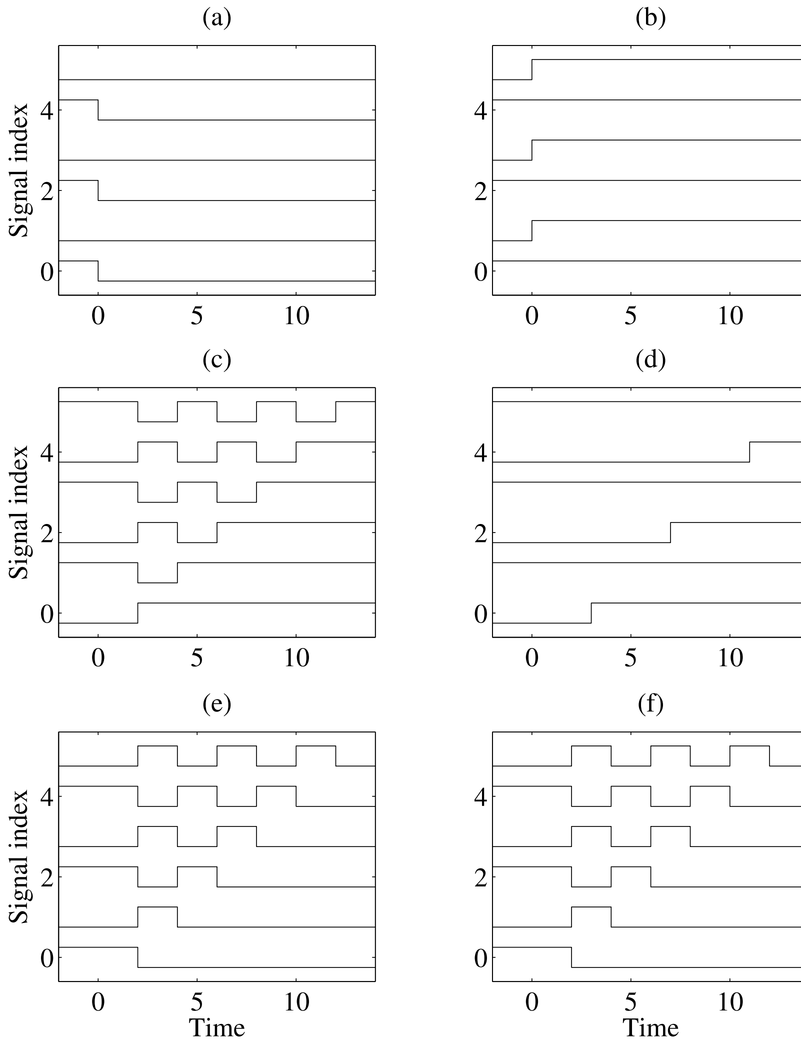


Figure 5.4 High-level simulation of a six-bit ripple-carry adder. The carry input, c_{in}, is constant zero, while the X and Y inputs are changed at time zero. (a) X input, (b) Y input, (c) s_i outputs with $s_{del} = c_{del} = 2$, (d) s_i outputs with $s_{del} = 3$ and $c_{del} = 2$, (e) c_i outputs with $s_{del} = c_{del} = 2$, and (f) c_i outputs with $s_{del} = 3$ and $c_{del} = 2$.

5.3 Switching Activity

In this section, the switching activity in ripple-carry adders is computed. First the generate phase is considered, and then the propagate phase. The

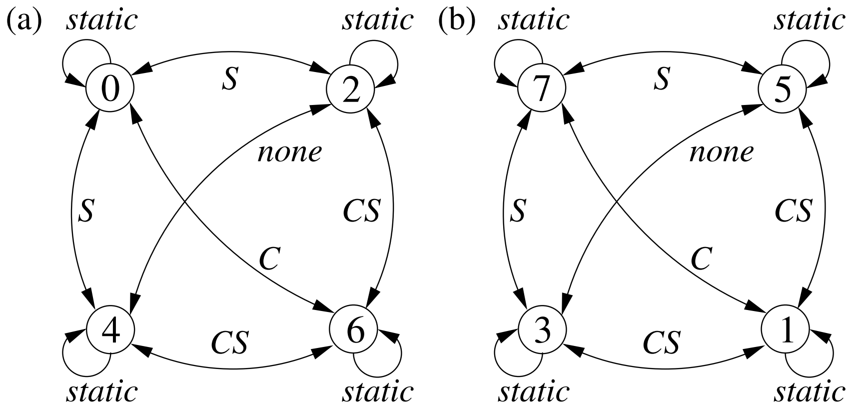


Figure 5.5 State transition graph for a full adder when the data inputs are active. (a) $c_{i-1} = 0$, and (b) $c_{i-1} = 1$.

contributions from the two phases are then merged. Finally, the result is simplified by assuming random inputs and the results are compared with former work.

5.3.1 Switching due to Change of Input

When new input data, x_i and y_i, is applied to a full adder, as shown in Fig. 5.2, the outputs, s_i and c_i, of the full adder may change. Note that the carry input, c_{i-1}, is constant during this phase. This is visualized in Fig. 5.5 using a state transition graph (STG), where the state represents the input values as described in Table 5.2. Each transition between two states result in a certain type of switching activity at the outputs. For example, a transition between state 2 and state 6 result in a transition at both outputs, c_i and s_i. Note that the situations where no changes of the outputs occur can be divided into the case when the inputs does not change, and the case when both inputs change, i.e., *static* and *none* transitions, respectively. One exception from the STG in Fig. 5.5 is the first full adder (LSB), for which all three inputs may change during the generation phase.

Transition Probabilities in the Generation Phase

The goal with the following computations is to find the probability for each possible transition, i.e., to obtain a probability matrix corresponding to Table 5.3. The row index j, column index k, and full adder index i will be used. The probability function, $P(u)$, will be used to state the probabil-

ity that a signal, u, have the logic value one. The function, $\alpha(u)$, will be used to state the switching activity of the signal u. The input data, x_i and y_i, are assumed to be random in the sense that the probabilities for zeros and ones are equal, i.e.,

$$P(x_i) = P(y_i) = \frac{1}{2} \tag{5.2}$$

The initial state (superscript I) and final state (superscript F) are defined by the input values as

$$\begin{cases} S^I_{i,j} = \begin{bmatrix} x^I_{i,j} & y^I_{i,j} & c^I_{i-1,j} \end{bmatrix} & \begin{cases} x^I_{i,j}, y^I_{i,j}, c^I_{i-1,j} \in \{0, 1\} \\ 0 \le j \le 7 \end{cases} \\ S^F_{i,k} = \begin{bmatrix} x^F_{i,k} & y^F_{i,k} & c^F_{i-1,k} \end{bmatrix} & \begin{cases} x^F_{i,k}, y^F_{i,k}, c^F_{i-1,k} \in \{0, 1\} \\ 0 \le k \le 7 \end{cases} \end{cases} \tag{5.3}$$

By applying the logic XOR operation, denoted by $\oplus$, to the state descriptions, a vector of length three containing ones for changed and zeros for unchanged inputs, is obtained as

$$\alpha_{i,j,k} = S^I_{i,j} \oplus S^F_{i,k} \qquad \begin{cases} 0 \le j \le 7 \\ 0 \le k \le 7 \end{cases} \tag{5.4}$$

Due to the assumption in (5.2), the carry input, c_{i-1}, is the only input that affects the state probabilities. Hence, there are only two different state probabilities, which are defined as

$$\begin{cases} P(S^0_i) = \frac{1}{4}(1 - P(c_{i-1})) \\ P(S^1_i) = \frac{1}{4}P(c_{i-1}) \end{cases} \tag{5.5}$$

where the superscript indicates the carry input value. The carry input probability is

$$P(c_{i-1}) = \begin{cases} P(c_{in}) & i = 0 \\ \frac{1}{4} + \frac{1}{2}P(c_{i-2}) & i > 0 \end{cases} \tag{5.6}$$

Elements in the probability matrix are denoted by the variable q, with indices to describe full adder, row, and column. Each value, q, is obtained by multiplying the corresponding state probability, according to (5.5), with the input switching activities, $\alpha(x_i)$ and $\alpha(y_i)$. This is done with respect to the switching vector defined in (5.4), so that the complementary switching activities, $1 - \alpha(x_i)$ and $1 - \alpha(y_i)$, are used if the corresponding input does not change. For the first full adder (corresponding to the LSB) the carry input, c_{in}, also has to be considered. Hence, each matrix element can be computed as

$$\begin{aligned} q_{i,j,k}^{gen} = {} & (c_{i-1,j}^{I} P(S_i^1) + \overline{c_{i-1,j}^{I}} P(S_i^0)) \cdot \\ & (\alpha_{i,j,k}(1)\alpha(x_i) + \overline{\alpha_{i,j,k}(1)}(1 - \alpha(x_i))) \cdot \\ & (\alpha_{i,j,k}(2)\alpha(y_i) + \overline{\alpha_{i,j,k}(2)}(1 - \alpha(y_i))) \cdot \\ & (\alpha_{i,j,k}(3)\alpha(c_{i-1}^{gen}) + \overline{\alpha_{i,j,k}(3)}(1 - \alpha(c_{i-1}^{gen}))) \end{aligned} \tag{5.7}$$

$$\text{where } \begin{cases} 0 \le j \le 7 \\ 0 \le k \le 7 \end{cases} \text{ and } \alpha(c_{i-1}^{gen}) = \begin{cases} \alpha(c_{in}) & i = 0 \\ 0 & i > 0 \end{cases}$$

since the carry input only can change for the first full adder (LSB) in the generate phase.

5.3.2 Switching due to Carry Propagation

In the propagate phase the data inputs, x_i and y_i, are constant, while the carry input, c_{i-1}, may change several times. In Fig. 5.6, the STG corresponding to the propagate phase is shown. Note that all cases without changes of the outputs are static.

Transition Probabilities in the Propagation Phase

In the following, the propagate part of the probability matrix will be derived. Since there are no transitions at the inputs of the first full adder

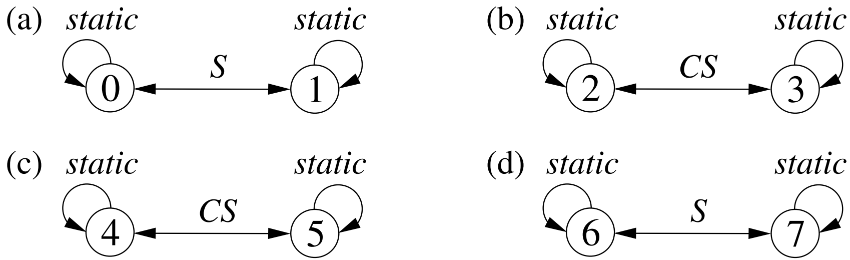


Figure 5.6 State transition graph for a full adder when the carry input is active. (a) $x_i = y_i = 0$, (b) $x_i = 0$, $y_i = 1$, (c) $x_i = 1$, $y_i = 0$, and (d) $x_i = y_i = 1$.

(LSB) in the propagate phase, the full adder index, *i,* is larger than zero in this section.

Let the variable β represent a weighted value for the total carry switching activity at the carry input, i.e., both *C* and *CS* transitions are included. Using the probability matrix of the preceding full adder, β is divided into equal parts for the eight states and computed as

$$\beta_i = \frac{1}{8}\sum_{j,k} q_{i-1,j,k} \qquad (j,k) \in C \cup CS \tag{5.8}$$

Since only the carry input can change in the generation phase, transitions are limited within 2×2 parts of the probability matrix. These transitions are described by

$$\begin{bmatrix} q^{pro}_{i,j,k} & q^{pro}_{i,j,k+1} \\ q^{pro}_{i,j+1,k} & q^{pro}_{i,j+1,k+1} \end{bmatrix} = \begin{bmatrix} iP(S_i^0) - \beta_i & \beta_i \\ \beta_i & iP(S_i^1) - \beta_i \end{bmatrix} \tag{5.9}$$
where $(j,k) \in \{(0,0),(2,2),(4,4),(6,6)\}$

For all elements in the matrix not covered by (5.9), we have

$$q^{pro}_{i,j,k} = 0 \tag{5.10}$$

5.3.3 Total Switching Activity

By adding the contributions from the generate and propagate phases, the total switching activity is obtained. Each matrix element is computed as

static	β	$P(S^0)\alpha(y)(1-\alpha(x))$	0	$P(S^0)\alpha(x)(1-\alpha(y))$	0	$P(S^0)\alpha(x)\alpha(y)$	0
β	*static*	0	$P(S^1)\alpha(y)(1-\alpha(x))$	0	$P(S^1)\alpha(x)(1-\alpha(y))$	0	$P(S^1)\alpha(x)\alpha(y)$
$P(S^0)\alpha(y)(1-\alpha(x))$	0	*static*	β	$P(S^0)\alpha(x)\alpha(y)$	0	$P(S^0)\alpha(x)(1-\alpha(y))$	0
0	$P(S^1)\alpha(y)(1-\alpha(x))$	β	*static*	0	$P(S^1)\alpha(x)\alpha(y)$	0	$P(S^1)\alpha(x)(1-\alpha(y))$
$P(S^0)\alpha(x)(1-\alpha(y))$	0	$P(S^0)\alpha(x)\alpha(y)$	0	*static*	β	$P(S^0)\alpha(y)(1-\alpha(x))$	0
0	$P(S^1)\alpha(x)(1-\alpha(y))$	0	$P(S^1)\alpha(x)\alpha(y)$	β	*static*	0	$P(S^1)\alpha(y)(1-\alpha(x))$
$P(S^0)\alpha(x)\alpha(y)$	0	$P(S^0)\alpha(x)(1-\alpha(y))$	0	$P(S^0)\alpha(y)(1-\alpha(x))$	0	*static*	β
0	$P(S^1)\alpha(x)\alpha(y)$	0	$P(S^1)\alpha(x)(1-\alpha(y))$	0	$P(S^1)\alpha(y)(1-\alpha(x))$	β	*static*

Table 5.6 Probability matrix for $i > 0$. The full adder index, i, is not included in this table.

$$q_{i,j,k} = q_{i,j,k}^{gen} + q_{i,j,k}^{pro} \qquad \begin{cases} 0 \le j \le 7 \\ 0 \le k \le 7 \end{cases} \tag{5.11}$$

The resulting probability matrix, for any of the full adders (except for the first one), is shown in Table 5.6. It can be shown that a static transition is negligible compared to the other types of transitions. Static transitions are therefore excluded in Table 5.6 and ignored in the energy model. When the probability matrix for each full adder is derived, the total energy can be computed by summing all products obtained by multiplying every matrix element with the corresponding transition energy, as given in Table 5.4. This method will be referred to as the matrix model. Since this is a rather complicated method, a simplified model will be discussed in the following.

Simplified Method Using Average Energies

The aim of this simplified method is to compute the probabilities for each kind of transition, and then compute the energy consumption using the average energies, as given in Table 5.5. The different types of switching activities are named, $\alpha(transition_i)$, where *transition* corresponds to the symbols defined in Table 5.1. By summing all C transitions (see Table 5.3) in Table 5.6, the switching activity for this type is obtained as

$$\alpha(C_i) = 2(P(S_i^0) + P(S_i^1))\alpha(x_i)\alpha(y_i) = \frac{1}{2}\alpha(x_i)\alpha(y_i) \qquad i > 0 \quad (5.12)$$

for each full adder. The switching activity corresponding to S transitions is computed as

$$\begin{aligned}\alpha(S_i) &= 4\beta_i + 2(P(S_i^0) + P(S_i^1))(\alpha(x_i)(1-\alpha(y_i)) + \\ &\quad \alpha(y_i)(1-\alpha(x_i))) = \\ &= 4\beta_i + \frac{1}{2}(\alpha(x_i) + \alpha(y_i)) - 2\alpha(C_i) \qquad i > 0\end{aligned} \quad (5.13)$$

Finally, the switching activities corresponding to the two remaining types of transitions are obtained directly from (5.12) and (5.13) according to

$$\begin{cases}\alpha(none_i) = \alpha(C_i) \\ \alpha(CS_i) = \alpha(S_i)\end{cases} \qquad i > 0 \quad (5.14)$$

In (5.13), the variable β_i is required. To compute this value according to (5.8), it is necessary to produce the probability matrix. However, the expression in (5.8) can now be simplified to

$$\beta_i = \frac{1}{8}(\alpha(C_{i-1}) + \alpha(CS_{i-1})) \qquad i > 0 \quad (5.15)$$

The transition activities are now fully specified for all full adders except the first (LSB). All elements in the probability matrix corresponding to the first full adder are computed according to (5.7), since there are no transitions in the propagation phase. From the probability matrix, the different types of transition activities are obtained as

$$\begin{cases}\alpha(C_0) = \frac{1}{2}(\alpha(x_0)\alpha(y_0) + (\alpha(x_0) + \alpha(y_0) - 3\alpha(x_0)\alpha(y_0))\alpha(c_{in})) \\ \alpha(none_0) = \alpha(C_0) \\ \alpha(S_0) = \frac{1}{2}(\alpha(x_0) + \alpha(y_0) + (1 - 3\alpha(x_0)\alpha(y_0))\alpha(c_{in})) - 2\alpha(C_0) \\ \alpha(CS_0) = \alpha(S_0) + \alpha(x_0)\alpha(y_0)\alpha(c_{in})\end{cases} \quad (5.16)$$

Note that the switching activities in (5.16) are equal to the corresponding expressions in (5.12), (5.13), and (5.14) if both $\alpha(c_{in})$ and β_i are zero. This can be used to also include the first full adder in a general equation according to

$$\begin{cases} \alpha(C_i) = \frac{1}{2}(\alpha(x_i)\alpha(y_i) + (\alpha(x_i) + \alpha(y_i) - 3\alpha(x_i)\alpha(y_i))\alpha(c_{in})) \\ \alpha(S_i) = 4\beta_i + \frac{1}{2}(\alpha(x_i) + \alpha(y_i) + (1 - 3\alpha(x_i)\alpha(y_i))\alpha(c_{in})) - 2\alpha(C_i) \\ \alpha(CS_i) = \alpha(S_i) + \alpha(x_i)\alpha(y_i)\alpha(c_{in}) \\ \alpha(none_i) = \alpha(C_i) \end{cases} \tag{5.17}$$

where $\beta_i = \begin{cases} 0 & i = 0 \\ \frac{1}{8}(\alpha(C_{i-1}) + \alpha(CS_{i-1})) & i > 0 \end{cases}$

$$\alpha(c_{in}) = 0 \qquad i > 0$$

5.3.4 Uncorrelated Input Data

If uncorrelated input data, x_i and y_i, are assumed, i.e.,

$$\alpha(x_i) = \alpha(y_i) = \frac{1}{2} \tag{5.18}$$

the transition activities can be simplified. Under this assumption the transition activities in the first full adder can be derived from (5.16) as

$$\begin{cases} \alpha(C_0) = \frac{1}{8}(1 + \alpha(c_{in})) \\ \alpha(S_0) = \frac{1}{8}(2 - \alpha(c_{in})) \end{cases} \text{ and } \begin{cases} \alpha(none_0) = \alpha(C_0) \\ \alpha(CS_0) = \frac{1}{8}(2 + \alpha(c_{in})) \end{cases} \tag{5.19}$$

From (5.12) and (5.13), the switching activities corresponding to C and S transitions for the remaining full adders are computed as

$$\alpha(C_i) = \frac{1}{8} \text{ and } \alpha(S_i) = 4\beta_i + \frac{1}{4} \text{ where } i > 0 \tag{5.20}$$

By summing the contribution from all full adders, a closed-form expression of the total switching activity corresponding to C transitions, $\alpha(C_{tot})$, is obtained as

$$\alpha(C_{tot}) = \alpha(C_0) + \sum_{i=1}^{n} \alpha(C_i) = \frac{1}{8}(1 + n + \alpha(c_{in})) \tag{5.21}$$

where $n + 1$ is the number of full adders, as illustrated in Fig. 5.1.

In (5.20), the variable β_i is required. This value can be computed according to (5.15) as

$$\begin{cases} \beta_1 = \frac{1}{8}(\alpha(C_0) + \alpha(CS_0)) = \frac{1}{64}(3 + 2\alpha(c_{in})) \\ \beta_i = \frac{1}{8}\left(\frac{1}{8} + 4\beta_{i-1} + \frac{1}{4}\right) = \frac{3}{64} + \frac{\beta_{i-1}}{2} \qquad i > 1 \end{cases} \tag{5.22}$$

The total switching activity corresponding to S transitions, $\alpha(S_{tot})$, is obtained by summing the contribution from all full adders as

$$\begin{aligned} \alpha(S_{tot}) &= \alpha(S_0) + \sum_{i=1}^{n} \alpha(S_i) = \alpha(S_0) + \frac{n}{4} + 4\sum_{i=1}^{n} \beta_i = \\ &= \alpha(S_0) + \frac{n}{4} + 4\left(\frac{3n}{32} + 2\left(\beta_1 - \frac{3}{32}\right)(1 - 2^{-n})\right) = \\ &= \frac{1}{8}(2 + 5n - \alpha(c_{in}) + (2\alpha(c_{in}) - 3)(1 - 2^{-n})) \end{aligned} \tag{5.23}$$

Finally, the switching activities corresponding to the two remaining types of transitions are obtained as

$$\begin{cases} \alpha(none_{tot}) = \alpha(C_{tot}) \\ \alpha(CS_{tot}) = \alpha(S_{tot}) + \frac{1}{4}\alpha(c_{in}) \end{cases} \tag{5.24}$$

Uncorrelated Carry Input

In this section, earlier presented models are derived by simplifying this new model. If the carry input, c_{in}, is assumed to be random, i.e.,

$$\alpha(c_{in}) = \frac{1}{2} \tag{5.25}$$

the total switching activities in (5.21), (5.23), and (5.24) are simplified to

$$\begin{cases} \alpha(C_{tot}) = \frac{1}{16}(3+2n) \\ \alpha(S_{tot}) = \frac{1}{8}\left(5n - \frac{1}{2} + 2^{1-n}\right) \end{cases} \text{ and } \begin{cases} \alpha(none_{tot}) = \alpha(C_{tot}) \\ \alpha(CS_{tot}) = \alpha(S_{tot}) + \frac{1}{8} \end{cases} \tag{5.26}$$

For the model presented in [35], an equation comparable to (5.26) was given. For example, if c_{in} is constant, i.e., 0 (adder) or 1 (subtractor), the value of $\alpha(c_{in})$ is zero. This case is handled by the closed-form expressions in (5.21), (5.23), and (5.24), but not by (5.26). Hence, the authors of [35] did not consider the carry input, c_{in}, but assumed it to be random as well.

For uncorrelated input data, it can be shown that

$$\lim_{i \to \infty} \beta_i = \frac{3}{32} \Rightarrow \lim_{i \to \infty} \alpha(S_i) = \lim_{i \to \infty} \alpha(CS_i) = \frac{5}{8} \tag{5.27}$$

The switching activity corresponding to the two remaining types of transitions can be obtained directly from (5.14) and (5.20) as

$$\lim_{i \to \infty} \alpha(C_i) = \lim_{i \to \infty} \alpha(none_i) = \frac{1}{8} \tag{5.28}$$

The values in (5.27) and (5.28) were used in the model presented in [97], where the energy estimation consequently was stated as a function of the wordlength without considering each full adder individually.

5.3.5 Summary

The differences between the discussed models are summarized in Table 5.7, where the model name TMMM stands for Transition Model

Model	Equations	Energy	Inputs	Carry
Matrix	(5.3) – (5.11)	Table 5.4	Corr.	Corr.
TMMM Corr.	(5.17)	Table 5.5	Corr.	Corr.
TMMM Random	(5.21), (5.23), (5.24)	Table 5.5	Random	Corr.
Carry chain II [35]	(5.26)	Table 5.5	Random	Random
Carry chain [97]	(5.27), (5.28)	Table 5.5	Random	–

Table 5.7 Comparison of different energy models for ripple-carry adders.

based on Matrix Model, and denotes the simplified model using average energy values.

Usually, switching activity is associated with a specific signal rather than a type of transition. However, the switching activities for a carry bit c_i and a sum bit s_i can be directly obtained from the transition activities according to

$$\begin{cases} \alpha_{c_i} = \alpha(C_i) + \alpha(CS_i) \\ \alpha_{s_i} = \alpha(S_i) + \alpha(CS_i) \end{cases} \tag{5.29}$$

5.4 Experimental Results

To validate the model, transistor level simulations were performed for an AMS 0.35 µm CMOS process using Spectre™. The full adder used was a mirror adder from [119]. The delay relation, which was discussed in Section 5.2.1, for this full adder is $1 < s_{del}/c_{del} < 2$. Hence, not all switches in the transistor level simulation will be rail-to-rail.

The energy consumed for each state change of the full adder is shown in Table 5.8, while the average energies for different change of the outputs are given in Table 5.9. In this characterization, the sum output was loaded with an inverter, while the carry output was loaded with the carry input of an identical full adder. All simulation results are average values over 1000 pairs of input samples at a data rate of 50 MHz.

State	0	1	2	3	4	5	6	7
0	0.624	168.8	179.7	391.4	211.2	512.6	533.8	324.6
1	316.6	0.477	87.32	401.4	106.0	417.5	512.9	146.9
2	368.5	102.7	0.923	458.6	92.03	478.8	441.0	195.4
3	426.9	579.3	555.8	0.713	559.1	103.9	123.8	207.6
4	385.0	123.3	66.09	487.5	1.952	476.9	482.6	239.0
5	439.4	618.8	580.6	78.56	623.1	0.967	163.6	237.9
6	467.4	637.8	634.5	132.6	646.1	122.2	1.008	283.7
7	722.8	605.4	632.2	381.5	519.4	350.6	329.4	2.948

Table 5.8 Energy, in fJ, corresponding to a transition from the state in the left column to the state in the heading row.

Transition	*CS*	*C*	*S*	*none*	*static*
Energy [fJ]	528.0	438.4	285.0	85.14	1.202

Table 5.9 Average energy, in fJ, corresponding to the different types of transitions.

5.4.1 Uncorrelated Data

The first validation is for uncorrelated input data. Here, we consider both constant and random carry input, c_{in}. The results are given in Table 5.10, and compared with the results obtained from [35] and [97].

It is worth noting that for random carry input, the method in [35] and the proposed simplified (TMMM) method gives identical results, as previously discussed. The differences between the proposed methods and the Spectre™ results are due to the non rail-to-rail switching present in the transistor level simulation. In [35], the results were compared to those obtained by HEAT [124], and a close correspondence was shown. However, HEAT is based on switching activities as well. Hence, all switches are assumed full swing in HEAT.

When each individual full adder is considered, the results in Table 5.11 are obtained for a four-bit ripple-carry adder. The difference for the most significant full adder is partly due to that the carry output is

Bits $n+1$	Constant $c_{in}=0$			Random c_{in}				Any c_{in}
	Sim.	Matrix	TMMM	Sim.	Matrix	TMMM	Carry chain II [35]	Carry chain [97]
4	368.5	435.5	430.5	383.5	464.5	465.0	465.0	573.6
8	420.5	499.0	497.8	422.3	514.8	516.3	516.3	573.6
12	431.3	522.8	522.8	433.3	533.5	535.2	535.2	573.6
16	434.1	535.0	535.5	440.6	542.9	544.8	544.8	573.6
20	440.1	542.2	543.1	443.1	548.5	550.6	550.6	573.6
24	439.3	547.0	548.2	444.3	552.3	554.4	554.4	573.6
32	443.8	553.1	554.5	446.9	557.0	559.2	559.2	573.6

Table 5.10 Average energy consumption per bit, in fJ, for ripple-carry adders with different wordlengths. Simulated and derived by the proposed methods, as well as the methods from [35] and [97].

