Digital Signal Processing and Machine Learning System Design using Stochastic Logic

A DISSERTATION
SUBMITTED TO THE FACULTY OF THE GRADUATE SCHOOL
OF THE UNIVERSITY OF MINNESOTA
BY

Yin Liu

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

Advisor: Keshab K. Parhi

July, 2017

© Yin Liu 2017
ALL RIGHTS RESERVED

Acknowledgements

First and foremost, I want to thank my advisor Prof. Keshab K. Parhi, for his continuing encouragement, tremendous guidance, and financial support throughout my entire Ph.D. study at the University of Minnesota. Throughout these years that he has been my advisor, he has served as an incredible inspiration and talented mentor in my educational development.

I also would like to thank Prof. Marc Riedel, Prof. Gerald E. Sobelman and Prof. Pen-Chung Yew at the University of Minnesota, for their support as members of my Ph.D. committee.

My sincere thank also goes to current and former members of my research group. I am grateful to Yingjie Lao, Zisheng Zhang, Sohini Roychowdhury, Tingting Xu, Bo Yuan, Sayed Ahmad Salehi, Shu-Hsien Chu, Sandhya Koteshwara and Bhaskar Sen. I also would like to thank Dongjiang You, Yuan Li, Qianying Tang and Bingzhe Li for their support during my Ph.D study.

Last but not the least, I am forever grateful to my parents. Without their support, I would not have earned my Ph.D. degree. I would like to express my dearest thanks to my girlfriend Sha Li who has always been extremely understanding and supportive during my studies. She also provides me significant help and advice on my career development. My career and life are more meaningful because of the love and care that I have been privileged to receive from my whole family.

Dedication

To my beloved Sha, for her advice, her faith, and her endless love.

Abstract

Digital signal processing (DSP) and machine learning systems play a crucial role in the fields of big data and artificial intelligence. The hardware design of these systems is extremely critical to meet stringent application requirements such as extremely small size, low power consumption, and high reliability. Following the path of Moore's Law, the density and performance of hardware systems are dramatically improved at an exponential pace. The increase in the number of transistors on a chip, which plays the main role in improvement in the density of circuit design, causes rapid increase in circuit complexity. Therefore, low area consumption is one of the key challenges for IC design, especially for portable devices. Another important challenge for hardware design is reliability. A chip fabricated using nanoscale complementary metal-oxide-semiconductor (CMOS) technologies will be prone to errors caused by fluctuations in threshold voltage, supply voltage, doping levels, aging, timing errors and soft errors. Design of nanoscale failure-resistant systems is currently of significant interest, especially as the technology scales below 10 nm. Stochastic Computing (SC) is a novel approach to address these challenges in system and circuit design.

This dissertation considers the design of digital signal processing and machine learning systems in stochastic logic. The stochastic implementations of finite impulse response (FIR) and infinite impulse response (IIR) filters based on various lattice structures are presented. The implementations of complex functions such as trigonometric, exponential, and sigmoid, are derived based on truncated versions of their Maclaurin series expansions. We also present stochastic computation of polynomials using stochastic subtractors and factorization. The machine learning systems including artificial neural network (ANN) and support vector machine (SVM) in stochastic logic are also presented.

First, we propose novel implementations for linear-phase FIR filters in stochastic logic. The proposed design is based on lattice structures. Compared to direct-form linear-phase FIR filters, linear-phase lattice filters require twice the number of multipliers but the same number of adders. The hardware complexities of stochastic implementations of linear-phase FIR filters for direct-form and lattice structures are comparable.

We propose stochastic implementation of IIR filters using lattice structures where the states are orthogonal and uncorrelated. We present stochastic IIR filters using basic, normalized and modified lattice structures. Simulation results demonstrate high signal-to-error ratio and fault tolerance in these structures. Furthermore, hardware synthesis results show that these filter structures require lower hardware area and power compared to two's complement realizations.

