LOW POWER AND LOW COMPLEXITY
CONSTANT MULTIPLICATION USING
SERIAL ARITHMETIC

Kenny Johansson
Low Power and Low Complexity
Constant Multiplication Using
Serial Arithmetic

Copyright © 2006 Kenny Johansson

Department of Electrical Engineering
Linköpings universitet
SE-581 83 Linköping
Sweden

ISBN 91-85523-76-3        ISSN 0280-7971
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 multiplication using serial arithmetic. The possibility to reduce the complexity and energy consumption is investigated. The main difference between serial and parallel arithmetic, which is of interest here, is that a shift operation in serial arithmetic require a flip-flop, while it can be hardwired in parallel arithmetic.

The possible ways to connect a certain number of adders is limited, i.e., for single-constant multiplication, the number of possible structures is limited for a given number of adders. Furthermore, for each structure there is a limited number of ways to place the shift operations. Hence, it is possible to find the best solution for each constant, in terms of complexity, by an exhaustive search. Methods to bound the search space are discussed. We show that it is possible to save both adders and shifts compared to CSD serial/parallel multipliers. Besides complexity, throughput is also considered by defining structures where the critical path, for bit-serial arithmetic, is no longer than one full adder.

Two algorithms for the design of multiple-constant multiplication using serial arithmetic are proposed. The difference between the proposed design algorithms is the trade-offs between adders and shifts. For both algorithms, the total complexity is decreased compared to an algorithm for parallel arithmetic.

The impact of the digit-size, i.e., the number of bits to be processed in parallel, in FIR filters is studied. Two proposed multiple-constant multiplication algorithms are compared to an algorithm for parallel arithmetic and separate realization of the multipliers. The results provide some guidelines for designing low power multiple-constant multiplication algorithms for FIR filters implemented using digit-serial arithmetic.
A method for computing the number of logic switchings in bit-serial constant multipliers is proposed. The average switching activity in all possible multiplier structures with up to four adders is determined. Hence, it is possible to reduce the switching activity by selecting the best structure for any given constant. In addition, a simplified method for computing the switching activity in constant serial/parallel multipliers is presented. Here it is possible to reduce the energy consumption by selecting the best signed-digit representation of the constant.

Finally, a data dependent switching activity model is proposed for ripple-carry adders. For most applications, the input data is correlated, while previous estimations assumed un-correlated data. Hence, the proposed method may be included in high-level power estimation to obtain more accurate estimates. In addition, the model can be used as cost function in multiple-constant multiplication algorithms. A modified model based on word-level statistics, which is accurate in estimating the switching activity when real world signals are applied, is also presented.
ACKNOWLEDGEMENTS

First, I thank my supervisor, Professor Lars Wanhammar, for giving me the opportunity to do my Ph.D. studies at Electronics Systems, and my co-supervisor, Dr. Oscar Gustafsson, who comes up with most of our research ideas. Our former co-worker, Dr. Henrik Ohlsson, for helping me a lot in the beginning of my period of employment, and for the many interesting, not only work related, discussions.

The rest of the staff at Electronics Systems for all the help and all the fun during coffee breaks.

Dr. Andrew Dempster for the support in the field of multiple-constant multiplication.

As always, the available time was limited, and therefore I am grateful to the proofreaders, Lars and Oscar, who did their job so well and, just as important, so fast. Thank you for all the fruitful comments that strongly improved this thesis.

I also thank the students I have supervised in various courses, particularly the international students. It has been interesting teaching you, and I hope that you have learned as much from me as I have learned from you.

The friends, Joseph Jacobsson, Magnus Karlsson, Johan Hedström, and Ingvar Carlson, who made the undergraduate studying enjoyable.

Teachers through the years. Special thanks to the upper secondary school teachers Jan Alvarsson and Arne Karlsson for being great motivators. Also, the classmates Martin Källström for being quite a competitor in all math related subjects, and Peter Eriksson for all the discussions about everything, but math related subjects during lessons.

My family, parents, Mona and Nils-Gunnar Johansson, sisters, Linda Johansson and Tanja Henze, and, of course also considered as family, all our lovely pets, especially the two wonderful Springer Spaniels, Nikki and Zassa, who are now eating bones in dog heaven.
The list of persons to thank is actually longer, however, the ones not mentioned so far, are here summarized as “friends who have made my life a more enlightening experience,” Thank You!

Finally, I thank the greatest source to happiness and grief, the hockey club, Färjestads BK. Please win the gold this year, but still, to qualify for the final six years in a row is magnificent!

This work was financially supported by the Swedish Research Council (Vetenskapsrådet).
# 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 Serial Arithmetic ....................................................................................... 6

1.4 Constant Multiplication ............................................................................ 7
   1.4.1 Single-Constant Multiplication ............................................................ 7
   1.4.2 Multiple-Constant Multiplication ....................................................... 8
   1.4.3 Graph Representation .......................................................................... 9
   1.4.4 Algorithm Terms ................................................................................ 10

1.5 Power and Energy Consumption ............................................................ 11

1.6 Outline and Main Contributions ............................................................. 12

2 Complexity of Serial Constant Multipliers ............................................. 15

2.1 Graph Multipliers .................................................................................... 15
   2.1.1 Multiplier Types ................................................................................ 17
   2.1.2 Graph Elimination ............................................................................ 18

2.2 Complexity Comparison – Single Multiplier .......................................... 19
   2.2.1 Comparison of Flip-Flop Cost .............................................................. 19
   2.2.2 Comparison of Building Block Cost ................................................... 22

2.3 Complexity Comparison – RSAG-n ....................................................... 24
   2.3.1 The Reduced Shift and Add Graph Algorithm .................................... 26
   2.3.2 Comparison by Varying the Wordlength ........................................... 29
   2.3.3 Comparison by Varying the Setsize .................................................... 32

2.4 Digit-Size Trade-Offs ............................................................................. 34
   2.4.1 Implementation Aspects ...................................................................... 34
   2.4.2 Specification of the Example Filter .................................................... 35
   2.4.3 Delay Elements .................................................................................. 36
2.4.4 Chip Area ................................................................. 37
2.4.5 Sample Rate ............................................................ 38
2.4.6 Energy Consumption ............................................... 39

2.5 Complexity Comparison – RASG-n .................................. 41
2.5.1 The Reduced Add and Shift Graph Algorithm .............. 42
2.5.2 Comparison by Varying the Wordlength ...................... 42
2.5.3 Comparison by Varying the Setsize ............................ 44

2.6 Logic Depth and Energy Consumption ............................ 46
2.6.1 Logic Depth ............................................................. 47
2.6.2 Energy Consumption ................................................ 48

3 Switching Activity in Bit-Serial Multipliers ............................. 55
3.1 Multiplier Stage ............................................................ 56
3.1.1 Preliminaries ............................................................ 56
3.1.2 Sum Output Switching Activity .................................. 57
3.1.3 Switching Activity Using STGs ................................. 58
3.1.4 Carry Output Switching Activity ............................... 62
3.1.5 Input-Output Correlation Probability ......................... 63

3.2 Graph Multipliers ......................................................... 66
3.2.1 Correlation Probability Look-Up Tables ..................... 66
3.2.2 The Applicability of the Equations ............................ 66
3.2.3 Example ................................................................. 67

3.3 Serial/Parallel Multipliers ............................................... 69
3.3.1 Simplification of the Switching Activity Equation .......... 70
3.3.2 Example ................................................................. 71

4 Energy Estimation for Ripple-Carry Adders .......................... 75
4.1 Background .................................................................... 76
4.2 Exact Method for Transitions in RCA ............................. 77
4.3 Energy Model ............................................................... 77
4.3.1 Timing Issues .......................................................... 81

4.4 Switching Activity ........................................................ 82
4.4.1 Switching due to Change of Input .............................. 82
4.4.2 Switching due to Carry Propagation ........................... 85
4.4.3 Total Switching Activity ............................................. 86
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.4.4</td>
<td>Uncorrelated Input Data</td>
<td>89</td>
</tr>
<tr>
<td>4.4.5</td>
<td>Summary</td>
<td>92</td>
</tr>
<tr>
<td>4.5</td>
<td>Experimental Results</td>
<td>92</td>
</tr>
<tr>
<td>4.5.1</td>
<td>Uncorrelated Data</td>
<td>93</td>
</tr>
<tr>
<td>4.5.2</td>
<td>Correlated Data</td>
<td>95</td>
</tr>
<tr>
<td>4.6</td>
<td>Adopting the Dual Bit Type Method</td>
<td>98</td>
</tr>
<tr>
<td>4.7</td>
<td>Statistical Definition of Signals</td>
<td>98</td>
</tr>
<tr>
<td>4.7.1</td>
<td>The Dual Bit Type Method</td>
<td>99</td>
</tr>
<tr>
<td>4.8</td>
<td>DBT Model for Switching Activity in RCA</td>
<td>101</td>
</tr>
<tr>
<td>4.8.1</td>
<td>Simplified Model Assuming High Correlation</td>
<td>103</td>
</tr>
<tr>
<td>4.9</td>
<td>Example</td>
<td>104</td>
</tr>
<tr>
<td>5</td>
<td>Conclusions</td>
<td>109</td>
</tr>
<tr>
<td></td>
<td>References</td>
<td>111</td>
</tr>
</tbody>
</table>
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 low battery weight is desirable. To obtain this the circuit must have low power consumption.

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 will be on single- and multiple-constant multiplication using serial arithmetic. Different design algorithms will be compared, not just to determine which algorithm that seems to be the best one, but also to increase the 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, this circle will not be completely closed within the scope of this thesis.

In this chapter, basics background about the design of digital filters using constant multiplication is presented. The information given here will be assumed familiar in the following chapters. Also, 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 [61], [62]. The filters studied here are assumed to be causal, linear, time-invariant filters.

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

\[ y(n) = \sum_{k=1}^{N} b_k y(n-k) + \sum_{k=0}^{N} a_k x(n-k) \]  

(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

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

(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. For a given specification the filter order, \(N\), is usually much 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 constant group delay.
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 is the same as the coefficients. The transfer function of an \( N \)th-order FIR filter can then be written as

\[
H(z) = \sum_{k=0}^{N} h(k)z^{-k}
\]  

(1.3)

A direct realization of (1.3) for \( N = 5 \) is shown in Fig. 1.1 (a). 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 [61]. The dashed boxes in Figs. 1.1 (a) and (b) mark a sum-of-product block and a multiplier block, 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 \( \omega T \) as [61]

\[
\Phi(\omega T) \propto \frac{N\omega T}{2}
\]  

(1.4)
Furthermore, linear-phase FIR filters have an (anti)symmetric impulse response, i.e.,

\[ h(n) = \begin{cases} h(N - n), & \text{symmetric} \\ -h(N - n), & \text{antisymmetric} \end{cases}, \quad n = 0, 1, \ldots, N \]  

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 \( x_{n-1} x_{n-2} \ldots x_1 x_0 \) where \( x_i \in \{0, 1\} \) correspond to the integer value, \( X \), as

\[ X = \sum_{i=0}^{n-1} x_i 2^i \]  

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, becomes complicated as a sequence of decisions have to be made [32].

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 $x_{n-1} x_{n-2} ... x_1 x_0$ where $x_i \in \{0, 1\}$ correspond to the integer value

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

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 [62]. Here each number has a unique representation, i.e., the CSD representation is nonredundant, where no two consecutive digits are nonzero. Consider, for example, the integer value eleven, which has the binary representation 1011 and the CSD representation 10$\bar{1}$0. Both these representations are also MSD representations, and so is 110$\bar{1}$.

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 was presented in [12].
1.3 Serial Arithmetic

In digit-serial arithmetic, each data word is divided into digits that are processed one digit at a time [18],[55]. The number of bits in each digit is the digit-size, $d$. This provides a trade-off between area, speed, and energy consumption [18],[56]. For the special case where $d$ equals the data word length 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 [47] or by folding bit-parallel processing elements [48]. In Fig. 1.2, a digit-serial adder, subtractor, and shift operation is shown, respectively. These are the operations that are required to implement constant multiplication, which will be discussed in the next section.

It is clear that serial architectures with a small digit-size have the advantage of area efficient processing elements. How speed and energy consumption depend on the digit-size is not as obvious. One main difference compared to parallel arithmetic is that the shift operations can be hardwired, i.e., without any flip-flops, in a bit-parallel architecture. However, the flip-flops included in serial shifts have the benefit to reduce the glitch propagation between subsequent adders/subtractors. To further prevent glitches pipelining can be introduced, which also increases the throughput. Note that fewer registers are required for pipelining in serial arithmetic compared to the parallel case. For example, in bit-serial arith-
metic only one flip-flop is required for each pipelining stage and, in addition, the available shift operations can be used to obtain an improved design, i.e., with a shorter critical path.

1.4 Constant Multiplication

Multiplication with a constant is commonly used in DSP circuits, such as digital filters [60]. It is possible to use shift-and-add operations [38] to efficiently implement this type of multiplication, i.e., shifts, adders and subtractors are used instead of a general multiplier. As the complexity is similar for adders and subtractors both will be referred to as adders, and adder cost will be used to denote the total number of adders/subtractors. A serial shift operation requires one flip-flop, as seen in Fig. 1.2 (c), hence, the number of shifts is referred to as flip-flop cost.

1.4.1 Single-Constant Multiplication

The general design of a multiplier is shown in Fig. 1.3. The input data, \(X\), is multiplied with a specific coefficient, \(a\), and the output, \(Y\), is the result.

\[
X \xrightarrow{\text{Network of shifts and adders}} Y = aX
\]

Figure 1.3 The principle of single-constant multiplication.

A method based on the CSD representation, which was discussed in Section 1.2.2, is widely-used to implement single-constant multipliers [20]. However, multipliers can in many cases be implemented more efficiently using other structures that require fewer operations [24]. Most existing work has focused on minimizing the adder cost [7],[14],[16], while shifts are assumed free as they can be hardwired in the implementation. This is true for bit-parallel arithmetic. However, in serial arithmetic shift operations require flip-flops, and therefore have to be taken into account.

Consider, for example, the coefficient 45, which has the CSD representation \(1010101\). The corresponding realization is shown in Fig. 1.4 (a). Note that a left shift correspond to a multiplication by two. If the realization in Fig. 1.4 (b) is used instead the adder cost is reduced from 3 to 2 and the flip-flop cost is reduced from 6 to 5.
Introduction

A commonly used method to design algorithms for single-constant multiplication is to use subexpression sharing. In the CSD representation of 45 the patterns 101 and 101, 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. 1.4 (c). Another set of subexpressions that can be found in the CSD representation of 45 is 10001 and 10001, which corresponds to $(16 - 1)(4 - 1)$, i.e., the two stages in Fig. 1.4 (c) are performed in reversed order. How to use all SD representations together with subexpression sharing to design single-constant multipliers was presented in [11].