FA no.	Sim.	Matrix	TMMM	Carry chain II [35]	Carry chain [97]
1	318.8	319.8	316.6	316.6	573.6
2	380.6	470.8	472.0	472.0	573.6
3	421.0	521.0	522.8	522.8	573.6
4	412.4	546.0	548.2	548.2	573.6

Table 5.11 Average energy consumption, in fJ, for individual full adders in a four-bit ripple-carry adder with random carry input, c_{in}.

now connected to an inverter. Note that the proposed methods are close to the simulation results for the first full adder, due to the fact that all switches in this full adder are rail-to-rail.

5.4.2 Correlated Data

The main advantage of the proposed method over previous methods is the ability to estimate the energy consumption for correlated data. Here, two different cases of correlated input signals are presented.

First, two input signals with typical two's-complement correlation, present in many real world applications are applied. The switching activities for the input signals are shown in Fig. 5.7 (a). In Figs. 5.7 (b) and (c), the resulting switching activities for all carry and sum output signals are shown, respectively. These are compared with simulated results from a VHDL model in ModelSim™, using two different delay relations. For the simulation using the same delay relation as the model was based upon, there is a good fit for the switching activities at the outputs. The difference of the switching activities at the sum outputs obtained in ModelSim™ is due to the number of full swing switches. Hence, the simulation with $s_{del}/c_{del} = 1.0$ shows the total number of switches, while the simulation with $s_{del}/c_{del} = 1.5$ shows the number of full swing switches. As was stated in Section 5.2.1, the delay relation is shown not to affect the switching activities at the carry outputs. Finally, in Fig. 5.7 (d), the estimated energy consumption for each full adder cell is shown together with the transistor level simulation results. As can be seen in Fig. 5.7 (d), the energy consumption is overestimated, except for the first full adder. Again, the differences are due to not all switches in the transistor level simulation being full swing. The differences between the full matrix method and the simplified TMMM method are small for this case, mainly due to the symmetry between the x_i and y_i inputs of the used full adder. Hence, the average energy consumption values in Table 5.9 are useful. However, for full adders with non-symmetric x_i and y_i inputs one can expect a larger deviation between the proposed methods, with the matrix method producing results that are more accurate.

To further validate the model, two more unrealistic input signals are applied. The switching activities for these signals are shown in Fig. 5.8 (a). The switching activity of the carry and sum outputs are given in Figs. 5.8 (b) and (c), respectively. Again, there is a very close match between the simulated results and the proposed method. Figure 5.8 (d) illustrates the estimated and simulated energy consumption.

5.5 Adopting the Dual Bit Type Method

When the previous discussed model, referred to as TMMM, is to be used, knowledge about the switching activity for all input bits is required. The reason for such models is, as stated before, that in real world signals, the switching activities are different for different bit positions. However, certain region properties can be observed [78],[89],[120]. The least signifi-

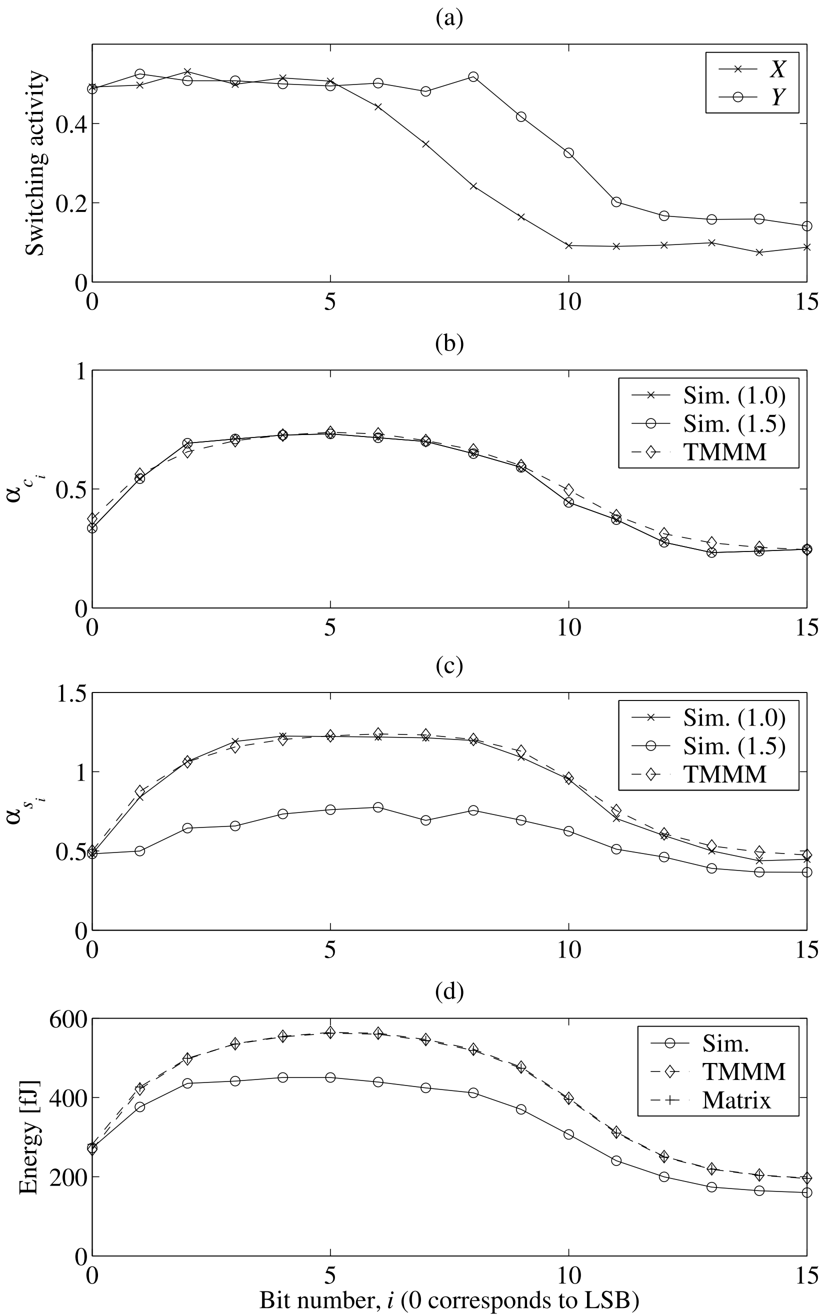


Figure 5.7 Switching activities of single bits for (a) input signals, (b) carry output, and (c) sum output. (d) Energy consumption. In (b) and (c), the value in brackets for the simulations is the delay relation, s_{del} / c_{del}, as discussed in Section 5.2.1.

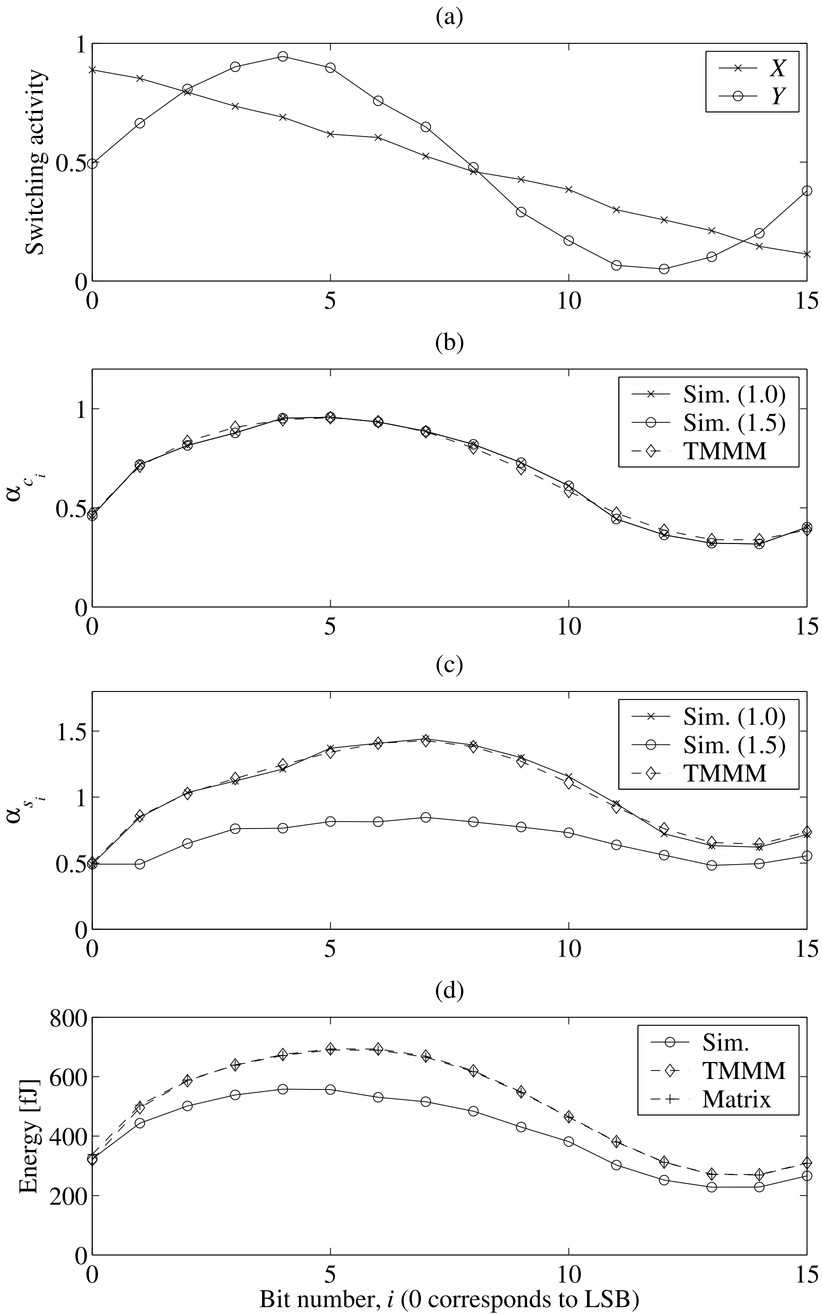


Figure 5.8 Switching activities of single bits for (a) input signals, (b) carry output, and (c) sum output. (d) Energy consumption. In (b) and (c), the value in brackets for the simulations is the delay relation, s_{del} / c_{del}, as discussed in Section 5.2.1.

cant bits have a random switching activity, while the switching activity of the most significant bits, which contain two's-complement sign information, depend on the correlation. Hence, a model that utilize the region properties and thereby provides a relevant base for high-level energy modeling and estimation of DSP systems would be appropriate.

In the following, such a model for estimation of the switching activity in ripple-carry adders with correlated input data is presented. This model is based on word-level statistics, i.e., mean, variance, and correlation, of the two input signals to be added. From this statistics, the region properties can be found for signals using the dual bit type method [78].

Finally, the switching activities for the four different types of transitions used in the previously presented models, i.e., *C*, *S*, *CS*, and *none* transitions, are computed for the different regions using equations based on TMMM. In [89], where sign magnitude representation was used, only one type of switching activity was considered, which is not sufficient to obtain an accurate energy estimate.

By comparison with high-level simulation, it is shown that the proposed model gives accurate results in terms of switches when the applied two's-complement represented inputs are real world signals.

5.5.1 Statistical Definition of Signals

Signals in real world applications can in many cases be approximated as Gaussian stochastic variables, i.e., the corresponding probability density function (PDF) has the characteristic illustrated in Fig. 5.9. As can be seen in Fig. 5.9 the probability is highest at the mean, μ, and is almost zero at three standard deviations (3σ) away from the mean. Note that the standard deviation, σ, can be obtained from the variance, σ^2.

The coefficient of correlation is defined as

$$\rho(A, B) = \frac{\mu_{AB} - \mu_A \mu_B}{\sigma_A \sigma_B} \tag{5.30}$$

In the case of a single signal, the temporal correlation, ρ, is computed from (5.30) where B takes the same values as A, but delayed one time unit.

To summarize, signals can be defined by the word-level statistics mean, μ, deviation, σ, and correlation, ρ.

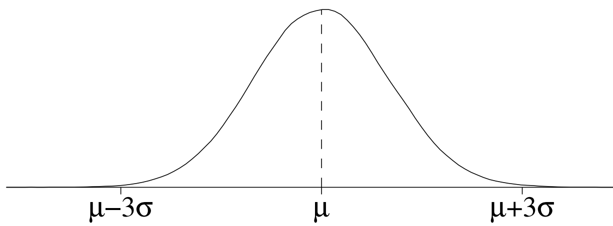


Figure 5.9 Probability density function for normal distribution.

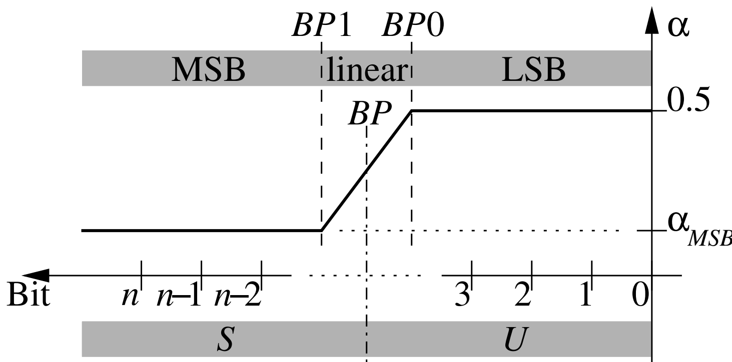


Figure 5.10 Breakpoints and regions used in the DBT model.

5.5.2 The DBT Method

The dual bit type (DBT) method is based on the fact that the binary representation of most real world signals, such as sound, can be divided into regions, where the bits of each region has a well defined switching activity. In [78], the three regions LSB, linear, and MSB were defined as illustrated in Fig. 5.10. Because of the linear approximation of the middle region, it was stated that two bit types is sufficient. Hence, the bits are divided into a uniform white-noise (UWN) region, *U*, and a sign region, *S*, as illustrated in Fig. 5.10.

The breakpoints that separate the regions, can be computed from word-level statistics as [78]

$$\begin{cases} BP0 = \log_2\sigma + \log_2\left(\sqrt{1-\rho^2} + \frac{|\rho|}{8}\right) \\ BP1 = \log_2(|\mu| + 3\sigma) \end{cases} \tag{5.31}$$

The intermediate breakpoint is here defined as

$$BP = \left\lfloor \frac{1}{2}(BP0 + BP1) \right\rfloor \tag{5.32}$$

The reason for an integer value as intermediate breakpoint is to enable a verification using TMMM.

If p_Q is the probability that a single-bit signal, Q, is one and the temporal correlation of Q is ρ_Q, then the switching activity, α_Q, of Q is defined as [120]

$$\alpha_Q = 2p_Q(1 - p_Q)(1 - \rho_Q) \tag{5.33}$$

The probability for a one is now assumed to be 1/2 for all bits in the two's-complement representation of a signal. Note that this is true if the mean, μ, is zero, which is the case if the two's-complement number range is utilized efficiently. Furthermore, it was stated in [120] that the temporal correlation for bits in the MSB region is close to the word-level correlation, ρ. Hence, the switching activity in the MSB region can be computed from (5.33) as

$$\alpha_{MSB} = \frac{1}{2}(1 - \rho) \tag{5.34}$$

In the case of a ripple-carry adder, the two signals to be added, X and Y, may have different word-level statistics. Hence, the intermediate breakpoints, computed from (5.32), may be different, resulting in the general case illustrated in Fig. 5.11. Without loss of generality, BP_Y is defined to be at least as large as BP_X. As can be seen in Fig. 5.11, there are three different regions to be considered. The number of bits in each region is obtained from the breakpoints as

$$\begin{cases} N_{UU} = BP_X + 1 \\ N_{SU} = BP_Y - BP_X \\ N_{SS} = n - BP_Y \end{cases} \tag{5.35}$$

The number of bits in each region are integer values since the breakpoints are rounded towards minus infinity. Note that the sum $N_{SS} + N_{SU} + N_{UU}$ is equal to the total number of bits, $n + 1$, in the ripple-carry adder.

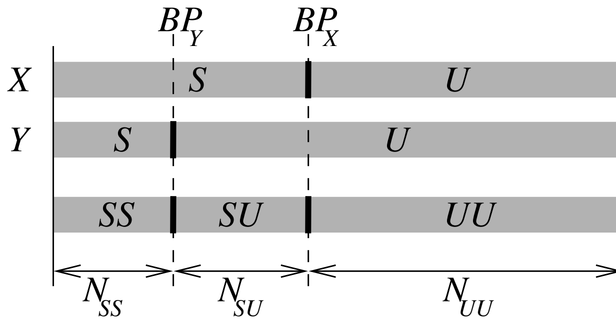


Figure 5.11 Model of the different regions for a ripple-carry adder (which of course is suitable for all two-input modules [78]).

5.5.3 DBT Model for Switching Activity in RCA

From the switching activity of each input bit, x_i, y_i, and c_{in}, the switching activity, α(*transition*), for each type of transition can be computed according to (5.17). This model, TMMM, was derived under the same assumption that was used in Section 5.5.2, i.e., that the probability for a one is 1/2 for all input bits, x_i and y_i. Note that this model requires that the switching activity of each input bit is known and that the computations must be performed in sequence, since the result from the previous bit position is required.

Here simplified equations, based on the region properties of real world signals, will be derived. From (5.17), it is clear that the switching activities corresponding to the transition types *CS* and *none* are easily obtained from the transition types *C* and *S*. Therefore, *CS* and *none* transitions will not be considered in this section.

In the first region, *UU*, the bits of both inputs are modeled to be random, hence

$$\alpha(x_i) = \alpha(y_i) = \frac{1}{2} \tag{5.36}$$

If (5.36) is applied to (5.17) and the sum is computed for all $i \in UU$, the total switching activities are obtained as

$$\begin{cases} \alpha(C_{UU}) = \frac{1}{8}(N_{UU} + \alpha(c_{in})) \\ \alpha(S_{UU}) = \frac{1}{8}(5N_{UU} - 3 - \alpha(c_{in}) + (2\alpha(c_{in}) - 3)(1 - 2^{1 - N_{UU}})) \end{cases} \tag{5.37}$$

In the second region, SU, one of the inputs, X, is modeled to be sign extended while the other input, Y, is random, i.e.,

$$\alpha(x_i) = \alpha_{MSB,X} \text{ and } \alpha(y_i) = \frac{1}{2} \tag{5.38}$$

Usually, the number of bits in the UU region, i.e., N_{UU}, is large, and therefore the probability that a change of the carry input, c_{in}, should propagate into the SU region is neglectable. Hence, to simplify the equations for the SU and SS regions it will be assumed that the carry input does not affect the switching activity, i.e., $\alpha(c_{in}) = 0$ is used. The total switching activities are obtained in the same way as before according to

$$\begin{cases} \alpha(C_{SU}) = \frac{1}{4}\alpha_{MSB,X}N_{SU} \\ \alpha(S_{SU}) = (1-2^{-N_{SU}})\Big(\frac{3}{4}(1-2^{-N_{UU}}) - \\ \qquad \frac{1}{2}(1+\alpha_{MSB,X})\Big) + \frac{1}{2}\Big(1+\frac{1}{2}\alpha_{MSB,X}\Big)N_{SU} \end{cases} \tag{5.39}$$

For the last region, SS, both inputs are sign extended, so that

$$\alpha(x_i) = \alpha_{MSB,X} \text{ and } \alpha(y_i) = \alpha_{MSB,Y} \tag{5.40}$$

and the total switching activities are

$$\begin{cases} \alpha(C_{SS}) = \frac{1}{2}\alpha_{MSB,X}\alpha_{MSB,Y}N_{SS} \\ \alpha(S_{SS}) = (1-2^{-N_{SS}})\Big(\frac{3}{4}(1-2^{-N_{UU}})2^{-N_{SU}} + \\ \qquad \frac{1}{2}(1+\alpha_{MSB,X})(1-2^{-N_{SU}}) - \\ \qquad \alpha_{MSB,X} - \alpha_{MSB,Y} + \alpha_{MSB,X}\alpha_{MSB,Y}\Big) + \\ \qquad \Big(\alpha_{MSB,X} + \alpha_{MSB,Y} - \frac{3}{2}\alpha_{MSB,X}\alpha_{MSB,Y}\Big)N_{SS} \end{cases} \tag{5.41}$$

Finally, the total switching activities for the RCA are obtained by summing the contributions from each region

$$\begin{cases} \alpha(C_{tot}) = \alpha(C_{UU}) + \alpha(C_{SU}) + \alpha(C_{SS}) \\ \alpha(S_{tot}) = \alpha(S_{UU}) + \alpha(S_{SU}) + \alpha(S_{SS}) \end{cases} \tag{5.42}$$

Note that the expressions in (5.42) are functions of the parameters $\alpha(c_{in})$, which is obtained from the architecture, and N_{UU}, N_{SU}, N_{SS}, $\alpha_{MSB,\,X}$, and $\alpha_{MSB,\,Y}$, which are obtained from the statistical knowledge of the inputs, X and Y, as discussed in Section 5.5.2. Furthermore, no iterative computations are required as opposed to TMMM.

Simplified Model Assuming High Correlation

If the correlation of the input signals is high, i.e., both ρ_X and ρ_Y are close to one, the switching activities of all sign bits will be ignorable. Hence, it can be assumed that

$$\alpha_{MSB,\,X} = \alpha_{MSB,\,Y} = 0 \tag{5.43}$$

which gives simplified equations for the SU and the SS regions as

$$\begin{cases} \alpha(C_{SU}) = \alpha(C_{SS}) = 0 \\ \alpha(S_{SU}) = (1 - 2^{-N_{SU}})\left(\frac{3}{4}(1 - 2^{-N_{UU}}) - \frac{1}{2}\right) + \frac{1}{2}N_{SU} \\ \alpha(S_{SS}) = (1 - 2^{-N_{SS}})\left(\frac{1}{4}(1 - 3 \cdot 2^{-N_{UU}})2^{-N_{SU}} + \frac{1}{2}\right) \end{cases} \tag{5.44}$$

5.5.4 Example

How the switching activity can be estimated using the proposed DBT model will be shown with an example, using real world audio signals. The architecture is a 16 bit ripple-carry adder, i.e., $n = 15$ and c_{in} is constant zero which gives $\alpha(c_{in}) = 0$.

Parts of two different music songs are used as X and Y input signals, which each contain 100 000 samples. The sample frequency is 44.1 kHz and the audio signals are encoded into real values of the numeric range [–1, +1] using 16 bits per sample.

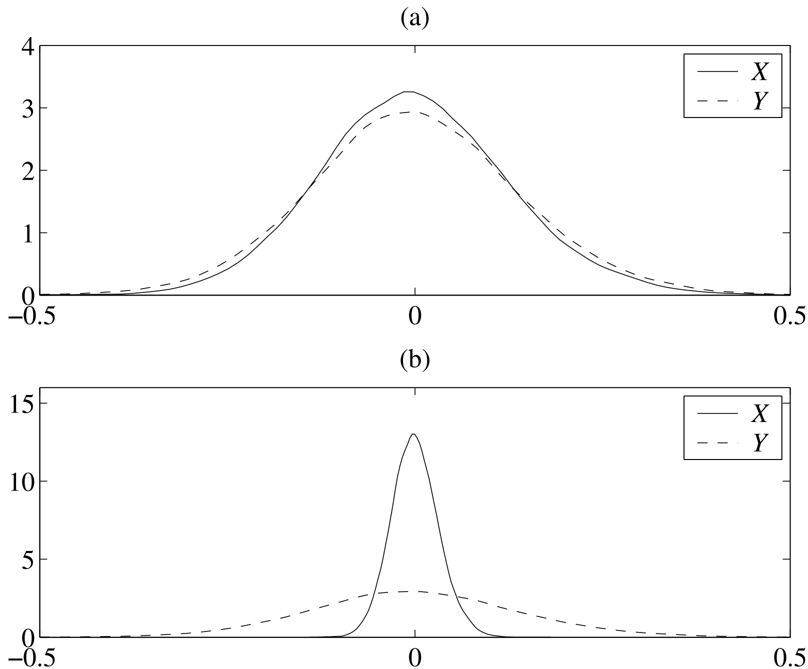


Figure 5.12 (a) Probability density functions for the input signals, X and Y. (b) The X values are divided by four.

Input	μ	σ	ρ
X	–0.00040	0.1269	0.9440
Y	–0.00062	0.1414	0.8652

Table 5.12 Word-level statistics for the two input signals.

The word-level statistics are stated in Table 5.12, and the probability density functions are illustrated in Fig. 5.12 (a). Both signals have a mean close to zero, which was assumed in the model. Since the PDFs for the two signals are similar, a modification is applied to make the example more interesting. The X input is divided by four and rounded to 16 bits, which gives 14 information bits and 2 sign extension bits. The resulting PDFs are shown in Fig. 5.12 (b).

The breakpoints and the switching activity in the MSB region are computed from the equations (5.31), (5.32), and (5.34). Note that (5.31) assume integer representation, i.e., the mean and standard deviation values in Table 5.12 are multiplied by 2^{n-2} (because of the previous division

Input	$BP0$	$BP1$	BP	α_{MSB}
X	8.86	11.61	10	0.028
Y	11.46	13.76	12	0.067

Table 5.13 Model parameters obtained from word-level statistics.

Type	UU	SU	SS	Total
C	1.3750	0.0140	0.0028	1.3918
S	6.1254	1.1907	0.6972	8.0133

Table 5.14 Switching activity for the transition types C and S.

Type	UU	SU	SS	Total
C	1.3750	0.0000	0.0000	1.3750
S	6.1254	1.1872	0.4921	7.8047

Table 5.15 Switching activities using the simplified DBT model.

by four) and 2^n for the X and Y signal, respectively. The results are stated in Table 5.13. The number of bits in each region, obtained from the intermediate breakpoints according to (5.35), is 11, 2, and 3 for the UU, SU, and SS region, respectively.

Finally, the total switching activity corresponding to the transition types C and S are obtained from (5.37), (5.39), and (5.41) for the UU, SU, and SS region, respectively. The sum of all regions gives the total switching activities of the RCA as stated in (5.42). Results are given in Table 5.14. As can be seen in Table 5.15, the corresponding values for the simplified model defined by (5.44) are similar.

The switching activities for the input signals are shown in Fig. 5.13. The dashed lines correspond to the three-region model defined by $BP0$ and $BP1$. However, the proposed model only depends on the intermediate breakpoints, BP, which are illustrated by the dotted lines in Fig. 5.13.

The switching activities for the carry and sum bits, computed according to (5.29), are shown in Fig. 5.14. The presented switching activity model for ripple-carry adders, referred to as DBT, is verified using the TMMM model. These models give identical results for adjusted input switching activities, i.e., according to the dotted lines in Fig. 5.13. Results obtained from a VHDL model in ModelSim™ is also shown in Fig. 5.14.

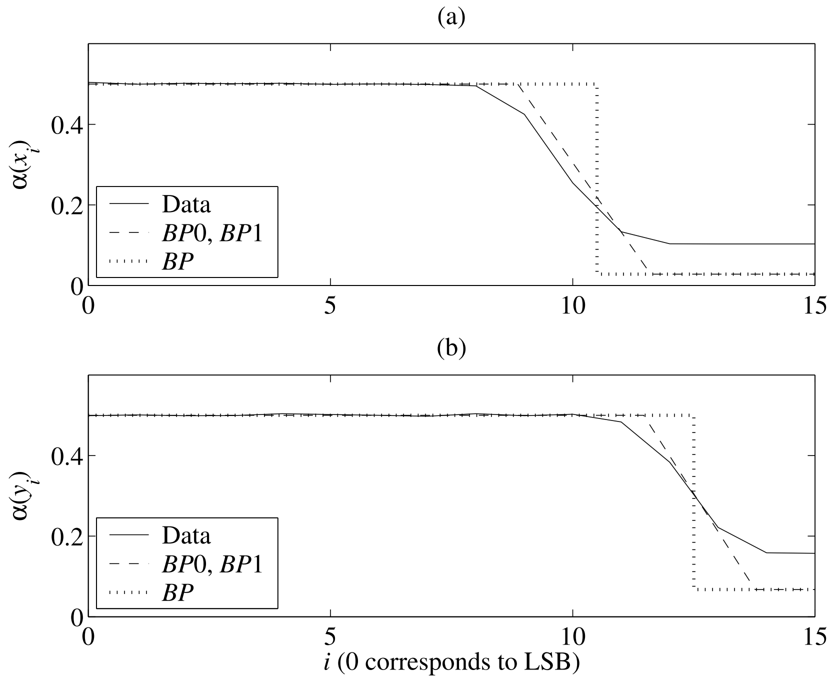


Figure 5.13 Switching activity for the inputs (a) *X* and (b) *Y*.