Second, We present stochastic logic implementations of complex arithmetic functions based on truncated versions of their Maclaurin series expansions. It is shown that a polynomial can be implemented using multiple levels of NAND gates based on Horner's rule, if the coefficients are alternately positive and negative and their magnitudes are monotonically decreasing. Truncated Maclaurin series expansions of arithmetic functions are used to generate polynomials which satisfy these constraints. The input and output in these functions are represented by unipolar representation. For a polynomial that does not satisfy these constraints, it still can be implemented based on Horner's rule if each factor of the polynomial satisfies these constraints. format conversion is proposed for arithmetic functions with input and output represented in different formats, such as $\cos \pi x$ given $x \in [0, 1]$ and sigmoid(x) given $x \in [-1, 1]$. Polynomials are transformed to equivalent forms that naturally exploit format conversions. The proposed stochastic logic circuits outperform the well-known Bernstein polynomial based and finite-state-machine (FSM) based implementations. Furthermore, the hardware complexity and the critical path of the proposed implementations are less than the Bernstein polynomial based and FSM based implementations for most cases.

Third, we address subtraction and polynomial computations using unipolar stochastic logic. It is shown that stochastic computation of polynomials can be implemented by using a stochastic subtractor and factorization. Two approaches are proposed to compute subtraction in stochastic unipolar representation. In the first approach, the subtraction operation is approximated by cascading multi-levels of OR and AND gates. The accuracy of the approximation is improved with the increase in the number of stages. In the second approach, the stochastic subtraction is implemented using a multiplexer and a stochastic divider. We propose stochastic computation of polynomials using factorization. Stochastic implementations of first-order and second-order factors are presented for different locations of polynomial roots. From experimental results, it

is shown that the proposed stochastic logic circuits require less hardware complexity than the previous stochastic polynomial implementation using Bernstein polynomials.

Finally, this thesis presents novel architectures for machine learning based classifiers using stochastic logic. Three types of classifiers are considered. These include: linear support vector machine (SVM), artificial neural network (ANN) and radial basis function (RBF) SVM. These architectures are validated using seizure prediction from electroencephalogram (EEG) as an application example. To improve the accuracy of proposed stochastic classifiers, an approach of data-oriented linear transform for input data is proposed for EEG signal classification using linear SVM classifiers. Simulation results in terms of the classification accuracy are presented for the proposed stochastic computing and the traditional binary implementations based datasets from two patients. It is shown that accuracies of the proposed stochastic linear SVM are improved by 3.88% and 85.49% for datasets from patient-1 and patient-2, respectively, by using the proposed linear-transform for input data. Compared to conventional binary implementation, the accuracy of the proposed stochastic ANN is improved by 5.89% for the datasets from patient-1. For patient-2, the accuracy of the proposed stochastic ANN is improved by 7.49% by using the proposed linear-transform for input data. Additionally, compared to the traditional binary linear SVM and ANN, the hardware complexity, power consumption and critical path of the proposed stochastic implementations are reduced significantly.

Contents

Acknowledgements **i**

Dedication **ii**

Abstract **iii**

List of Tables **xi**

List of Figures **xv**

1 Introduction **1**

1.1 Introduction 1

1.2 Summary of Contributions 3

1.2.1 FIR Digital Filter Design in Stochastic Logic 3

1.2.2 IIR Digital Filter Design in Stochastic Logic 4

1.2.3 Computing Arithmetic Functions using Stochastic Logic by Series Expansion 5

1.2.4 Computing Polynomials using Unipolar Stochastic Logic 5

1.2.5 Machine Learning Systems in Stochastic Logic 6

1.3 Outline of the Thesis 7

2 FIR Digital Filters in Stochastic Logic **9**

2.1 Background 9

2.1.1 SC Inner-Product 9

2.1.2 Implementation Considerations for Stochastic Digital Filters . . 12

2.2 Stochastic Lattice Implementation of Linear-phase FIR Filters 13
2.2.1 Lattice structure for FIR filters 13
2.2.2 Linear-phase lattice FIR filters 17
2.2.3 Stochastic implementation of linear-phase lattice FIR filters . . . 18
2.3 Experimental Results 22
2.3.1 Simulation Results 22
2.3.2 Synthesis Results 25
2.3.3 Fault Tolerance Analysis 27
2.4 Conclusion 28