1.4.2 Multiple-Constant Multiplication

In some applications one signal is to be multiplied with several coefficients, as shown in Fig. 1.5. 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 CSD representation. However, multiplier blocks can be effectively implemented using structures that make use of redundant partial results between the coefficients, and thereby reduce the required number of components.
This problem has received considerable attention during the last decade and is referred to as multiple-constant multiplication (MCM). The MCM algorithms can be divided into three groups based on the operation of the algorithm: subexpression sharing [19],[50], graph based [2],[8], and difference methods [15],[41],[45]. Most work has focused on minimizing the number of adders. However, for example, logic depth [9] and power consumption [5],[6] have also been considered. An algorithm that considers the number of shifts may yield digit-serial filter implementations with smaller overall complexity.

By transposing the multiplier block a sum-of-products block is obtained as illustrated by the dashed box in Fig. 1.1 (a), i.e., a multiplier block together with structural adders correspond to a sum-of-products block. Hence, MCM techniques can be applied to both direct and transposed direct form FIR filters. In [10] the design of FIR filters using subexpression sharing and all SD representations was considered.

1.4.3 Graph Representation

The graph representation of constant multipliers was introduced in [2]. As discussed previously a multiplier, i.e., single- or multiple-constant multiplication, is composed of a network of shifts and adders. The network corresponding to a multiplier can be represented using directed acyclic graphs with the following characteristics [7],[14]

- The input is the vertex that has in-degree zero and vertices that have out-degree zero are outputs. However, 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 and a subsequent addition or subtraction.

Figure 1.5  The principle of multiple-constant multiplication.
Introduction

An example of the graph representation is shown in Fig. 1.6, where vertices that correspond to adders are marked. Note that this illustration is simpler than Fig. 1.4 although it contains the same information.

In [14] a vertex reduced representation of graph multipliers was introduced, but since the placement of shift operations is of importance here the original graph representation will be used.

1.4.4 Algorithm Terms

Here terms that will be used in algorithm descriptions are introduced. Consider the graph shown in Fig. 1.6 (a). The fundamental set, $F$, of this graph is

$$F = \begin{bmatrix} 1 & 3 & 11 & 45 \end{bmatrix} \quad (1.8)$$

which are all vertex values. The input vertex value, 1, is always included in the fundamental set. The interconnection table, $G$, of the graph in Fig. 1.6 (a) is

$$G = \begin{bmatrix} 3 & 1 & 1 & -1 & 4 \\ 11 & 1 & 3 & -1 & 4 \\ 45 & 11 & 1 & 4 \end{bmatrix} \quad (1.9)$$

where column 1 is the vertex value, column 2 and 3 are the values of the input vertices, and column 4 and 5 are the values of the input edges. In [6] such an interconnection table, which also includes the logic depth in a sixth column, was referred to as the Dempster format.

From $G$ and $F$ the flip-flop cost, $N_{ff}$, can be computed as
where $M + 1$ is the length of $F$. The vector $e$ contains the largest absolute edge value at the output of each vertex, hence, $e(i)$ is computed for each fundamental in $F$ as

$$e(i) = \max\{1, |G_{4,5}(k)|\}, \begin{cases} k \in \{1, \ldots, 2M\} \\ \forall k |G_{2,3}(k) = F(i) \end{cases}$$

(1.11)

where $G_{i,j}$ is a vector containing the elements in column $i$ and $j$ of $G$. Finally, the flip-flop cost for the graph in Fig. 1.6 (a) is obtained as

$$e = \begin{bmatrix} 4 & 4 & 4 & 1 \end{bmatrix} \Rightarrow N_{ff} = 6$$

(1.12)

1.5 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. However, since the static part mainly depends on the process rather than the design the focus will be on the dynamic part. Furthermore, the power figures of interest is the average power, as opposed to peak power, as this determine the battery life time.

The average dynamic power can be approximated by

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

(1.13)

where $V_{DD}$ is the supply voltage, $f_c$ is the clock frequency, $C_L$ is the load capacitance and $\alpha$ is the switching activity. All these parameters, except $\alpha$, are directly defined by the layout and specification of the circuit.

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

\[ E = \frac{P}{f_c} \]  

(1.14)

However, as the number of clock cycles required to perform one computation varies with the digit-size, we are in this work using energy per computation or energy per sample as comparison measure.

1.6 Outline and Main Contributions

Here the outline of the rest of this thesis is given. In addition, related publications are specified.

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:


**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, for some graphs look-up tables are required. Related publications:


**Chapter 4**

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. Finally, the method is simplified by adopting the dual bit type method [33]. Parts of this work was previously published in:


2

COMPLEXITY OF SERIAL CONSTANT MULTIPLIERS

In this chapter, the possibilities to minimize the complexity of bit-serial single-constant multipliers are investigated [24]. This is done in terms of number of required building blocks, which includes adders and shifts. The multipliers are described using a graph representation. It is shown that it is possible to find a minimum set of graphs required to obtain optimal results.

In the case of single-constant multipliers, the number of possible solutions can be limited because of the limited number of graph topologies. However, if shift-and-add networks containing more coefficients are required different heuristic algorithms can be used to reduce the complexity. Here two algorithms, suitable for bit- and digit-serial arithmetic, for realization of multiple-constant multiplication are presented [23],[29]. It is shown that the new algorithms reduce the total complexity significantly. Comparisons considering area and energy consumption, with respect to the digit-size, are also performed [28].

2.1 Graph Multipliers

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

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 1 to 4 are presented in Fig. 2.1 [7].

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 larger for larger digit-size, hence, such trade-offs are of most interest for a small digit-size.
2.1.1 Multiplier Types

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

- **CSD – Canonic Signed-Digit multiplier**  
  Multiplier based on the CSD representation, as discussed in Section 1.4.1.

- **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 was 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 adders as corresponding CSD/MSD multiplier, and can by that lower 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 adders. This property, which is always obtained for CSD/MSD multipliers, gives high throughput.

**Example**

To describe the difference between the defined multiplier types, different realizations of the coefficient 2813 are shown in Fig. 2.2. Note that there are other possible solutions for all types except the CSD multiplier. 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 can be shared, as for 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) to an extra cost of 0 and 1 register, respectively. Note that the flip-flop cost includes both shifts and pipelining registers, as both corresponds 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 possible to find a minimum set of graphs that is sufficient to always obtain an optimal result. 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 expression

\[ 1 + 2^{b} + 2^{c} + 2^{a+b} + 2^{a+c} \]  \hspace{1cm} (2.1)

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

\[ 1 + 2^{z} + 2^{x+z} + 2^{y+z} + 2^{x+y+z} \]  \hspace{1cm} (2.2)

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 positive, but even if signs are considered, the graphs have the same coefficient set [14].
It is also possible to set up conditions to describe 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 otherwise 2. 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 otherwise 1.

From the coefficient sets and flip-flop cost descriptions, it is possible to eliminate graphs that are not necessary to obtain optimal results. This covering problem [42] has different solutions, and one minimal graph set for each multiplier type is shown 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 result 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

In this section we will compare the complexity of different multiplier types. Due to the fact that adder cost has been discussed before [7],[14], we will focus 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.

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

The results for MAG and CSDAG multipliers are shown in Fig. 2.6 and Fig. 2.5, respectively. In Fig. 2.6, it can be seen that it is possible to save, not only adders [7], but also flip-flops by using the graph multipliers.
Complexity of Serial Constant Multipliers

instead of CSD/MSD multipliers. This is true as long as the multipliers need not to be pipelined. In Fig. 2.5, we do not have the minimum adder cost requirement, but still no more adders than for the corresponding CSD/MSD multiplier is allowed. Since it for all coefficients here is possible to select the same structure as CSD/MSD (this is not completely true as we will see soon) also the pipelined graph multiplier has a lower

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. (Arrows correspond to edges with shifts.)
flip-flop cost. 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. 2.7 it can be seen that the percentage improvement in flip-flop cost for the CSDAG multiplier is almost constant, independent of the number of coefficient bits, and around 9%. For the MAG and PL CSDAG multipliers the savings does not increase as fast as the average flip-flop cost, which result in a decreasing percentage improvement for larger number of coefficient bits. The average flip-flop cost for the PL MAG multiplier is increasing faster than for the CSD multiplier, and for 12 coefficient bits they have approximately the same average flip-flop cost, i.e., the improvement is insignificant.

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. 2.8, the four different graph multiplier types are compared to the CSD multiplier, considering all coefficients with 12 bits. In Fig. 2.8 (a) we can see that one shift is saved for 52% of the coefficients, and that two shifts are saved for 19% of the coefficients.

**Figure 2.5** Average flip-flop cost for CSDAG multipliers compared to CSD/MSD multipliers.
in the CSDAG case. The corresponding histogram for MAG is shown in Fig. 2.8 (b) where the savings in the flip-flop cost is significantly smaller, one shift for 46% and two shifts for 2% of the coefficients, because of the minimum adder cost requirement. If a pipelined multiplier is required the savings becomes smaller, since this is inherent for the CSD multipliers, as shown in Figs. 2.8 (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. 2.8 (c). The explanation to this is that the CSD multipliers for some coefficients have to use more than four adders, which is not allowed for the studied graph multipliers. So in the cases where the PL CSDAG multiplier has a higher flip-flop cost, a lower adder cost is guaranteed.

### 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 minimized or at least not higher than for corresponding CSD multiplier. To get a total complexity measure we have to consider both shifts and adders. The cost difference between
Complexity Comparison – Single Multiplier

adders and shifts in terms of chip area and energy consumption depend on the implementation. From the results in [22] a general rule can be formulated stating that an adder, in terms of energy consumption, is more expensive than a shift, but less expensive than two shifts. Note that this is only valid for the bit-serial case. In the following comparison, we assume an equal cost for adders and shifts. Hence, we study the savings in number of building blocks, which is shown in Fig. 2.9. 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 100101010, for which the MAG realization requires two adders and two shifts less than the CSD realization as shown in Fig. 2.10.

The histograms in Figs. 2.9 (a) and (b) are almost identical. 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. 2.9 (c) and (d), are similar to each other for the same reason.

Figure 2.7 Average improvement in flip-flop cost for graph multipliers over CSD multipliers.
As was shown in Fig. 2.9, the savings in building blocks are similar for MAG and CSDAG multipliers. The difference in adder cost and flip-flop cost is shown in Table 2.1, where 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 requires 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. This shows that a minimum number of adders not necessarily result in an optimal solution.

### 2.3 Complexity Comparison – RSAG-$n$

In this section, an MCM algorithm suitable for serial arithmetic will be presented and compared to a well-known algorithm, referred to as RAG-$n$ [8], in terms of adder and flip-flop costs.
Figure 2.9 Graph multipliers compared to the CSD multiplier in terms of building block cost considering all coefficients with 12 bits.

Figure 2.10 Different realizations of the coefficient 2739. (a) CSD using 18 building blocks and (b) MAG using 14 building blocks.
In [8] 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 number of adders. Based on this algorithm an $n$-dimensional Reduced Shift and Add Graph (RSAG-$n$) algorithm has been developed [23]. 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. The termination condition of the algorithm is that the coefficient set is empty. The steps in the RSAG-$n$ algorithm are as follows:

1. Check the input vector, i.e., the coefficient set. Remove zeros, ones, and repeated coefficients from the coefficient set.

2. For each coefficient, $c$, with adder cost zero, i.e., $c$ is a power-of-two, add $c$ to the fundamental set, add the row $[c \ 0 \ c \ 0]$ to the interconnection table, and remove $c$ from the coefficient set.

3. Compute a sum matrix based on power-of-two multiples of the values in the fundamental set. At start this matrix is

<table>
<thead>
<tr>
<th>MAG vs. CSDAG</th>
<th>PL MAG vs. PL CSDAG</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Flip-flops</strong></td>
<td><strong>Coefficients</strong></td>
</tr>
<tr>
<td>3</td>
<td>0 0</td>
</tr>
<tr>
<td>2</td>
<td>0 55</td>
</tr>
<tr>
<td>1</td>
<td>0 1550</td>
</tr>
<tr>
<td>0</td>
<td>2490 0</td>
</tr>
<tr>
<td><strong>Adders</strong></td>
<td><strong>Coefficients</strong></td>
</tr>
<tr>
<td></td>
<td>0 1</td>
</tr>
<tr>
<td></td>
<td>3 0 41</td>
</tr>
<tr>
<td></td>
<td>2 0 355</td>
</tr>
<tr>
<td></td>
<td>1 0 1001</td>
</tr>
<tr>
<td></td>
<td>0 2698 0</td>
</tr>
<tr>
<td></td>
<td>Adders 0 1</td>
</tr>
</tbody>
</table>

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

**2.3.1 The Reduced Shift and Add Graph Algorithm**

In [8] 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 number of adders. Based on this algorithm an $n$-dimensional Reduced Shift and Add Graph (RSAG-$n$) algorithm has been developed [23]. 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. The termination condition of the algorithm is that the coefficient set is empty. The steps in the RSAG-$n$ algorithm are as follows:

1. Check the input vector, i.e., the coefficient set. Remove zeros, ones, and repeated coefficients from the coefficient set.

2. For each coefficient, $c$, with adder cost zero, i.e., $c$ is a power-of-two, add $c$ to the fundamental set, add the row $[c \ 0 \ c \ 0]$ to the interconnection table, and remove $c$ from the coefficient set.

3. Compute a sum matrix based on power-of-two multiples of the values in the fundamental set. At start this matrix is
and it is extended when new fundamentals are added. The cost zero coefficients found in step 2 can be ignored since they are powers-of-two, and therefore included in the matrix at the start. If any coefficients are found in the matrix, compute the flip-flop cost according to (1.10). Find the coefficients that require the lowest number of additional shifts, and select the smallest of those. Add this coefficient to the fundamental set and the interconnection table, and remove it from the coefficient set.

4. Repeat step 3 until no new coefficient is found in the sum matrix.

5. For each remaining coefficient, check if it can be obtained by the strategies illustrated in Fig. 2.11. For these two cases, two new adders are required. If any coefficients are found, choose the smallest coefficient of those that require the lowest number of additional shifts. Add this coefficient and the extra fundamental to the fundamental set and the interconnection table. Remove the coefficient from the coefficient set.

6. Repeat step 4 and 5 until no new coefficient is found.

7. Choose the smallest coefficient 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. For each set, remove fundamentals 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 those, select the set with smallest sum. Add

Figure 2.11  The coefficient, $c$, 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.)
this set and the coefficient to the fundamental set and the interconnection table. Remove the coefficient from the coefficient set.