FA output	Simulation	TMMM	DBT	DBT simple
c_i	0.5875	0.5999	0.5878	0.5737
s_i	1.0124	1.0277	1.0017	0.9756

Table 5.16 Average switching activity per bit at the c_i and s_i outputs.

The DBT model follows the simulation closely. Note that the switching activity is overestimated in the *SU* region and underestimated in the *SS* region. Hence, on average the model will give accurate results as is evident from the values in Table 5.16. This compensation between the regions is the reason that only two signal regions need to be considered in the dual bit type method [78].

5.6 Switching Activity in Constant Multipliers

As illustrated by the examples in Section 4.4.7, the number of adders is often not proportional to the energy consumption. It is therefore of inter-

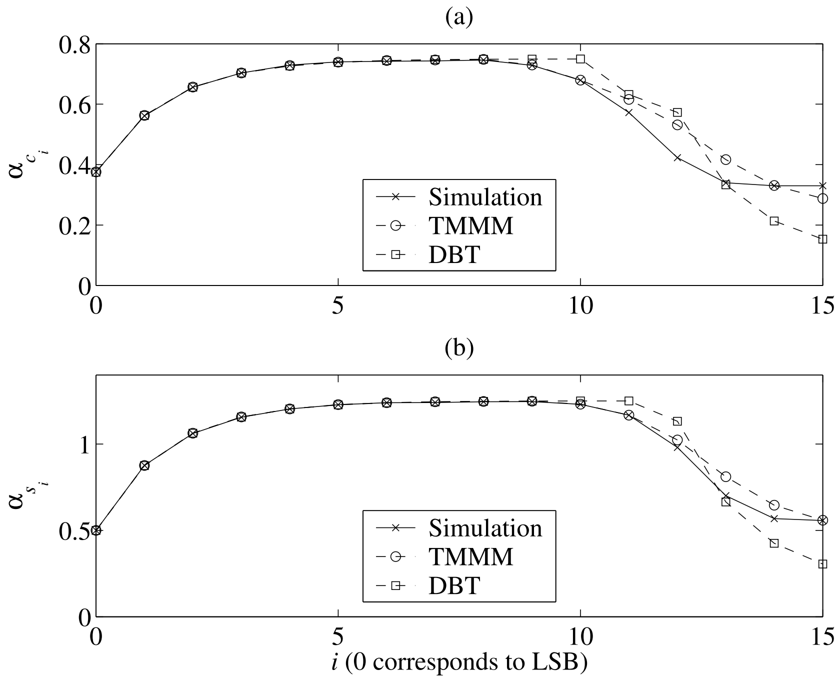


Figure 5.14 The proposed DBT model compared with the TMMM model and high-level simulation. Switching activity for (a) the carry output bits, c_i, and (b) the sum output bits, s_i.

est to determine accurate high-level models for the switching activity to be used in MCM algorithms, aiming at selecting extra fundamentals that will result in low energy implementations. A very simple model, referred to as the glitch path (GP) count, was proposed in [21]. A more detailed version was presented in [22], where empirically derived coefficient values were used in the computation. However, these models are not always in good correlation with the actual switching activity. Here, the goal is to contribute to a more accurate high-level model.

5.6.1 Addition with High Correlation

The model presented in Section 5.3 assumes that the two signals to be added are uncorrelated. However, this will never truly occur in a multiplier block, since all signals origin from the same input data. The highest degree of correlation is, of course, obtained if both inputs to the adder are exactly the same signal, i.e., one is a shifted version of the other. An example of this is the two adders at the lowest depth in Fig. 1.14, where

both adder inputs are connected to the input of the multiplier block. This is an important situation because the adders at depth one will affect the rest of the multiplier. Therefore, a switching activity model for this special case will be derived and compared to simulations in the following.

The input data are assumed to be random, i.e., no correlation between following words is considered. This is not completely true in most applications, since following values of a signal usually are correlated. However, most of the bits will still have a random behavior [78]. Furthermore, the switching activity of the correlated most significant bits is low and therefore does not contribute as much to the overall energy consumption, which justifies this simplification. In the same way as discussed in Section 5.2.1, $0 < s_{del}/c_{del} \leq 1$ is also assumed here for the computation delay of the sum and carry outputs.

If the input is X and the number of shifts is d, there are three possible operations

- $X + 2^d X$, in the following referred to as OP I.
- $X - 2^d X$, in the following referred to as OP II.
- $2^d X - X$, in the following referred to as OP III.

The three cases are illustrated in Fig. 5.15, with $d = 2$. These are the same structures as in Fig. 4.3 with both a_i and b_i connected to the input bit x_i. Note that right shifts never are used in the first stage, i.e., the structures in Fig. 4.4 are not of interest here. As can be seen, half adders are required for the OP III case. However, OP II and OP III only differ by the sign of the result and OP II can therefore be used instead of OP III, as discussed in Section 4.1.3. Hence, OP III will not be considered.

In this section, the sum and carry bits of equal significance will have the same index, i.e., Fig. 5.1 does not apply here. Furthermore, m is defined as the full adder index where the first full adder have index zero, i.e., $m = z$ correspond to the output bit s_{z+d}. With W_0 as the input wordlength, this means that the maximum full adder index, m, is W_0 and $W_0 - 1$ for the OP I and OP II case, respectively, i.e.

$$\begin{cases} 0 \leq m \leq W_0 & \text{OP I} \\ 0 \leq m \leq W_0 - 1 & \text{OP II} \end{cases} \tag{5.45}$$

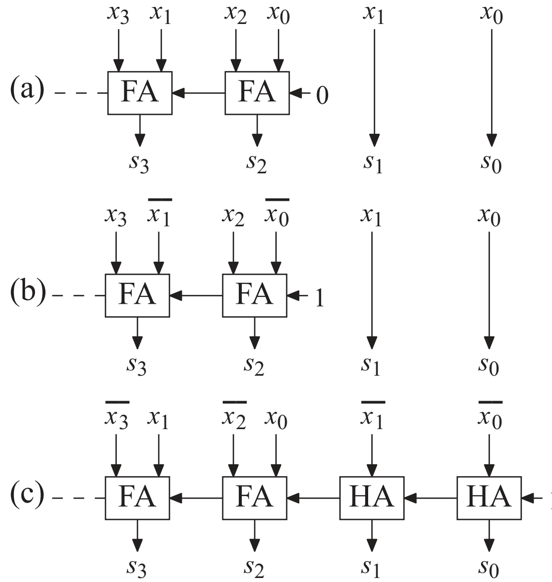


Figure 5.15 Adding the least significant bits for OP I–III, with $d = 2$

This is explained by (4.1) and the fact that the magnitude of f_i is less for a subtraction than an addition, using the same number of shifts d. Hence, for these specific operations, (4.1) can be rewritten according to

$$W_i = W_0 + \lceil \log_2(|f_i|) \rceil = \begin{cases} W_0 + d + 1 & \text{OP I} \\ W_0 + d & \text{OP II} \end{cases} \tag{5.46}$$

The equations presented in the following are all derived by studying many different combinations of d and W_0, using state transition graphs, similar to the techniques used in Sections 3.1 and 5.3. The exact values, obtained as rational numbers, have then been mapped to mathematical expressions. However, only the resulting formulas are stated here without repeating the theoretical background for the complete procedure.

Inverted Carry Correlation

For every pair of subsequent carry bits, the inverted carry correlation, ξ_m, is defined as

$$\xi_m = prob(c_m \neq c_{m-1}) \tag{5.47}$$

which means the probability that the logic values of the carry bits are different.

Three variables, a, b, and c, which can be computed from knowledge of the operation, the input wordlength, W_0, and the number of shifts, d, are here defined as

$$
\begin{aligned}
a &= (W_0 \bmod d) + d\delta(W_0 \bmod d) \quad \text{where} \quad \delta(q) = \begin{cases} 1 & q = 0 \\ 0 & q \neq 0 \end{cases} \\
b &= \begin{cases} -1 & \text{OP I} \\ 1 & \text{OP II} \end{cases} \\
c &= b^{\frac{W_0 - a}{d}}
\end{aligned}
\tag{5.48}
$$

Two partial result values, which will be multiplied by a power-of-two to obtain a certain inverted carry correlation, can then be defined as

$$\xi_A = \frac{1 - c2^{a - W_0}}{2^d - b} \quad \text{and} \quad \xi_B = \xi_A + \frac{c}{2^{W_0 - 1}} \tag{5.49}$$

Finally, the inverted carry correlation is computed as

$$
\xi_m = \begin{cases} \dfrac{1 - b^j 2^{-dj}}{(2^d - b)2^{2-d}} & 0 \leq m \leq W_0 - d - 1 \\ 2^k \xi_A & W_0 - d \leq m \leq W_0 - a - 1 \\ 2^k \xi_B & W_0 - a \leq m \leq W_0 - 2 \end{cases} \tag{5.50}
$$

$$\text{where} \quad \begin{cases} j = \dfrac{m - (m \bmod d)}{d} + 1 \\ k = W_0 - m - 3 \end{cases}$$

Carry Switching Activity

Using the inverted carry correlation, as defined in (5.50), the carry switching activity is computed as

$$\alpha_{c_m} = \begin{cases} 0 & m = 0 \\ \dfrac{1 + 2\alpha_{c_{m-1}} + 2\xi_{m-1}}{4} & 1 \le m \le W_0 - 1 \\ \dfrac{1}{2} & m = W_0 \quad \text{(OP I)} \end{cases} \tag{5.51}$$

Note that the switching activity of the least significant full adder, $m = 0$, is zero as the carry input is constant. Furthermore, the most significant full adder, $m = W_0$, which is only included for the OP I case, can actually be removed since the sign always will be equal to the sign of X, as was noted in Section 4.1.2.

Sum Switching Activity

The switching activity at the sum output is easily computed from the carry switching activity as

$$\alpha_{s_m} = \begin{cases} \alpha_{c_m} + 0.5 & 0 \le m \le W_0 - 2 \\ \alpha_{c_m} & m = W_0 - 1 \\ \alpha_{c_m} & m = W_0 \quad \text{(OP I)} \end{cases} \tag{5.52}$$

Note that the switching activity of the d least significant bits, which are not included in (5.52), is 0.5, since these sum output bits are directly connected to the X input, which is assumed to be a random signal.

5.6.2 Results

Here, the accuracy of the proposed switching activity model is investigated. The six different single adder multipliers defined in Table 5.17 and illustrated in Fig. 5.16 are considered. The input wordlength, W_0, is selected to be 8 bits, which means that 9 and 8 full adder cells are required for the OP I and OP II case, respectively. The implementations for two of the example multipliers are illustrated in Fig. 5.17. As was mentioned above, the most significant full adder may be eliminated for the OP I case as marked by the dashed contour in Fig. 5.17 (a), i.e., $s_{10} = x_7$ can be used.

Using (5.51) and (5.52), the carry and sum switching activities can be computed. The theoretical results are compared to high-level simulations of VHDL code using 1 000 000 random input values. As can be seen in

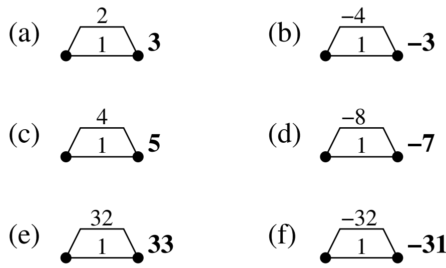


Figure 5.16 Single addition examples for six different coefficients.

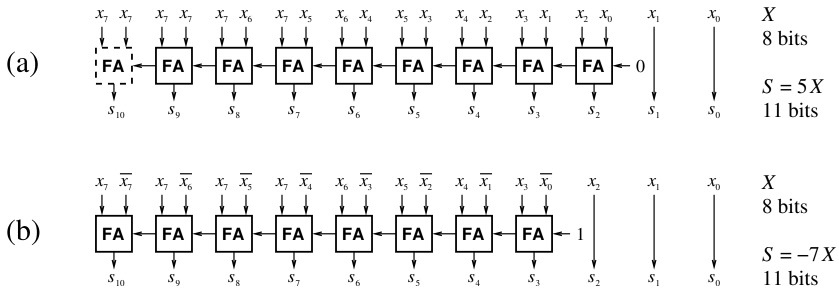


Figure 5.17 Implementations for the multipliers in Figs. 5.16 (c) and (d).

	(a)	(b)	(c)	(d)	(e)	(f)
OP	I	II	I	II	I	II
d	1	2	2	3	5	5

Table 5.17 Single addition multiplier examples.

	(a)	(b)	(c)	(d)	(e)	(f)
s_i	0.23% (8)	0.24% (9)	0.13% (7)	0.11% (7)	0.26% (6)	0.14% (11)
c_i	0.23% (8)	0.24% (9)	0.15% (7)	0.19% (10)	0.22% (6)	0.20% (12)

Table 5.18 Absolute maximum percentage deviation.

Fig. 5.18, the model agrees well with the simulations for all cases. The maximum percentage deviation is given in Table 5.18 for each case. The bit position, i, for which the largest error was obtained is given in brackets. Again, it is clear that the proposed model is accurate.

Similar to the model in Section 5.3, it is also here assumed that all switches will be full swing. Hence, the worst case is assumed, while in an

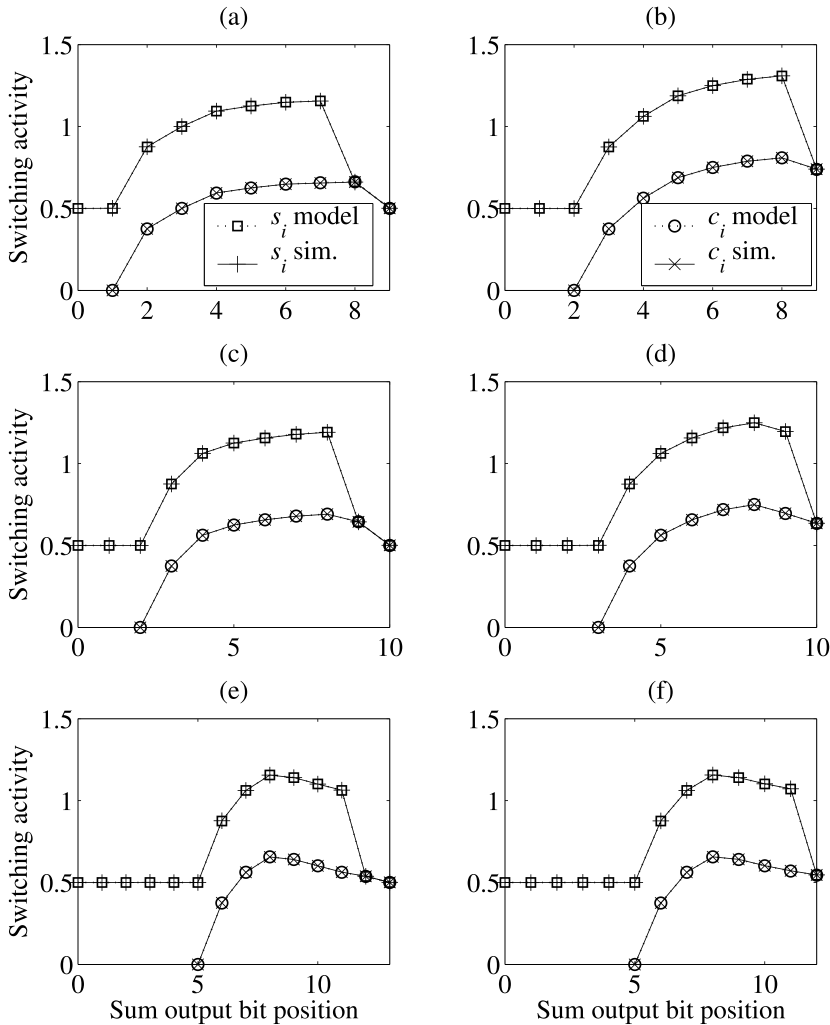


Figure 5.18 Switching activity from simulations and using the proposed model, respectively, for the realizations in Fig. 5.16.

actual implementation many glitches will be suppressed before they reach full swing, as illustrated by the examples in Section 5.4. Furthermore, if the carry is faster than the sum computation, i.e., $c_{del} < s_{del}$, which normally is the case for full adder cells used in parallel arithmetic, the switching activity of the sum output will be reduced. Note that the carry does not depend on this timing relation, given that all full adder cells are identical and that all bits of the input, X, will arrive simultaneously. Thus, the proposed model will still be valid for the carry signal, but some compensation for the sum signal might be required.

5.6.3 Discussion on Switching Activity in MCM

The model presented in Section 5.6.1 for highly correlated signals, is a good starting point for accurate modeling of complete multiplier blocks. However, there are still some issues that need to be investigated. To give an idea about how switching activity in multiplier blocks should be modeled, an example is considered in the following.

In Fig. 5.19, the coefficient 2625 is realized in two different ways. Note that these realizations only are used as illustrative examples and that the coefficient 2625 can be obtained more efficient, for example, from the two single adder values 5 and 65 as $5 \cdot 2^9 + 65$, using only three adders in total and with a critical path of only two adders.

In Table 5.19, information about the two example realizations is given. The required number of output bits, W_i, from each adder is computed using (4.1), assuming an input wordlength, W_0, of 16 bits. Note that the actual number of full adder cells is $W_i - d$ since d bits do not pass through any full adders, as illustrated in Fig. 5.15. Furthermore, using the input sign bit directly for positive fundamentals would save 4 and 3 full adders for the (a) and (b) realization, respectively. More details on the complexity of multiplier blocks in terms of full adders were addressed in Section 4.1.

Included in Table 5.19 is also the results obtained by the switching activity estimation model in [21], referred to as glitch path (GP) count. This model is based on the fact that the switching activity will increase with the adder depth. The total GP count is 10 and 26 for the (a) and (b) realization, respectively. Comparing this with the simulation results in Fig. 5.20, it is clear that this simple model is not suitable for these realizations. In fact, considering case (b) the GP count model suggests that the switching activity increases exponentially in the cascaded adders, while the simulation indicate the opposite.

Considering the average switching activity, Fig. 5.20 shows that case (b) is preferable. However, the number of full adder cells is larger for this case, 81 (78 with directly connected sign bits) compared to 69 (65) for multiplier (a). Hence, both complexity and switching activity should be taken into account.

For the graph in Fig. 5.19 (a), the critical paths for the two inputs of all adders (except the first adder) are different. If the difference in arrival time is large, the model in Section 5.3 can be used, by setting the switching activity for one of the inputs to zero. Even though the critical paths

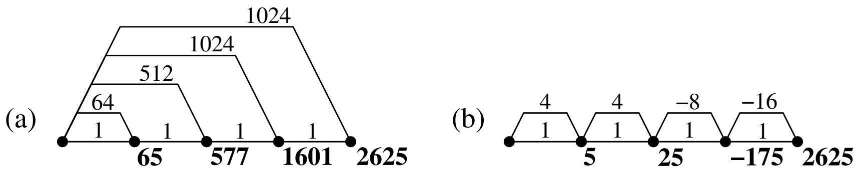


Figure 5.19 Two different realizations of the coefficient 2625.

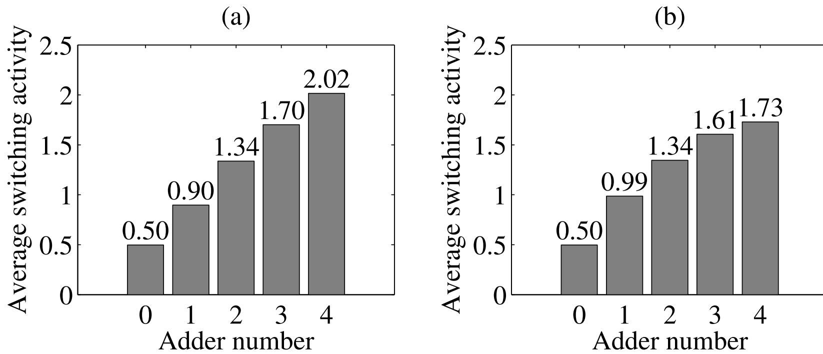


Figure 5.20 Average switching activity in each adder for the multipliers in Fig. 5.19, obtained from simulations. Adder number zero correspond to the input node.

Multiplier	(a)				(b)			
Adder	1	2	3	4	1	2	3	4
OP	I	I	I	I	I	I	II	II
d	6	9	10	10	2	2	3	4
W_i	23	26	27	28	19	21	24	28
FA cells	17	17	17	18	17	19	21	24
GP count	1	2	3	4	1	3	7	15

Table 5.19 Complexity and GP count measures for the multiple addition multipliers in Fig. 5.19.

appear to be equal for the realization in Fig. 5.19 (b), some signal bits will arrive faster since they do not pass through any full adders. This shows that some sort of timing must be included in an accurate model for MCM.

5.6.4 Time Instant Model

As concluded in the previous section, timing must be taken into account when modeling the switching activity for constant multiplication using parallel arithmetic. Consider the first two adders used in the implementation of the multiplier realization in Fig. 5.19 (a). This is illustrated in Fig. 5.21. Assuming an equal delay for the sum and carry computations according to $s_{del} = c_{del} = t_0$, the events that will occur for the seven full adders are listed in Table 5.20. For example, FA 4 will have an event at time zero since the two inputs, x_4 and x_{10}, might change. There will then be events on the carry input at time instants t_0, $2t_0$, $3t_0$, and $4t_0$ due to the possibility of a carry being propagated from the preceding full adders.

Note that the last event for each full adder gives the critical path from the input to that cell. Hence, this timing model can be used to estimate the throughput of MCM implementations.

Event-Based Full Adder Model

The transition activities for a specific full adder can be obtained in the same way as for the LSB in an RCA. Hence, the sequence of events obtained from the timing model are processed one at a time and the transition activity at a certain time instant is computed according to (5.16). The switching activities for the carry and sum signals are obtained by (5.29), which then are used as input activities for subsequent full adders in the following time step.

Since the carry propagate through the adder, each full adder can make many transitions for each applied input signal, X. Naturally, the magnitude of the switching activities will decrease gradually, since the probability of long carry propagation paths is low. For example, assuming $\alpha(x_i) = 0.5$, the carry signal c_3 in Fig. 5.21 will have the switching activities 3/8, 3/16, 3/32, and 3/64 at time instants t_0, $2t_0$, $3t_0$, and $4t_0$, respectively.

It can be shown that this model give accurate results as long as there is no correlation between input signals to the full adder. The main correlation problem is illustrated by the dashed lines in Fig. 5.21. Consider the y_1 and c_0 signals, which are inputs to FA 1 at depth two. Only the x_1 input has an event at time zero, and there will not exist any correlation during the second event. However, it can be seen that both signals are affected by x_3 and x_9 at time instant $2t_0$. A similar correlation will be present at the three following events. This will significantly reduce the switching activ-

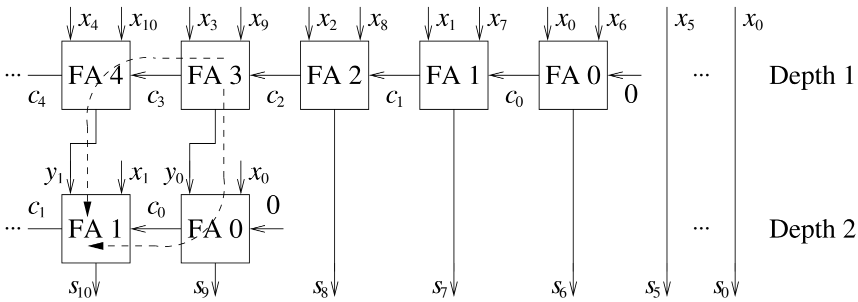


Figure 5.21 Architecture using two adder levels for computing the least significant bits of $S = 577X = (64 + 1)X + 512X$.

Time instant	Depth 1					Depth 2	
	FA 4	FA 3	FA 2	FA 1	FA 0	FA 1	FA 0
0	x_4, x_{10}	x_3, x_9	x_2, x_8	x_1, x_7	x_0, x_6	x_1	x_0
t_0	c_3	c_2	c_1	c_0		y_1, c_0	y_0
$2t_0$	c_3	c_2	c_1			y_1, c_0	y_0
$3t_0$	c_3	c_2				y_1, c_0	y_0
$4t_0$	c_3					y_1, c_0	y_0
$5t_0$						y_1, c_0	

Table 5.20 Events at the different inputs for the full adders in Fig. 5.21.

ity at the sum output, s_{10}, for random input data, X. Note that the correlated y_1 and c_0 signals will arrive simultaneously to FA 1 even if $s_{del} \neq c_{del}$, since both paths have the delay $s_{del} + c_{del}$.

Hence, a correlation factor, similar to the one derived in Section 3.1.5 for bit-serial constant multipliers, should be defined and incorporated to achieve a complete model. Furthermore, correlation will also occur in the most significant bits due to sign extension.

5.7 Conclusions

A data dependent switching activity model was presented for the ripple-carry adder. Previous work have only considered random input data, but for most applications the input data are correlated. Hence, the method could be included in high-level energy estimation to give results that are

more accurate. It was shown by examples that the proposed model agrees very well with simulations in estimating the switching activity of the output signals. However, the energy consumption was overestimated since not all switches in the implemented adder were rail-to-rail (full swing).

A simplified model based on word-level statistics, was also presented for estimation of the switching activity in ripple-carry adders. The model is accurate in estimating the switching activity when real world signals are applied, as was illustrated in an example.

Furthermore, a switching activity model for single adder multipliers was presented. The model was shown to agree well with high-level simulation of VHDL code descriptions, with an error of at most 0.26% for the studied test cases. Since the single adder multiplier is a common part in multiplier blocks, the model is suitable to be used in energy consumption aware algorithms for constant multiplication. Finally, a general model for estimation of the switching activity in MCM implementations was suggested. It was concluded that an event-based model considering each full adder individually gives accurate results under the condition that the full adder inputs are uncorrelated.

6

FUNCTION APPROXIMATION BY A SUM OF BIT-PRODUCTS

Here, a method to approximate elementary functions as a sum of weighted bit-products is presented [65]. For smooth functions, the weights vary over several orders of magnitude and many of the bit-products can therefore be neglected. When not all bit-products are included, the accuracy can be significantly increased by optimization [41].

The function approximation computation can be mapped to a corresponding low complexity hardware architecture. Different ways to divide the architecture into sub-blocks, which may be turned off to reduce the energy consumption, are investigated [64].

The method can be used to implement functions that are required for conversion and addition in logarithmic number systems (LNS). A sign transformation, suitable for functions with logarithmic characteristics, is presented [68],[71].

In [151], an alternative derivation of the presented function approximation approach is used for sine and cosine. Here, several angle rotations are expressed as a sum of weighted bit-products.

It is shown that for larger wordlengths, the area can be significantly decreased compared to ROM implementations. Furthermore, comparing the presented method with a straightforward unfolded CORDIC implementation for sine and cosine computation, the required area is halved.

6.1 Background

Computation of various functions, for example, trigonometric, logarithm, exponential, and square root, are commonly used in digital signal and video processing systems. Another usage is the functions used to implement addition and subtraction in the logarithm number system. In real-time applications, software routines are often too slow, and, hence, dedicated hardware is used for computation of elementary functions [116].