3 IIR Digital Filters in Stochastic Logic 31
3.1 Stochastic State-space Implementation for IIR Digital Filters 32
3.1.1 Background of Lattice IIR Filter 32
3.1.2 An Example of 3rd-order Stochastic IIR Filter with State-Space Description 32
3.1.3 The Lattice-to-State-Space Algorithm for Arbitrary Order IIR Filters 35
3.2 Stochastic Lattice Implementation 38
3.3 Normalized Stochastic Lattice IIR Digital Filters 40
3.3.1 State-Space Implementation with Scaling 40
3.3.2 Stochastic Lattice Implementation 41
3.4 Optimized Stochastic Lattice IIR Filters 43
3.5 The Stochastic Implementation of Normalized Lattice IIR Filter Using Traditional SC Units 48
3.5.1 Inner-product for stochastic IIR filter design 48
3.5.2 The Stochastic Implementation of Normalized Lattice IIR Filters 49
3.5.3 Analysis of Hardware Complexity 51
3.6 The Stochastic Implementation of modified lattice IIR filters 52
3.6.1 The Modified Lattice Structure 52
3.6.2 The design of stochastic modified lattice IIR filters 57
3.6.3 State overflow and input scaling for the modified lattice structure 58
3.7 Experimental Results 59

3.7.1 Simulation Results 59
3.7.2 Synthesis Results 64
3.7.3 Fault Tolerance Analysis 68
3.8 Conclusion 70

4 Computing Arithmetic Functions using Stochastic Logic 71
4.1 Theoretical Foundations for Stochastic logic 72
4.1.1 Implementing Polynomials using Horner's Rule 72
4.1.2 Implementation using Factoring and Factor-Combining 73
4.1.3 Format Conversion 76
4.2 Horner's rule for Maclaurin expansions 77
4.3 Implementations using Factorization 82
4.3.1 The Implementation of $\sin \pi x$ 82
4.3.2 The Implementation of e^{-ax} $(a > 1)$ 84
4.3.3 Generalization of Stochastic Implementation for e^{-ax} with $a > 1$ 85
4.3.4 The Implementation of $\tanh ax$ and sigmoid(ax) for $a > 1$ 87
4.4 Implementation of Functions with Input and Output Represented using Different Formats 88
4.4.1 Unipolar Input and Bipolar Output 88
4.4.2 Bipolar Input and Unipolar Output 91
4.5 Experimental Results 93
4.5.1 Previous Work 93
4.5.2 Performance Comparison 96
4.5.3 Hardware Complexity and Critical Path Delay comparisons . . . 98
4.6 Conclusion 99

5 Polynomial Computation in Unipolar Stochastic Logic 101
5.1 Subtraction in Stochastic Unipolar Representation 101
5.1.1 Unipolar Subtraction using Multi-Level Logic 102
5.1.2 Computing Subtraction Based on Stochastic Unipolar Division . 106
5.2 Polynomial Computation Using Unipolar Subtraction 110
5.2.1 Case-I: all positive coefficients 111
5.2.2 Case-II: positive and negative coefficients 113

5.3 Polynomial Computation Using Factorization 114
5.3.1 Location of complex roots: $u < 0$ 116
5.3.2 Location of complex roots: $(u^2 + v^2 - 2u \geq 0)$ and $(u \geq 0.5)$ . . 118
5.3.3 Location of complex roots: $(0 < u < 0.5)$ and $(u^2 + v^2 \geq 1)$ . . . 118
5.3.4 Location of complex roots: $(u^2 + v^2 - 2u < 0)$ and $(u^2 + v^2 > 1)$ and $(u \geq 0.5)$ 120
5.3.5 Location of complex roots: $((u-0.5)^2 + v^2 \geq 0.25)$ and $(u^2+v^2 < 1)$ and $(u > 0)$ 121
5.3.6 Location of complex roots: $(u - 0.5)^2 + v^2 < 0.25$ 122
5.4 Comparison of simulation and synthesis results 124
5.4.1 Stochastic polynomial computations 125
5.4.2 Complex arithmetic functions based on stochastic polynomial computations 128
5.4.3 Comparison with the STRAUSS implementation 131
5.5 Comparison of polynomial computations using unipolar and bipolar format 134
5.6 Conclusions 137