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

The basic ideas for the RAG-\( n \) [8] and RSAG-\( n \) algorithms are similar, but the resulting difference is significant. 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 fundamental values 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 RSAG-\( n \) only adds one coefficient at a time to be able to minimize the number of shifts in an effective way, while RAG-\( n \) adds all possible coefficients that can be realized with one additional adder each. The result is that RSAG-\( n \) is slower, due to more iterations to add the same number of coefficients. Another contribution to the run time is the repeated counting of shifts. This is performed according to (1.10) and requires that the interconnection table is computed in parallel, which is not necessary for the RAG-\( n \) algorithm.

It is worth noting that if all coefficients are realized before step 5 of the algorithm, the corresponding implementation is optimal in terms of adder cost [8].

**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) as

\[
C = \begin{bmatrix} 387 & 144 & 64 & 745 & 805 \end{bmatrix}
\]  

(2.4)

The resulting fundamental sets are
\[ F_{\text{RAG-}n} = \begin{bmatrix} 1 & 9 & 23 & 745 & 15 & 805 & 19 & 387 & 64 & 144 \end{bmatrix} \]
\[ F_{\text{RSAG-}n} = \begin{bmatrix} 1 & 64 & 144 & 129 & 387 & 31 & 805 & 29 & 745 \end{bmatrix} \] (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. 2.12 (a) a realization of 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 shown in Fig. 2.12 (b) is obtained. Here, the number of adders is the same, while the number of shifts is reduced to 9. It can be seen in Fig. 2.12 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.

![Realizations of the same coefficient set using different algorithms, (a) RAG-\(n\) and (b) RSAG-\(n\). The largest absolute edge value (except ones) for each vertex is in bold.](image)

**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. 2.13, 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 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. 2.13, on average more than six shifts are saved for every extra adder. Such a trade-off should be advantageous in most implementations.

In Figs. 2.14 and 2.15 histograms of the savings in adder and flip-flop costs using 7 and 10 bits coefficients, respectively, are shown. For 7 bits 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 bits 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 2.14 Frequency of savings in adder and flip-flop costs using RSAG-\(n\) compared with RAG-\(n\). 100 sets of 25 coefficients of wordlength 7 are used.

Figure 2.15 Frequency of savings in adder and flip-flop costs using RSAG-\(n\) compared with RAG-\(n\). 100 sets of 25 coefficients of wordlength 10 are used.
Complexity of Serial Constant Multipliers

2.3.3 Comparison by Varying the Setsize

In Fig. 2.16, 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 optimal adder cost. This is due to the fact that more coefficients give more flexibility in step 3 of the algorithm.

The flip-flop cost for RSAG-$n$ has a maximum for setsize 20. When more fundamentals are available, coefficients are more likely to be obtained without 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 small number of extra adders. For the worst case coefficient setsize, an average of six shifts are saved for each extra adder.

In Figs. 2.17 and 2.18, histograms of the savings in adder and flip-flop costs using sets of 10 and 40 coefficients, respectively, are shown. In

Figure 2.16  Average adder and flip-flop costs for 10 bits coefficients. Sets containing from 5 to 45 coefficients are used.
Figure 2.17 Frequency of savings in adder and flip-flop costs using RSAG-\(n\) compared with RAG-\(n\). 100 sets of 10 coefficients of wordlength 10 are used.

Figure 2.18 Frequency of savings in adder and flip-flop costs using RSAG-\(n\) compared with RAG-\(n\). 100 sets of 40 coefficients of wordlength 10 are used.
Fig. 2.17 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 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 the adder cost is significant smaller for RAG-\( n \) when 10 coefficients are used. The savings in shifts for RSAG-\( n \) compared to RAG-\( n \) are significant 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 [20],[35],[56],[58]. 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.

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. 1.2. Hence, the number of adders will have larger effects on the complexity compared to shifts for increasing digit-size.

In the rest of this section the effects 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-\( n \) [8] and the modified version of this algorithm that drastically reduces the number of shifts, referred to as RSAG-\( n \) [23], 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.

#### 2.4.1 Implementation Aspects

The transposed direct form FIR filter is mapped to a hardware structure using a direct mapping. The wordlength is selected as an integer multiple
of the digit-size. It is possible to use an arbitrary wordlength, but this requires a more complex structure of each processing element [49]. 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 $W_d$ bits and that the maximum number of fractional bits of the coefficients is $W_f$, the total wordlength, $W_T$, is

$$W_T = \left\lfloor \frac{W_d + W_f}{d} \right\rfloor d$$

(2.6)

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

$$W_e = W_T - W_d - W_f$$

(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 $W_T - W_d$ bits.

### 2.4.2 Specification of the Example Filter

A 27th-order lowpass linear-phase FIR filter with passband edge $0.15\pi$ rad and stopband edge $0.4\pi$ rad is used for the comparison. The maximum passband ripple is 0.01, while the stopband attenuation is 80 dB. The 28 tap filter has the symmetric coefficients $\{4, 18, 45, 73, 72, 6, -132, -286, -334, -139, 363, 1092, 1824, 2284\}/8192$. 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, $W_{dp}$, is selected to 16 bits. The number of fractional bits of the coefficients, $W_{pf}$, is 13 bits. Nine different values of the digit-size, $d = \{1, 2, 3, 4, 5, 6, 8, 10, 15\}$, are considered. The total wordlength, $W_{TP}$, is computed for each digit-size from (2.6) as $W_T = \{29, 30, 30, 32, 30, 30, 32, 30, 30\}$.

The multiplier block is designed using the two MCM algorithms RSAG-$n$ [23] and RAG-$n$ [8]. For comparison, an approach using CSD representation serial/parallel multipliers [59],[62] is also considered.
Here, each multiplier is realized independent of other coefficients. The required number of adders, $n_{ADD}$, and shifts, $n_{SH}$, for the different approaches is shown in Table 2.2. As expected the RAG-$n$ algorithm requires the lowest number of adders (which for this coefficient set is optimal), while the RSAG-$n$ algorithm requires the lowest number of shifts. The number of structural adders is 27. 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 Delay Elements

Each delay element is implemented as $W_T$ digit-serial shifts. This implies that a delay element will contain $W_T$ flip-flops and have the structure shown in Fig. 2.20. For larger digit-size, 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.

![Figure 2.19](image-url) (a) Magnitude response for the example filter. (b) Passband.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>$n_{ADD}$</th>
<th>$n_{SH}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>RSAG-$n$</td>
<td>14</td>
<td>19</td>
</tr>
<tr>
<td>RAG-$n$</td>
<td>12</td>
<td>30</td>
</tr>
<tr>
<td>CSD</td>
<td>28</td>
<td>98</td>
</tr>
</tbody>
</table>

Table 2.2 Complexity of the multiplier block for the example filter.
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 as the number of read and written bits then is independent of the digit-size.

### 2.4.4 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) is computed as

\[
\begin{align*}
\text{n}_{FA,MB} &= d\text{n}_{ADD} \\
\text{n}_{FF,MB} &= \text{n}_{ADD} + \text{n}_{SH}
\end{align*}
\]  

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

\[
\begin{align*}
\text{n}_{FA,FIR} &= \text{n}_{FA,MB} + \text{dN} = d(\text{n}_{ADD} + \text{N}) \\
\text{n}_{FF,FIR} &= \text{n}_{FF,MB} + \text{N} + W_T\text{N} = \text{n}_{ADD} + \text{n}_{SH} + \text{N}(1 + W_T)
\end{align*}
\]  

From (2.9) it is clear that the area is more affected by the number of adders for larger digit-size. Note that the complexity of the delay elements and structural adders 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 \( \mu \)m standard cell library. In Fig. 2.21 (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 FIR filter area, as shown in Fig. 2.21 (b). The fact that the FIR filter area is smaller for digit-size five than for digit-size four is explained by the difference in total wordlength, \( W_T \).

### 2.4.5 Sample Rate

In Fig. 2.22 (a) the maximum clock frequency, \( f_{clk} \), obtained by the synthesis tool is shown. The clock frequency decrease for larger digit-size, as the critical path includes more and more full adders. For the CSD algorithm there is at most \( d \) full adders in the critical path of the multiplier block, as no adders are cascaded without at least two shifts between them. This result in a higher maximum clock frequency compared to the other two algorithms. In a similar way, the maximum clock frequency is lower for RSAG-\( n \), as there are more cascaded adders than compared with RAG-\( n \).

The maximum sample frequency, \( f_s \), is shown in Fig. 2.22 (b), and can be computed as

\[
f_s = \frac{f_{clk}}{W_T/d} \tag{2.10}
\]
The sample frequency increases for larger digit-size, since the overhead in delay caused by the flip-flops is relatively smaller for larger digit-size.

2.4.6 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 [31],[63] or any of the different implementation strategies discussed in Section 2.4.3 should be used. Hence, the focus of the energy analysis is 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. 2.23 for different parts of the implementation. The implementations now include a clock tree.

Considering the multiplier block, for increasing digit-size the glitches increase as the carry-propagation paths becomes longer. Furthermore, the leakage increase with the digit-size, as more full adders are used. However, the number of added bits is constant, resulting in small differences in energy consumption for different digit-size, as shown in Fig. 2.23 (a).

For the shifts, the average energy per sample is shown in Fig. 2.23 (b). Note that the adders in the multiplier block consume less energy with RAG-n than RSAG-n, while the situation for the shifts is the opposite.

By adding the energy for the adders and the shifts, the energy for the multiplier block is obtained, which is shown in Fig. 2.23 (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.
Complexity of Serial Constant Multipliers

using the RSAG-$n$ algorithm, while RAG-$n$ is preferable for larger digit-size.

The structural adders, i.e., the adders in the delay section, consume most energy for RSAG-$n$, as shown in Fig. 2.23 (d). This is because the outputs from the multiplier block is more likely to be directly connected from an adder without an intermediate shift, compared to the RAG-$n$ and CSD algorithms. Hence, more glitches are propagated to the structural adders.

Figure 2.23 Energy per sample for (a) adders in multiplier block, (b) shifts in multiplier block, (c) multiplier block, (d) structural adders, and (e) all arithmetic parts.
Combining the energy consumption for the multiplier block and the structural adders results in the energy consumption for all arithmetic parts, this is shown in Fig. 2.23 (e). 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 a lower energy consumption than the one designed by RAG-$n$ for $d = 1$. Hence, it is clear that not only the number of adders is important, but also the glitch propagation.

The savings in energy for RSAG-$n$ and RAG-$n$ compared to CSD are shown in Fig. 2.24. From Fig. 2.24 (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-$n$ is in the range 60 to 64 percent. The energy savings for all arithmetic parts are shown in Fig. 2.24 (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 to minimize the number of adders, while keeping the number of shifts low, would be preferable in most cases.

In this section, a new algorithm that reduces the number of shifts while the number of adders is on average the same as for RAG-$n$ [8] 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-\(n\) [23], respectively. The three algorithms are compared in terms of complexity, including adders and shifts. Average results are shown for 100 random coefficient sets.

2.5.1 The Reduced Add and Shift Graph Algorithm

The new algorithm is a hybrid of the RAG-\(n\) [8] and RSAG-\(n\) [23] algorithms, and is referred to as the \(n\)-dimensional Reduced Add and Shift Graph (RASG-\(n\)) algorithm [29]. As the RSAG-\(n\) algorithm was described in detail in Section 2.3.1, only the main differences will be discussed here.

RASG-\(n\) work with odd integer coefficients, like RAG-\(n\). Hence, even coefficients are divided by two until odd at start. RASG-\(n\) save the number of times each coefficient is divided. These shifts at the outputs can be considered free when other coefficients are realized. RASG-\(n\) only adds one coefficient at a time, like RSAG-\(n\). When it is possible to add 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 effective way, as will be shown in the following sections.

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 coefficient using the RASG-\(n\) algorithm is shown in Fig. 2.25 (a). Coefficients that can be realized with no adders include zeros, power-of-twos, and repeated coefficients. Most coefficients can be realized with only one additional adder. The number of adders is assured to be optimal for all coefficient sets of wordlengths up to 8 bits as shown in Fig. 2.25 (b). Corresponding statistics for the other two algorithms would look similar.

In Fig. 2.26 (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 shown in Fig. 2.26 (b). The
Complexity Comparison – RASG-n

same coefficient sets as in Section 2.3 was used, hence, the lines for RAG-n and RSAG-n are the same as in Fig. 2.13.

The required number of adders using 10 bits coefficients is illustrated by a histogram in Fig. 2.27 (a). RASG-n and RAG-n only have a different number of adders in one out of the 100 cases. As can be seen in Fig. 2.27 (b) RASG-n have on average more than 11 shifts less than RAG-n. RSAG-n has the highest number of adders and the lowest number of shifts.

Figure 2.25 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 optimal.

Figure 2.26 Average number of (a) adders and (b) shifts for sets of 25 coefficients. Wordlengths from 6 to 12 bits are used.
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. 2.28 (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 shown in Fig. 2.28 (b). For a large setsize, most coefficients can be realized with only one additional adder, and the probability that the total number of adders is optimal is high.

In Fig. 2.29 (a), the average number of adders for the three algorithms is shown (the lines for RAG\(_n\) and RSAG\(_n\) are the same as in Fig. 2.16). 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 small 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 larger setsize as shown in Fig. 2.29 (b). RSAG-\(n\) takes full advantage of the fact that coefficients are more likely to be obtained without 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. 2.30 (a), a histogram for the required number of adders using sets of 5 coefficients is shown. RASG-\(n\) and RAG-\(n\) have the same number of adders in 70 out of the 100 cases. For the remaining 30 cases the differences even out, resulting in almost the same number of adders.
Complexity of Serial Constant Multipliers

on average. As can be seen in Fig. 2.30 (b) RASG-$n$ have on average almost 7 shifts less than RAG-$n$.

For sets of 40 coefficients, RASG-$n$ and RAG-$n$ have the same number of adders in all 100 cases as shown in Fig. 2.31 (a). RASG-$n$ has on average almost 18 shifts less than RAG-$n$ as illustrated in Fig. 2.31 (b).

2.6 Logic Depth and Energy Consumption

In [4] and [5] methods to predict the number of transitions in multiplier blocks was introduced. These methods are based on the fact that high logic depth results in more transitions, and consequently higher energy consumption.

In this section, the RAG-$n$ algorithm is compared to the two hybrid algorithms suited for serial arithmetic in terms of logic depth. In addition, the relation between logic depth and energy consumption is discussed.
2.6.1 Logic Depth

The characteristics for varying coefficient wordlength considering average and maximum logic depth are shown in Figs. 2.32 (a) and (b), respectively. The same coefficient sets as in Section 2.5 are used. It is clear that RAG-$n$ has the lowest logic depth among the three algorithms. However, for larger wordlengths the difference becomes smaller. For comparison, the results obtained when all coefficients are implemented separately using the CSD representation and realized as a binary adder tree, which gives a lower bound [16], are also included. 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 logic depth for all algorithms.