Several general methods for evaluation of elementary functions have been proposed, for example, additive/multiplicative normalization [75], [146] and rational approximations [74]. Polynomial approximations, which naturally are closely related to rational approximations, have also been extensively investigated [6],[86],[126]. Using tables and multipliers, several elementary functions can be computed with the same scheme [34]. In [137], the complete generation of polynomial approximations, including optimization of coefficients and data-paths, was described.

The input range can be partitioned into regions, and different polynomial approximations can then be used in each segment. This technique results in a trade-off between memory size and arithmetic complexity [106]. Using a specialized squaring unit, second-degree interpolation of the different segments was used in [117]. When only first-degree polynomials are used, this method is referred to as piecewise linear approximation, which require two look-up tables, one for the offset and one for the slope, one multiplication, and one addition. In [136] a modification was proposed for obtaining a power of an operand, using only one table and a multiplier, i.e., no adder was required. A generalization of the approach in [80] resulted in a piecewise linear approximation for arbitrary functions, using a multiplierless architecture where the general multiplier is replaced by configurable shifts and a multi-operand adder [44].

If the slope is constant for subsequent segments within a certain region, a special form of piecewise linear approximation, referred to as the bipartite method, is obtained. This method was first used in [133] and then re-discovered in [123]. The corresponding architecture consists of two parallel look-up tables and an adder. In [132], it was shown that the total memory requirement could be reduced by using more look-up tables and a multi-operand adder. By combining the ideas in [99] and [132], a general multipartite table method that require even smaller tables was obtained in [33].

Also range-reduction, i.e., scaling the input argument to be within the convergence region of an algorithm, has been studied. There are two types of range reduction; additive and multiplicative. Normally, multiplicative reduction is straightforward, while additive reduction is often problematic. In [19] the modular range reduction (MRR) algorithm, which can be used for exponential and trigonometric functions, was analyzed and implemented using a modified multiplier. An on-the-fly range reduction algorithm, with the same accuracy as MRR, for the case when the input argument is given bit-serially has also been proposed [83]. To use different algorithms depending on the size of the input argument, and by that obtain high-speed for arguments in the most common range, was suggested in [5].

Most work on function approximation have considered single and double-precision floating-point operands, for computations in either hardware or software. Here, the focus is on high-speed, low area, fixed-point realizations for systems implemented in either FPGA or ASIC technology. Applications may include direct digital frequency synthesizers (DDFS) [81], conversion to/from and addition/subtraction in logarithmic number systems [32], and various multimedia algorithms. Furthermore, iterative algorithms, such as Newton-Raphson, are commonly used in floating-point systems. Less complicated fixed-point function approximation methods can then be used to find a starting point with high accuracy, which result in fewer iterations.

6.1.1 PPA Methods

The general idea here is to express the approximation in the form of a partial product array (PPA). The PPA is composed of boolean elements, often referred to as bit-products, which are selected so that added together, with appropriate significances, the desired function is obtained.

This method was introduced in [131], where a divider was implemented. The operation is first expressed as a multiplication, which is also simple for some other functions beside division, for example, the square root [128]. However, for many other functions this is more complicated and may then be obtained by instead expressing a related function as a multiplication, for example, for sine the derivative of the corresponding inverse trigonometric function was used in [127]. The multiplication is expanded into a PPA where each column gives one equation, and the operand digits are then back-solved from this set of equations. The preci-

sion is increased by back-solving as many digits as possible. However, the complexity increases exponentially with the number of digits.

In [131], each quotient digit was dependent on all more significant digits. This dependency was removed by substitutions in [94].

In [128], it was shown that errors are introduced due to carry-outs from less significant columns into the back-solved columns. Hence, the accuracy can be increased by adding error compensating elements to the PPA. Since the computation includes summation of partial products, it is possible to reuse the hardware of an existing multiplier by modifying the PPA into a suitable shape, i.e., so that no columns include too many bit-products [128]. It was also noted in [128], that instead of deriving the PPA by back-solving, a piecewise linear approximation can be transformed into a PPA representation.

In [10], the computation of sine and cosine in a DDFS circuit was implemented by polynomial expansion with fixed coefficients. The method was referred to as CSD hyperfolding, since the coefficients are represented in CSD code and the derivation of the bit-products is related to folding techniques used in squarers. It was shown that a third-degree polynomial, $P(X) = AX^3 + BX^2 + CX + D$ where X is composed of N bits, can be evaluated as a sum of $(N^3 + 5N)/6$ weighted bit-products. The polynomial coefficient corresponding to the term with highest degree, strongly affects the number of partial products, which in turn is directly related to the hardware complexity. While keeping the complexity low, by limiting the number of nonzero digits in the crucial coefficient, the polynomial coefficients were optimized in order to maximize the spurious free dynamic range (SFDR).

Also in [49], the starting point to obtain a PPA is a polynomial form, or more accurately, functions that can be expressed as converging series, according to

$$f(X) = \sum_{i=0}^{\infty} c_i X^i \tag{6.1}$$

The PPA is then obtained by expressing all powers of X in bit notation. The main idea in [49] is an algorithm to transform the PPA into a sum of less complicated functions, for which look-up tables can be used. Hence, similar to [33] and [132], the function is computed using several parallel look-up tables and additions. Furthermore, bounds for the error introduced by splitting X into several smaller inputs was also derived in [49].

The function approximation approach considered in the rest of this chapter is implemented in the same way as the methods in [10] and [128], i.e., as a summation of partial products. However, the presented approach is more general than these methods. It is not derived from a polynomial expression as in [10], which prevent both the possibility to include larger bit-products and to arbitrary vary the weights. Furthermore, it can be used for any function, even a function composed by random values, which is not possible for the method in [128] due to the complicated back-solving technique.

6.2 Function Approximation Approach

In this section, an approach for approximating elementary functions is presented. By rewriting the function as a sum of weighted bit-products, an efficient implementation is obtained. Since the function is arbitrary, the method can in practice be seen as an alternative method to implement a look-up table. For most functions, a majority of the bit-products can be neglected and still obtain good accuracy. The method is suitable for high-speed implementation of fixed-point functions.

How efficient the obtained implementation is depends on the characteristics of the function. The possibility to reduce the complexity, using optimization methods, is investigated for some common functions.

6.2.1 General Formulation

In the following, the method proposed in [65] for function approximation using a weighted sum of bit-products is presented.

Assume a function, $f(X)$, where X is an integer as defined in (1.6), i.e., composed of N bits, x_i, so that

$$X = \sum_{i=0}^{N-1} x_i 2^i \quad \text{where} \quad x_i \in \{0, 1\} \tag{6.2}$$

Then there are 2^N function values, $f(X)$, where $0 \le X \le 2^N - 1$. By rewriting the corresponding look-up table expression where X is used as address, the function values can be computed as a weighted sum of bit-products according to

$$\begin{aligned} f(X) &= f(0)\ldots\bar{x}_2\bar{x}_1\bar{x}_0 + f(1)\ldots\boldsymbol{\bar{x}_2\bar{x}_1x_0} + \ldots + f(2^N-1)\ldots x_2x_1x_0 = \\ &= f(0)(\ldots) + f(1)(\ldots(\mathbf{1}-\boldsymbol{x_2})(\mathbf{1}-\boldsymbol{x_1})\boldsymbol{x_0}) + \ldots + f(2^N-1)(\ldots) = \\ &= f(0)(\ldots) + f(1)((\ldots)(\boldsymbol{x_0} - \boldsymbol{x_1x_0} - \boldsymbol{x_2x_0} + \boldsymbol{x_2x_1x_0})) + \ldots = \\ &= f(0) + (-f(0)+f(1))x_0 + (-f(0)+f(2))x_1 + \ldots = \sum_{j=0}^{2^N-1} c_jp_j \end{aligned} \quad (6.3)$$

where c_j corresponds to the weights and p_j to the bit-products. Hence, the function is evaluated as a sum of 2^N terms, each consisting of a fixed constant times a product of binary variables. Note that the weights are differences of function values, and, hence, the magnitude of the weights, and thereby the efficiency of this method, depends on the characteristics of the function. The bit-products can be formed by considering all possible combinations of the bits in the input argument. For example, for a 3-bit input $X = 4x_2 + 2x_1 + x_0$, there are $2^3 = 8$ bit-products: 1, x_0, x_1, x_2, x_1x_0, x_2x_0, x_2x_1, and $x_2x_1x_0$. The bit-product 1 corresponds to the case where no binary variables are selected, i.e., the corresponding weight is a constant that is always included in the sum.

Each bit-product, p_j, is composed of the bits x_i that are one in the binary representation of j, hence

$$p_j = \prod_{i=0}^{N-1} (\bar{a}_i + a_ix_i) \quad \text{where} \quad \sum_{i=0}^{N-1} a_i2^i = j \quad \text{and} \quad a_i \in \{0, 1\} \quad (6.4)$$

Note that the bit-products can be computed using simple logic AND-operations, for example, $p_{25} = x_4x_3x_0$ corresponds to a 3-input AND gate.

The weights, c_j, which in the following also are referred to as coefficients, are computed by vector multiplications according to

$$c_j = Q_N^j \cdot F = \begin{bmatrix} q_{j,0} & q_{j,1} & \cdots & q_{j,2^N-1} \end{bmatrix} \cdot \begin{bmatrix} f(0) \\ f(1) \\ \ldots \\ f(2^N-1) \end{bmatrix} \quad (6.5)$$

where F is a column vector containing all function values. The row vector Q_N^j is the jth row of the $2^N \times 2^N$ matrix Q_N, which is obtained by

$$Q_i = \begin{cases} 1 & i = 0 \\ \begin{bmatrix} 1 & 0 \\ -1 & 1 \end{bmatrix} \otimes Q_{i-1} & 1 \le i \le N \end{cases} \tag{6.6}$$

where the $\otimes$ operation denotes the Kronecker product. Note that (6.5) may be modified into the matrix multiplication $Q_N F$, which gives all coefficient values in the resulting vector.

Using (6.5), the coefficients are directly obtained from the function values, for example, $c_0 = f(0)$, $c_1 = -f(0) + f(1)$, and so on. Thus, it is not necessary to rewrite the function expression as in (6.3).

From (6.6) it is clear that the elements $q_{j,\,k}$ in (6.5) are restricted so that

$$q_{j,\,k} \in \{-1, 0, 1\} \tag{6.7}$$

Hence, each coefficient c_j is a sum of function values. Moreover, it can be shown that each coefficient can be expressed as a sum of differences of function values with a fixed distance. This is defined as

$$c_j = \sum_{n=1}^{j} s_n(f(n) - f(n - 2^i)) \qquad \begin{cases} 1 \le j \le 2^N - 1 \\ s_n \in \{-1, 0, 1\} \end{cases} \tag{6.8}$$

where i is equal to the index of the least significant bit x_i included in the bit-product p_j corresponding to the coefficient c_j. The values for s_n can be derived from (6.6). Note that (6.8) implies that many coefficients will be small if the difference between adjacent function values is small. It will be shown in Section 6.2.3 that many coefficients can be ignored, i.e., set to zero, for smooth functions.

As discussed above, for functions with certain properties many of the coefficients, c_j, will be small, i.e., truncated to zero for a limited coefficient wordlength. Hence, we can approximate a function $f(X)$ with $\hat{f}(X)$ as

$$\hat{f}(X) = \sum_{j \in G} c_j p_j \tag{6.9}$$

where G is the set of indices for the M coefficients with largest absolute value, i.e., the number of included bit-products, $|G|$, is equal to M and no

discarded bit-product has a weight that is larger than any included one according to

$$\min_{j \in G}\{|c_j|\} \geq \max_{j \notin G}\{|c_j|\} \tag{6.10}$$

Note that the coefficient values, c_j, as computed from (6.5) only are optimal as long as all 2^N bit-products are included. How to find good coefficient values when only M bit-products are included will be discussed in Section 6.2.2.

Example

The method will here be illustrated by an example. Consider the sequence of Fibonacci numbers as

$$f(X) = \begin{cases} 3 & X = 0 \\ 5 & X = 1 \\ f(X-1) + f(X-2) & 2 \leq X \leq 7 \end{cases} \tag{6.11}$$

where X is composed of $N = 3$ bits.

The function values, together with bit-products and coefficients derived according to (6.4) and (6.5), respectively, are presented in Table 6.1. Note that the magnitude of the coefficients is significantly smaller than the magnitude of the function values. The reason for this is that the coefficients are sums of differences of function values, as discussed in the previous section. For example, c_5 can be computed according to (6.8) as

$$c_5 = -(f(1) - f(0)) + (f(5) - f(4)) = -2 + 13 = 11 \tag{6.12}$$

The function can now be obtained as a weighted sum of bit-products according to (6.3) as

$$\begin{aligned} f(X) = {} & 3 + 2x_0 + 5x_1 + 3x_1x_0 + 18x_2 + \\ & 11x_2x_0 + 29x_2x_1 + 18x_2x_1x_0 \end{aligned} \tag{6.13}$$

This expression is visualized in Fig. 6.1. How to efficiently implement a circuit to compute $f(X)$ will be discussed in Section 6.3.1.

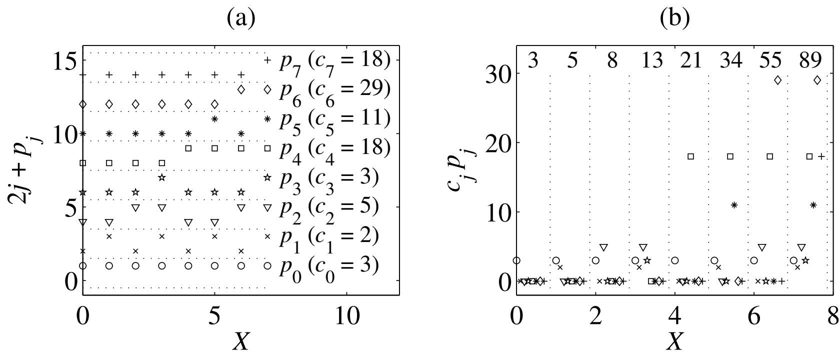


Figure 6.1 (a) Illustration describing for which input arguments the different bit-products are active. (b) The contribution from all weighted bit-products, which summed together result in the function values given at the top (for clarity the markers are horizontally shifted).

j	a_i	p_j	Q_N^j								$f(X)$	c_j
0	000	1	1	0	0	0	0	0	0	0	3	3
1	001	x_0	–1	1	0	0	0	0	0	0	5	2
2	010	x_1	–1	0	1	0	0	0	0	0	8	5
3	011	x_1x_0	1	–1	–1	1	0	0	0	0	13	3
4	100	x_2	–1	0	0	0	1	0	0	0	21	18
5	101	x_2x_0	1	–1	0	0	–1	1	0	0	34	11
6	110	x_2x_1	1	0	–1	0	–1	0	1	0	55	29
7	111	$x_2x_1x_0$	–1	1	1	–1	1	–1	–1	1	89	18

Table 6.1 Computation of the coefficients for the example function.

6.2.2 Optimization

In the previous example, we have not gained any significant reduction in implementation cost since the number of terms in (6.13) is the same as would be required if we stored the function values directly in a look-up table. However, in Section 6.2.3 it will be shown that for many functions, the value of the weights, c_j, varies over several orders of magnitude, and, hence, not all of them need to be used.

Although the values computed for c_j from (6.5) are optimal when all 2^N bit-products are considered, it is possible to find better values for the

case in (6.9), i.e., when only M bit-products are included. Hence, optimization techniques should be used to find the solution that minimizes the error. It was shown in [41] that most optimization formulations, including minimizing the maximum absolute error, can be solved using standard linear or quadratic programming methods.

Using M out of the total 2^N bit-products, as discussed in Section 6.2.1, the maximum error, ε, can be bounded according to

$$|f(X) - \hat{f}(X)| \le \varepsilon \qquad X = 0, \ldots, 2^N - 1 \tag{6.14}$$

Using (6.9), this can easily be transformed into a linear optimization problem on the standard form as

$$\begin{aligned} &\text{minimize } \varepsilon \\ &\text{subject to } \sum_{j \in G} c_j p_j - \varepsilon \le f(X) \qquad X = 0, \ldots, 2^N - 1 \\ &\qquad\quad -\sum_{j \in G} c_j p_j - \varepsilon \le -f(X) \qquad X = 0, \ldots, 2^N - 1 \end{aligned} \tag{6.15}$$

Using continuous variables for ε and c_j, the problem can be solved with standard linear programming techniques, such as the simplex algorithm.

Based on the maximum absolute error, ε, an accuracy measure is defined as

$$\text{Accuracy} = -\log_2 \varepsilon \tag{6.16}$$

which means that an accuracy of B bits correspond to an error, ε, of at most 2^{-B}, i.e., the weight of the Bth fractional bit. Hence, it is guaranteed to be at least $B - 1$ correct fractional bits in the output.

There are two main issues to be considered in terms of quantization. First, the implementation complexity is primarily reduced by decreasing the number of bit-products used in the computation. The main problem is to determine the minimum number of bit-products required for a certain accuracy. Furthermore, which bit-products to use is also a problem. The straightforward approach, as mentioned in Section 6.2.1, using the M bit-products with the largest absolute weights before optimization is a good heuristic. It should be noted, that it in general is possible to obtain slightly higher accuracy by replacing some of the bit-products. However, the problem of selecting the M best bit-products instead of the M with largest

original weights is hard, and the gains are often small. It was shown in [41] that choosing the M best out of the $M + 10$ bit-products with the largest absolute weights increased the accuracy with about one half bit for the studied cases.

The second issue, is the use of finite wordlength for representing the coefficients. This reduces the possible accuracy. Also, as the range of the magnitude of the weights is large, it may lead to that certain bit-products are removed as their corresponding weights are quantized to zero. In the implementation, we would like to represent the coefficients by fixed-point numbers. One straightforward way is to simply use rounding of the coefficients obtained by the optimization procedure in (6.15). However, we can obtain better results if the coefficients are optimized as fixed-point coefficients. By replacing c_j in (6.15) by $d_j 2^{-W}$, where d_j and W are integer numbers, we obtain a mixed-integer linear programming (MILP) problem using coefficients with W fractional bits. This problem can be solved using, for example, branch-and-bound techniques. As will be shown in Section 6.4.2, there is a trade-off between the number of bit-products and the coefficient wordlength. Often it is advantageous to reduce the coefficient wordlength by increasing the number of bit-products.

Furthermore, the implementation complexity is roughly directly proportional to the number of partial products to be added, i.e., the number of nonzero digits in the CSD representation of the coefficients, c_j. In [41] an optimization problem to find the solution that minimizes the number of nonzero digits, while keeping the maximum error below a certain limit, was formulated. However, due to the increasing complexity of solving the resulting MILP problem, this is only feasible for few bit-products and/or short wordlengths.

6.2.3 Results for Some Elementary Functions

In this section, the proposed method is applied to a number of various functions used in digital signal processing systems. First, the selected functions are characterized by studying the magnitude of the weights. Then the possibilities to reduce the implementation complexity using optimization techniques are investigated.

Characterization of Functions

The considered functions are defined in Table 6.2 and plotted in Fig. 6.2. The range of the input value is $0 \leq X < 1$. Also given in Table 6.2 is the

	$f(X)$	Range of $f(X)$
(a)	$\sin(\pi X/2)$	$0 \leq f(X) < 1$
(b)	$\tan(\pi X/4)$	$0 \leq f(X) < 1$
(c)	$\text{atan}(X)$	$0 \leq f(X) < \pi/4$
(d)	$\log_2(X+1)$	$0 \leq f(X) < 1$
(e)	2^X	$1 \leq f(X) < 2$
(f)	$1/(X+1)$	$1/2 < f(X) \leq 1$
(g)	$\sqrt{X}$	$0 \leq f(X) < 1$
(h)	$\sqrt{X+1}$	$1 \leq f(X) < 2$

Table 6.2 Definition of some elementary functions with $0 \leq X < 1$.

range of the function values, $f(X)$. Figure 6.3 shows the distribution of the magnitude of the coefficients, c_j, for the studied functions when the input value, X, is represented using nine bits, i.e., $N = 9$. From Fig. 6.3 it is clear that the coefficient magnitude variation is large for most functions. Hence, for a given accuracy many of the coefficients can be neglected.

The magnitude decrease fast for the 2^X and $\sqrt{X+1}$ functions, which only have 102 and 98 coefficients of a magnitude greater than 10^{-5}, respectively. This indicates that these functions will require fewer bit-products for a given accuracy. Especially for $\sqrt{X+1}$ it is clear from Fig. 6.2 (h) that this is a simple function, since it is close to a straight line.

The function that is least suitable for the proposed approach is $\sqrt{X}$, where all coefficients, except one, are greater than 10^{-2}. The difference between adjacent function values is large in the beginning of the interval for $\sqrt{X}$, as can be seen in Fig. 6.2 (g). Hence, the reason for the large coefficients is that the difference between $f(0)$ and another function value is included in the computation of all coefficients c_j, except c_0. A method that enables efficient implementation of functions with such characteristics will be presented in Section 6.4.1.

Complexity Reduction by Optimization

The number of correct output bits based on maximum absolute error, i.e., the accuracy as defined in (6.16), obtained when neglecting the smaller bit-products is shown in Fig. 6.4. The dashed lines correspond to the original coefficients, c_j, computed according to (6.5), while the solid lines

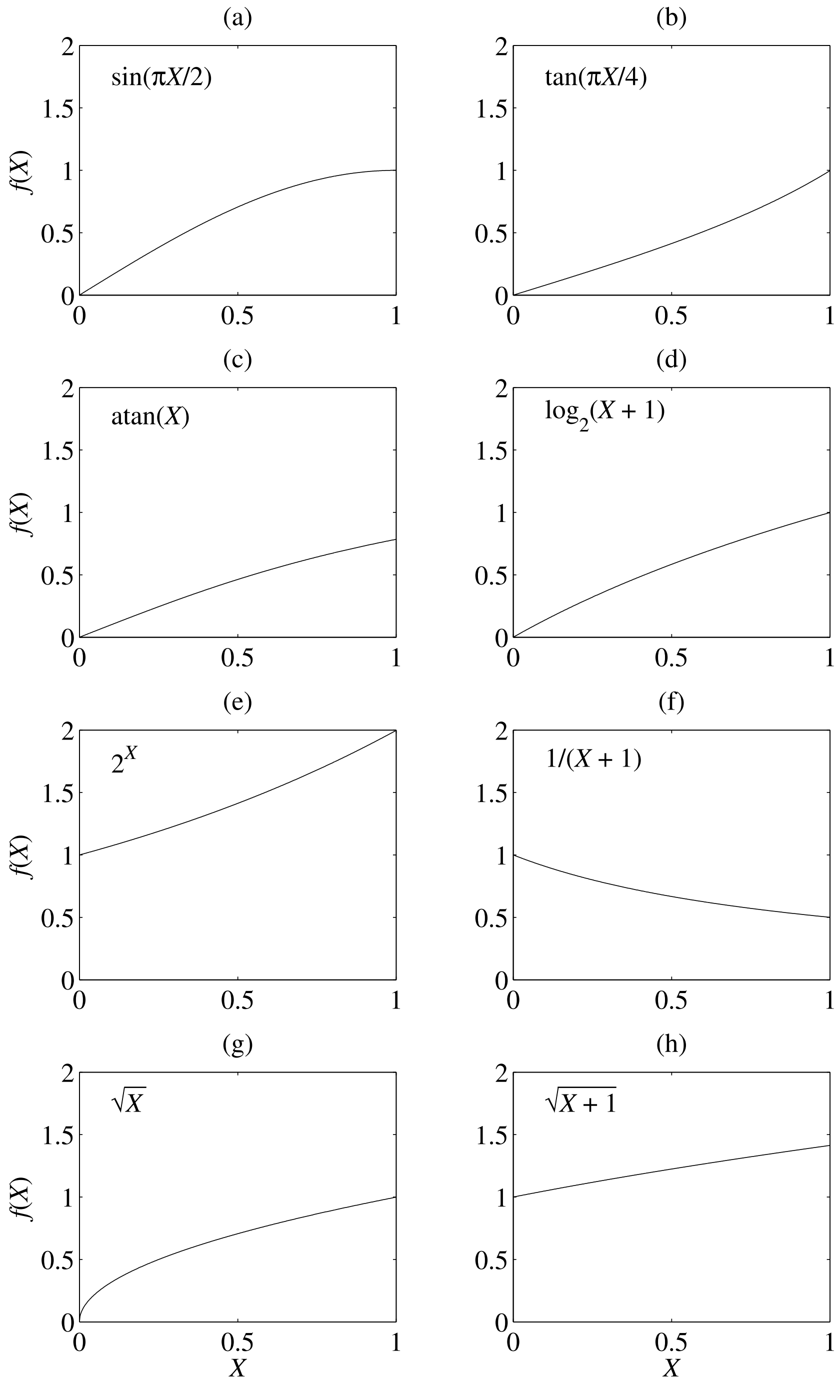


Figure 6.2 The studied functions as defined in Table 6.2.

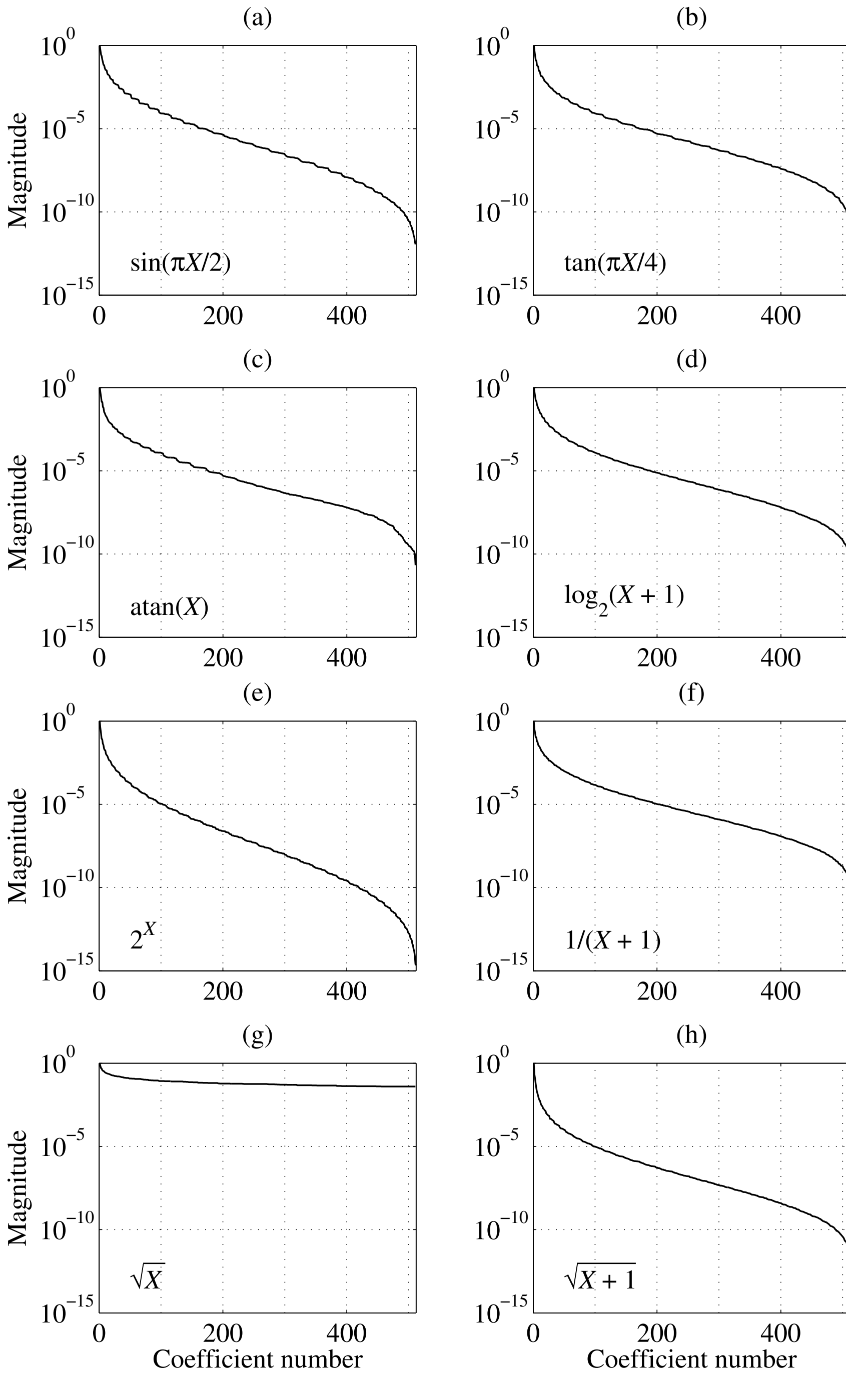


Figure 6.3 Normalized distribution of the coefficients, c_j, for the functions defined in Table 6.2.