6 Machine Learning **138**
6.1 Background of Machine Learning Classifiers 139
6.2 Stochastic Implementation of linear SVM classifier 141
6.2.1 The Architecture of Stochastic Implementaion 141
6.2.2 EEG Signal Classification using Stochastic Linear SVM 143
6.3 Stochastic Implementation of ANN classifier 149
6.3.1 The Architecture of Stochastic Implementation 149
6.3.2 EEG Signal Classification using Stochastic ANN Classifier 151
6.4 Computing RBF Kernel for SVM Classification using Stochastic Logic . 155
6.4.1 Stochastic Implementation of RBF Kernel using Finite State Machine 156
6.4.2 Stochastic RBF Kernel based on format conversion 159
6.4.3 Test RBF Kernel of SVM classifier based on EEG Signals 162
6.5 conclusion 163

7 Conclusion and Future Work **164**

References **166**

List of Tables

2.1 Area consumption comparison of two implementations for stochastic direct-form FIR filters in terms of equivalent 2-input NAND gates. 13

2.2 The output SNR (dB) for linear-phase FIR filters of different orders and cut-off frequencies. 25

2.3 The hardware complexity (in terms of equivalent 2-input NAND gates), power consumption and critical path delay of various implementations of linear phase FIR filters. 27

2.4 The output SNR (dB) due to random bit-flipping for different implementations for a 7th-order low-pass linear-phase FIR filter with cut-off frequency 0.1π. 28

3.1 The area of each single module in a stochastic lattice stage in terms of equivalent 2-input NAND gates. 51

3.2 The area composition of the stochastic implementation for a 3rd-order normalized lattice IIR filter using traditional SC units in terms of equivalent 2-input NAND gates. 52

3.3 The output SER (dB) of 3rd-order stochastic IIR filters for different implementations. 62

3.4 The output SER (dB) of 6th-order stochastic IIR filters for different implementations. 62

3.5 The output MSE and SNR (dB) of (a) 3rd-order low-pass stochastic IIR filters, (b) 3rd-order high-pass stochastic IIR filters for various implementations. 64

3.6 The power consumption comparison and area consumption comparison in terms of equivalent 2-NAND gates for different implementations of IIR filters. 66
3.7 The hardware complexity comparison in terms of equivalent 2-NAND gates for different implementations of IIR filters. 67
3.8 The delay (ns) of critical path for different implementations of IIR filters. 67
3.9 The output SER (dB) with random bit-flipping for different implementations of a 3rd-order low-pass butterworth IIR filter with cut-off frequency 0.3π. 69
4.1 The arithmetic functions implemented in this paper. 72
4.2 The computational results and formats of internal nodes in Fig. 4.21. . . 91
4.3 The computational results and formats of internal nodes and the output in Fig. 4.22. 93
4.4 The output mean absolute error (MAE) of stochastic implementations for different functions using the proposed method, the FSM-based method and the Bernstein polynomial method with different orders. 97
4.5 The hardware complexity and critical path delay (ns) of stochastic implementations for different functions using the proposed method, the FSM-based method and the Bernstein polynomial method with different orders. 100
5.1 The corresponding Boolean and arithmetic operations for AND, OR and NOT gates. 102
5.2 The output Mean Absolute Error (MAE) of stochastic subtractions $x_1 - x_2$ for different values of x_1. 108
5.3 Synthesis results of the subtractor based on stochastic divider (subtractor 1), the subtractor using 4-stage enhancement units (subtractor 2) and stochastic scaled adder. 110
5.4 Output mean absolute error (MAE) of two proposed implementations and previous implementation using Bernstein polynomials for $f(x)$. . . . 128
5.5 Synthesis results for different stochastic implementations of $f(x)$. 128
5.6 Output mean absolute error (MAE) of different stochastic implementations of $g(x)$. 130
5.7 Hardware complexity for different stochastic implementations of $g(x)$. . 130