The logic depth actually decrease for larger coefficient sets for the RAG-$n$ algorithm, as can be seen in Figs. 2.32 (c) and (d). This is not surprising because, as was mentioned in Section 2.3.3, more coefficients give

Figure 2.31 Frequency of the number of (a) adders and (b) shifts for the three different algorithms using sets of 40 coefficients of wordlength 10.
more flexibility. In the original algorithm, this flexibility is used to obtain low logic depth, while for the serial algorithms it is used to reduce the number of shifts. For large coefficient sets, RAG-$n$ is close to the CSD realizations.

### 2.6.2 Energy Consumption

The energy consumption is studied by the use of an example filter implemented by logic synthesis of VHDL code using a 0.35 μm CMOS standard cell library. The 27th-order FIR defined in Section 2.4.2 is used for the evaluation. Again, the filter is implemented using the transposed direct form structure shown in Fig. 1.1 (b). However, only the multiplier block is considered here.

The required number of adders and shifts for the three different algorithms is shown in Table 2.3 (where only the RASG-$n$ algorithm is added compared to Table 2.2). The number of shifts has been divided into an internal part, corresponding to shifts within the shift-and-add network,
and an output part, corresponding to additional shifts at the outputs of the shift-and-add network. As the even coefficient values are maintained for the RSAG-n algorithm, the number of output shifts is low. The RAG-n and RSAG-n algorithms require 12 adders, which is optimal for this coefficient set.

The power consumption was again obtained using Nanosim™ with 100 random input samples, i.e., the results for the RAG-n and RSAG-n algorithms are the same as was shown in Fig. 2.23. As can be seen in

\[
\begin{array}{|c|c|c|c|c|}
\hline
\text{Algorithm} & \text{Adders} & \text{Shifts} & \text{Total} \\
\hline
\text{RAG-n} & 12 & 20 & 10 & 30 \\
\text{RSAG-n} & 14 & 18 & 1 & 19 \\
\text{RASG-n} & 12 & 14 & 9 & 23 \\
\hline
\end{array}
\]

Table 2.3 Multiplier block complexity for the example filter.

Figure 2.33 Consumed energy per sample for (a) shifts, (b) adders, and (c) the total multiplier block.
Fig. 2.33 (a) the energy per sample for the shifts in the multiplier block is smallest for RSAG-$n$ and largest for RAG-$n$. The energy per sample for the adders in the multiplier block is shown in Fig. 2.33 (b). RAG-$n$ consumes less energy for any digit-size. Finally, by adding the energy for the adders and the shifts, the energy for the multiplier block is obtained, as shown in Fig. 2.33 (c). RSAG-$n$ consumes less energy for digit-size one and two and RAG-$n$ for larger digit-size. Note that the energy consumption corresponding to shifts and adders dominates for small and large values of the digit-size, respectively.

A surprising result is that the energy consumed by the adders is larger for RASG-$n$ than RSAG-$n$, as can be seen in Fig. 2.33 (b), although the number of adders is smaller. The reason for this will be discussed in the following.

In Fig. 2.34, the realizations using the different algorithms are shown. The realizations are illustrated using the graph representation discussed in Section 1.4.3. Note that if all coefficients are divided by two until odd, and the absolute value is used, the set will be \{1, 9, 45, 73, 9, 3, 33, 143, 167, 139, 363, 273, 57, 571\}. The fact that both 3 and 6 are included in the realization shown in Fig. 2.34 (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 required.

From the realizations the number of internal and output shifts, as stated in Table 2.3, can be found. For example, for the RSAG-$n$ realization, shown in Fig. 2.34 (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. 2.34 (b). However, 72 is obtained by shifting of 18 and these shifts are included in the shift-and-add network. Hence, no extra shifts are required at the output of node 18.

The number of cascaded adders gives the logic depth for each coefficient. For example, the longest path from the input to the node corresponding to coefficient 363 only passes three nodes for the realizations in Fig. 2.34 (a) and (b), but as many as six in (c). In Fig. 2.35, the logic depth of the example filter using the three different algorithms is illustrated. It is clear that RASG-$n$ has larger logic depth than RSAG-$n$, which explains the higher energy consumption. RAG-$n$ has the lowest logic depth.
The fact that logic depth is highly correlated with energy consumption is established when the energy consumed in each adder is investigated. This is shown in Figs. 2.36 (a) and (b) for digit-size one and five, respec-

Figure 2.34 Multiplier block realizations using the (a) RAG-$n$, (b) RSAG-$n$, and (c) RASG-$n$ algorithm. Coefficient values are in bold.

The fact that logic depth is highly correlated with energy consumption is established when the energy consumed in each adder is investigated. This is shown in Figs. 2.36 (a) and (b) for digit-size one and five, respec-
Complexity of Serial Constant Multipliers

Note that the RSAG-$n$ implementation includes two extra adders, hence, the total energy is larger than illustrated in Fig. 2.36. However, the energy consumption also depends on other factors. Consider, for example, the coefficients 363 and 2284 (4×571). Although both have logic depth six in the RASG-$n$ realization, the adder that generates...
the output corresponding to the coefficient 363 consumes more than two times more energy for the implementation with digit-size five. The explanation can be found in Fig. 2.34. 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, only logic depth as cost measure is not enough to develop an algorithm optimized for low energy consumption.
Complexity of Serial Constant Multipliers
In this chapter, a method for computing the switching activity in bit-serial, constant multipliers is presented [25].

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

Functions to compute the theoretical 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 obtained directly from the derived equations. The remaining switching activities are obtained by using look-up tables, which also can be generated using the Chapman-Kolmogorov equations. The switching activities are useful to estimate and optimize the energy consumption, and make it possible to select the best energy efficient multiplier structure.

Finally, simplifications of the method for bit-serial constant serial/parallel multipliers are derived [21]. The method here makes it possible to select an energy efficient coefficient representation.
3.1 Multiplier Stage

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

The generic building block that is required for such multipliers is the multiplier stage shown in Fig. 3.1. 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

\[
(b_A, b_B) \in \{(1, 1), (1, -1), (-1, 1)\} \quad (3.1)
\]

If $b_A$ or $b_B$ is –1 a subtraction is performed (given that the corresponding carry register 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, if this integration was done already here there would be three different cases to be studied separately. Therefore the signals $A_0$ and $B_0$ are 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 then the sign variables can be compensated for as 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 full adder cir-
circuit called mirror adder \[51\] the glitching activity was investigated in [22]. However, glitches will not be considered here. Hence, 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 is denoted \( p_{XY} \). The switching activity at a node \( x \) is then defined as

\[
\alpha(x) = 1 - p_{x(n), x(n-1)}
\]  

which means 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 Section 1.5 can be rewritten as

\[
P_{\text{dyn}} = \frac{1}{2} V_{\text{DD}}^2 f_c \sum_{i=1}^{N} C_i \alpha(x_i)
\]  

where \( C_i \) is the load capacitance at node \( x_i \) and \( N \) is the number of outputs of logic cells in the circuit. The switching activity at the output of a flip-flop is the same as at the input. Thus, the only switching activities that has to be computed in a multiplier is the sum and carry outputs of the full adders.

### 3.1.2 Sum Output Switching Activity

A multiplier stage, as shown 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 states. For each of these states we have that

\[
S = A_0 \otimes f(\text{state}) = \begin{cases} 
  f(\text{state}) , & A_0 = 0 \\
  \overline{f(\text{state})} , & A_0 = 1 
\end{cases}
\]  

where a bar is used to denote the inverted value. What this means is, that for each specific state, \( \text{prob}(S = 0) = \text{prob}(S = 1) \) if \( \text{prob}(A_0 = 0) = \text{prob}(A_0 = 1) \).
prob\(A_0 = 1\). For the first stage this relation is trivial as \(A\) 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 have equal probability for ones and zeros. Hence, the relation is also true for the second stage, and so on. The conclusion of this discussion is that

\[
\alpha(S) = \alpha(A_0) = \alpha(A) = \frac{1}{2}
\]  

(3.5)

for all multiplier stages in a bit-serial constant multiplier. Thus, the only switching activities that remain to determine are for the carry outputs of the full adders.

### 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 [36] and sequential [57] 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 [46]

\[
Q\Pi = Q, \quad \sum q_i = 1
\]  

(3.6)

where \(\Pi\) is the transition matrix and \(Q\) is the steady-state probability vector. The transition matrix is defined such that each entry, \(\pi_{ij}\), 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 [57], by iteration [30], from the eigenvalues associated with the transition matrix [53], or with an algorithm based on algebraic decision diagrams (ADD) [17].
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 signal, i.e., a multiplication with three, as shown in Fig. 3.2. Note that for the first stage $A$ is always equal to $B$ and, hence, $p_{A,B} = 1$. There are two delay elements in this circuit, and therefore four possible states. The STG corresponding to the circuit is shown in Fig. 3.3, where also the output signals are indicated ($v_1$ and $v_2$ refers to the values stored in the flip-flops).
The system of equations corresponding to the STG in Fig. 3.3 is

\[
Q\Pi = Q \quad , \quad \Pi = \frac{1}{2} \begin{bmatrix}
1 & 0 & 1 & 0 \\
1 & 0 & 0 & 1 \\
1 & 0 & 0 & 1 \\
0 & 1 & 0 & 1
\end{bmatrix}, \quad \sum_{i=0}^{3} q_i = 1 \quad (3.7)
\]

which have the solution

\[
Q = \begin{bmatrix} q_0 & q_1 & q_2 & q_3 \end{bmatrix} = \begin{bmatrix} \frac{1}{3} & \frac{1}{3} & \frac{1}{6} & \frac{1}{6} \end{bmatrix} \quad (3.8)
\]

Finally, when the steady-state probabilities are known, it is straightforward to compute the carry switching activity from the STG. As marked in Fig. 3.3, 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 as

\[
\alpha(C) = \frac{1}{2} (q_1 + q_2) = \frac{1}{6} \quad (3.9)
\]

**Example 2**

In Fig. 3.4, a multiplier stage with \( d = 1 \) and \((b_A, b_B) = (1, 1)\) is shown. The corresponding STG is shown in Fig. 3.5 (\( v_1 \) and \( v_2 \) refers 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), where the transition matrix corresponds to the STG in Fig. 3.5 as

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

\[
\begin{align*}
q_0 &= q_3 = \frac{p_{A,B}}{2p_{A,B} + 1} \\
q_1 &= q_2 = \frac{1}{2(2p_{A,B} + 1)}
\end{align*}
\]

(3.11)
Finally, the carry switching activity can be derived from Fig. 3.5 as

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

### 3.1.4 Carry Output Switching Activity

The number of shifts, i.e., the number of clock periods, \(d\), that the signal \(B_0\) is delayed, may vary. Furthermore, the correlation probability associated with the stage input signals, \(A_0\) and \(B_0\), can also vary. For each case, it is possible to set up and solve the corresponding system of equations. Some of the results are given in Table 3.1. From this a general expression for the carry switching activity can be derived as

\[
\alpha(C) = \frac{2^d}{4(2^d + 2p_{A,B}B_0 - 1)} \tag{3.13}
\]

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

\[
p_{A_0B_0} = \begin{cases} p_{A,B} , & b_Ab_B = 1 \\ 1 - p_{A,B} , & b_Ab_B = -1 \end{cases} \tag{3.14}
\]

For simplicity a variable, \(\lambda\), that 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 \(\lambda\) is 1 if \(A\) and \(B\) is 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 as

\[
\lambda_0 = 2p_{A_0B_0} - 1 = b_Ab_B\lambda \tag{3.16}
\]
The carry switching activity for a multiplier stage can then be expressed as

\[ \alpha(C) = \frac{2^d}{4(2^d + b_A b_B \lambda)} \]  

(3.17)

This relation is illustrated in Fig. 3.6, for both addition and subtraction operations. Some conclusions that can be drawn from this figure is that \( \alpha(C) \) is 1/4 when \( A \) and \( B \) are uncorrelated and that \( \alpha(C) \) can be far from 1/4 when \( d \) is small.

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

### 3.1.5 Input-Output Correlation Probability

In this section the correlation probability between the inputs, \( A \) and \( B \), and the output, \( S \), of a multiplier stage will be derived. First, consider Exam-
ple 1 above, where \( d = 1 \) and \( \lambda = 1 \). From the STG, shown in Fig. 3.3, and the steady-state probabilities, given in (3.8), the correlation probability is obtained as

\[
p_{X,S} = q_0 + q_3 = \frac{2}{3}
\]  

(3.18)

For the second example, Example 2, with arbitrary input correlation, i.e., \( \lambda = 2p_{A,B} - 1 \), the STG in Fig. 3.5 and the steady-state probabilities in (3.11) gives

\[
\begin{align*}
p_{A,S} &= q_0 + q_3 = \frac{2p_{A,B}}{2p_{A,B} + 1} = \frac{1 + \lambda}{2 + \lambda} \\
p_{B,S} &= p_{A,B}(q_0 + q_3) + (1 - p_{A,B})(q_1 + q_2) = \\
&= \frac{2p_{A,B}^2 - p_{A,B} + 1}{2p_{A,B} + 1} = \frac{2 + \lambda + \lambda^2}{2(2 + \lambda)}
\end{align*}
\]  

(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

\[
\begin{align*}
    p_{A_0, S} &= \frac{2^d + 2\lambda_0}{2(2^d - \lambda_0)} \\
    p_{B_0, S} &= \frac{2^d + \lambda_0 + \lambda_0^2}{2(2^d - \lambda_0)}
\end{align*}
\]  

(3.20)

When a subtraction is performed (3.16) gives that \(\lambda_0 = -\lambda\). Furthermore, if \(b_A = -1\), in addition, \(A\) and \(A_0\) are complementary signals, hence

\[
p_{A, S} = 1 - p_{A_0, S} = \frac{2^d}{2(2^d - \lambda)}
\]

(3.21)

and, for the same reason, if \(b_B = -1\)

\[
p_{B, S} = 1 - p_{B_0, S} = \frac{2^d - \lambda - \lambda^2}{2(2^d - \lambda)}
\]

(3.22)

Merging the different cases gives

\[
\begin{align*}
    p_{A, S} &= \frac{2^d + (1 + b_A)b_B\lambda}{2(2^d + b_A b_B\lambda)} \\
    p_{B, S} &= \frac{2^d + b_A b_B\lambda + b_B\lambda^2}{2(2^d + b_A b_B\lambda)}
\end{align*}
\]

(3.23)

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 (3.23).
3.2 Graph Multipliers

In this section, the possibility to compute the switching activity, using the equations derived in the previous section, is 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 shown in Fig. 3.7. The number of states in an STG including the first two adders in this graph is $2^{d_1+d_2+2}$, where $d_1$ and $d_2$ represent the number of flip-flops in the first two stages. Besides the large number of possible flip-flop combinations, there are $3^2$ different sign combinations of the first two stages that have to be considered. Some of the resulting values, assuming that the first two stages correspond to additions, are shown in Table 3.2. Notice that the table is symmetric and that the values converge towards 1/2.