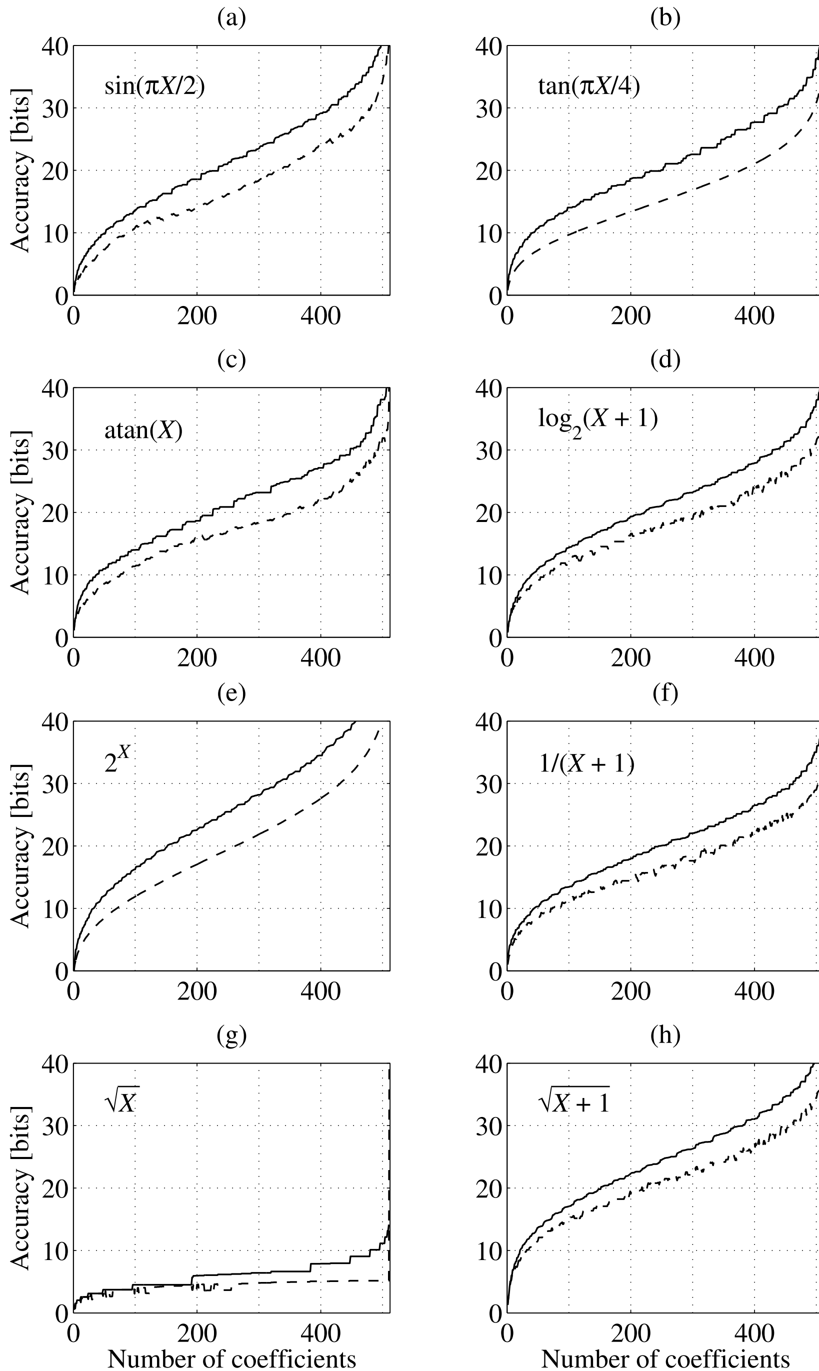


Figure 6.4 Accuracy vs. the number of bit-products for the studied functions. Coefficients taken from (6.5) (dashed) and optimized (solid).

correspond to optimized continuous weights minimizing the maximum absolute error according to (6.15). It is seen that an accuracy corresponding to more than 20 correct output bits is obtained for all functions, except $\sqrt{X}$, when neglecting half of the bit-products.

Furthermore, it is clear from Fig. 6.4 that optimization increases the accuracy significantly when not all bit-products are used, as discussed in Section 6.2.2. For example, considering the 2^X function, an accuracy of almost 12 bits is obtained including 100 bit-products out of the total 512, when using the coefficients computed by (6.5). Applying the linear optimization approach, increases the accuracy to more than 16 bits.

6.3 Architecture

In this section, an effective architecture to implement the expression in (6.3), suitable for high-speed implementations, is presented. Furthermore, different ways to split the architecture into sub-blocks that may be turned off to decrease the energy consumption are investigated.

6.3.1 Implementation for a Sum of Bit-Products

The architecture is illustrated in Fig. 6.5. The input bits, x_i, are connected to the AND-stage, which is used to compute the required bit-products, p_j, according to (6.4).

In the next step, the partial product generation stage shifts and possibly inverts the outputs of the AND-stage, according to the corresponding coefficients, c_j. Hence, the bit-products are included in the columns corresponding to nonzero digits in the coefficient representation. By using a minimum signed-digit (MSD) representation, the number of partial products to be added is minimized.

Here, the coefficients are represented using the canonic signed-digit (CSD) representation [149]. For bit-positions corresponding to a positive digit in the CSD representation, the bit-product is added in that column. For negative digits in the CSD representation, a one-bit two's-complement representation is used. A compensation vector is used to avoid sign-extension. Consider subtraction of a bit x. Sign-extending it two positions gives xxx. This can be rewritten as $00\bar{x} + 111$, where the 111 part is the compensation vector. What this means is that $-x$ is computed as $\bar{x} - 1$. Hence, only the inverted bit-product is added. All compensation vectors, and also the weight corresponding to the constant bit-product, can be

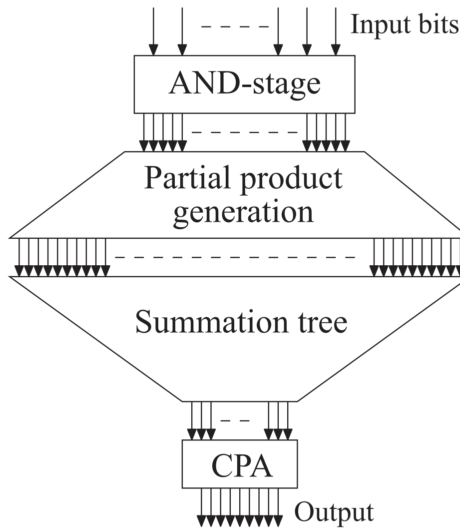


Figure 6.5 Architecture for computing a function as a weighted sum of bit-products.

accumulated into a single value that is added in the summation of partial products. According to the above, the contribution to the compensation value, for a negative digit of significance 2^k is the corresponding negative value, i.e., -2^k.

In the third stage, a summation tree, for example implemented as a Wallace tree [145], sums up the partial products. Note that it is sometimes advantageous to use other SD representations than CSD, to obtain a good distribution of the partial products among the different columns. Finally, a carry propagation adder (CPA), sometimes denoted vector merging adder (VMA), is used to form a non-redundant representation of the adder tree output.

The latency of this architecture is low, which is important in many applications. Furthermore, it can easily be pipelined to an arbitrary degree to obtain the desired throughput.

Example

Here the example from Section 6.2.1 is continued. To reduce the number of partial products, the coefficients are represented using the CSD code, as given in Table 6.3.

As an example, consider the weight for x_2x_1, which is 29 and has the CSD representation $100\bar{1}01$. To avoid sign extension for the negative digit, the bit-product is negated and $-4 = 1111100$ is added to the com-

j	p_j	c_j	CSD	Compensation
0	1	3	–	3 = 0000011
1	x_0	2	000010	
2	x_1	5	000101	
3	x_1x_0	3	$00010\bar{1}$	–1 = 1111111
4	x_2	18	010010	
5	x_2x_0	11	$010\bar{1}0\bar{1}$	–1 = 1111111 –4 = 1111100
6	x_2x_1	29	$100\bar{1}01$	–4 = 1111100
7	$x_2x_1x_0$	18	010010	+
			Total compensation vector:	–7 = 1111001

Table 6.3 Computation of the compensation vector.

Column weight	2^6	2^5	2^4	2^3	2^2	2^1	2^0
Partial products							x_1
			x_2		x_1		$\overline{x_1x_0}$
			x_2x_0		x_1x_0	x_0	$\overline{x_2x_0}$
		x_2x_1	$x_2x_1x_0$		$\overline{x_2x_0}$	x_2	x_2x_1
	1	1	1	1	$\overline{x_2x_1}$	$x_2x_1x_0$	1

Table 6.4 Partial products for the example function.

pensation value. The total compensation vector, which also includes the constant 3, is found to be –7 = 1111001.

The resulting set of bit-products to be added is shown in Table 6.4. In this case, there are in total 20 partial products, which need to be reduced to 7 output bits. Hence, 12 full adders are required since each full adder eliminates one partial product and there is one carry output bit. How many half adders that are required depends on the used reduction method. For example, an adder tree implemented according to the original reduction scheme introduced by Wallace in [145] require significantly more half adders than the modified Dadda scheme presented in [4].

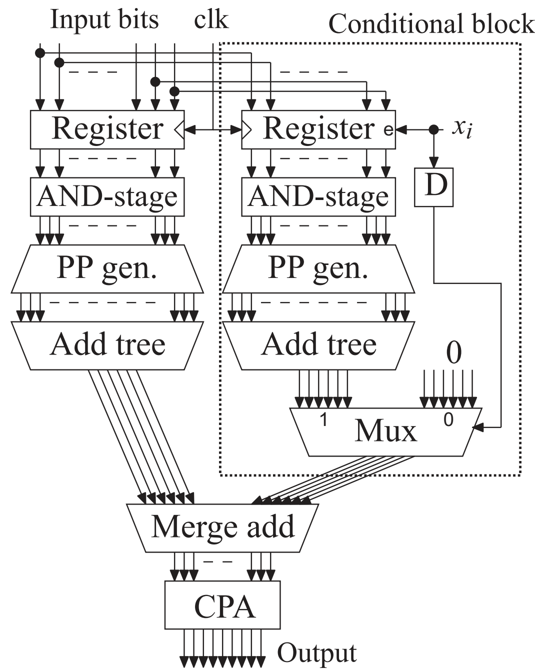


Figure 6.6 Architecture with one conditional block, controlled by x_i.

6.3.2 Conditional Blocks

The architecture shown in Fig. 6.5 can be divided into multiple parts. For example, all bit-products that include the input bit x_i can be separated into a block with input registers enabled by x_i, as shown in Fig. 6.6. Hence, this block, denoted conditional block, will be inactive when x_i is zero, which for random input data will be the case half of the time. It is even more effective to separate bit-products that have more than one common input bit. A block that is controlled by the AND-product $x_i x_j$ will be inactive 75% of the time for random data.

The architecture can be divided into an arbitrary number of blocks. If no conditional blocks are used, the whole circuit will be active all the time. On the other hand, if many conditional blocks are used, the merging operation, as shown in Fig. 6.6, will be complex. Hence, one of the objectives here is to find where the threshold in terms of energy dissipation, with respect to the number of conditional blocks, occurs.

It is advantageous if the architecture can be divided so that the different blocks have an equal number of levels in the summation tree, since it

is then more likely that the critical path is not significantly longer compared to the original architecture. Furthermore, balanced trees will reduce the switching activity in the merging operation, as the timing between the different blocks then will be synchronized.

To reduce the number of bits that are passed through the multiplexer, a CPA can be introduced between the summation tree and the multiplexer in the conditional block of Fig. 6.6. This will decrease the number of outputs of the multiplexer, leading to a less complex merging adder. This also reduces the switching activity, as an old value is passed through the multiplexer before the new output of the sub-block is available when the conditional block becomes enabled.

6.3.3 Results Using Conditional Blocks

Here, results for different implementations of the cosine function, using a 9 bit input corresponding to angles between 0 and $\pi/4$ rad, will be given. The number of bit-products in this example function, i.e., the number of terms, M, in (6.9), is 91. The optimized coefficients have a wordlength of 20 bits and there are in total 296 nonzero digits in the CSD representation, i.e., on average 3.25 nonzero digits per coefficient. The obtained accuracy of the output is 16 bits. The results presented are based on synthesis of VHDL code using a 0.35 μm CMOS standard cell library. The power consumption was obtained for 100 random input values at a clock frequency of 20 MHz using NanoSim™.

Designs

The specifications for the different designs that will be implemented are given in Table 6.5. The first design, D1, is the straightforward architecture as shown in Fig. 6.5, i.e., this design contains no conditional blocks. The summation tree has eight levels.

In the second design, D2, the architecture is divided in two blocks of which one block is controlled by the most significant input bit, x_8. The reason that x_8 is selected as control signal is because it is included in more bit-products than any other input bit. Since there are 44 bit-products that include the x_8 bit, the partial products will be almost equal divided between the two blocks. The summation tree has six levels for both blocks, and the merging tree has two levels. Hence, the total depth is the same as for D1.

Design	BPs, levels	Control for conditional blocks (bit-products in the block, levels)			Merge levels	Balanced levels	Extra CPA
D1	91, 8	–	–	–	–	–	–
D2	47, 6	x_8 (44, 6)	–	–	2	Yes	No
D3	28, 5	x_8 (44, 6)	x_7 (19, 4)	–	3	No	No
D4	28, 5	x_8 (24, 5)	x_7 (39, 6)	–	3	No	No
D5	28, 5	x_8 (31, 5)	x_7 (32, 5)	–	3	Yes	No
D6	28, 5	x_8 (24, 5)	x_7 (19, 4)	x_8x_7 (20, 4)	4	No	No
D7	20, 4	x_8 (20, 4)	x_7 (26, 4)	x_6 (25, 4)	4	Yes	No
D8	28, 5	x_8 (31, 5)	x_7 (32, 5)	–	2	Yes	Yes
D9	20, 4	x_8 (20, 4)	x_7 (26, 4)	x_6 (25, 4)	3	Yes	Yes

Table 6.5 Design specifications. BPs is the number of bit-products in block 1, i.e., the block without control signal.

In D3 the 19 bit-products that include x_7 but not x_8, have formed a third block. Here, the merging tree has three levels resulting in a higher total depth. The bit-products will have a more balanced distribution if all bit-products that include x_8 but not x_7 are separated instead, as in D4. However, the total depth is not decreased by this.

An algorithm to divide the bit-products, which include both x_7 and x_8, between the two blocks with respect to the required number of levels in the summation trees has been developed. Note that this is a more complicated problem than to just assign the same number of bit-products to each block, since the depth of the tree depends on the complexity, i.e., the number of nonzero digits and the column positions of these digits, in the coefficients, c_i, from (6.9). Furthermore, the number of levels also depends on how the coefficients are represented, i.e., other signed-digit representations may be advantageous in some cases. However, only the CSD representation has been considered here. This algorithm resulted in the design D5, where all three blocks have five levels in the summation tree.

In D6 the 20 bit-products, which include both x_7 and x_8 have formed a third conditional block. Here, the merging tree has four levels, which results in an increased overall depth.

If x_6 is used as control signal to the third block, bit-products that include more than one of x_6, x_7, and x_8 can be assigned to the different

Design	Block 1		Conditional blocks						Merge		CPA		Total	
	FA	HA	FA	HA	FA	HA	FA	HA	FA	HA	FA	HA	FA	HA
D1	261	71	–	–	–	–	–	–	–	–	15	1	276	72
D2	125	48	117	50	–	–	–	–	34	7	18	1	294	106
D3	67	37	117	50	38	29	–	–	67	8	17	1	306	125
D4	67	37	52	57	97	52	–	–	70	8	17	1	303	155
D5	67	37	65	53	85	43	–	–	71	6	17	1	305	140
D6	67	37	52	57	38	29	40	30	101	26	17	1	315	180
D7	39	23	43	39	55	25	56	23	100	24	17	1	310	135
D8	67	37	65	53	85	43	–	–	36	5	51	4	304	142
D9	39	23	43	39	55	25	56	23	53	21	62	12	308	143

Table 6.6 Full and half adders for the different designs.

blocks so that balanced levels are obtained. In D7, all blocks have a four level tree.

Finally, designs including a CPA in the conditional blocks, as discussed in Section 6.3.2, have been implemented. The two most promising designs were selected for this modification, hence, D8 and D9 originate from D5 and D7, respectively.

Implementations

In this part, the different architectures will be compared in terms of area, speed, and energy consumption.

In Table 6.6, the number of full and half adders is given. Since there is one compensation vector for each block, the number of partial products, and, hence, the number of required full adders, is increased with the number of blocks. The number of required half adders is less predictable as it depends on the reduction scheme, and can probably be reduced by using a less strict method [4].

The area and critical path (CP) reported by the synthesis tool are given in Table 6.7. Naturally, the straightforward design, D1, requires the smallest area, as no control logic or extra registers are needed. Furthermore, the summation tree contains fewer full and half adders than the summation trees and merging operation for the divided architectures. For the same

Design	Area [mm^2]	CP* [ns]	CP [ns]	Frequency [MHz]	Energy [nJ]
D1	0.1161	8.95	16.63	60.13	0.2583
D2	0.1313	10.18	18.61	53.73	0.2439
D3	0.1417	10.36	18.95	52.77	0.2346
D4	0.1470	9.45	18.04	55.43	0.2311
D5	0.1445	8.69	17.28	57.87	0.2237
D6	0.1562	8.98	17.57	56.92	0.2313
D7	0.1502	8.49	17.08	58.55	0.2277
D8	0.1417	–	17.48	57.21	0.2183
D9	0.1470	–	17.20	58.14	0.2230

Table 6.7 Area, timing, and energy consumption. CP* is the portion of the total critical path excluding the final CPA.

reasons, the area increases with the number of conditional blocks. None of the divided designs has an area that is more than 35% larger than D1.

Also when speed is considered, the straightforward design, D1, is advantageous, having a critical path corresponding to a clock frequency of 60 MHz. However, the critical path is not increased by more than 14% for any of the divided designs. Furthermore, the architectures can easily be pipelined to an arbitrary degree to obtain a desired throughput. The column labeled CP* in Table 6.7 gives the length of the critical part except for the final CPA. As can be seen, the values are approximately half of the total critical path, so the speed may be doubled if pipelining is introduced before the CPA. Note that this is not true for the designs D8 and D9, as the delay path of the conditional blocks now also includes a CPA. Here, a simple ripple-carry adder has been used as CPA, and, hence, the speed can be increased if a faster CPA is selected.

The last column in Table 6.7 gives the simulated energy consumption. As can be seen, all divided architectures consume less energy than the straightforward design, D1. The lowest energy consumption is obtained for D8, which consists of three blocks with the same number of levels and has a CPA after the summation trees in the two conditional blocks. For D8, the energy consumption is decreased with more than 15% compared to D1. The savings comes with a cost of increased area and critical path

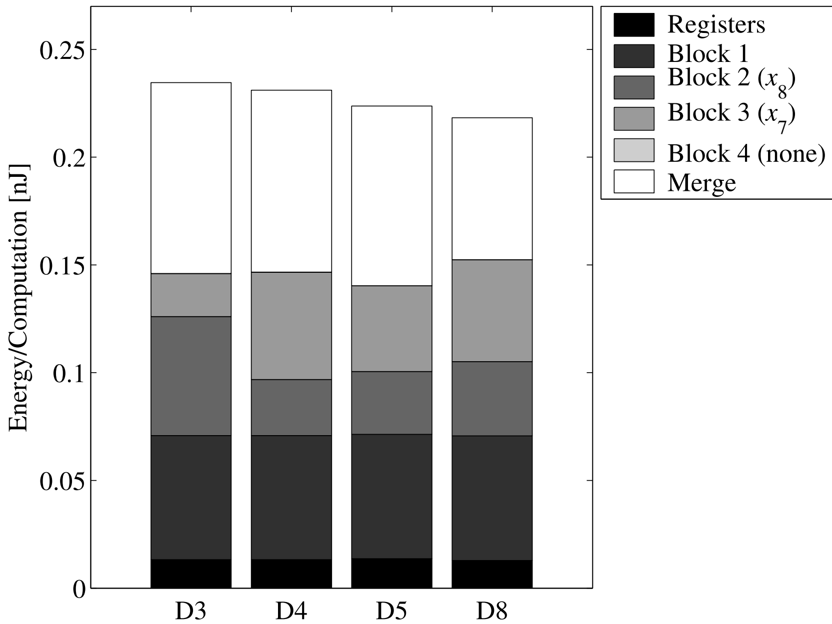


Figure 6.7 Breakdown of the energy consumption for designs containing three blocks.

by 22% and 5%, respectively. The lowest energy consumption among the designs without the introduced CPA is obtained for D5, where the savings are more than 13% compared to D1.

In the following, the distribution of the energy consumption between the different parts of the designs will be discussed.

In Fig. 6.7, the distribution of the energy consumption for the four different designs that consists of three blocks is shown. Naturally, there are no differences for the registers and the unconditional block.

The sum of the two conditional blocks is equal for D3 and D4. However, the merging consumes more energy for D3 because the summation trees here have a different number of levels for all three blocks, which increases the switching activity.

When all blocks have the same number of levels, as is the case for D5, the glitching energy is reduced due to balanced delays between propagation paths. If a CPA is introduced after the summation trees in the conditional blocks of D5, the energy for the blocks including the CPA increases, while the energy for the merging decreases, as can be seen for D8 in Fig. 6.7.

In Fig. 6.8, the designs that consist of four blocks are compared. For D6 the energy consumption of the unconditional block is the same as for

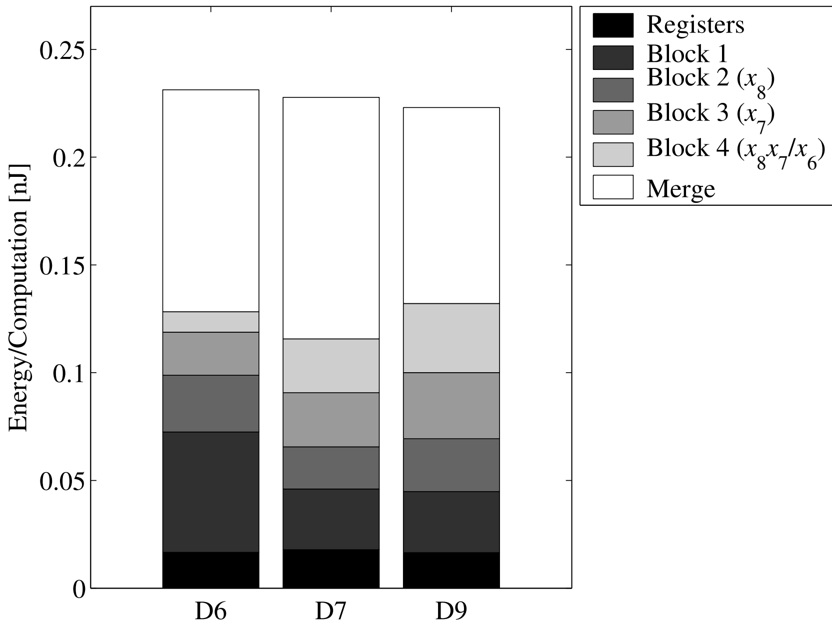


Figure 6.8 Breakdown of the energy consumption for designs containing four blocks.

the designs with three blocks, while it is significantly lower for D7 and D9 as all bit-products which include the input bit x_6 have been removed. Note the low energy consumption for the conditional block controlled by x_8x_7, i.e., block 4 in D6, which is active only 25% of the time. Comparing D7 and D9, the same effect as was discussed for D5 and D8 above can be seen.

The energy consumption for the D1, D2, D8, and D9 designs is illustrated in Fig. 6.9. As can be seen, the energy consumed in registers increases with the number of blocks. For D2, the unconditional block consumes approximately half the energy compared to D1, i.e., the energy consumption is proportional to the number of bit-products. Hence, the unconditional block consumes less energy when the number of blocks is increased as the number of included bit-products then is decreased, as can be seen for D8 and D9 in Fig. 6.9.

Furthermore, the conditional block in D2 consumes approximately half the energy compared to the unconditional block, i.e., the energy consumption is proportional to the activity.

For D1, the merge part illustrated in Fig. 6.9 is a single CPA, while for the divided designs it consists of a summation tree and a following CPA. Note that the merging part increases with the number of blocks. However,

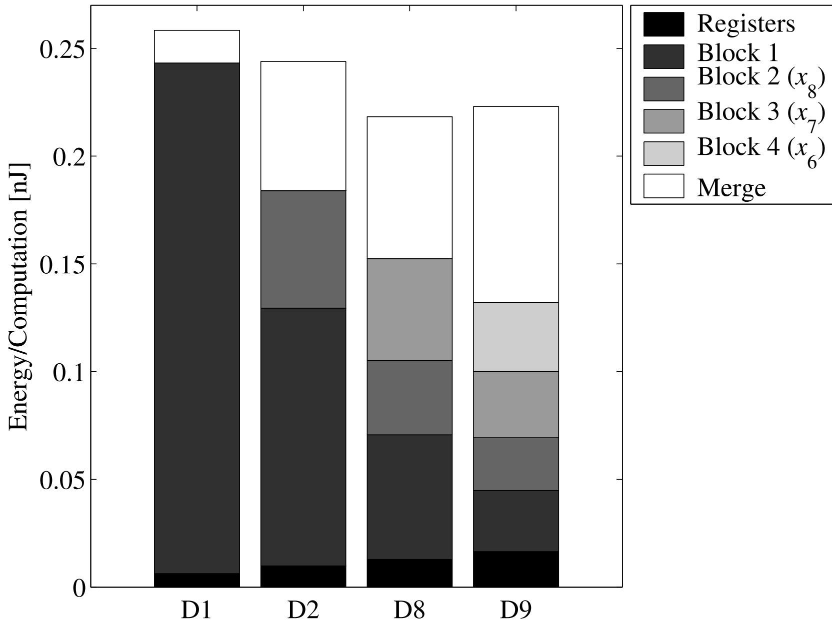


Figure 6.9 Distribution of the energy consumption for the best design containing 1, 2, 3, and 4 blocks, respectively.

the difference between D2 and D8 is small because the results from the two conditional blocks are partially merged by the introduced CPAs in D8.

6.4 Functions for LNS

As discussed in Section 1.3, computations in logarithmic number systems require realizations of four different elementary functions. The straightforward method is to use look-up tables implemented in ROMs. However, many table reducing strategies have been proposed for the precomputed $\Phi(x)$ values, required during addition and subtraction, for example, interpolation techniques [15] and biased addition [139]. Also, computation of logarithms and antilogarithms needed for the conversions to and from LNS have been studied. Proposed methods are, for example, a bit-by-bit computation [76] and a combination of linear and non-linear approximation [77].