5.8 Output mean absolute error (MAE) of different stochastic implementations of $p_1(x)$. 133
5.9 Hardware complexity for different stochastic implementations of $p_1(x)$. . 134
5.10 The signal-to-error ratio (SER) in dB for unipolar and bipolar implementations of $f_1(x)$. 135
6.1 The confusion matrix of classification for patient-1 (4-features) using *conventional binary* linear SVM (16-bit fixed point implementation). 145
6.2 The confusion matrix of classification for patient-1 (4-features) using *stochastic* linear SVM with l_1 scaling for input data. 145
6.3 The confusion matrix of classification for patient-2 (5-features) using *conventional binary* linear SVM (16-bit fixed point implementation). 146
6.4 The confusion matrix of classification for patient-2 (5-features) using *stochastic* linear SVM with l_1 scaling for input data. 146
6.5 The confusion matrix of classification for patient-1 (4-features) using *stochastic* linear SVM with *linear transform* for input data. 148
6.6 The confusion matrix of classification for patient-2 (5-features) using *stochastic* linear SVM with *linear transform* for input data. 148
6.7 Synthesis results of conventional binary and stochastic linear SVM classifiers for dataset-1 with 4 features and dataset-2 with 5 features. 149
6.8 The confusion matrix of classification for patient-1 (4-features) using *conventional binary* ANN with 16-bit fixed point implementation. 152
6.9 The confusion matrix of classification for patient-1 (4-features) using *stochastic* ANN with l_1 scaling for input data. 152
6.10 The confusion matrix of classification for patient-1 (4-features) using *stochastic* ANN with *linear transform* for input data. 152
6.11 The confusion matrix of classification for patient-2 (5-features) using *conventional binary* ANN with 16-bit fixed point implementation. 153
6.12 The confusion matrix of classification for patient-2 (5-features) using *stochastic* ANN with l_1 scaling for input data. 153
6.13 The confusion matrix of classification for patient-2 (5-features) using *stochastic* ANN with *linear transform* for input data. 153

6.14 Synthesis results of conventional binary and stochastic ANN classifiers for dataset-1 with 4 features and dataset-2 with 5 features. 155
6.15 The output mean absolute error (MAE) of two implementations of stochastic RBF kernel. 163

List of Figures

1.1 (a) The circuit diagram of a basic stochastic number generator (SNG) and (b) the symbol representing an SNG, where RNG stands for random number generator. . . . 2

1.2 Fundamental stochastic computational elements. (a) $y = 1-x$ in unipolar format or $y = -x$ in bipolar format. (b) Unsigned multiplication in unipolar format: $y = x_1 \cdot x_2$. (c) Scaled addition in unipolar/bipolar format: $y = \frac{a}{|a|+|b|} \cdot x_1 + \frac{b}{|a|+|b|} \cdot x_2$. (d) Signed multiplication in bipolar format: $y = x_1 \cdot x_2$. . . . 3

2.1 (a) circuit diagram, and (b) structure description of traditional stochastic inner-product scaled by 1/2. . . . 10

2.2 (a) Circuit diagram, and (b) structure description of proposed implementation of stochastic inner-product. . . . 10

2.3 The architecture of a stochastic inner-product with input vector size of 4. 11

2.4 Two approaches to delaying input signals in stochastic digital filters: the input samples are delayed in (a) stochastic representation, (b) binary representation. . . . 12