![Figure 3.7](image)

Figure 3.7 Graph number 4 with 3 adders.

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 3.3 is statistics on the equations applicability with respect to all multiplier stages presented. 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 3 adders are produced, the carry switching activity in
more than 92% of the multiplier stages can be derived. Which multiplier stages that are solvable is shown in Table 3.4 for each graph in Fig. 2.1 (the symbols are explained in Table 3.3).

### 3.2.3 Example

How the switching activity can be significantly reduced by selecting the best graph representation will be shown with an example. Two different graphs, both implementing the coefficient 87, are shown in Fig. 3.8. Note that the graph in Fig. 3.8 (a) corresponds to the commonly used CSD representation. The multiplier in Fig. 3.8 (b) must be pipelined to prevent propagation of glitches, but this will not affect the carry switching activities.

---

<table>
<thead>
<tr>
<th>$d_1$</th>
<th>$d_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0.5556</td>
</tr>
<tr>
<td>2</td>
<td>0.5333</td>
</tr>
<tr>
<td>3</td>
<td>0.5185</td>
</tr>
<tr>
<td>4</td>
<td>0.5098</td>
</tr>
<tr>
<td>5</td>
<td>0.5051</td>
</tr>
</tbody>
</table>

**Table 3.2** Input correlation probability, $p_{A,B}$, for the last stage of the graph shown in Fig. 3.7.

<table>
<thead>
<tr>
<th>Definition of the stage types, and the symbols used in Table 3.4</th>
<th>Adders</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 Same input signals, i.e., $\lambda = 1$</td>
<td>1</td>
<td>3 13  68  85</td>
</tr>
<tr>
<td>$\times$ The derived equations applies</td>
<td>0</td>
<td>1 6  36  43</td>
</tr>
<tr>
<td>$\theta_4$ Same type as graph 4, 3 adders</td>
<td>0</td>
<td>0 1  5  6</td>
</tr>
<tr>
<td>$\theta_7$ Same type as graph 7, 3 adders</td>
<td>0</td>
<td>0 1  4  5</td>
</tr>
<tr>
<td>$\theta$ Unsolvable with equations</td>
<td>0</td>
<td>0 0  12 12</td>
</tr>
<tr>
<td>$\Theta$ Consequence error occur</td>
<td>0</td>
<td>0 0  3  3</td>
</tr>
<tr>
<td>Total number of stages</td>
<td>1</td>
<td>4 21 128 154</td>
</tr>
</tbody>
</table>

**Table 3.3** Statistics on the classification of stages.
Table 3.4  Classification of multiplier stages in all graph topologies with up to four adders. The graphs are shown in Fig. 2.1.
3.3 Serial/Parallel Multipliers

The design of a constant serial/parallel multiplier [59] is based on shifted sums of the input data. Only the structures corresponding to graph multiplier number 1 with different number of adders are used here. Furthermore, the shifts, i.e., the delay elements, are placed so that no adders are directly connected without any intermediate shifts, as shown in Fig. 3.9. 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 the same as the number of nonzero bits in the coefficient minus one. A common method to implement the multipliers is based on the CSD representation, which requires a minimum number of adders. However, other MSD representations may result in more energy efficient implementations.

As indicated in Fig. 3.9 some of the multiplier stages may perform a subtraction instead of an addition. As mentioned in Section 3.1, a subtraction is implemented by adding an inverter at the input of the full adder and initiate the carry flip-flop to one.

All stages of a bit-serial, constant serial/parallel multiplier, except the first stage, have two different input signals, the multiplier input (i.e. the

---

Table 3.5 Switching activities for the graphs in Fig. 3.8.

<table>
<thead>
<tr>
<th></th>
<th>$\alpha(C_1)$</th>
<th>$\alpha(C_2)$</th>
<th>$\alpha(C_3)$</th>
<th>Average</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multiplier (a)</td>
<td>0.3333</td>
<td>0.2727</td>
<td>0.2529</td>
<td>0.2863</td>
</tr>
<tr>
<td>Multiplier (b)</td>
<td>0.1667</td>
<td>0.2526</td>
<td>0.1667</td>
<td>0.1953</td>
</tr>
</tbody>
</table>

---

Figure 3.9 The structure of a bit-serial, constant serial/parallel multiplier.
signal \( X \) in Fig. 3.9) 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. 3.10 two intermediate stages of a bit-serial, constant serial/parallel multiplier is shown. The main difference compared to the general stage shown in Fig. 3.1 is that only one sign variable, \( b_i \), defined so that \( b_i = 1 \) (\( b_i = -1 \)) correspond to an addition (subtraction), is required for each stage \( i \). Because of this, it is easier to deduce a suitable expression for the carry switching activity taking the general equation where the sign variables are excluded as a starting point. Hence, (3.13) is rewritten using (3.16) as

\[
\alpha(C) = \frac{2^d}{4(2^d + \lambda_0)} \quad (3.24)
\]

To simplify the computations it is desirable to eliminate the term \( \lambda_0 \) in (3.24). From Fig. 3.10 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 equal. Hence, the relation for \( \lambda_0 \) given in (3.16) can be rewritten for a specific stage \( i \) as
\[ \lambda_{0,i} = 2p_{A_0\rightarrow B_0,i} - 1 = \begin{cases} 2p_{A_0,i-1,s_{i-1}} - 1, & b_i = b_{i-1} \\ 2(1 - p_{A_0,i-1,s_{i-1}}) - 1, & b_i \neq b_{i-1} \end{cases} = \]
\[ = b_i b_{i-1} (2p_{A_0,i-1,s_{i-1}} - 1) \] (3.25)

Using (3.24) the expression for \( p_{A_0,S} \) given in (3.20) can be rewritten as
\[ p_{A_0,S} = \frac{2^d + 2\lambda_0}{2(2^d + \lambda_0)} = \frac{2(2^d + \lambda_0) - 2^d}{2(2^d + \lambda_0)} = 1 - 2\alpha(C) \] (3.26)

Using (3.26) in (3.25) gives
\[ \lambda_{0,i} = b_i b_{i-1} (1 - 4\alpha(C_{i-1})) \] (3.27)

It is now possible to derive a function for the carry switching activity that only depends on the structure, i.e., the performed operation and the number of shifts, and the carry switching activity in the preceding stage. If conditions so that also the first stage is covered are introduced, 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. 3.9 is obtained. This equation is expressed as
\[ \alpha(C_i) = \frac{2^d}{4(2^d + b_i b_{i-1} (1 - 4\alpha(C_{i-1})))}, \quad \begin{cases} 1 \leq i \leq N \\ b_0 = 1 \\ \alpha(C_0) = 0 \end{cases} \] (3.28)

where \( N \) is the number of stages.

### 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 at least four nonzero digits. Three different MSD representations are
Switching Activity in Bit-Serial Multipliers

177_{10} = 10110001_{MSD1} = 10\overline{0}001_{MSD2} = 110\overline{0}001_{MSD3} \quad (3.29)

where a bar is used to denote negative digits. Note that MSD1 and MSD2 are equal to 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 shown in Fig. 3.11, can be expressed as

$$
c = 2^{d_1} + \cdots + 2^{d_N} + \sum_{i=2}^{N} (b_{i-1} 2^{d_i} + \cdots + d_N) + b_N \quad (3.30)
$$

The variables corresponding to the three coefficient representations are given in Table 3.6. From (3.28) the switching activities are computed, and the result is presented in Table 3.7. The average carry switching activity can be decreased with more than 20% by using binary instead of CSD representation of the coefficient for this example.

<table>
<thead>
<tr>
<th>Representation</th>
<th>$b_1$</th>
<th>$b_2$</th>
<th>$b_3$</th>
<th>$d_1$</th>
<th>$d_2$</th>
<th>$d_3$</th>
</tr>
</thead>
<tbody>
<tr>
<td>MSD1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>MSD2</td>
<td>-1</td>
<td>-1</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>MSD3</td>
<td>1</td>
<td>-1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>4</td>
</tr>
</tbody>
</table>

**Table 3.6** Variables for different coefficient representations.
<table>
<thead>
<tr>
<th>Representation</th>
<th>$\alpha(C_1)$</th>
<th>$\alpha(C_2)$</th>
<th>$\alpha(C_3)$</th>
<th>Average</th>
</tr>
</thead>
<tbody>
<tr>
<td>MSD1</td>
<td>0.2000</td>
<td>0.2273</td>
<td>0.2486</td>
<td>0.2253</td>
</tr>
<tr>
<td>MSD2</td>
<td>0.3333</td>
<td>0.2727</td>
<td>0.2486</td>
<td>0.2849</td>
</tr>
<tr>
<td>MSD3</td>
<td>0.1667</td>
<td>0.2727</td>
<td>0.2486</td>
<td>0.2293</td>
</tr>
</tbody>
</table>

Table 3.7  Carry switching activities for different coefficient representations.
In this chapter, modeling of the energy consumption for ripple-carry adders implemented in CMOS, is considered [26]. Based on the switching activity of each input bit, two switching models, one full and one simplified, are derived. Combined with transition energies obtained by simulation, these switching models can be used to derive the average energy consumed for one operation. This work extends previous results by introducing a data dependent energy model, i.e., correlated input data is considered. Examples show that the model is accurate if all switching in the ripple-carry adder is rail-to-rail (full swing), but in the actual implementation this is not always the case. Hence, our model overestimates the energy consumption.

In [33], 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, for numbers 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 [27].
4.1 Background

Energy modeling and estimation of digital circuits have received considerable interest during the last decade [39],[44]. Especially for DSP systems, energy modeling with correlated data has been considered [34],[37].

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. 4.1. 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 low-power full adder cells have been presented and recent comparisons are found in [1] and [54].

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 [3] and [43]. However, only numerical results are presented for the power consumption.

In [40], 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 [40] was extended in [13], 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 [53]. Again, random inputs were assumed.

In this chapter, an energy model for ripple-carry adders with correlated input data is derived. In real world signals, the switching activity is different for different bit positions [34],[37]. Hence, the proposed model provides a more relevant base for higher-level energy modeling and estimation of energy consumption in ripple-carry adders.

![Figure 4.1](image_url)  
Ripple-carry adder of \( n + 1 \) bits.
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 for the fed back carry is derived.

### 4.2 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 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 [13], while further simplifications will lead to the model in [40]. 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 [13] and [40], where applicable. It is shown that the proposed models gives accurate results in terms of switchings, while the energy consumption is overestimated due to the fact that not all switching is rail-to-rail. However, this is a problem for the methods in [13] and [40] as well.

### 4.3 Energy Model

A ripple-carry adder is composed of a number of cascaded full adders, as shown in Fig. 4.1. When at least one of the three inputs of a full adder changes value, the outputs perform a transition according to Table 4.1. 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 does not switch, and static means that not even the inputs switch.

The state of a full adder, as shown in Fig. 4.2, is here defined as the logic value at its three inputs, in the order $X_i$, $Y_i$, and $C_{i-1}$, which is
Energy Estimation for Ripple-Carry Adders

Table 4.1 Type of transitions at the two outputs of a full adder for different changes of the number of ones at the three inputs.

<table>
<thead>
<tr>
<th>Change of input ones</th>
<th>Transition type</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 ↔ 2, 1 ↔ 3</td>
<td>C</td>
</tr>
<tr>
<td>0 ↔ 1, 2 ↔ 3</td>
<td>S</td>
</tr>
<tr>
<td>0 ↔ 3, 1 ↔ 2</td>
<td>CS</td>
</tr>
<tr>
<td>1 → 1, 2 → 2</td>
<td>none or static</td>
</tr>
<tr>
<td>0 → 0, 3 → 3</td>
<td>static</td>
</tr>
</tbody>
</table>

Table 4.2 Truth table for the full adder in Fig. 4.2.

<table>
<thead>
<tr>
<th>State</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>X_i</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Y_i</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>C_{i-1}</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>C_i</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>S_i</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Figure 4.2 Full adder used in a ripple-carry adder.

described in Table 4.2. In Table 4.3, 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. In the generate phase the data inputs are changed, and in the propagate phase the carry outputs are stabilized in the chain of full adders. This implies that transitions where both the data inputs and the carry input change are not possible. These transitions are in parentheses in Table 4.3. 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 shown in Table 4.4. 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, as shown in Table 4.5, are used.

<table>
<thead>
<tr>
<th>State</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>static</td>
<td>$S$</td>
<td>$S$</td>
<td>(C)</td>
<td>$S$</td>
<td>(C)</td>
<td>$C$</td>
<td>(CS)</td>
</tr>
<tr>
<td>1</td>
<td>$S$</td>
<td>static</td>
<td>(none)</td>
<td>CS</td>
<td>(none)</td>
<td>CS</td>
<td>(CS)</td>
<td>C</td>
</tr>
<tr>
<td>2</td>
<td>$S$</td>
<td>(none)</td>
<td>static</td>
<td>CS</td>
<td>none</td>
<td>(CS)</td>
<td>CS</td>
<td>(C)</td>
</tr>
<tr>
<td>3</td>
<td>(C)</td>
<td>CS</td>
<td>CS</td>
<td>static</td>
<td>(CS)</td>
<td>none</td>
<td>(none)</td>
<td>S</td>
</tr>
<tr>
<td>4</td>
<td>$S$</td>
<td>(none)</td>
<td>none</td>
<td>(CS)</td>
<td>static</td>
<td>CS</td>
<td>CS</td>
<td>(C)</td>
</tr>
<tr>
<td>5</td>
<td>(C)</td>
<td>CS</td>
<td>(CS)</td>
<td>none</td>
<td>CS</td>
<td>static</td>
<td>(none)</td>
<td>S</td>
</tr>
<tr>
<td>6</td>
<td>$C$</td>
<td>(CS)</td>
<td>CS</td>
<td>(none)</td>
<td>CS</td>
<td>(none)</td>
<td>static</td>
<td>S</td>
</tr>
<tr>
<td>7</td>
<td>(CS)</td>
<td>$C$</td>
<td>(C)</td>
<td>$S$</td>
<td>(C)</td>
<td>$S$</td>
<td>$S$</td>
<td>static</td>
</tr>
</tbody>
</table>

Table 4.3 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.
Energy Estimation for Ripple-Carry Adders

The average transition energies are computed as

\[
\begin{align*}
E_{CS} &= \frac{1}{12} \sum_{j,k} E_{j \to k} \quad , \ (j, k) \in CS \text{ and } j + k \neq 7 \\
E_{C} &= \frac{1}{4} \sum_{j,k} E_{j \to k} \quad , \ (j, k) \in \{(0, 6), (1, 7), (6, 0), (7, 1)\} \\
E_{S} &= \frac{1}{12} \sum_{j,k} E_{j \to k} \quad , \ (j, k) \in S \\
E_{\text{none}} &= \frac{1}{4} \sum_{j,k} E_{j \to k} \quad , \ (j, k) \in \{(2, 4), (3, 5), (4, 2), (5, 3)\} \\
E_{\text{static}} &= \frac{1}{8} \sum_{j,k} E_{j \to k} \quad , \ (j, k) \in \text{static}
\end{align*}
\]  

(4.1)

where, for example, the CS set includes all pairs of \((j, k)\) corresponding to a CS transition according to Table 4.3.