Here, the function approximation method based on a weighted sum of bit-products is used to implement the look-up tables that are required for conversions and additions in LNS. A sign transformation that drastically

improves the results for functions with logarithmic characteristics is presented in the next section. It is shown that the considered method can be used to efficiently realize the different LNS functions. Furthermore, implementation results show that significant savings in area and energy can be obtained using optimization techniques.

The same variable names and definitions introduced for LNS in Section 1.3 will be used in the rest of this section.

6.4.1 Sign Transformation

Consider the logarithm function with $N = 10$ shown in Fig. 6.10 (a). For this function, the differences between the first function values are large, which gives large values of the coefficients, c_j, as illustrated in Fig. 6.10 (c). Hence, this function is not suitable to be implemented using the described method, since no accuracy will be obtained unless all bit-products are included, as confirmed by Fig. 6.10 (e).

However, the function values can be taken in the reversed order, as shown in Fig. 6.10 (b). This is compensated for by using the input data X instead of the original A according to

$$X = \overline{2^l|A| - 1} \tag{6.17}$$

which means that a least significant bit is subtracted, and all bits are then inverted. Note that this transformation does not add any hardware as X is obtained as the fractional bits if $A < 0$ and as the negated value if $A > 0$, i.e., a negation is performed for positive instead of negative values of A. In Fig. 6.10 (d), it can be seen that the magnitude of the coefficients decreases significantly if this transformation is used.

Comparing Figs. 6.10 (e) and (f) the efficiency of the sign transformation is clear. For the original function, all coefficients must be used, while for the reversed function a subset of the coefficients can be selected depending on the required accuracy. It will be shown in Section 6.4.2 that the area and energy consumption is proportional to the number of coefficients. Hence, large savings can be obtained by using the sign transformation.

Note that this method is implicitly applied for the antilogarithm function and the $\Phi(x)$ functions illustrated in Figs. 1.2 (b) and 1.4, respectively, as only negative values of E_A and x occur, i.e., the sign bit is discarded.

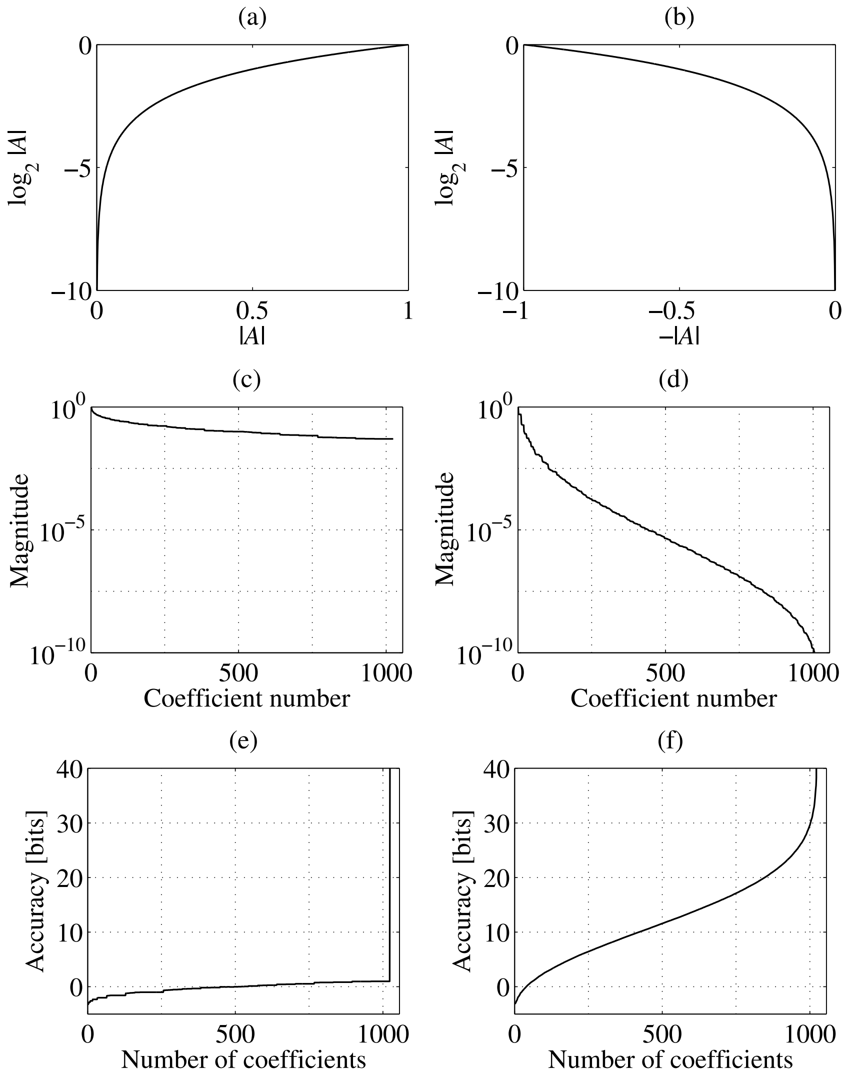


Figure 6.10 (a) The logarithm function and (b) the reversed function. Normalized distribution of the coefficients, c_j, (c) before and (d) after sign transformation. Accuracy vs. the number of bit-products (e) before and (f) after sign transformation.

Example

The sign transformation, will here be illustrated in an example. The function $\log_2 |A|$ is implemented using three fractional bits in the representation of A and three bits both in the integer and the fractional part of the

A	X	p_j	Q_N^j								$\log_2\lvert A\rvert$	c_j
–1	000	1	1	0	0	0	0	0	0	0	0	0
±7/8	001	x_0	–1	1	0	0	0	0	0	0	–1/4	–1/4
±3/4	010	x_1	–1	0	1	0	0	0	0	0	–3/8	–3/8
±5/8	011	x_1x_0	1	–1	–1	1	0	0	0	0	–5/8	0
±1/2	100	x_2	–1	0	0	0	1	0	0	0	–1	–1
±3/8	101	x_2x_0	1	–1	0	0	–1	1	0	0	–11/8	–1/8
±1/4	110	x_2x_1	1	0	–1	0	–1	0	1	0	–2	–5/8
±1/8	111	$x_2x_1x_0$	–1	1	1	–1	1	–1	–1	1	–3	–5/8

Table 6.8 Parameters for the logarithm function.

result. Function values, bit-products, and coefficients are presented in Table 6.8.

After conversion of A to X according to (6.17), the function values, $f(X)$, are obtained in the form of (6.3) as

$$f(X) = -\frac{1}{4}x_0 - \frac{3}{8}x_1 - x_2 - \frac{1}{8}x_2x_0 - \frac{5}{8}x_2x_1 - \frac{5}{8}x_2x_1x_0 \qquad (6.18)$$

To implement the function defined by (6.18), a suitable representation of the coefficients, c_j, must be found. In the same way as for the example in Section 6.3.1, the CSD representation together with a compensation vector in order to avoid sign extension will be used, as described in Table 6.9.

In Table 6.10 it is shown how the partial products should be added, for the case when $A = \pm 3/8$. Hence, the two's-complement representation of A is 0.011 or 1.101, which both gives $X = 101$. The obtained result, 110.101, is equal to –11/8 as expected.

Finally, the architecture in Fig. 6.5 is used to implement the logarithm function, which result in the design shown in Fig. 6.11. The number of partial products is given in bold. In total there are nine nonzero digits in the representation of the coefficients, and four ones in the compensation vector. Hence, there are in total 13 partial products to be added, both the first and the second level in the summation tree eliminates two partial products. After the CPA, the six required output bits are obtained. The

j	p_j	c_j	CSD	Compensation
0	1	0	0.000	
1	x_0	–1/4	$0.0\bar{1}0$	–1/4 = 111.110
2	x_1	–3/8	$0.\bar{1}01$	–1/2 = 111.100
3	x_1x_0	0	0.000	
4	x_2	–1	$\bar{1}.000$	–1 = 111.000
5	x_2x_0	–1/8	$0.00\bar{1}$	–1/8 = 111.111
6	x_2x_1	–5/8	$0.\bar{1}0\bar{1}$	–1/8 = 111.111 –1/2 = 111.100
7	$x_2x_1x_0$	–5/8	$0.\bar{1}0\bar{1}$	–1/8 = 111.111 + –1/2 = 111.100
Total compensation vector:				–25/8 = 100.111

Table 6.9 Coefficient representation.

2^2	2^1	2^0	2^{-1}	2^{-2}	2^{-3}
					x_1
			$\overline{x_1}$		$\overline{x_2x_0}$
			$\overline{x_2x_1}$		$\overline{x_2x_1}$
			$\overline{x_2x_1x_0}$	$\overline{x_0}$	$\overline{x_2x_1x_0}$
+ 1		$\overline{x_2}$	1	1	1

$\Rightarrow$ X = 101

2^2	2^1	2^0	2^{-1}	2^{-2}	2^{-3}
					0
			1		0
			1		1
			1	0	1
+ 1		0	1	1	1
1	1	0	1	0	1

Table 6.10 Partial products to be added in the summation tree.

numbers in italic corresponds to the performed operation when the input X = 101, i.e., the obtained result is the same as in Table 6.10. Note that the ones would be propagated slightly different if both zero valued partial products in the rightmost column were inputs to the top right full adder, since the carry to the next column would then be generated at the next tree level instead. For large implementations, it is possible to significantly reduce the energy consumption, by including partial products with high switching activity as late as possible in the summation tree [135].

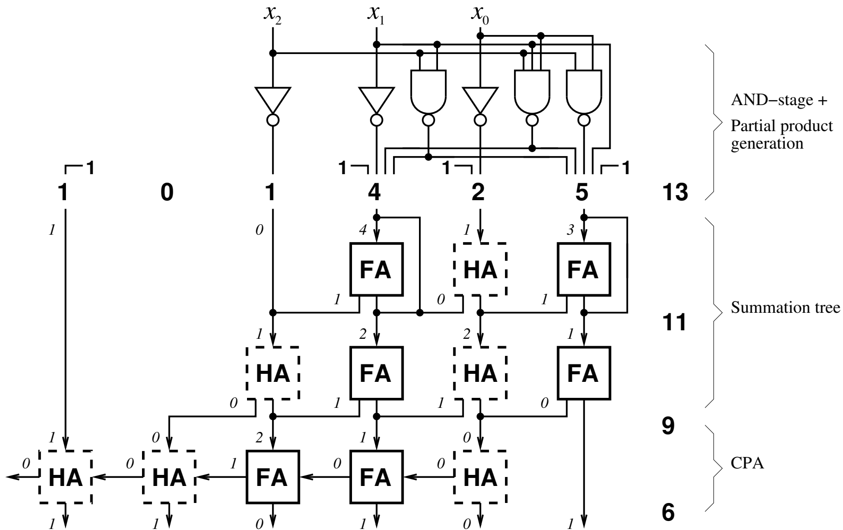


Figure 6.11 Implementation of the example function.

6.4.2 Results for the LNS Functions

In this section, the efficiency of the presented approximation method is investigated for the LNS functions. The number of integer and fractional bits are selected so that $k = 1$ and $l = 8$ in the linear domain, which according to (1.10) and (1.11) gives $K = 4$ and $L = 8$, respectively, in the logarithmic domain.

Accuracy vs. Complexity

The functions that are required in conversions and addition are shown in Figs. 1.2 and 1.4, respectively. The distribution of the coefficients for these four LNS functions is given in Fig. 6.12. It is clear that the coefficient magnitude variation is large for all functions. Hence, for a given accuracy many of the bit-products can be neglected.

The number of correct output bits based on maximum absolute error, i.e., the accuracy as defined in (6.16), obtained when neglecting the smaller coefficients, is shown in Fig. 6.13. It is clear that optimization increases the accuracy significantly when not all bit-products are used, i.e., a given accuracy can be obtained using fewer bit-products.

The number of bit-products that are required to obtain at least eight correct fractional bits, i.e., $\varepsilon \leq 2^{-9}$, is given in Table 6.11 for three of the studied functions. Large reductions are obtained for all functions, for

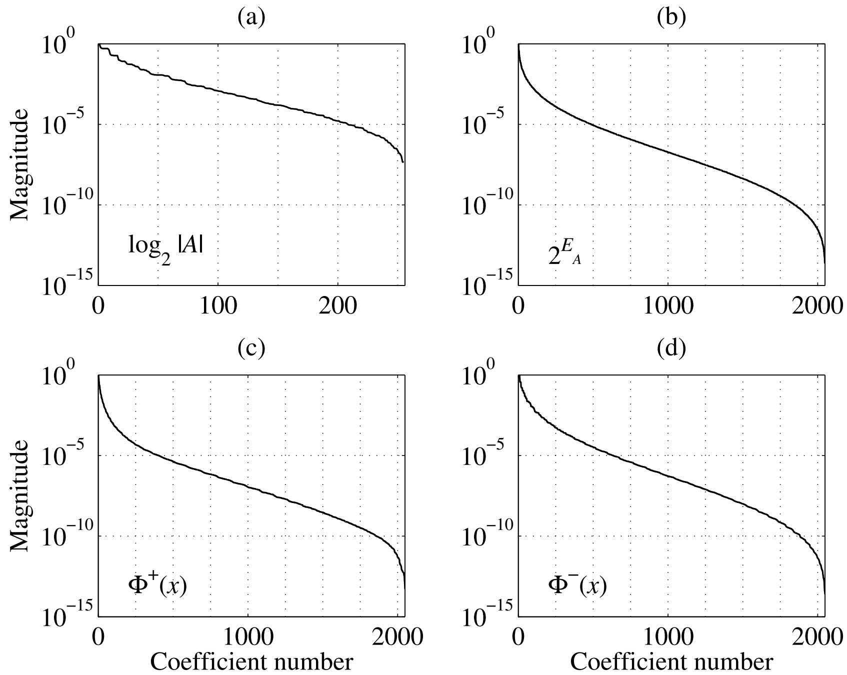


Figure 6.12 Normalized distribution of the coefficients, c_j.

<table>
<tr><th rowspan="2">Function</th><th rowspan="2">Number of input bits, N</th><th rowspan="2">Total number of coefficients, 2^N</th><th colspan="4">Required number of coefficients</th></tr>
<tr><th colspan="2">From (6.5)</th><th colspan="2">Optimized</th></tr>
<tr><td>$\log_2 |A|$</td><td>8</td><td>256</td><td>163</td><td>(64%)</td><td>99</td><td>(39%)</td></tr>
<tr><td>$\Phi^{+}(x)$</td><td>11</td><td>2048</td><td>192</td><td>(9%)</td><td>70</td><td>(3%)</td></tr>
<tr><td>$\Phi^{-}(x)$</td><td>11</td><td>2048</td><td>502</td><td>(25%)</td><td>227</td><td>(11%)</td></tr>
</table>

Table 6.11 Required number of coefficients, taken directly from (6.5) and optimized, respectively.

example, the $\Phi^{+}(x)$ function is realized with an error of at most 2^{-9} using only 3.4% out of the 2048 bit-products for optimized coefficients.

Implementations

In Table 6.12, design details are given. Both implementations with an accuracy of 9 bits, as found in Table 6.11, and with 12 bits using optimized coefficients are included. High-level simulations have been performed to verify the functionality and to check the accuracy. It was found

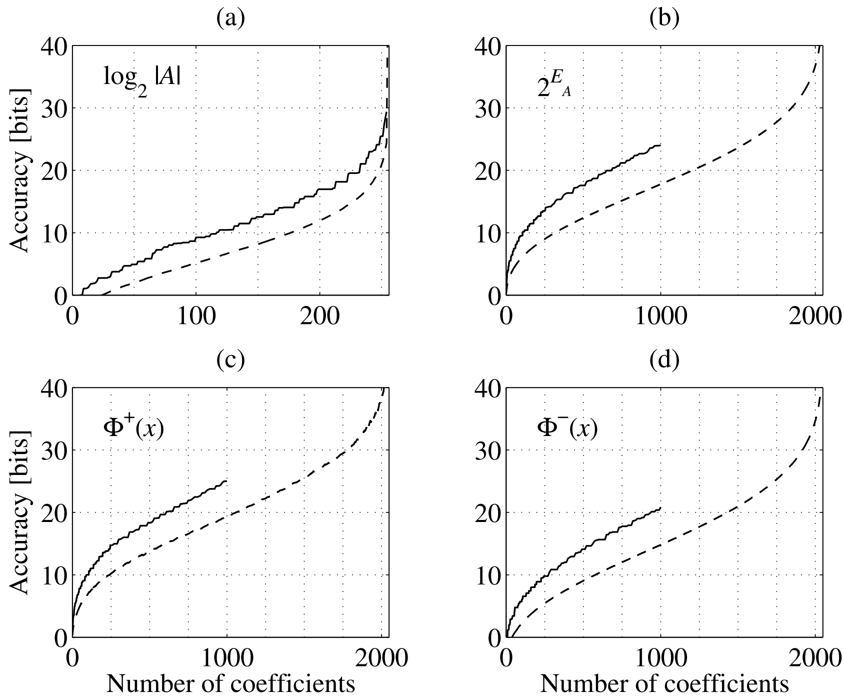


Figure 6.13 Accuracy vs. the number of bit-products. Coefficients from (6.5) (dashed) and optimized (solid).

that the absolute maximum error, ε, for all possible input data is less than 2^{-9} for the different functions, using the number of coefficients stated in Table 6.11.

The number of coefficient bits, W, given in Table 6.12 refers to the magnitude of the smallest coefficient included in the implementation, i.e., the smallest integer coefficient, d_j, is equal to 1 and the smallest coefficient, c_j, is equal to 2^{-W}, where $c_j = d_j 2^{-W}$ as defined in Section 6.2.2. For the optimized coefficients, fewer bit-products are required, i.e., larger coefficients are used, and, hence, the number of coefficient bits, W, is reduced. However, it is also possible to increase the number of bit-products and, by using optimization, find a solution requiring fewer coefficient bits, as illustrated by the two versions of the $\Phi(x)$ functions with 9 bit accuracy.

Also, the number of full and half adders in the summation tree, and the number of tree levels are given in Table 6.12. Again, for obvious reasons, there is a close relation between the number of nonzero digits in the coefficients and the required number of full adders.

Description	Function	Accuracy [bits]	Coeff. bits, W	BPs	PPs	FA	HA	Tree levels
9 bit accuracy, coefficients from (6.5)	$\log_2 \lvert A \rvert$	9.000	13	163	401	376	60	9
	$\Phi^+(x)$	9.244	14	192	341	316	66	9
	$\Phi^-(x)$	9.458	15	502	1129	1097	100	12
9 bit accuracy, optimized coefficients	$\log_2 \lvert A \rvert$	9.072	11	99	260	237	56	8
	$\Phi^+(x)$	9.010	13	70	171	150	46	7
	$\Phi^+(x)$	9.259	11	79	143	123	42	7
	$\Phi^-(x)$	9.147	12	227	574	550	72	10
	$\Phi^-(x)$	9.003	11	229	483	460	66	10
12 bit accuracy, optimized coefficients	$\log_2 \lvert A \rvert$	12.000	14	149	458	430	74	9
	$\Phi^+(x)$	12.121	14	169	342	320	62	9
	$\Phi^-(x)$	12.150	15	394	1043	1012	88	11

Table 6.12 Design specifications for the studied functions with 9 and 12 bit accuracy. BPs is the number of bit-products and PPs is the number of nonzero digits in the coefficients.

The obtained area and maximum sample frequency presented in Table 6.13 is based on synthesis of VHDL code using a 0.35 μm CMOS standard cell library. Naturally, the throughput can be increased by pipelining. By simulations at a clock frequency of 20 MHz, the power consumption was obtained using NanoSim™ with 100 random input samples. The same data were used for all implementations with the same number of input bits.

In Table 6.13, it can be seen that the area and energy consumption for the $\Phi(x)$ functions are reduced by more than 50% using optimization. Comparing the two optimized versions of the $\Phi(x)$ functions with an accuracy of 9 bits, it is clear that the implementations using fewer coefficient bits gives better results. The implementations of the $\Phi(x)$ functions with 12 bit accuracy have lower area and energy consumption than the non-optimized versions, i.e., three extra bits accuracy is obtained using optimization.

Most of the results given in Tables 6.12 and 6.13 are illustrated in Fig. 6.14. A linear relation can be observed between the number of

Description	Function	Accuracy [bits]	Area [mm^2]	Frequency [MHz]	Energy [nJ]
9 bit accuracy, coefficients from (6.5)	$\log_2 \lvert A \rvert$	9.000	0.1376	55.74	0.3484
	$\Phi^+(x)$	9.244	0.1258	58.86	0.3406
	$\Phi^-(x)$	9.458	0.3875	43.03	0.9187
9 bit accuracy, optimized coefficients	$\log_2 \lvert A \rvert$	9.072	0.0927	64.98	0.2125
	$\Phi^+(x)$	9.010	0.0626	70.57	0.1571
	$\Phi^+(x)$	9.259	0.0532	78.00	0.1153
	$\Phi^-(x)$	9.147	0.2056	48.95	0.4291
	$\Phi^-(x)$	9.003	0.1776	51.60	0.3533
12 bit accuracy, optimized coefficients	$\log_2 \lvert A \rvert$	12.000	0.1584	55.34	0.3759
	$\Phi^+(x)$	12.121	0.1237	64.68	0.3270
	$\Phi^-(x)$	12.150	0.3646	44.82	0.8705

Table 6.13 Implementation results using a 0.35 μm process.

included bit-products and the number of adder cells, area, sample frequency, and energy consumption. All these results are expected, since when the number of coefficients is increased, more adder cells are required to compute the final sum, which then result in a larger area. The critical path is increased as more levels are required in the summation tree, and, hence, the maximum sample frequency is decreased. Finally, because more cells are used, and, also, because the increased depth gives rise to more glitches, the energy consumption is increased. Note that the results in Figs. 6.14 (b), (d), and (f) are similar, i.e., if the number of non-zero digits in the coefficients is reduced, then the area and energy consumption are also decreased with the same factor.

6.4.3 Comparison with ROM

Here, the function approximation method is compared to look-up tables using a ROM generator in a 0.13 μm CMOS process. To make a fair comparison, the proposed architectures are also implemented using a 0.13 μm CMOS standard cell library (0.35 μm was used in the previous section to enable reliable power consumption simulations). As can be seen in

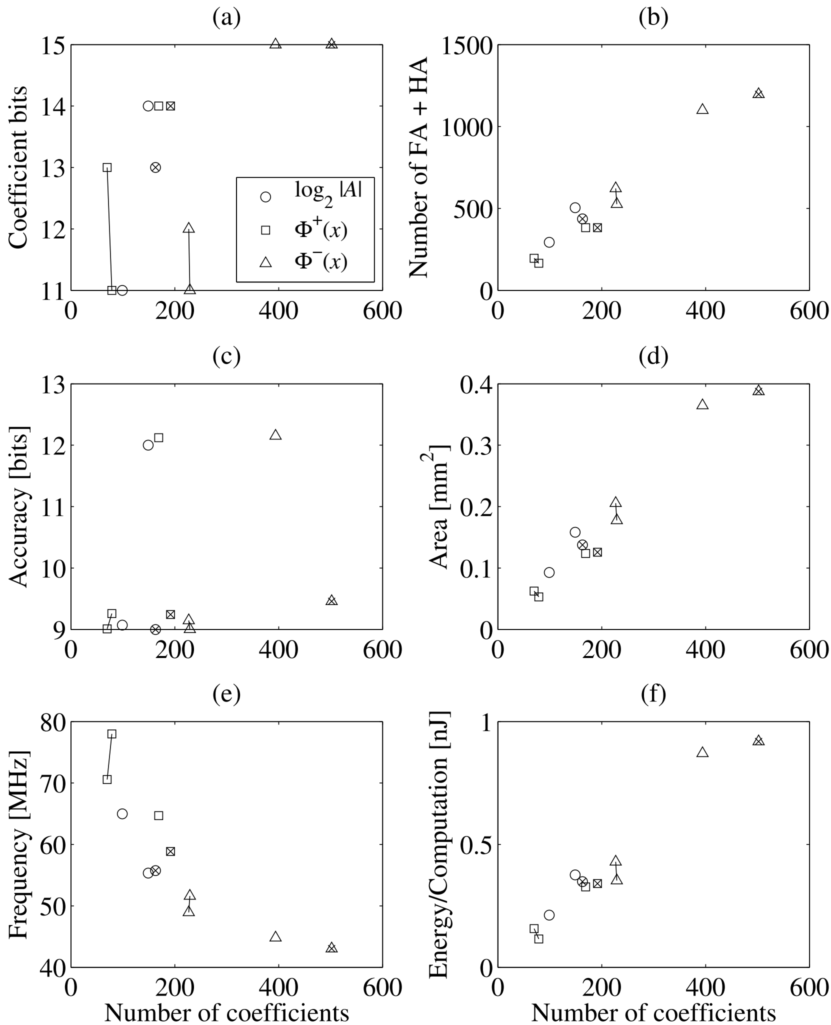


Figure 6.14 (a) Coefficient bits, (b) total number of full and half adders, (c) accuracy, (d) area, (e) maximum frequency, and (f) energy consumption vs. the number of bit-products. Symbols corresponding to implementations using coefficients directly from (6.5) are marked with an ×. The two versions of the $\Phi(x)$ functions with 9 bit accuracy are connected with a line.

Table 6.14, the ROM implementation is advantageous for the logarithm function, which only has eight input bits, i.e., the ROM contains 256 words of 11 bits each (3 integer and 8 fractional bits). For the $\Phi(x)$ functions, ROMs with 2048 words and 12 and 9 output bits, respectively, are

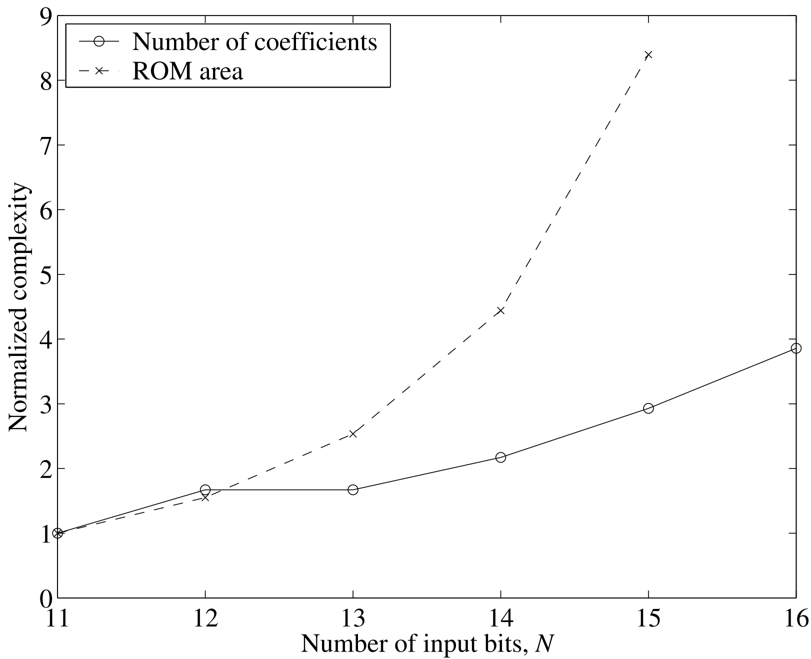


Figure 6.15 Complexity comparison between ROM and the presented method for the $\Phi^+(x)$ function.

Function	Area [μm²]		Savings
	ROM	Sum of bit-products	
$\log_2 \lvert A \rvert$	11 999	12 341	–3%
$\Phi^+(x)$	26 642	7 210	73%
$\Phi^-(x)$	30 587	22 778	26%

Table 6.14 Area results for the studied functions with 9 bit accuracy, implemented using a 0.13 μm process.

required. For these functions, the presented approximation method results in significant area reductions.