2.5 A typical basic lattice stage for FIR filters. . . . 14

2.6 An alternative implementation of basic lattice stage for FIR filters. . . . 15

2.7 The implementation of normalized lattice stage corresponding to equations (2.7). . . . 15

2.8 The implementation of normalized lattice stage corresponding to equations (2.8). . . . 16

2.9 The implementation of modified lattice stage corresponding to equations (2.11). . . . 16

2.10 The implementation of modified lattice stage corresponding to equations (2.12). 17
2.11 The block diagram of an N-tap FIR lattice filter. 17
2.12 The block diagram of an N-tap linear-phase FIR lattice filter using *basic* lattice structure. 18
2.13 The block diagram of an N-tap linear-phase FIR lattice filter using arbitrary proposed lattice stage. 18
2.14 Zeros of $H(z) = 0.0264+0.1405z^{-1}+0.3331z^{-2}+0.3331z^{-3}+0.1405z^{-4}+0.0264z^{-5}$, which describes a linear-phase 5th-order FIR filter with cut-off frequency at 0.1π. The lattice coefficients are given by $k = [0.3909, \mathbf{12.6123}, 0, 0, 1]$. 19
2.15 Stochastic inner-products to compute (a) $w_1(n) = ax(n) + by(n)$ and (b) $w_2(n) = c \cdot ax(n) + c \cdot by(n)$ 19
2.16 The architecture of a stochastic implementation for an N-tap linear-phase FIR lattice filter using lattice stages shown in Fig. 2.5, Fig. 2.7 and Fig. 2.9. 21
2.17 The architecture of a stochastic implementation for an N-tap linear-phase FIR lattice filter using lattice stages shown in Fig. 2.6 and Fig. 2.10. . . 21
2.18 The 2's complement implementation corresponding to stochastic lattice FIR filter shown in Fig. 2.17. 21
2.19 The spectrums of input signal, filter frequency response, ideal output, stochastic direct-form output, stochastic lattice-I output, and stochastic lattice-II output for filtering using a 3rd-order linear-phase FIR filters with cut-off frequency at 0.1π. 24
2.20 Output Mean Squared Error (MSE) and SNR of a specified filter with various sequence lengths for (a) stochastic direct-form implementation, (b) stochastic lattice implementation I, and (c) stochastic lattice implementation II. 26
2.21 The architectures for (a) traditional binary FIR filter, (b) stochastic direct-form FIR filter, (c) stochastic lattice implementation-I, and (d) stochastic lattice implementation-II of linear-phase FIR filter, where random bit-flippings occur at the nodes marked (SNG and S2B modules are not shown in this figure). 29

2.22 Fault-tolerance test results of different implementations for a 7th-order low-pass linear-phase FIR filters with cut-off frequency 0.1π. 30
3.1 The 3rd-order basic lattice filter structure. 33
3.2 The circuit diagram of a 3rd-order stochastic IIR lattice filter with state space implementation. 34
3.3 The data path to compute internal state $x_i(n+1)$. 36
3.4 (a) Original 2's complement implementation of the basic lattice module, and (b) stochastic implementation of the basic lattice module. 38
3.5 The transformed basic lattice filter structure to be used for stochastic implementation. 39
3.6 The circuit diagram of a 3rd-order stochastic IIR lattice filter with lattice module implementation. 39
3.7 (a) A filter with unscaled node x, and (b) a filter with scaled node x_s. . 41
3.8 The 3rd-order normalized lattice filter structure. 41
3.9 (a) Original 2's complement implementation of the normalized lattice structure, and (b) stochastic implementation of the normalized lattice structure. 42
3.10 The circuit diagram of normalized 3rd-order stochastic IIR lattice filter with lattice module implementation. 43
3.11 The implementation of stochastic basic lattice stage without binary multiplier. 44
3.12 The transformed basic lattice filter structure using one binary multiplier for stochastic implementation. 45
3.13 The circuit diagram of optimized implementation for 3rd-order stochastic IIR lattice filter. 46
3.14 An Nth-order lattice IIR filter consisting of $\lfloor \frac{N}{3} \rfloor$ 3-stage lattice blocks and one m-stage lattice block. 48
3.15 The stochastic inner-products implemented using (a) traditional SC units, and (b) the coefficients-based scaling method. (S2B modules are not shown in this figure.) 49
3.16 (a) The 2's complement implementation of a lattice stage, and (b) stochastic implementation of a lattice stage using traditional SC units. . 50