<table>
<thead>
<tr>
<th>State</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>(E_{0 \to 0})</td>
<td>(E_{0 \to 1})</td>
<td>(E_{0 \to 2})</td>
<td>(E_{0 \to 3})</td>
<td>(E_{0 \to 4})</td>
<td>(E_{0 \to 5})</td>
<td>(E_{0 \to 6})</td>
<td>(E_{0 \to 7})</td>
</tr>
<tr>
<td>1</td>
<td>(E_{1 \to 0})</td>
<td>(E_{1 \to 1})</td>
<td>(E_{1 \to 2})</td>
<td>(E_{1 \to 3})</td>
<td>(E_{1 \to 4})</td>
<td>(E_{1 \to 5})</td>
<td>(E_{1 \to 6})</td>
<td>(E_{1 \to 7})</td>
</tr>
<tr>
<td>2</td>
<td>(E_{2 \to 0})</td>
<td>(E_{2 \to 1})</td>
<td>(E_{2 \to 2})</td>
<td>(E_{2 \to 3})</td>
<td>(E_{2 \to 4})</td>
<td>(E_{2 \to 5})</td>
<td>(E_{2 \to 6})</td>
<td>(E_{2 \to 7})</td>
</tr>
<tr>
<td>3</td>
<td>(E_{3 \to 0})</td>
<td>(E_{3 \to 1})</td>
<td>(E_{3 \to 2})</td>
<td>(E_{3 \to 3})</td>
<td>(E_{3 \to 4})</td>
<td>(E_{3 \to 5})</td>
<td>(E_{3 \to 6})</td>
<td>(E_{3 \to 7})</td>
</tr>
<tr>
<td>4</td>
<td>(E_{4 \to 0})</td>
<td>(E_{4 \to 1})</td>
<td>(E_{4 \to 2})</td>
<td>(E_{4 \to 3})</td>
<td>(E_{4 \to 4})</td>
<td>(E_{4 \to 5})</td>
<td>(E_{4 \to 6})</td>
<td>(E_{4 \to 7})</td>
</tr>
<tr>
<td>5</td>
<td>(E_{5 \to 0})</td>
<td>(E_{5 \to 1})</td>
<td>(E_{5 \to 2})</td>
<td>(E_{5 \to 3})</td>
<td>(E_{5 \to 4})</td>
<td>(E_{5 \to 5})</td>
<td>(E_{5 \to 6})</td>
<td>(E_{5 \to 7})</td>
</tr>
<tr>
<td>6</td>
<td>(E_{6 \to 0})</td>
<td>(E_{6 \to 1})</td>
<td>(E_{6 \to 2})</td>
<td>(E_{6 \to 3})</td>
<td>(E_{6 \to 4})</td>
<td>(E_{6 \to 5})</td>
<td>(E_{6 \to 6})</td>
<td>(E_{6 \to 7})</td>
</tr>
<tr>
<td>7</td>
<td>(E_{7 \to 0})</td>
<td>(E_{7 \to 1})</td>
<td>(E_{7 \to 2})</td>
<td>(E_{7 \to 3})</td>
<td>(E_{7 \to 4})</td>
<td>(E_{7 \to 5})</td>
<td>(E_{7 \to 6})</td>
<td>(E_{7 \to 7})</td>
</tr>
</tbody>
</table>

Table 4.4  
Energy matrix.

<table>
<thead>
<tr>
<th>Transition</th>
<th>CS</th>
<th>C</th>
<th>S</th>
<th>none</th>
<th>static</th>
</tr>
</thead>
<tbody>
<tr>
<td>Energy</td>
<td>(E_{CS})</td>
<td>(E_{C})</td>
<td>(E_{S})</td>
<td>(E_{\text{none}})</td>
<td>(E_{\text{static}})</td>
</tr>
</tbody>
</table>

Table 4.5  
Average energy table for the different types of transitions.
4.3.1 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. 4.3, where \( S_{\text{del}} \) is the delay corresponding to a sum computation and \( C_{\text{del}} \) is the delay corresponding to a carry computation. ModelSim\textsuperscript{TM} was used as a high-level simulation tool. The switching activity in Fig. 4.3 is quantified because a different value is obtained depending on the number of carry computations that is 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, i.e., \( 0 < S_{\text{del}} / C_{\text{del}} \leq 1 \). This implies that the energy will be overestimated. However, this is still a realistic model as every change at the carry input will in fact affect the sum output, although not always resulting in full swing switching. An example of the difference in high-level simulation between \( S_{\text{del}} / C_{\text{del}} = 1 \) and \( S_{\text{del}} / C_{\text{del}} = 1.5 \) is shown in Fig. 4.4. 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 \( S_{\text{del}} / C_{\text{del}} = 1 \).
4.4 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 contributions from the two phases are then merged. Finally, the result is simplified by assuming random inputs and the results are compared with earlier presented models.

4.4.1 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. 4.2, 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

Figure 4.4  High-level simulation of a six-bit ripple-carry adder. The carry input, $C_{i-1}$, is constant zero, while the $X$ and $Y$ inputs are changed at time zero. (a) $X$ input, (b) $Y$ input, (c) $S$ output with $S_{del} = C_{del} = 2$, (d) $S$ output with $S_{del} = 3$ and $C_{del} = 2$, (e) $C$ output with $S_{del} = C_{del} = 2$, and (f) $C$ output with $S_{del} = 3$ and $C_{del} = 2$.
Switching Activity

Fig. 4.5 using a state transition graph (STG), where the state represents the input values as described in Table 4.2. 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. 4.5 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 4.3. The row index \( j \), column index \( k \), and full adder index \( i \) will be used. The probability function, \( P(x) \), will be used to state the probability that a signal, \( x \), have the logic value one. The function, \( \alpha(x) \), will be used to state the switching activity of the signal \( x \). 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.

\[
P(X_i) = P(Y_i) = \frac{1}{2}
\]  

(4.2)
The initial state (superscript $I$) and final state (superscript $F$) are defined by the input values as

$$
\begin{align*}
S_{i,j}^I &= \left[ X_{i,j}^I, Y_{i,j}^I, C_{i-1,j}^I \right], \quad X_{i,j}, Y_{i,j}, C_{i-1,j} \in \{0, 1\}, \quad 0 \leq j \leq 7 \\
S_{i,k}^F &= \left[ X_{i,k}^F, Y_{i,k}^F, C_{i-1,k}^F \right], \quad X_{i,k}, Y_{i,k}, C_{i-1,k} \in \{0, 1\}, \quad 0 \leq k \leq 7
\end{align*}
\tag{4.3}
$$

By applying the $\text{xor}$ function to the state descriptions, a vector containing ones for changed and zeros for unchanged inputs is obtained as

$$
\alpha_{i,j,k} = S_{i,j}^I \otimes S_{i,k}^F, \quad \left\{ \begin{array}{l} 0 \leq j \leq 7 \\
0 \leq k \leq 7 \end{array} \right. \tag{4.4}
$$

Due to the assumption in (4.2), the carry input, $C_{i-1}$, is the only input that effects the state probabilities. Hence, there are only two different state probabilities, which are defined as

$$
\begin{align*}
P(S_i^0) &= \frac{1}{4}(1 - P(C_{i-1})) \\
P(S_i^1) &= \frac{1}{4}P(C_{i-1})
\end{align*}
\tag{4.5}
$$

where the superscript indicates the carry input value. The carry input probability is

$$
P(C_{i-1}) = \begin{cases} 
P(C_{in}), & i = 0 \\
\frac{1}{4} + \frac{1}{2}P(C_{i-2}), & i > 0 \end{cases}
\tag{4.6}
$$

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 (4.5), with the input switching activities, $\alpha(X_i)$ and $\alpha(Y_i)$. This is done with respect to the switching vector defined in (4.4), 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 also has to be considered. Hence, each matrix element can be computed as

$$
q_{i,j,k}^{\text{gen}} = (C_{i-1,j}^{\text{gen}} P(S_i^1) + C_{i-1,j}^{\text{gen}} P(S_i^0)) \cdot \left( \begin{array}{c} \alpha_{i,j,k}(1) \alpha(X_i) + \alpha_{i,j,k}(1)(1 - \alpha(X_i)) \\ \alpha_{i,j,k}(2) \alpha(Y_i) + \alpha_{i,j,k}(2)(1 - \alpha(Y_i)) \\ \alpha_{i,j,k}(3) \alpha(C_{i-1}^{\text{gen}}) + \alpha_{i,j,k}(3)(1 - \alpha(C_{i-1}^{\text{gen}})) \end{array} \right) \cdot \left( \begin{array}{c} 0 \\ 0 \\ 0 \end{array} \right),
$$

where

$$
\alpha(C_{i-1}^{\text{gen}}) = \begin{cases} \alpha(C_{in}), & i = 0 \\ 0, & i > 0 \end{cases}
$$

since the carry input only can change for the first full adder (LSB) in the generate phase.

### 4.4.2 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. 4.6 the STG corresponding to the propagate phase is shown. Note that all cases without changes of the outputs are static.

![Figure 4.6 State transition graph for a full adder when the carry input is active.](image)

(a) $X_i = Y_i = 0$, (b) $X_i = 0, Y_i = 1$, (c) $X_i = 1, Y_i = 0$, and (d) $X_i = Y_i = 1$.  

85
Transition Probabilities in the Propagation Phase

In the following the propagate part of the probability matrix will be derived. As there are no transition at the inputs of the first full adder (LSB) in the propagate phase, the full adder index, \( i \), is larger than zero in this section.

Let the variable \( \beta \) represent a weighted value, divided into equal parts for the eight different states, of the total carry switching activity at the carry input, i.e., both \( C \) and \( CS \) transitions are included. Using the probability matrix of the previous full adder, \( \beta \) is computed as

\[
\beta_i = \frac{1}{8} \sum_{j,k} q_{i-1,j,k} \quad (j,k) \in C \cup CS
\]  

(4.8)

Since only the carry input can change in the generation phase, transitions are limited within 2 x 2 parts of the probability matrix. These transitions are described by

\[
\begin{bmatrix}
q^\text{pro}_{i,j,k} & q^\text{pro}_{i,j,k+1} \\
q^\text{pro}_{i+1,j,k} & q^\text{pro}_{i+1,j,k+1}
\end{bmatrix} = \begin{bmatrix}
iP(S^0_i) - \beta_j & \beta_j \\
\beta_j & iP(S^1_i) - \beta_j
\end{bmatrix},
\]  

(4.9)

\( (j,k) \in \{(0,0), (2,2), (4,4), (6,6)\} \)

For all matrix elements not covered by (4.9) we have

\[
q^\text{pro}_{i,j,k} = 0
\]  

(4.10)

4.4.3 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

\[
q_{i,j,k} = q^\text{gen}_{i,j,k} + q^\text{pro}_{i,j,k} \quad \begin{cases} 
0 \leq j \leq 7 \\
0 \leq k \leq 7
\end{cases}
\]  

(4.11)

The resulting probability matrix, for any of the full adders (except for the first one), is shown in Table 4.6. It can be shown that a static transition
Switching Activity

is negligible compared to the other types of transitions. Static transitions are therefore excluded in Table 4.6 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 shown in Table 4.4. This method will be referred to as the matrix model. As this is a comparatively complex 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 shown in Table 4.5. The different types of switching activities are named, $\alpha(\text{transition})$, where transition correspond to the symbols used in Table 4.3. By summing all $C$ transitions (see Table 4.3) in Table 4.6, the switching activity for this type is obtained as

$$\alpha(C_i) = 2(P(S_i^0) + P(S_i^1)) \alpha(X_i) \alpha(Y_i) = \frac{1}{2} \alpha(X_i) \alpha(Y_i), \ i > 0 \ (4.12)$$

<table>
<thead>
<tr>
<th>$\beta$</th>
<th>static</th>
<th>$0$</th>
<th>$P(S^0 \mu(X) \ (1-\alpha(Y)))$</th>
<th>$0$</th>
<th>$P(S^0 \mu(X) \ (1-\alpha(Y)))$</th>
<th>$0$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P(S^0 \mu(Y) \ (1-\alpha(X)))$</td>
<td>$0$</td>
<td>static</td>
<td>$\beta$</td>
<td>$P(S^0 \mu(X) \ (1-\alpha(Y)))$</td>
<td>$0$</td>
<td>$P(S^0 \mu(X) \ (1-\alpha(Y)))$</td>
</tr>
<tr>
<td>$0$</td>
<td>$P(S^0 \mu(Y) \ (1-\alpha(X)))$</td>
<td>$\beta$</td>
<td>static</td>
<td>$0$</td>
<td>$P(S^0 \mu(X) \ (1-\alpha(Y)))$</td>
<td>$0$</td>
</tr>
<tr>
<td>$0$</td>
<td>$P(S^0 \mu(Y) \ (1-\alpha(X)))$</td>
<td>$0$</td>
<td>$P(S^0 \mu(X) \ (1-\alpha(Y)))$</td>
<td>$\beta$</td>
<td>static</td>
<td>$0$</td>
</tr>
<tr>
<td>$0$</td>
<td>$P(S^0 \mu(Y) \ (1-\alpha(X)))$</td>
<td>$0$</td>
<td>$P(S^0 \mu(X) \ (1-\alpha(Y)))$</td>
<td>$0$</td>
<td>static</td>
<td>$\beta$</td>
</tr>
</tbody>
</table>