For the $\Phi^+(x)$ function, the number of required bit-products has been investigated for different cases, using from 11 ($K = 4$, $L = 8$) up to 16 ($K = 5$, $L = 12$) input bits. The number of bit-products then varies from 70 up to 270. As discussed in the previous section, there is a linear relation between the number of included bit-products and the area. Hence, this

indicates a growth of the area by a factor 3.86. A comparison with the corresponding ROM implementations is shown in Fig. 6.15 (due to limits in the ROM generator, no implementation could be obtained for $N = 16$). Note that although both curves start in 1 for $N = 11$, the proposed method here has 73% less chip area for the $\Phi^{+}(x)$ function. For the other functions, the normalized curves would have the same characteristics, but correspond to different area values. Hence, for increasing wordlength the area savings will be significant using the proposed method compared to ROM look-up tables.

6.5 Sine and Cosine Functions

Computation of trigonometric functions, mainly sine and cosine values from an angle, have several applications in digital signal processing systems. For example, for computing the twiddle factors in fast Fourier transforms (FFT) and for converting angle to phase in direct digital frequency synthesizers (DDFS) [81]. Several different digital approaches for the angle to sine/cosine conversion have previously been proposed. In [81] an overview of different applicable techniques is given. Other work published after this overview includes, for example, [10], [11], and [148].

Here, the approximation approach presented in Section 6.2.1 will be used to implement the sine and cosine functions. However, the computation corresponding to (6.3) is here derived using trigonometric identities. The approach is based on angle rotation and decomposition to simultaneously compute sine and cosine. It is shown that it is possible to express several angle rotations as a weighted sum of bit-products. Again, the bit-products with smallest weights can be neglected, and the same low-complexity realization as before is thereby obtained.

6.5.1 Angle Rotation Based Approach

The aim is to simultaneously compute sine and cosine of the angle input argument A, which is expressed in terms of the angle resolution, α. Here, $\alpha = 2\pi/2^{N}$, for a resolution of 2^{N} equally spaced points around the unit circle. Using an N-bit unsigned binary representation for X as defined in (6.2), the argument can be written as

$$A = \alpha X = \alpha \sum_{i=0}^{N-1} x_i 2^i \qquad (6.19)$$

Consider the problem of computing sin(A) and cos(A), i.e., evaluate e^{jA}, by performing angle rotations. Using (6.19) this can be expressed as

$$e^{jA} = e^{j\alpha \sum_{i=0}^{N-1} x_i 2^i} = e^{j2^{N-1}\alpha x_{N-1}} e^{j2^{N-2}\alpha x_{N-2}} \ldots e^{j4\alpha x_2} e^{j2\alpha x_1} e^{j\alpha x_0} \qquad (6.20)$$

Using standard trigonometric formulas, the sine and cosine values of A can then be written in matrix form as

$$\begin{bmatrix} \sin(A) \\ \cos(A) \end{bmatrix} = \prod_{k=1}^{N-1} \begin{bmatrix} \cos(2^k \alpha x_k) & \sin(2^k \alpha x_k) \\ -\sin(2^k \alpha x_k) & \cos(2^k \alpha x_k) \end{bmatrix} \begin{bmatrix} \sin(\alpha x_0) \\ \cos(\alpha x_0) \end{bmatrix} \qquad (6.21)$$

This expression represents N binary weighted angle rotations of the unit vector, $e^{j\alpha x_0}$. Hence, each matrix represent that the angle of the unit vector is either unchanged or increased with a factor $2^k\alpha$. These rotations can easily be implemented directly using either real, complex, or distributed arithmetic [149]. Note that the coordinate rotation digital computer (CORDIC) algorithm [143], can be obtained by rewriting (6.21) so that the matrix multiplications becomes a sequence of simple additions and subtractions.

Now, since $x_i \in \{0, 1\}$, it is possible to write

$$\sin(2^k \alpha x_k) = \sin(2^k \alpha) x_k \qquad (6.22)$$

and

$$\cos(2^k \alpha x_k) = 1 - (1 - \cos(2^k \alpha)) x_k \qquad (6.23)$$

Extracting the binary variables, by applying (6.22) and (6.23) to (6.21), and performing the matrix multiplications gives an equation that only contains products of bit variables, x_i, multiplied with constants, i.e., $\sin(2^k\alpha)$ and so on. Hence, the expressions for sin(A) and cos(A) can be written as a weighted sum of bit-products as

$$\begin{bmatrix} \sin(A) \\ \cos(A) \end{bmatrix} = \begin{bmatrix} \sum_{j=0}^{2^N-1} s_j p_j \\ \sum_{j=0}^{2^N-1} c_j p_j \end{bmatrix} \qquad (6.24)$$

where s_i and c_i are the weights of the bit-products for the sine and cosine expressions, respectively. Again, the bit-products, p_j, are computed using simple AND-operations according to (6.4). Similar as for all other studied functions in this chapter, it was shown in [151] that also for sine and cosine the magnitude of most of the coefficients s_i and c_i is small, and many can therefore be neglected.

6.5.2 Octant Mapping

When computing both sine and cosine values for the complete circle ($0 \leq A \leq 2\pi$), the symmetric properties of the sine and cosine functions may be used. To evaluate e^{jA} at 2^N equally spaced points on the unit circle, it is only necessary to evaluate the first $2^N/8 + 1$ points. This is because each octant can be mapped to the first octant through a rotation, or a rotation combined with a mirroring operation. Hence, we only need to compute sin(A) and cos(A) in the first octant. This octant symmetry technique leading to that only sine and cosine values between 0 and $\pi/4$ are required has been frequently used in, for example, quadrature DDFS [81],[92].

By using the three most significant bits of X, we can convert any angle in the range $0 \leq A < 2\pi$ to a corresponding angle in the range $0 \leq A' \leq \pi/4$ as indicated by Table 6.15. Hence, when considering A' as the input argument, there are originally only $2^{N-3} + 1$ terms in each part of (6.24), which simplify the optimization procedure. Consequently, the term $X' = A'/\alpha$ denotes a representation using the $N - 2$ least significant bits of X. The third most significant bit, x_{N-3}, is zero for all angles in the range except for $A' = \pi/4$. Note that using standard trigonometric functions as, for example, $\sin(-A') = -\sin(A')$ is not possible in Table 6.15, as this would result in A' being outside the first octant, i.e., the valid range.

Using the relations in Table 6.15 for computing sine and cosine over all possible angles, the complexity can be significantly reduced by conditionally negating the least significant part of the input, i.e., X', and condi-

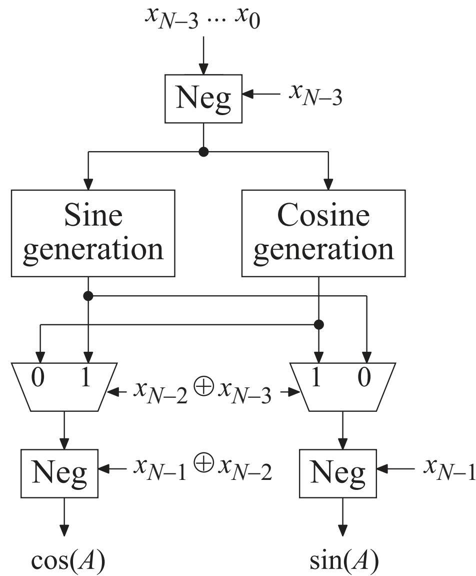


Figure 6.16 Architecture for computing sine and cosine simultaneously using only one octave of input values due to symmetry.

Octant	MSBs	cos(A)	sin(A)
$0 \le A < \pi/4$	000	cos(A')	sin(A')
$\pi/4 \le A < \pi/2$	001	sin(–A')	cos(–A')
$\pi/2 \le A < 3\pi/4$	010	–sin(A')	cos(A')
$3\pi/4 \le A < \pi$	011	–cos(–A')	sin(–A')
$\pi \le A < 5\pi/4$	100	–cos(A')	–sin(A')
$5\pi/4 \le A < 3\pi/2$	101	–sin(–A')	–cos(–A')
$3\pi/2 \le A < 7\pi/4$	110	sin(A')	–cos(A')
$7\pi/4 \le A < 2\pi$	111	cos(–A')	–sin(–A')

Table 6.15 Computation of sine and cosine using octant mapping.

tionally interchanging and negating sin(A') and cos(A'). Negation of the bits in X' should be handled as for two's complement, i.e., inverting and adding a one to the LSB position. The resulting implementation is illustrated in Fig. 6.16. The sine and cosine generation parts are realized in the same way as before, i.e., as shown in Fig. 6.5. However, note that the

AND-stages of the sine and cosine generation, and possibly also parts of the summation trees, now may be merged.

6.5.3 Comparison with CORDIC

As an example, consider a twiddle factor generator for a 4096-point FFT processor. The accuracy should be 16 bits or better for both sine and cosine.

The coefficients are optimized for a small number of nonzero bits in the CSD representation, using 20 bit coefficients. The number of unique bit-products for the terms in (6.24), with nonzero coefficients after optimization, is 111 in total. There are 69 bit-products that are used for both sine and cosine, with 20 unique for sine and 22 unique for cosine. Thus, sine has 89 and cosine 91 weighted bit-products. There are 273 and 296 nonzero digits in the CSD representation of the coefficients for sine and cosine, respectively. Note that this is the same design of the cosine function as was used for the different implementations in Section 6.3.3. Hence, the energy consumption could also here be decreased by using conditional blocks.

The summation tree for sine has 7 levels and consists of 241 FA and 67 HA, while the summation tree for cosine has 8 levels and require 261 FA and 71 HA. Each summation tree approximately corresponds to a 16×16-bit multiplier. The carry propagation adder and all negations are realized using ripple-carry adders.

The total number of gates, for a straightforward VHDL synthesis, is 2650 cells with an area of 0.23 mm^2 using a 0.35 μm CMOS standard cell library. The critical path without pipelining is 27 ns, corresponding to a throughput of 37.2 Msample/s. Of course, the throughput can be increased by pipelining, circuit optimization, and using a faster carry propagation adder.

For comparison, a straightforward unfolded CORDIC implementation is used. A general CORDIC architecture can be used to compute a large variety of trigonometric functions [2] and transforms [51]. However, the design implemented here has been simplified to only be able to compute sine and cosine. The wordlengths are 22 bits for the data accumulators and 21 bits for the angle accumulator. The CORDIC algorithm is unfolded 18 times. This gives a worst case accuracy of 16.15 bits. The implementation consists of 3876 standard cells, with a total area of

Implementation	Gates	Area [mm^2]	Critical path [ns]	Frequency [MHz]	Energy [nJ]
Sum of bit-products	2650	0.2310	26.85	37.24	0.5948
CORDIC	3876	0.4059	185.07	5.40	9.5238

Table 6.16 Two implementations of a 4096-point twiddle factor generator.

0.41 mm^2. The critical path is 185 ns, corresponding to a throughput of 5.4 MSample/s.

The implementation results are summarized in Table 6.16, where also the energy consumption obtained using NanoSim™ with 100 random input samples at an input rate of 2 MHz is given. The CORDIC design consumes 16 times more energy than the presented approach. The main reason for this large difference is the long propagation paths in the CORDIC architecture, which result in high switching activity. This can be verified by studying the number of output transitions, which is 15 and 13 times more for the CORDIC design at the sine and cosine outputs, respectively. Hence, the energy would probably decrease significantly if pipelining was introduced in the CORDIC architecture.

Even though it is possible to enhance the CORDIC architecture in several ways, this example shows the viability of the presented approach. For this case, it results in a chip area that is 43% smaller compared with the straightforward unfolded CORDIC.

6.6 Conclusions

In this chapter, a method to rewrite an arbitrary function as a sum of weighted bit-products was presented. It was shown that for many functions a majority of the bit-products can be neglected while still maintaining reasonable high accuracy, since the weights are significantly smaller than the allowed error. Furthermore, it was also shown that the complexity, i.e., the number of included terms, and thereby the area and energy consumption, can be significantly decreased by optimization.

The function approximation algorithms can be implemented using a fast, low latency, hardware architecture, which can easily be pipelined to an arbitrary degree for high throughput. Furthermore, different ways to divide the architecture into multiple summation trees, where sub-blocks composed of bit-products that all include a certain input bit may be turned

off, were evaluated. It was shown that by using such architectures, the energy consumption was decreased up to 15% for the example implementation.

The function approximation method was also evaluated when used to implement the look-up tables that are required for conversions and addition in LNS. A sign transformation, which significantly improves the efficiency for functions where the difference between subsequent function values is large in the beginning of the number range, was presented.

An approach to simultaneously compute both sine and cosine by a sum of weighted bit-products was derived. To reduce the implementation complexity, an arbitrary angle in the interval 0 to 2π rad is mapped to the first octant. It was noted that parts of the architectures used to compute the sine and cosine, respectively, may be shared.

It is not straightforward to compare the presented approach with table look-up methods like [33], as the implementation cost of a table differs significantly with implementation technology. However, for specific functions and specific implementation technologies it is, of course, possible. It was shown that significant area savings can be obtained compared to ROM implementations, especially for larger wordlengths. However, a combination of the method used here and other proposed table reducing techniques may give improved results. Furthermore, it was shown that the method is efficient for sine and cosine computation. Compared to a straightforward unfolded CORDIC implementation, the throughput for the presented method, without pipelining, was increased by a factor of seven, while the required area was essentially halved. These results were obtained for an input wordlength of 12 bits and an output accuracy of 16 bits.

7

CONCLUSIONS

7.1 Summary

In this thesis, the possibilities to improve the complexity and the energy consumption of arithmetic operations in digital circuits were investigated. More specific, the focus was on single- and multiple-constant multiplications, which are implemented using shift-and-add operations. Both serial and parallel arithmetic were considered. The main difference, which was of interest here, is that shift operations require flip-flops in serial arithmetic, while it can be hardwired in parallel arithmetic.

The possible ways to connect a certain number of adders is limited, i.e., the number of possible shift-and-add structures is finite for single-constant multiplication. Hence, it is possible to find the best solution for each constant, in terms of complexity, by an exhaustive search. We proposed a minimum set of graphs that are required to obtain optimal results in terms of complexity, for different multiplier types that are constrained by adder cost and throughput. Here, the throughput is considered by defining structures where the critical path, for bit-serial arithmetic, is no longer than one full adder. The results show that it is possible to save both adders and shifts compared to CSD serial/parallel multipliers. However, there is a clear trade-off between the adder and flip-flop costs. Two algorithms for the design of multiplier blocks using serial arithmetic were proposed. The difference between these algorithms is the priority of reducing the number of adders and shifts, respectively. For the first algorithm, the

number of shifts can be significantly reduced, while the number of adders is slightly increased, compared to one of the best-known algorithms for parallel arithmetic. For the second algorithm, the number of shifts can be reduced, while the number of adders is on average the same. Hence, for both algorithms, the total complexity of multiplier blocks is decreased. The impact of the digit-size was studied by implementing an FIR filter with varying digit-size. Besides the two proposed multiplier block algorithms and an algorithm for parallel arithmetic, separate realization of the multipliers using CSD serial/parallel multipliers was used. The focus was on the arithmetic parts, i.e., the multiplier block and the structural adders. The results provide some guidelines for designing low power MCM algorithms for FIR filters implemented using digit-serial arithmetic. The placement of shifts is crucial since they reduce the number of glitches. Possibly, except for bit-serial processing, it is more important to minimize the number of adders than the number of shifts. Furthermore, the relation between energy consumption and adder depth was discussed.

The energy consumption is proportional to the switching activity. A method for computing the number of logic switches in bit-serial constant multipliers was proposed. Although the method is only used for single-constant multiplication here, it could also be useful for multiple-constant multiplication, for example, as cost function in heuristic algorithms. The average switching activity in all graph multipliers with up to four adders can be determined. The derived equations can be applied to more than 83% of the adders in the graph topologies. For the remaining cases, look-up tables, which are generated with the same method as the equations were derived, can be used. Hence, it is possible to reduce the switching activity by selecting the best structure for any given constant to be implemented. In addition, a simplified method for computing the switching activity in constant serial/parallel multipliers was presented. Here it is possible to reduce the energy consumption by selecting the best signed-digit representation of the constant.

For parallel arithmetic, a detailed complexity model for MCM blocks was proposed. The model counts the number of full and half adder cells required to realize an MCM block. A transformation that can be used to eliminate the use of half adders, at no extra cost, was introduced. Based on this model, a novel algorithm for the MCM problem was proposed, which provides significantly improved results compared with previous algorithms. The complexity model can also be used for single constant coefficient multipliers, constant matrix multipliers, and FIR filters. Fur-

thermore, interconnection strategies that can be applied independently of which algorithm that is used to solve the MCM problem were presented. It was shown that the complexity in terms of full and half adders could be significantly reduced. A main factor for energy consumption in multiplier blocks is adder depth, i.e., the number of cascaded adders. Hence, the proposed interconnection algorithm that given an MCM solution result in minimum depth, while the complexity is also considered, is advantageous in most cases. However, it is usually possible to reduce the depth by selecting a different MCM solution. Therefore, we proposed an algorithm for MCM problems, where all multiplier coefficients are guaranteed to be realized at the theoretically lowest possible adder depth. For an FIR filter example, it is shown that this algorithm result in a multiplier block with around 25% lower energy consumption compared to MCM solutions using fewer word level adders.

A data dependent switching activity model was derived for ripple-carry adders. For most applications, the input data are correlated, while previous estimations assumed uncorrelated data. Hence, the proposed method may be included in high-level power estimation to obtain results that are more accurate. The model is accurate in estimating the switching activity of the carry and sum signals. However, the energy consumption was overestimated since not all switches in the implemented adders were rail-to-rail (full swing). In addition, a modified model based on word-level statistics was presented. This model is accurate in estimating the switching activity when real world signals are applied. In MCM blocks, there is in many cases also a high correlation between the two signals to be added. A switching activity model for the single adder multiplier was proposed. The model was shown to agree well with high-level simulations, with an error of at most 0.26% for the studied test cases. Since the single adder multiplier is a common part in multiplier blocks, the proposed model is suitable to be used as cost function in energy consumption aware MCM algorithms. Furthermore, it was concluded that an event-based model considering each full adder individually gives accurate results under the condition that the inputs are uncorrelated. Thus, correlation between signals is the main problem when developing a general model for estimation of the switching activity in MCM implementations.

Finally, a method to rewrite an arbitrary function as a sum of weighted bit-products was presented. It was shown that for many elementary functions, a majority of the bit-products could be neglected while still maintaining reasonable high accuracy, since the weights are significantly

smaller than the allowed error. Furthermore, it was also shown that the complexity, i.e., the number of included terms, and thereby the area and energy consumption, can be significantly reduced by optimization. Significant area savings can be obtained compared to ROM implementations, especially for larger wordlengths. The function approximation algorithms can be implemented using a fast, low latency, hardware architecture, which can easily be pipelined to an arbitrary degree for high throughput. Furthermore, different ways to divide the architecture into multiple summation trees, where the sub-blocks are conditionally turned off, were evaluated. It was shown that by using such architectures, the energy consumption was decreased with up to 15% for the example implementation. The function approximation method was evaluated when used to implement the look-up tables that are required for conversions and additions in logarithmic number systems. A sign transformation, which significantly improves the results for functions with logarithmic characteristics, was presented. Furthermore, an alternative approach to simultaneously derive both sine and cosine by a sum of weighted bit-products was given.

7.2 Future Work

Most of the methods presented in this thesis can be refined further. For example, the switching activity estimation for bit-serial multipliers would be more realistic if sign extension was considered. For parallel arithmetic, models for other adder structures than the RCA could be developed. To solve the correlation problem in MCM is a great challenge. By combining the models for bit-serial and parallel arithmetic, a model for digit-serial implementations could be obtained. Estimation of the switching activity using carry save arithmetic is also an interesting topic.

When an accurate and complete switching activity model is derived, it should be used in an MCM algorithm. To integrate the filter design would also give improved results for specified filter requirements. For example, the filter coefficients can be optimized using complexity and activity models for MCM. Furthermore, quantization should also be considered.

The function approximation method could be improved by searching for coefficients that will reduce the complexity by locating the nonzero digits in favorable columns. Comparison with as many as possible of all other available methods is also a subject for future work.

REFERENCES

[1] M. Alioto and G. Palumbo, "Analysis and comparison on full adder block in submicron technology," *IEEE Trans. VLSI Syst.*, vol. 10, no. 6, pp. 806–823, Dec. 2002.

[2] R. Andraka, "A survey of CORDIC algorithms for FPGA based computers," in *Proc. ACM/SIGDA Int. Symp. Field Programmable Gate Arrays*, Monterey, CA, Feb. 22–25, 1998, pp. 191–200.

[3] Ch. Basetas, I. Kouretas, and V. Paliouras, "Low-power digital filtering based on the logarithmic number system," in *Proc. Int. Workshop Power Timing Modeling Optimization Simulation*, Gothenburg, Sweden, Sept. 3–5, 2007, pp. 546–555.

[4] K. A. C. Bickerstaff, M. Schulte, and E. E. Swartzlander Jr., "Reduced area multipliers," in *Proc. Int. Conf. Application-Specific Array Processors*, Oct. 25–27, 1993, pp. 478–489.

[5] N. Brisebarre, D. Defour, P. Kornerup, J.-M. Muller, and N. Revol, "A new range-reduction algorithm," *IEEE Trans. Computers*, vol. 54, no. 3, pp. 331–339, Mar. 2005.

[6] N. Brisebarre, J.-M. Muller, and A. Tisserand, "Computing machine-efficient polynomial approximations," *ACM Trans. Mathematical Software*, vol. 32 , no. 2, pp. 236–256, June 2006.

[7] D. R. Bull and D. H. Horrocks, "Primitive operator digital filters," *IEE Proc. G*, vol. 138, no. 3, pp. 401–412, June 1991.

[8] T. K. Callaway and E. E. Swartzlander Jr., "Optimizing arithmetic elements for signal processing," in *Proc. IEEE Workshop VLSI Signal Processing*, Oct. 28–30, 1992, pp. 91–100.

[9] T. K. Callaway and E. E. Swartzlander Jr., "Estimating the power consumption of CMOS adders," in *Proc. 11th IEEE Symp. Comp. Arithmetic*, Windsor, Canada, June 29–July 2, 1993, pp. 210–216.

[10] D. De Caro, E. Napoli, and A. G. M. Strollo, "Direct digital frequency synthesizers with polynomial hyperfolding technique," *IEEE Trans. Circuits Syst. II*, vol. 51, no. 7, pp. 337–344, July 2004.

[11] D. De Caro and A. G. M. Strollo, "High-performance direct frequency synthesizers using piecewise-polynomial approximation," *IEEE Trans. Circuits Syst. I*, vol. 52, no. 2, pp. 324–337, Feb. 2005.

[12] A. P. Chandrakasan, S. Sheng, and R. W. Brodersen, "Low-power CMOS digital design," *IEEE J. Solid-State Circuits*, vol. 27, no. 4, pp. 473–484, Apr. 1992.

[13] A. P. Chandrakasan, M. Potkonjak, R. Mehra, J. Rabaey, and R. W. Brodersen, "Optimizing power using transformations," *IEEE Trans. Computer-Aided Design*, vol. 14, no. 1, pp. 12–31, Jan. 1995.

[14] A. P. Chandrakasan and R. W. Brodersen, "Minimizing power consumption in digital CMOS circuits," *Proc. IEEE*, vol. 83, no. 4, pp. 498–523, Apr. 1995.

[15] J. N. Coleman, "Simplification of table structure in logarithmic arithmetic," *Electronics Letters*, vol. 31, no. 22, pp. 1905–1906, Oct. 1995.

[16] J. N. Coleman, E. I. Chester, C. I. Softley, and J. Kadlec, "Arithmetic on the European logarithmic microprocessor," *IEEE Trans. Computers*, vol. 49, no. 7, pp. 702–715, July 2000.

[17] J. N. Coleman, C. I. Softley, J. Kadlec, R. Matousek, M. Tichy, Z. Pohl, A. Hermanek, and N. F. Benschop, "The European logarithmic microprocesor," *IEEE Trans. Computers*, vol. 57, no. 4, pp. 532–546, Apr. 2008.

[18] O. Coudert, "On solving covering problems," in *Proc. 33rd Design Automation Conf.*, June 3–7, 1996, pp. 197–202.

[19] M. Daumas, C. Mazenc, X. Merrheim, and J.-M. Muller, "Modular range reduction: a new algorithm for fast and accurate computation of the elementary functions," *J. Universal Computer Science*, vol. 1, no. 3, pp. 162–175, Mar. 1995.

[20] L. S. DeBrunner, "Reducing complexity of FIR filter implementations for low power applications," in *Proc. Asilomar Conf. Signals Syst. Comp.*, Pacific Grove, CA, Nov. 4–7, 2007, pp. 1407–1411.

[21] S. S. Demirsoy, A. G. Dempster, and I. Kale, "Transition analysis on FPGA for multiplier-block based FIR filter structures," in *Proc. IEEE Int. Conf. Electronics Circuits Syst.*, Lebanon, Dec. 17–20, 2000, vol. 2, pp. 862–865.

[22] S. S. Demirsoy, A. G. Dempster, and I. Kale, "Power analysis of multiplier blocks," in *Proc. IEEE Int. Symp. Circuits Syst.*, Scottsdale, AZ, May 26–29, 2002, vol. 1, pp. 297–300.

[23] S. S. Demirsoy, A. G. Dempster, and I. Kale, "Power consumption behaviour of multiplier block algorithms," in *Proc. IEEE Midwest Symp. Circuits Syst.*, Tulsa, OK, Aug. 4–7, 2002, vol. 3, pp. 1–4.

[24] A. G. Dempster and M. D. Macleod, “Constant integer multiplication using minimum adders,” *IEE Proc. Circuits Devices Syst.*, vol. 141, no. 5, pp. 407–413, Oct. 1994.

[25] A. G. Dempster and M. D. Macleod, “Use of minimum-adder multiplier blocks in FIR digital filters,” *IEEE Trans. Circuits Syst. II*, vol. 42, no. 9, pp. 569–577, Sept. 1995.

[26] A. G. Dempster, S. S. Demirsoy, and I. Kale, “Designing multiplier blocks with low logic depth,” in *Proc. IEEE Int. Symp. Circuits Syst.*, Scottsdale, AZ, May 26–29, 2002, vol. 5, pp. 773–776.

[27] A. G. Dempster, O. Gustafsson, and J. O. Coleman, “Towards an algorithm for matrix multiplier blocks,” in *Proc. European Conf. Circuit Theory Design*, Kraków, Poland, Sept. 1–4, 2003.

[28] A. G. Dempster and M. D. Macleod, “Digital filter design using subexpression elimination and all signed-digit representations,” in *Proc. IEEE Int. Symp. Circuits Syst.*, Vancouver, Canada, May 23–26, 2004, vol. 3, pp. 169–172.

[29] A. G. Dempster and M. D. Macleod, “Using all signed-digit representations to design single integer multipliers using subexpression elimination,” in *Proc. IEEE Int. Symp. Circuits Syst.*, Vancouver, Canada, May 23–26, 2004, vol. 3, pp. 165–168.

[30] A. G. Dempster and M. D. Macleod, “Generation of signed-digit representations for integer multiplication,” *IEEE Signal Processing Letters*, vol. 11, no. 8, pp. 663–665, Aug. 2004.

[31] A. G. Dempster and M. D. Macleod, “Multiplication by two integers using the minimum number of adders,” in *Proc. IEEE Int. Symp. Circuits Syst.*, Kobe, Japan, May 23–26, 2005, vol. 2, pp. 1814–1817.

[32] J. Detrey and F. de Dinechin, “A VHDL library of LNS operators,” in *Proc. Asilomar Conf. Signals Syst. Comp.*, Nov. 9–12, 2003, vol. 2, pp. 2227–2231.

[33] F. de Dinechin and A. Tisserand, “Multipartite table methods,” *IEEE Trans. Computers*, vol. 54, no. 3, pp. 319–330, Mar. 2005.