3.17 The circuit diagram of the stochastic implementation for a 3rd-order normalized lattice IIR filter using traditional SC units. 51
3.18 The lattice stages obtained by choosing (a) $s_i = 1 - k_i$, and (b) $s_i = 1 + k_i$. 53
3.19 The circuit diagram of the stochastic modified lattice stage with $s_i = 1 - k_i$ for the case of $k_i > 0$. 55
3.20 The circuit diagram of the stochastic modified lattice stage with $s_i = 1 - k_i$ for the case of $k_i < 0$. 56
3.21 The architecture of the stochastic implementation of a 3rd-order high-pass modified lattice Butterworth IIR filter with cut-off frequency $\omega_c = 0.8\pi$. 58
3.22 The 3rd-order modified lattice filter structure. 58
3.23 The filtering results of (a) a low-pass 3rd-order IIR filter with cutoff-frequency 0.4π (stochastic implementation: NSS), and (b) a high-pass 6th-order IIR filter with cutoff-frequency 0.6π (stochastic implementation: NLI). 61
3.24 The counts of state overflow for the basic lattice structure (BSS and BLI) for the 3rd-order low-pass IIR filter. 63
3.25 The filtering results of different stochastic implementations for the high-pass 3rd-order IIR filter with cut-off frequency at 0.7π. 65
3.26 The output MSE and SNR for the 3rd-order low-pass MOD implementation with the cut-off frequency at 0.3π. 65
3.27 Fault-tolerance test results of traditional binary direct-form, normalized lattice, basic lattice, stochastic NSS and stochastic OBLI implementations for a 3rd-order low-pass butterworth IIR filter with cut-off frequency 0.3π. 69
4.1 Fundamental blocks for stochastic unipolar with no scaling: (a) $p_1(x)$ is implemented using NAND gates, (b) $p_2(x)$ is implemented using two levels of NAND gates, (c) $p(x)$ is implemented using multiple levels of NAND gates. 73
4.2 Three possible locations of a real root (r) of $p(x)$: (a) $r \leq 0$, (b) $0 < r \leq 1$ and (c) $1 \leq r$ 75
4.3 Fundamental stochastic computational elements with unipolar input and bipolar output, where (a) $y = 2f(x) - 1$ and (b) $y = 1 - 2f(x)$. 76

4.4 Fundamental stochastic computational elements with bipolar input and unipolar output, where (a) $y = \frac{1}{2}g(x) + \frac{1}{2}$ and (b) $y = \frac{1}{2} - \frac{1}{2}g(x)$. 77
4.5 The SC square operation: $y = x^2$. 78
4.6 The circuit diagram of stochastic implementation of $\sin x$ using the 7th-order Maclaurin polynomial (4.7). 79
4.7 The circuit diagram of stochastic implementation of $\cos x$ using the 8th-order Maclaurin polynomial (4.8). Replacing 4 delays by one delay for the input would also satisfy decorrelation. 80
4.8 The circuit diagram of stochastic implementation of $\tanh x$ using the 9th-order Maclaurin polynomial (4.9). 80
4.9 The circuit diagram of stochastic implementation of $\log(1 + x)$ using the 5th-order Maclaurin polynomial (10). 81
4.10 The circuit diagram of stochastic implementation of e^{-x} using the 5th-order Maclaurin polynomial (4.11). 81
4.11 The circuit diagram of stochastic implementation of sigmoid(x) for $x \in [0, 1]$ using the 5th-order Maclaurin polynomial. 82
4.12 The circuit diagram of stochastic implementation of $\sin \pi x$ using the 9th-order Maclaurin polynomial. 84
4.13 The circuit diagram of stochastic implementation of $e^{-1.9x}$. 85
4.14 The circuit diagram of stochastic implementation of e^{-2x}. 86
4.15 The circuit diagram of stochastic implementation of e^{-ax} $(a > 1)$ by using e^{-bx} $(b \leq 1)$ and $n - 1$ cascaded AND gates. 86
4.16 The two stochastic implementations of e^{-10x}. If one decorrelating delay is used everywhere, this circuit will not function correctly. 87
4.17 Simulation results of (a) e^{-2x} and (b) e^{-8x} using the proposed method. 87
4.18 (a) The circuit diagram of stochastic implementation of $\tanh ax$ $(a > 1)$ using e^{-2ax} and a JK flip-flop. (b) The function $y = \frac{x_1}{x_1+x_2}$ implemented using a JK flip-flop. 88
4.19 An alternative design of $\tanh ax$ in stochastic logic, with sigmoid($2ax$) computed at an internal node. 88
4.20 The simulation result of stochastic implementation of $\tanh 4x$ using the proposed method. 89