Table 4.6 Probability matrix for $i > 0$. The full adder index, $i$, is not included in this table.
for each full adder. The switching activity corresponding to \( S \) transitions is computed as

\[
\alpha(S_i) = 4\beta_i + 2(P(S_i^0) + P(S_i^1))(\alpha(X_i)(1 - \alpha(Y_i)) + \\
\alpha(Y_i)(1 - \alpha(X_i))) = 4\beta_i + \frac{1}{2}(\alpha(X_i) + \alpha(Y_i)) - 2\alpha(C_i), \quad i > 0
\]  

(4.13)

Finally, the switching activities corresponding to the two remaining types of transitions are obtained directly from (4.12) and (4.13) as

\[
\begin{align*}
\alpha(\text{none}_i) &= \alpha(C_i) \\
\alpha(CS_i) &= \alpha(S_i)
\end{align*}
\]

(4.14)

In (4.13) the variable \( \beta_i \) is required. To compute this value according to (4.8) it is necessary to produce the probability matrix. However, the expression in (4.8) can now be simplified to

\[
\beta_i = \frac{1}{8}(\alpha(C_{i-1}) + \alpha(CS_{i-1})), \quad i > 0
\]

(4.15)

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 (4.7), as there are no transitions in the propagation phase. From the probability matrix, the different types of transition activities are obtained as

\[
\begin{align*}
\alpha(C_0) &= \frac{1}{2}(\alpha(X_0)\alpha(Y_0) + (\alpha(X_0) + \alpha(Y_0) - 3\alpha(X_0)\alpha(Y_0))\alpha(C_{in})) \\
\alpha(\text{none}_0) &= \alpha(C_0) \\
\alpha(S_0) &= \frac{1}{2}(\alpha(X_0) + \alpha(Y_0) + (1 - 3\alpha(X_0)\alpha(Y_0))\alpha(C_{in})) - 2\alpha(C_0) \\
\alpha(CS_0) &= \alpha(S_0) + \alpha(X_0)\alpha(Y_0)\alpha(C_{in})
\end{align*}
\]

(4.16)

Note that the switching activities in (4.16) are equal to the corresponding expressions in (4.12), (4.13), and (4.14) if both \( \alpha(C_{in}) \) and \( \beta_i \) are zero.
This can be used to also include the first full adder in a general equation as

\[
\begin{align*}
\alpha(C_i) &= \frac{1}{2} (\alpha(X_i)\alpha(Y_i) + (\alpha(X_i) + \alpha(Y_i) - 3\alpha(X_i)\alpha(Y_i))\alpha(C_{in})) \\
\alpha(S_i) &= 4\beta_i + \frac{1}{2} (\alpha(X_i) + \alpha(Y_i) + (1 - 3\alpha(X_i)\alpha(Y_i))\alpha(C_{in})) - 2\alpha(C_i) \\
\alpha(CS_i) &= \alpha(S_i) + \alpha(X_i)\alpha(Y_i)\alpha(C_{in}) \\
\alpha(\text{none}_i) &= \alpha(C_i)
\end{align*}
\] (4.17)

where \( \beta_i = \begin{cases} 
0 & , \ i = 0 \\
\frac{1}{8} (\alpha(C_{i-1}) + \alpha(CS_{i-1})) & , \ i > 0
\end{cases} \)

\( \alpha(C_{in}) = 0, \ i > 0 \)

**4.4.4 Uncorrelated Input Data**

If uncorrelated input data, \( X_i \) and \( Y_i \), are assumed, i.e.,

\[
\alpha(X_i) = \alpha(Y_i) = \frac{1}{2}
\] (4.18)

the transition activities can be simplified. Under this assumption the transition activities in the first full adder can be derived from (4.16) as

\[
\begin{align*}
\alpha(C_0) &= \frac{1}{8} (1 + \alpha(C_{in})) \\
\alpha(S_0) &= \frac{1}{8} (2 - \alpha(C_{in})) \\
\alpha(\text{none}_0) &= \alpha(C_0) \\
\alpha(CS_0) &= \frac{1}{8} (2 + \alpha(C_{in}))
\end{align*}
\] (4.19)

From (4.12) and (4.13) the switching activities corresponding to \( C \) and \( S \) transitions for the remaining full adders are computed as

\[
\begin{align*}
\alpha(C_i) &= \frac{1}{8} \ , \ \alpha(S_i) = 4\beta_i + \frac{1}{4} \ , \ i > 0
\end{align*}
\] (4.20)
By summing the contribution from all full adders, a closed-form expression of the total switching activity corresponding to $C$ transitions, $\alpha(C_{\text{tot}})$, is obtained as

$$\alpha(C_{\text{tot}}) = \alpha(C_0) + \sum_{i=1}^{n} \alpha(C_i) = \frac{1}{8} (1 + n + \alpha(C_{\text{in}})) \quad (4.21)$$

where $n + 1$ is the number of full adders.

In (4.20) the variable $\beta_i$ is required. This value can be computed according to (4.15) as

$$\begin{align*}
\beta_1 &= \frac{1}{8} (\alpha(C_0) + \alpha(CS_0)) = \frac{1}{64} (3 + 2\alpha(C_{\text{in}})) \\
\beta_i &= \frac{1}{8} \left(\frac{1}{8} + 4\beta_{i-1} + \frac{1}{4}\right) = \frac{3}{64} + \frac{\beta_{i-1}}{2}, \quad i > 1
\end{align*} \quad (4.22)$$

The total switching activity corresponding to $S$ transitions, $\alpha(S_{\text{tot}})$, is obtained by summing the contribution from all full adders as

$$\begin{align*}
\alpha(S_{\text{tot}}) &= \alpha(S_0) + \sum_{i=1}^{n} \alpha(S_i) = \alpha(S_0) + \frac{n}{4} + 4 \sum_{i=1}^{n} \beta_i = \\
&= \alpha(S_0) + \frac{n}{4} + 4 \left(\frac{3n}{32} + 2 \left(\beta_1 - \frac{3}{32}\right) (1 - 2^{-n})\right) \\
&= \frac{1}{8} (2 + 5n - \alpha(C_{\text{in}}) + (2\alpha(C_{\text{in}}) - 3)(1 - 2^{-n})) \quad (4.23)
\end{align*}$$

Finally, the switching activities corresponding to the two remaining types of transitions are obtained as

$$\begin{align*}
\alpha(\text{none}_{\text{tot}}) &= \alpha(C_{\text{tot}}) \\
\alpha(CS_{\text{tot}}) &= \alpha(S_{\text{tot}}) \pm \frac{1}{4} \alpha(C_{\text{in}}) \quad (4.24)
\end{align*}$$
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.,

$$\alpha(C_{in}) = \frac{1}{2} \quad (4.25)$$

the total switching activities in (4.21), (4.23), and (4.24) are simplified to

$$\begin{align*}
\alpha(C_{tot}) &= \frac{1}{16}(3 + 2n) \\
\alpha(S_{tot}) &= \frac{1}{8}(5n - \frac{1}{2} + 2^{1-n}) \\
\alpha(CS_{tot}) &= \alpha(S_{tot}) + \frac{1}{8} \quad (4.26)
\end{align*}$$

In the model presented in [13], an equation comparable to (4.26) was given. If, for example, $C_{in}$ 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 (4.21), (4.23), and (4.24), but not by (4.26). Hence, the authors of [13] do not consider the carry input, $C_{in}$, but assumes it to be random as well.

For uncorrelated input data it can be shown that

$$\lim_{i \to \infty} \beta_i = \frac{3}{32} \quad \Rightarrow \quad \lim_{i \to \infty} \alpha(S_i) = \lim_{i \to \infty} \alpha(CS_i) = \frac{5}{8} \quad (4.27)$$

The switching activity corresponding to the two remaining types of transitions can be obtained directly from (4.14) and (4.20) as

$$\lim_{i \to \infty} \alpha(C_i) = \lim_{i \to \infty} \alpha(\text{none}_i) = \frac{1}{8} \quad (4.28)$$

The values in (4.27) and (4.28) was used in the model presented in [40], where the energy estimation consequently was stated as a function of the wordlength without considering each full adder individually.
4.4.5 Summary

The differences between the discussed models are summarized in Table 4.7, where the model name TMMM stands for Transition Model 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 switching activities as

$$
\begin{align*}
\alpha_{C_i} &= \alpha(C_i) + \alpha(CS_i) \\
\alpha_{S_i} &= \alpha(S_i) + \alpha(CS_i)
\end{align*}
$$

\hspace{1cm} (4.29)

### Table 4.7 Comparison of different energy models for ripple-carry adders.

<table>
<thead>
<tr>
<th>Model</th>
<th>Equations</th>
<th>Energy</th>
<th>Inputs</th>
<th>Carry</th>
</tr>
</thead>
<tbody>
<tr>
<td>TMMM Random</td>
<td>(4.21), (4.23), (4.24)</td>
<td>Table 4.5</td>
<td>Random</td>
<td>Corr.</td>
</tr>
<tr>
<td>Carry chain II [13]</td>
<td>(4.26)</td>
<td>Table 4.5</td>
<td>Random</td>
<td>Random</td>
</tr>
<tr>
<td>Carry chain [40]</td>
<td>(4.27), (4.28)</td>
<td>Table 4.5</td>
<td>Random</td>
<td>–</td>
</tr>
</tbody>
</table>

4.5 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 [51]. The delay relation, which was discussed in Section 4.3.1, for this full adder was $1 < S_{del}/C_{del} < 2$. Hence, not all switching 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 4.8, while the average energy for different change of the outputs is shown in Table 4.9. In this characterization the sum output was loaded with an inverter, while the carry output was loaded with the carry input of...
Experimental Results

4.5.1 Uncorrelated Data

The first validation is for uncorrelated input data. Here, we consider both constant and random $C_{in}$. The results are shown in Table 4.10, where it is also compared with the results obtained from [13] and [40].

It is worth noting that for random $C_{in}$ the method in [13] 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 [13] the results were compared to those obtained by HEAT [53], and a close correspondence was shown. However, HEAT is based on switching activities as well. Hence, all switching are assumed full swing in HEAT.

### Table 4.8

<table>
<thead>
<tr>
<th>State</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0.624</td>
<td>168.8</td>
<td>179.7</td>
<td>391.4</td>
<td>211.2</td>
<td>512.6</td>
<td>533.8</td>
<td>324.6</td>
</tr>
<tr>
<td>1</td>
<td>316.6</td>
<td>0.477</td>
<td>87.32</td>
<td>401.4</td>
<td>106.0</td>
<td>417.5</td>
<td>512.9</td>
<td>146.9</td>
</tr>
<tr>
<td>2</td>
<td>368.5</td>
<td>102.7</td>
<td>0.923</td>
<td>458.6</td>
<td>92.03</td>
<td>478.8</td>
<td>441.0</td>
<td>195.4</td>
</tr>
<tr>
<td>3</td>
<td>426.9</td>
<td>579.3</td>
<td>555.8</td>
<td>0.713</td>
<td>559.1</td>
<td>103.9</td>
<td>123.8</td>
<td>207.6</td>
</tr>
<tr>
<td>4</td>
<td>385.0</td>
<td>123.3</td>
<td>66.09</td>
<td>487.5</td>
<td>1.952</td>
<td>476.9</td>
<td>482.6</td>
<td>239.0</td>
</tr>
<tr>
<td>5</td>
<td>439.4</td>
<td>618.8</td>
<td>580.6</td>
<td>78.56</td>
<td>623.1</td>
<td>0.967</td>
<td>163.6</td>
<td>237.9</td>
</tr>
<tr>
<td>6</td>
<td>467.4</td>
<td>637.8</td>
<td>634.5</td>
<td>132.6</td>
<td>646.1</td>
<td>122.2</td>
<td>1.008</td>
<td>283.7</td>
</tr>
<tr>
<td>7</td>
<td>722.8</td>
<td>605.4</td>
<td>632.2</td>
<td>381.5</td>
<td>519.4</td>
<td>350.6</td>
<td>329.4</td>
<td>2.948</td>
</tr>
</tbody>
</table>

### Table 4.9

<table>
<thead>
<tr>
<th>Transition</th>
<th>$CS$</th>
<th>$C$</th>
<th>$S$</th>
<th>none</th>
<th>static</th>
</tr>
</thead>
<tbody>
<tr>
<td>Energy [fJ]</td>
<td>528.0</td>
<td>438.4</td>
<td>285.0</td>
<td>85.14</td>
<td>1.202</td>
</tr>
</tbody>
</table>

an identical full adder. All simulation results are average values over 1000 pairs of input samples at a data rate of 50 MHz.

4.5.1 Uncorrelated Data
When each individual full adder is considered, the results in Table 4.11 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 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 switching in this full adder is rail-to-rail.

<table>
<thead>
<tr>
<th>Bits $n+1$</th>
<th>Constant $C_{in} = 0$</th>
<th>Random $C_{in}$</th>
<th>Any $C_{in}$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Sim.</td>
<td>Matrix</td>
<td>TMMM</td>
</tr>
<tr>
<td>4</td>
<td>73.7</td>
<td>87.1</td>
<td>86.1</td>
</tr>
<tr>
<td>8</td>
<td>168.2</td>
<td>199.6</td>
<td>199.1</td>
</tr>
<tr>
<td>12</td>
<td>258.8</td>
<td>313.7</td>
<td>313.7</td>
</tr>
<tr>
<td>16</td>
<td>347.3</td>
<td>428.0</td>
<td>428.4</td>
</tr>
<tr>
<td>20</td>
<td>440.1</td>
<td>542.2</td>
<td>543.1</td>
</tr>
<tr>
<td>24</td>
<td>527.2</td>
<td>656.4</td>
<td>657.8</td>
</tr>
<tr>
<td>32</td>
<td>710.1</td>
<td>884.9</td>
<td>887.2</td>
</tr>
</tbody>
</table>

Table 4.10 Average power consumption, in $\mu$W, for a ripple-carry adder. Simulated and derived by the proposed methods, as well as the methods from [13] and [40].

<table>
<thead>
<tr>
<th>FA no.</th>
<th>Sim.</th>
<th>Matrix</th>
<th>TMMM</th>
<th>Carry chain II [13]</th>
<th>Carry chain [40]</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>15.94</td>
<td>15.99</td>
<td>15.83</td>
<td>15.83</td>
<td>28.68</td>
</tr>
<tr>
<td>2</td>
<td>19.03</td>
<td>23.54</td>
<td>23.60</td>
<td>23.60</td>
<td>28.68</td>
</tr>
<tr>
<td>3</td>
<td>21.05</td>
<td>26.05</td>
<td>26.14</td>
<td>26.14</td>
<td>28.68</td>
</tr>
<tr>
<td>4</td>
<td>20.62</td>
<td>27.30</td>
<td>27.41</td>
<td>27.41</td>
<td>28.68</td>
</tr>
</tbody>
</table>

Table 4.11 Average power consumption, in $\mu$W, for individual full adders in a four-bit ripple-carry adder with random carry input, $C_{in}$.