[34] M. D. Ercegovac, T. Lang, J.-M. Muller, and A. Tisserand, “Reciprocation, square root, inverse square root, and some elementary functions using small multipliers,” *IEEE Trans. Computers*, vol. 49, no. 7, pp. 628–637, July 2000.

[35] R. Freking and K. K. Parhi, “Theoretical estimation of power consumption in binary adders,” in *Proc. IEEE Int. Symp. Circuits Syst.*, Hong Kong, 1998, vol. 2, pp. 453–457.

[36] O. Gustafsson, H. Ohlsson, and L. Wanhammar, "Minimum-adder integer multipliers using carry-save adders," in *Proc. IEEE Int. Symp. Circuits Syst.*, Sydney, Australia, May 6–9, 2001, vol. 2, pp. 709–712.

[37] O. Gustafsson, A. G. Dempster, and L. Wanhammar, "Extended results for minimum-adder constant integer multipliers," in *Proc. IEEE Int. Symp. Circuits Syst.*, Scottsdale, AZ, May 26–29, 2002, vol. 1, pp. 73–76.

[38] O. Gustafsson and L. Wanhammar, "A novel approach to multiple constant multiplication using minimum spanning trees," in *Proc. IEEE Midwest Symp. Circuits Syst.*, Tulsa, OK, Aug. 4–7, 2002, vol. 3, pp. 652–655.

[39] O. Gustafsson, A. G. Dempster, and L. Wanhammar, "Multiplier blocks using carry-save adders," in *Proc. IEEE Int. Symp. Circuits Syst.*, Vancouver, Canada, May 23–26, 2004, vol. 2, pp. 473–476.

[40] O. Gustafsson, H. Ohlsson, and L. Wanhammar, "Improved multiple constant multiplication using minimum spanning trees," in *Proc. Asilomar Conf. Signals Syst. Comp.*, Pacific Grove, CA, Nov. 7–10, 2004, pp. 63–66.

[41] O. Gustafsson, K. Johansson, and L. Wanhammar, "Optimization and quantization effects for sine and cosine computation using a sum of bit-products," in *Proc. Asilomar Conf. Signals Syst. Comp.*, Pacific Grove, CA, Oct. 30–Nov. 2, 2005, pp. 1347–1351.

[42] O. Gustafsson, A. G. Dempster, K. Johansson, M. D. Macleod, and L. Wanhammar, "Simplified design of constant coefficient multipliers," *Circuits Syst. Signal Processing*, vol. 25, no. 2, pp. 225–251, Apr. 2006.

[43] O. Gustafsson, "A difference based adder graph heuristic for multiple constant multiplication problems," in *Proc. IEEE Int. Symp. Circuits Syst.*, New Orleans, LA, May 27–30, 2006, pp. 1097–1100.

[44] O. Gustafsson and K. Johansson, "Multiplierless piecewise linear approximation of elementary functions," in *Proc. Asilomar Conf. Signals Syst. Comp.*, Pacific Grove, CA, Oct. 29–Nov. 1, 2006, pp. 1678–1681.

[45] O. Gustafsson, "Lower bounds for constant multiplication problems," *IEEE Trans. Circuits Syst. II*, vol. 54, no. 11, pp. 974–978, Nov. 2007.

[46] G. D. Hachtel, E. Macii, A. Pardo, and F. Somenzi, "Markovian analysis of large finite state machines," *IEEE Trans. Computer-Aided Design*, vol. 15, no. 12, pp. 1479–1493, Dec. 1996.

[47] R. I. Hartley and K. K. Parhi, *Digit-Serial Computation*, Kluwer, 1995.

[48] R. I. Hartley, "Subexpression sharing in filters using canonic signed digit multipliers," *IEEE Trans. Circuits Syst. II*, vol. 43, no. 10, pp. 677–688, Oct. 1996.

[49] H. Hassler and N. Takagi, "Function evaluation by table look-up and addition," in *Proc. 12th Symp. Comp. Arithmetic*, July 19–21, 1995, pp. 10–16.

[50] S. He and M. Torkelson, "FPGA implementation of FIR filters using pipelined bit-serial canonical signed digit multipliers," in *Proc. IEEE Custom Integrated Circuits Conf.*, May 1994, pp. 81–84.

[51] Y. H. Hu, "CORDIC-based VLSI architectures for digital signal processing," *IEEE Signal Processing Magazine*, vol. 9, no. 3, pp. 16–35, July 1992.

[52] S.-W. Jeong and F. Somenzi, "A new algorithm for the binate covering problem and its application to the minimization of boolean relations," *IEEE/ACM Int. Conf. Computer-Aided Design*, Nov. 8–12, 1992, pp. 417–420.

[53] Z. Jiang and A. N. Willson Jr., "Efficient digital filtering architectures using pipelining/interleaving," *IEEE Trans. Circuits Syst. II*, vol. 44, no. 2, pp. 110–119, Feb. 1997.

[54] K. Johansson, O. Gustafsson, and L. Wanhammar, "Switching activity in bit-serial constant coefficient serial/parallel multipliers," in *Proc. IEEE NorChip Conf.*, Riga, Latvia, Nov. 10–11, 2003, pp. 260–263.

[55] K. Johansson, O. Gustafsson, and L. Wanhammar, "Power estimation for bit-serial constant coefficient multipliers," in *Proc. Swedish System-on-Chip Conf.*, Båstad, Sweden, Apr. 13–14, 2004.

[56] K. Johansson, O. Gustafsson, A. G. Dempster, and L. Wanhammar, "Algorithm to reduce the number of shifts and additions in multiplier blocks using serial arithmetic," in *Proc. IEEE Mediterranean Electrotechnical Conf.*, Dubrovnik, Croatia, May 12–15, 2004, vol. 1, pp. 197–200.

[57] K. Johansson, O. Gustafsson, and L. Wanhammar, "Low-complexity bit-serial constant-coefficient multipliers," in *Proc. IEEE Int. Symp. Circuits Syst.*, Vancouver, Canada, May 23–26, 2004, vol. 3, pp. 649–652.

[58] K. Johansson, O. Gustafsson, and L. Wanhammar, "Switching activity in bit-serial constant coefficient multipliers," in *Proc. IEEE Int. Symp. Circuits Syst.*, Vancouver, Canada, May 23–26, 2004, vol. 2, pp. 469–472.

[59] K. Johansson, O. Gustafsson, and L. Wanhammar, "Power estimation for ripple-carry adders with correlated input data," in *Proc. Int. Workshop Power Timing Modeling Optimization Simulation*, Santorini, Greece, Sept. 15–17, 2004, pp. 662–674.

[60] K. Johansson, O. Gustafsson, and L. Wanhammar, “Estimation of switching activity for ripple-carry adders adopting the dual bit type method,” in *Proc. Swedish System-on-Chip Conf.*, Tammsvik, Sweden, Apr. 18–19, 2005.

[61] K. Johansson, O. Gustafsson, and L. Wanhammar, “Implementation of low-complexity FIR filters using serial arithmetic,” in *Proc. IEEE Int. Symp. Circuits Syst.*, Kobe, Japan, May 23–26, 2005, vol. 2, pp. 1449–1452.

[62] K. Johansson, O. Gustafsson, A. G. Dempster, and L. Wanhammar, “Trade-offs in low power multiplier blocks using serial arithmetic,” in *Proc. National Conf. Radio Science (RVK)*, Linköping, Sweden, June 14–16, 2005, pp. 271–274.

[63] K. Johansson, O. Gustafsson, and L. Wanhammar, “A detailed complexity model for multiple constant multiplication and an algorithm to minimize the complexity,” in *Proc. European Conf. Circuit Theory Design*, Cork, Ireland, Aug. 28–Sept. 2, 2005, vol. 3, pp. 465–468.

[64] K. Johansson, O. Gustafsson, and L. Wanhammar, “Low power architectures for sine and cosine computation using a sum of bit-products,” in *Proc. IEEE NorChip Conf.*, Oulu, Finland, Nov. 21–22, 2005, pp. 161–164.

[65] K. Johansson, O. Gustafsson, and L. Wanhammar, “Approximation of elementary functions using a weighted sum of bit-products,” in *Proc. IEEE Int. Symp. Circuits Syst.*, Kos Island, Greece, May 21–24, 2006, pp. 795–798.

[66] K. Johansson, O. Gustafsson, and L. Wanhammar, “Trade-offs in multiplier block algorithms for low power digit-serial FIR filters,” in *Proc. WSEAS Int. Conf. Circuits*, Vouliagmeni, Greece, July 10–12, 2006.

[67] K. Johansson, O. Gustafsson, and L. Wanhammar, “Multiple constant multiplication for digit-serial implementation of low power FIR filters,” *WSEAS Trans. Circuits Syst.*, vol. 5, no. 7, pp. 1001–1008, July 2006.

[68] K. Johansson, O. Gustafsson, and L. Wanhammar, “Conversion and addition in logarithmic number systems using a sum of bit-products,” in *Proc. IEEE NorChip Conf.*, Linköping, Sweden, Nov. 20–21, 2006, pp. 39–42.

[69] K. Johansson, O. Gustafsson, and L. Wanhammar, “Bit-level optimization of shift-and-add based FIR filters,” in *Proc. IEEE Int. Conf. Electronics Circuits Syst.*, Marrakech, Morocco, Dec. 11–14, 2007, pp. 713–716.

[70] K. Johansson, O. Gustafsson, and L. Wanhammar, “Switching activity estimation for shift-and-add based constant multipliers,” in *Proc. IEEE Int. Symp. Circuits Syst.*, Seattle, WA, May 18–21, 2008, pp. 676–679.

[71] K. Johansson, O. Gustafsson, and L. Wanhammar, “Implementation of elementary functions for logarithmic number systems,” *IET Computers Digital Techniques (Selected Papers NorChip 2006)*, vol. 2, no. 4, pp. 295–304, July 2008.

[72] M. Kerttu, P. Lindgren, M. Thornton, and R. Drechsler, “Switching activity estimation of finite state machines for low power synthesis,” in *Proc. IEEE Int. Symp. Circuits Syst.*, Scottsdale, AZ, May 26–29, 2002, vol. 4, pp. 65–68.

[73] U. Ko and P. T. Balsara, “High-performance energy-efficient D-flip-flop circuits,” *IEEE Trans. VLSI Syst.*, vol. 8, no. 1, pp. 94–98, Feb. 2000.

[74] I. Koren and O. Zinaty, “Evaluating elementary functions in a numerical coprocessor based on rational approximations,” *IEEE Trans. Computers*, vol. 39, no. 8, pp. 1030–1037, Aug. 1990.

[75] I. Koren, *Computer Arithmetic Algorithms*, A. K. Peters, Natick, MA, 2002.

[76] D. K. Kostopoulos, “An algorithm for the computation of binary logarithms,” *IEEE Trans. Computers*, vol. 40, no. 11, pp. 1267–1270, Nov. 1991.

[77] S. K. Lam, D. K. Chaudhary, and T. Srikanthan, “Low cost logarithmic techniques for high-precision computations,” in *Proc. IEEE Int. Symp. Circuits Syst.*, Bangkok, Thailand, 25–28 May, 2003, vol. 5, pp. 125–128.

[78] P. E. Landman and J. M. Rabaey, “Architectural power analysis: the dual bit type method,” *IEEE Trans. VLSI Syst.*, vol. 3, no. 2, pp. 173–187, June 1995.

[79] P. E. Landman and J. M. Rabaey, “Activity-sensitive architectural power analysis,” *IEEE Trans. Computer-Aided Design*, vol. 15, no. 6, pp. 571–587, June 1996.

[80] J. M. P. Langlois and D. Al-Khalili, “Novel approach to the design of direct digital frequency synthesizers based on linear interpolation,” *IEEE Trans. Circuits Syst. II*, vol. 50, no. 9, pp. 567–578, Sept. 2003.

[81] J. M. P. Langlois and D. Al-Khalili, “Phase to sinusoid amplitude conversion techniques for direct digital frequency synthesis,” *IEE Proc. Circuits Devices Syst.*, vol. 151, no. 6, pp. 519–528, Dec. 2004.

[82] H. Lee and G. E. Sobelman, “FPGA-based FIR filters using digit-serial arithmetic,” in *Proc. IEEE Int. ASIC Conf.*, Sept. 7–10, 1997, pp. 225–228.

[83] V. Lefèvre and J.-M. Muller, “On-the-fly range reduction,” *J. VLSI Signal Processing*, vol. 33, pp. 31–35, Jan. 2003.

[84] J. Leijten, J. van Meerbergen, and J. Jess, "Analysis and reduction of glitches in synchronous networks," in *Proc. European Design Test Conf.*, Paris, France, Mar. 6–9, 1995, pp. 398–403.

[85] D. Li, "Minimum number of adders for implementing a multiplier and its application to the design of multiplierless digital filters," *IEEE Trans. Circuits Syst. II*, vol. 42, pp. 453–460, July 1995.

[86] R.-C. Li, "Near optimality of Chebyshev interpolation for elementary function computations," *IEEE Trans. Computers*, vol. 53, no. 6, pp. 678–687, June 2004.

[87] Y. C. Lim and S. R. Parker, "Discrete coefficient FIR digital filter design based upon an LMS criteria," *IEEE Trans. Circuits Syst.*, vol. 30, no. 10, pp. 723–739, Oct. 1983.

[88] J.-Y. Lin, T.-C. Liu, and W.-Z. Shen, "A cell-based power estimation in CMOS combinational circuits," in *Proc. Int. Conf. Computer Aided Design*, Nov. 1994, pp. 304–309.

[89] M. Lundberg, K. Muhammad, K. Roy, and S. K. Wilson, "A novel approach to high-level swithing activity modeling with applications to low-power DSP system synthesis," *IEEE Trans. Signal Processing*, vol. 49, no. 12, pp. 3157–3167, Dec. 2001.

[90] G. K. Ma and F. J. Taylor, "Multiplier policies for digital signal processing," *IEEE ASSP Mag.*, no. 1, pp. 6–20, Jan. 1990.

[91] E. Macii, M. Pedram, and F. Somenzi, "High-level power modeling, estimation, and optimization," *IEEE Trans. Computer-Aided Design*, vol. 17, no. 11, pp. 1061–1079, Nov. 1998.

[92] A. Madisetti, A. Y. Kwentus, and A. N. Willson Jr., "A 100-MHz, 16-b, direct digital frequency synthesizer with a 100-dBc spurious-free dynamic range," *IEEE J. Solid-State Circuits*, vol. 34, no. 8, pp. 1034–1043, Aug. 1999.

[93] M. Magar and L. S. DeBrunner, "Balancing the tradeoffs between coefficient quantization and internal quantization in FIR digital filters," in *Proc. Asilomar Conf. Signals Syst. Comp.*, Pacific Grove, CA, Nov. 7–10, 2004, vol. 1, pp. 493–497.

[94] D. M. Mandelbaum, "Some results on a SRT type division scheme," *IEEE Trans. Computers*, vol. 42, no. 1, pp. 102–106, Jan. 1993.

[95] M. Martínez-Peiró, E. Boemo, and L. Wanhammar, "Design of high speed multiplierless filters using a nonrecursive signed common subexpression algorithm," *IEEE Trans. Circuits Syst. II*, vol. 49, no. 3, pp. 196–203, Mar. 2002.

[96] D. L. Maskell, "Design of efficient multiplierless FIR filters," *IET Circuits Devices Syst.*, vol. 1, no. 2, pp. 175–180, Apr. 2007.

[97] L. A. Montalvo, K. K. Parhi, and J. H. Satyanarayana, "Estimation of average energy consumption of ripple-carry adder based on average length carry chains," in *Proc. IEEE Workshop VLSI Signal Processing*, 1996, pp. 189–198.

[98] K. Muhammad and K. Roy, "A graph theoretic approach for synthesizing very low-complexity high-speed digital filters," *IEEE Trans. Computer-Aided Design*, vol. 21, no. 2, pp. 204–216, Feb. 2002.

[99] J.-M. Muller, "A few results on table-based methods," *Reliable Computing*, vol. 5, no. 3, pp. 279–288, 1999.

[100] S. Mutoh, T. Douseki, Y. Matsuya, T. Aoki, S. Shigematsu, and J. Yamada, "1-V power supply high-speed digital circuit technology with multithreshold-voltage CMOS," *IEEE J. Solid-State Circuits*, vol. 30, no. 8, pp. 847–854, Aug. 1995.

[101] C. J. Myers, *Asynchronous Circuit Design*, John Wiley & Sons, Inc., 2001.

[102] C. Nagendra, M. J. Irwin, and R. M. Owens, "Area-time-power tradeoffs in parallel adders," *IEEE Trans. Circuits Syst. II*, vol. 43, no. 10, pp. 689–702, Oct. 1996.

[103] F. Najm, "A survey of power estimation techniques in VLSI circuits," *IEEE Trans. VLSI Syst.*, vol. 2, no. 4, pp. 446–455, Dec. 1994.

[104] P. Nilsson, "Arithmetic and architectural design to reduce leakage in nano-scale digital circuits," in *Proc. European Conf. Circuit Theory Design*, Seville, Spain, Aug. 27–30, 2007, pp. 372–375.

[105] P. Nilsson, "Reducing leakage power in fixed coefficient arithmetic," in *Proc. IEEE Int. Conf. Electronics Circuits Syst.*, Marrakech, Morocco, Dec. 11–14, 2007, pp. 306–309.

[106] A. S. Noetzel, "An interpolating memory unit for function evaluation: analysis and design," *IEEE Trans. Computers*, vol. 38, no. 3, pp. 377–384, Mar. 1989.

[107] T. G. Noll, "Carry-save arithmetic for high-speed digital signal processing," in *Proc. IEEE Int. Symp. Circuits Syst.*, May 1–3, 1990, vol. 2, pp. 982–986.

[108] H. Ohlsson, O. Gustafsson, and L. Wanhammar, "Implementation of low complexity FIR filters using a minimum spanning tree," in *Proc. IEEE Mediterranean Electrotechnical Conf.*, Dubrovnik, Croatia, May 12–15, 2004, vol. 1, pp. 261–264.

[109] V. Paliouras, J. Karagiannis, G. Aggouras, and T. Stouraitis, "A very-long instruction word digital signal processor based on the logarithmic number system," in *Proc. IEEE Int. Conf. Electronics Circuits Syst.*, Sept. 7–10, 1998, vol. 3, pp. 59–62.

[110] A. Papoulis, *Probability, Random Variables, and Stochastic Processes*, 3rd ed., McGraw-Hill, 1991.

[111] K. K. Parhi, "A systematic approach for design of digit-serial signal processing architectures," *IEEE Trans. Circuits Syst.*, vol. 38, no. 4, pp. 358–375, Apr. 1991.

[112] K. K. Parhi, C.-Y. Wang, and A. P. Brown, "Synthesis of control circuits in folded pipelined DSP architectures," *IEEE J. Solid-State Circuits*, vol. 27, pp. 29–43, Jan. 1992.

[113] K. K. Parhi, *VLSI Digital Signal Processing Systems: Design and Implementation*, Wiley, 1998.

[114] I.-C. Park and H.-J. Kang, "Digital filter synthesis based on an algorithm to generate all minimal signed digit representations," *IEEE Trans. Computer-Aided Design*, vol. 21, no. 12, pp. 1525–1529, Dec. 2002.

[115] R. Pasko, P. Schaumont, V. Derudder, S. Vernalde, and D. Durackova, "A new algorithm for elimination of common subexpressions," *IEEE Trans. Computer-Aided Design*, vol. 18, no. 1, pp. 58–68, Jan. 1999.

[116] J.-A. Piñeiro, M. D. Ercegovac, and J. D. Bruguera, "Algorithm and architecture for logarithm, exponential, and powering computation," *IEEE Trans. Computers*, vol. 53, no. 9, pp. 1085–1096, Sept. 2004.

[117] J.-A. Piñeiro, S. F. Oberman, J.-M. Muller, and J. D. Bruguera, "High-speed function approximation using a minimax quadratic interpolator," *IEEE Trans. Computers*, vol. 54, no. 3, pp. 304–318, Mar. 2005.

[118] M. Potkonjak, M. B. Srivastava, and A. P. Chandrakasan, "Multiple constant multiplications: efficient and versatile framework and algorithms for exploring common subexpression elimination," *IEEE Trans. Computer-Aided Design*, vol. 15, no. 2, pp. 151–165, Feb. 1996.

[119] J. M. Rabaey, *Digital Integrated Circuits: A Design Perspective*, Prentice Hall, 1996.

[120] S. Ramprasad, N. R. Shanbhag, and I. N. Hajj, "Analytical estimation of transition activity from word-level signal statistics," in *Proc. 34th Des. Automat. Conf.*, June 1997, pp. 582–587.

[121] J. R. Sacha and M. J. Irwin, "The logarithmic number system for strength reduction in adaptive filtering," in *Proc. Int. Symp. Low Power Electronics Design*, Monterey, CA, Aug. 10–12, 1998, pp. 256–261.

[122] T. Sakuta, W. Lee, and P. T. Balsara, "Delay balanced multipliers for low power/low voltage DSP core," in *Proc. IEEE Symp. Low Power Electronics*, San José, CA, Oct. 9–11, 1995, pp. 36–37.

[123] D. Das Sarma and D. W. Matula, “Faithful bipartite ROM reciprocal tables,” in *Proc. 12th Symp. Comp. Arithmetic*, July 19–21, 1995, pp. 17–28.

[124] J. H. Satyanarayana and K. K. Parhi, “Power estimation of digital data paths using HEAT,” *IEEE Design Test Computers*, vol. 17, no. 2, pp. 101–110, Apr. 2000.

[125] C. V. Schimpfle, S. Simon, and J. A. Nossek, “Optimal placement of registers in data paths for low power design,” in *Proc. IEEE Int. Symp. Circuits Syst.*, Hong Kong, June 9–12, 1997, vol. 3, pp. 2160–2163.

[126] M. J. Schulte and E. E. Swartzlander Jr., “Hardware designs for exactly rounded elementary functions,” *IEEE Trans. Computers*, vol. 43, no. 8, pp. 964–973, Aug. 1994.

[127] E. M. Schwarz and M. J. Flynn, “Approximating the sine function with combinational logic,” in *Proc. Asilomar Conf. Signals Syst. Comp.*, Oct. 26–28, 1992, vol. 1, pp. 386–390.

[128] E. M. Schwarz and M. J. Flynn, “Hardware starting approximation method and its application to the square root operation,” *IEEE Trans. Computers*, vol. 45, no. 12, pp. 1356–1369, Dec. 1996.

[129] A. M. Shams, T. K. Darwish, and M. A. Bayoumi, “Performance analysis of low-power 1-bit CMOS full adders cells,” *IEEE Trans. VLSI Syst.*, vol. 10, no. 1, pp. 20–29, Feb. 2002.

[130] S. G. Smith and P. B. Denyer, *Serial-Data Computation*, Kluwer, 1988.

[131] R. Stefanelli, “A suggestion for a high-speed parallel binary divider,” *IEEE Trans. Computers*, vol. 21, no. 1, pp. 42–55, Jan. 1972.

[132] J. E. Stine and M. J. Schulte, “The symmetric table addition method for accurate function approximation,” *J. VLSI Signal Processing*, vol. 21, no. 2, pp. 167–177, June 1999.

[133] D. A. Sunderland, R. A. Strauch, S. S. Wharfield, H. T. Peterson, and C. R. Cole, “CMOS/SOS frequency synthesizer LSI circuit for spread spectrum communications,” *IEEE J. Solid-State Circuits*, vol. 19, no. 4, pp. 497–506, Aug. 1984.

[134] H. Suzuki, Y.-N. Chang, and K. K. Parhi, “Performance tradeoffs in digit-serial DSP systems,” in *Proc. Asilomar Conf. Signals Syst. Comp.*, Nov. 1–4, 1998, vol. 2, pp. 1225–1229.

[135] S. Tahmasbi Oskuii, K. Johansson, O. Gustafsson, and P. G. Kjeldsberg, “Power optimization of weighted bit-product summation tree for elementary function generator,” in *Proc. IEEE Int. Symp. Circuits Syst.*, Seattle, WA, May 18–21, 2008.

[136] N. Takagi, "Powering by a table look-up and a multiplication with operand modification," *IEEE Trans. Computers*, vol. 47, no. 11, pp. 1216–1222, Nov. 1998.

[137] A. Tisserand, "High-performance hardware operators for polynomial evaluation," *Int. J. High Performance Systems Architecture*, vol. 1, no. 1, pp. 14–23, Apr. 2007.

[138] C.-Y. Tsui, J. Monteiro, M. Pedram, S. Devadas, A. Despain, and B. Lin, "Power estimation methods for sequential logic circuits," *IEEE Trans. VLSI Syst.*, vol. 3, no. 3, pp. 404–416, Sept. 1995.

[139] O. Vainio, "Biased logarithmic arithmetic in FIR filters," *Electronics Letters*, vol. 41, no. 10, pp. 580–581, May 2005.

[140] J. Valls, M. M. Peiro, T. Sansaloni, and E. Boemo, "Design and FPGA implementation of digit-serial FIR filters," in *Proc. IEEE Int. Conf. Electronics Circuits Syst.*, Sept. 7–10, 1998, vol. 2, pp. 191–194.

[141] M. Vesterbacka, K. Palmkvist, and L. Wanhammar, "Realization of serial/parallel multipliers with fixed coefficients," in *Proc. National Conf. Radio Science (RVK)*, Lund, Sweden, Apr. 5–7, 1993, pp. 209–212.

[142] M. Vesterbacka, *On Implementation of Maximally Fast Wave Digital Filters*, Diss., no. 487, Linköping University, Sweden, June 1997.

[143] J. E. Volder, "The CORDIC trigonometric computing technique," *IRE Trans. Electronic Computers*, vol. 8, no. 3, pp. 330–334, Sept. 1959.

[144] Y. Voronenko and M. Püschel, "Multiplierless multiple constant multiplication," *ACM Trans. Algorithms*, vol. 3, no. 2, article 11, May 2007.

[145] C. S. Wallace, "A suggestion for a fast multiplier," *IEEE Trans. Electronic Computers*, vol. 13, no. 1, pp. 14–17, Feb. 1964.

[146] J. S. Walther, "A unified algorithm for elementary functions," in *Proc. AFIPS Spring Joint Computer Conf.*, 1971, pp. 379–385.

[147] Y. Wan and C.-L. Wey, "Efficient algorithms for binary logarithmic conversion and addition," in *Proc. IEEE Int. Symp. Circuits Syst.*, Monterey, CA, 31 May–3 June, 1998, vol. 5, pp. 233–236.

[148] C.-C. Wang, Y.-L. Tseng, H.-C. She, C.-C. Li, and R. Hu, "A 13-bit resolution ROM-less direct digital frequency synthesizer based on a trigonometric quadruple angle formula," *IEEE Trans. VLSI Systems*, vol. 12, no. 9, pp. 895–900, Sept. 2004.

[149] L. Wanhammar, *DSP Integrated Circuits*, Academic Press, 1999.

[150] L. Wanhammar and H. Johansson, *Digital Filters*, Linköping University, 2002.

[151] L. Wanhammar, K. Johansson, and O. Gustafsson, “Efficient sine and cosine computation using a weighted sum of bit-products,” in *Proc. European Conf. Circuit Theory Design*, Cork, Ireland, Aug. 28–Sept. 2, 2005, vol. 1, pp. 139–142.

[152] A. Wróblewski, C. V. Schimpfle, and J. A. Nossek, “Automated transistor sizing algorithm for minimizing spurious switching activities in CMOS circuits,” in *Proc. IEEE Int. Symp. Circuits Syst.*, Geneva, Switzerland, May 28–31, 2000, vol. 3, pp. 291–294.

[153] P. Zhao, T. K. Darwish, and M. A. Bayoumi, “High-performance and low-power conditional discharge flip-flop,” *IEEE Trans. VLSI Syst.*, vol. 12, no. 5, pp. 477–484, May 2004.