When each individual full adder is considered, the results in Table 4.11 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 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 switching in this full adder is rail-to-rail.
### 4.5.2 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. 4.7 (a). In Figs. 4.7 (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 switchings. Hence, the simulation with $S_{del}/C_{del} = 1.0$ shows the total number of switchings, while the simulation with $S_{del}/C_{del} = 1.5$ shows the number of full swing switchings. As was stated in Section 4.3.1, the delay relation is shown not to affect the switching activities at the carry outputs. Finally, in Fig. 4.7 (d), the estimated power consumption for each full adder cell is shown together with the transistor level simulation results. As can be seen in Fig. 4.7 (d) the power consumption is overestimated, except for the first full adder. Again, the differences are due to not all switching 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 $X$ and $Y$ inputs of the used full adder. Hence, the average energy consumption in Table 4.9 is useful. However, for full adders with non-symmetric $X$ and $Y$ 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. 4.8 (a). The switching activity of the carry and sum outputs are shown in Figs. 4.8 (b) and (c), respectively. Again, there is a very close match between the simulated results and the proposed method. Figure 4.8 (d) shows the estimated and simulated power consumption.
Figure 4.7  Switching activities of single bits for (a) input signals, (b) carry output, and (c) sum output. (d) Power consumption at 50 MHz. In (b) and (c) the value in parentheses for the simulations is the delay relation, $S_{\text{del}} / C_{\text{del}}$, as discussed in Section 4.3.1.
Experimental Results

Figure 4.8  Switching activities of single bits for (a) input signals, (b) carry output, and (c) sum output. (d) Power consumption at 50 MHz. In (b) and (c) the value in parentheses for the simulations is the delay relation, $S_{del} / C_{del}$, as discussed in Section 4.3.1.
4.6 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 [33],[37],[52]. The least significant 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 [33].

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 [37], 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 switchings when the applied two’s-complement represented inputs are real world signals.

4.7 Statistical Definition of Signals

Signals in real world applications can in many cases be approximated as a Gaussian stochastic variable, i.e., the corresponding probability density function (PDF) has the characteristic shape illustrated in Fig. 4.9. As can be seen in Fig. 4.9 the probability is highest at the mean, $\mu$, and is almost zero at three standard deviations ($3\sigma$) away from the mean. Note that the standard deviation, $\sigma$, can be obtained from the variance, $\sigma^2$.

The coefficient of correlation is defined as
4.7.1 The Dual Bit Type 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 a few regions, where the bits of each region has a well defined switching activity. In [33] the three regions LSB, linear, and MSB was defined as illustrated in Fig. 4.10. 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 shown in Fig. 4.10.

The breakpoints that separate the regions, can be computed from word-level statistics as \[33\]

$$
\begin{align*}
BP_0 &= \log_2 \sigma + \log_2 \left( \sqrt{1 - \rho^2 + \frac{|P|}{8}} \right) \\
BP_1 &= \log_2 (|\mu| + 3\sigma)
\end{align*}
$$

(4.31)

The intermediate breakpoint is here defined as

$$BP = \left\lfloor \frac{1}{2} (BP_0 + BP_1) \right\rfloor$$

(4.32)

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 $\rho_Q$, then the switching activity, $\alpha_Q$, of $Q$ is defined as \[52\]

$$
\alpha_Q = 2p_Q(1 - p_Q)(1 - \rho_Q)
$$

(4.33)

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, $\mu$, is zero, which is the case if the two’s-complement number range is utilized efficiently. Furthermore, it was stated in \[52\] that the temporal correlation for bits in the MSB region is close to the word-level correlation, $\rho$. Hence, the switching activity in the MSB region can be computed from (4.33) as

$$\alpha_{MSB} = \frac{1}{2}(1 - \rho)$$

(4.34)

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 (4.32), may be different, resulting in the general case illustrated in Fig. 4.11. Without loss of generality, $BP_Y$ is defined to
be at least as large as $BP_X$. As can be seen in Fig. 4.11 there are three different regions to be considered. The number of bits in each region is obtained from the breakpoints as

$$\begin{align*}
N_{UU} &= BP_X + 1 \\
N_{SU} &= BP_Y - BP_X \\
N_{SS} &= n - BP_Y
\end{align*}$$

(4.35)

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.

### 4.8 DBT Model for Switching Activity in RCA

From the switching activity of each input bit, $X_i$, $Y_i$, and $C_i$, the switching activity, $\alpha$(transition), for each type of transition can be computed according to (4.17). This model, TMMM, was derived under the same assumption that was used in Section 4.7.1, 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, as 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 (4.17) 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

\[
\alpha(X_i) = \alpha(Y_i) = \frac{1}{2}
\]  
(4.36)

If (4.36) is applied to (4.17) and the sum is computed for all \( i \in UU \) the total switching activities are obtained as

\[
\left\{ \begin{array}{l}
\alpha(C_{UU}) = \frac{1}{8}(N_{UU} + \alpha(C_{in})) \\
\alpha(S_{UU}) = \frac{1}{8}(5N_{UU} - 3 - \alpha(C_{in}) + (2\alpha(C_{in}) - 3)(1 - 2^{-N_{UU}}))
\end{array} \right.
\]  
(4.37)

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.,

\[
\alpha(X_i) = \alpha_{MSB,X}, \quad \alpha(Y_i) = \frac{1}{2}
\]  
(4.38)

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 negligible. 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 as

\[
\left\{ \begin{array}{l}
\alpha(C_{SU}) = \frac{1}{4}\alpha_{MSB,X}N_{SU} \\
\alpha(S_{SU}) = (1 - 2^{-N_{SU}})\left(\frac{3}{4}(1 - 2^{-N_{UU}}) - \frac{1}{2}(1 + \alpha_{MSB,X}) + \frac{1}{2}(1 + \frac{1}{2}\alpha_{MSB,X})N_{SU}\right)
\end{array} \right.
\]  
(4.39)

For the last region, \( SS \), both inputs are sign extended, so that
\[ \alpha(x_i) = \alpha_{\text{MSB}, X}, \quad \alpha(y_i) = \alpha_{\text{MSB}, Y} \] (4.40)

and the total switching activities are

\[
\begin{align*}
\alpha(C_{SS}) &= \frac{1}{2} \alpha_{\text{MSB}, X} \alpha_{\text{MSB}, Y} N_{SS} \\
\alpha(S_{SS}) &= (1 - 2^{-N_{SS}}) \left( \frac{3}{4} (1 - 2^{-N_{UU}}) \right) 2^{-N_{SU}} + \\
&\quad \frac{1}{2} (1 + \alpha_{\text{MSB}, X})(1 - 2^{-N_{SU}}) - \\
&\quad \alpha_{\text{MSB}, X} - \alpha_{\text{MSB}, Y} + \alpha_{\text{MSB}, X} \alpha_{\text{MSB}, Y} \\
&\quad \left( \alpha_{\text{MSB}, X} + \alpha_{\text{MSB}, Y} - \frac{3}{2} \alpha_{\text{MSB}, X} \alpha_{\text{MSB}, Y} \right) N_{SS}
\end{align*}
\] (4.41)

Finally, the total switching activities of the RCA are obtained by summing the contributions from each region

\[
\begin{align*}
\alpha(C_{\text{tot}}) &= \alpha(C_{UU}) + \alpha(C_{SU}) + \alpha(C_{SS}) \\
\alpha(S_{\text{tot}}) &= \alpha(S_{UU}) + \alpha(S_{SU}) + \alpha(S_{SS})
\end{align*}
\] (4.42)

Note that the expressions in (4.42) are functions of the parameters \( \alpha(C_{in}) \), which is obtained from the architecture, and \( N_{UU}, N_{SU}, N_{SS}, \alpha_{\text{MSB}, X}, \) and \( \alpha_{\text{MSB}, Y} \) which are obtained from the statistical knowledge of the inputs, \( X \) and \( Y \), as discussed in Section 4.7.1. Furthermore, no iterative computations are required as opposed to TMMM.

### 4.8.1 Simplified Model Assuming High Correlation

If the correlation of the input signals is high, i.e., both \( \rho_X \) and \( \rho_Y \) is close to one, the switching activities of all sign bits will be ignorable. Hence, it can be assumed that

\[ \alpha_{\text{MSB}, X} = \alpha_{\text{MSB}, Y} = 0 \] (4.43)
which gives simplified equations for the SU and the SS regions as

\[
\begin{align*}
\alpha(C_{SU}) &= \alpha(C_{SS}) = 0 \\
\alpha(S_{SU}) &= (1 - 2^{-N_{SU}}) \left(\frac{3}{4}(1 - 2^{-N_{UU}}) - \frac{1}{2}\right) + \frac{1}{2}N_{SU} \\
\alpha(S_{SS}) &= (1 - 2^{-N_{SS}}) \left(\frac{1}{4}(1 - 3 \cdot 2^{-N_{UU}})2^{-N_{SU}} + \frac{1}{2}\right)
\end{align*}
\]  

(4.44)

### 4.9 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.

The word-level statistics are stated in Table 4.12 and the probability density functions are shown in Fig. 4.12 (a). Both signals have a mean close to zero, as was assumed in the model. Since the PDFs for the two signals are similar, a modification is done 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. 4.12 (b).

The breakpoints and the switching activity in the MSB region are computed from the equations (4.31), (4.32), and (4.34). Note that (4.31) assume integer representation, hence the mean and standard deviation values in Table 4.12 are multiplied by \( 2^{n-2} \) (because of the previous division by four) and \( 2^n \) for the \( X \) and \( Y \) signal, respectively. The results are stated in Table 4.13. The number of bits in each region, obtained from the intermediate breakpoints according to (4.35), 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 (4.37), (4.39), and (4.41) for the \( UU, SU, \) and \( SS \) region, respectively. The sum of all regions gives the total switch-
ing activities of the RCA as stated in (4.42). Results are shown in Table 4.14. As can be seen in Table 4.15 the corresponding values for the simplified model defined in Section 4.8.1 are similar.

The switching activity for the input signals is shown in Fig. 4.13. Here the dashed line corresponds to the three-region model defined by $BP_0$ and

\begin{table}
\centering
\begin{tabular}{|c|c|c|}
\hline
Input & $\mu$ & $\sigma$ & $\rho$ \\
\hline
$X$ & $-0.00040$ & 0.1269 & 0.9440 \\
$Y$ & $-0.00062$ & 0.1414 & 0.8652 \\
\hline
\end{tabular}
\caption{Word-level statistics for the two input signals.}
\end{table}

\begin{table}
\centering
\begin{tabular}{|c|c|c|c|}
\hline
Input & $BP_0$ & $BP_1$ & $BP$ & $\alpha_{MSB}$ \\
\hline
$X$ & 8.86 & 11.61 & 10 & 0.028 \\
$Y$ & 11.46 & 13.76 & 12 & 0.067 \\
\hline
\end{tabular}
\caption{Model parameters obtained from word-level statistics.}
\end{table}

Figure 4.12 (a) Probability density functions for the input signals, $X$ and $Y$. (b) The $X$ values are divided by four.
However, the proposed model only depends on the intermediate breakpoint, $BP$, which is illustrated by the dotted line in Fig. 4.13.

The switching activities for carry and sum bits, computed according to (4.29), are shown in Fig. 4.14. The proposed switching activity model for ripple-carry adders, referred to as DBT, was plotted and verified using the simplification defined by (4.44).

Table 4.14 Switching activity for the transition types $C$ and $S$.

<table>
<thead>
<tr>
<th>Type</th>
<th>$UU$</th>
<th>$SU$</th>
<th>$SS$</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>$C$</td>
<td>1.3750</td>
<td>0.0140</td>
<td>0.0028</td>
<td>1.3918</td>
</tr>
<tr>
<td>$S$</td>
<td>6.1254</td>
<td>1.1907</td>
<td>0.6972</td>
<td>8.0133</td>
</tr>
</tbody>
</table>

Table 4.15 Switching activity for the transition types $C$ and $S$ using the simplified DBT model defined by (4.44).

$BP1$. However, the proposed model only depends on the intermediate breakpoint, $BP$, which is illustrated by the dotted line in Fig. 4.13.

The switching activities for carry and sum bits, computed according to (4.29), are shown in Fig. 4.14. The proposed switching activity model for ripple-carry adders, referred to as DBT, was plotted and verified using the
TMMM model, which give identical results for adjusted input switching activities, i.e., as the dotted lines in Fig. 4.13. Simulated results from a VHDL model in ModelSim is also shown in Fig. 4.14. 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 4.16. This is the reason that only two signal regions need to be considered in the dual bit type method [33].

![Graph](image)

**Figure 4.14** 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, $S_i$.

<table>
<thead>
<tr>
<th>FA output</th>
<th>Simulation</th>
<th>TMMM</th>
<th>DBT</th>
<th>DBT simple</th>
</tr>
</thead>
<tbody>
<tr>
<td>$C$</td>
<td>0.5875</td>
<td>0.5999</td>
<td>0.5878</td>
<td>0.5737</td>
</tr>
<tr>
<td>$S$</td>
<td>1.0124</td>
<td>1.0277</td>
<td>1.0017</td>
<td>0.9756</td>
</tr>
</tbody>
</table>

**Table 4.16** Average switching activity per bit at the $C$ and $S$ outputs.
CONCLUSIONS

In this thesis, the possibilities to improve the complexity and energy consumption of constant multiplication using serial arithmetic were investigated. The main difference between serial and parallel arithmetic, that is of interest here, is that a shift operation in serial arithmetic requires a flip-flop, while it can be hardwired in parallel arithmetic.

For single-constant multipliers, 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. 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 adder cost and flip-flop cost.

Two algorithms for the design of multiplier blocks using serial arithmetic were proposed. The difference between the proposed 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 an algorithm 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.

An implementation example of a digit-serial FIR filter was investigated by varying the digit-size. Besides the two proposed multiplier block
Conclusions

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 multiplier block 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 logic depth of the different arithmetic algorithms, and the relation between energy consumption and logic depth, was discussed.

A method for computing the switching activity 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 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 graph representation for any given constant to be implemented. In addition, a simplified method for 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.

A data dependent switching activity model was proposed for ripple-carry adders. For most applications, the input data is correlated, while previous estimations assumed un-correlated data. Hence, the proposed method may be included in high-level power estimation to give more accurate estimates. Also, the model can be used as cost function in multiple-constant multiplication algorithms for both serial and parallel arithmetic. The proposed model is accurate in estimating the switching activity of the carry and sum. However, the energy consumption was overestimated as not all switching in the implemented adders was 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.
REFERENCES