4.21 The circuit diagram of stochastic implementation of $\cos \pi x$ using the 10th-order Maclaurin polynomial. 90
4.22 The circuit diagram of stochastic implementation of sigmoid(x) using the 5th-order Maclaurin polynomial. 92
4.23 An example of stochastic implementation based on 3rd-order Bernstein polynomials. Stochastic bit streams x_1, x_2 and x_3 encode the input value x. Stochastic bit streams z_0, z_1, z_2 and z_3 encode the corresponding Bernstein coefficients. 94
4.24 The state transition diagram of the FSM implementing the stochastic $\tanh(\frac{G}{2}x)$, where G is the number of states. 95
4.25 The state transition diagram of the FSM topology proposed in [1]. . . . 95
4.26 The complete circuit for implementation of stochastic functions [1]. . . . 96
5.1 The implementations of stochastic subtraction using (a) a NOR gate, (b) the enhancement unit and a NOR gate, and (c) iterative enhancement units and a NOR gate. 103
5.2 3-stage versions of subtraction with (a) one delay for each stage, and (b) increasing delays for each stage. 105
5.3 The unipolar subtractor using multiple levels of combinational logic with one delay for each stage. 105
5.4 Simulation results of proposed stochastic subtractors using multi-level combinational logic gates are given as functions of x_2 for different values of x_1, where (a) $x_1 = 0.9$ and (b) $x_1 = 0.7$. 106
5.5 The implementation of stochastic subtraction using equation (5.15). . . 107
5.6 (a) Stochastic divider in unipolar format where $p_1 < p_2$, (b) Stochastic subtractor using unipolar divider. 107
5.7 Simulation results of proposed stochastic subtractors based on division and using iterative enhancement units are given as functions of x_2 for different values of x_1, where (a) $x_1 = 0.3$ (b) $x_1 = 0.5$ (c) $x_1 = 0.7$ and (b) $x_1 = 0.9$. 109
5.8 Stochastic implementation of polynomial (19) using multi-levels of multiplexers. 112

5.9 Comparison of simulation results of the proposed stochastic implementation for polynomial (19) and theoretical results. 113
5.10 Stochastic implementation of polynomial (26) based on unipolar subtractor. 114
5.11 Comparison of simulation results of the proposed stochastic implementation for polynomial (26) and theoretical results. 115
5.12 Three possible locations of a real root (r) of $p(x)$: (a) $r \leq 0$, (b) $0 < r \leq 1$ and (c) $1 \leq r$ 116
5.13 Various locations of complex conjugate roots, which are determined by constraints of u and v: (a) $u < 0$, (b) $(u^2 + v^2 - 2u \geq 0)\&\&(u \geq 0.5)$, (c) $(0 < u < 0.5)\&\&(u^2 + v^2 \geq 1)$, (d) $(u^2 + v^2 - 2u < 0)\&\&(u^2 + v^2 > 1)\&\&(u \geq 0.5)$, (e) $((u - 0.5)^2 + v^2 \geq 0.25)\&\&(u^2 + v^2 < 1)\&\&(u > 0)$, and (f) $(u - 0.5)^2 + v^2 < 0.25$. 117
5.14 Stochastic implementation of second-order factor $1 - ax + bx^2$ using equation (5.34). 118
5.15 Stochastic implementation of second-order factor $1 - ax + bx^2$ using equation (5.41). 120
5.16 Stochastic implementation of second-order factor $1 - ax + bx^2$ using equation (5.43). 121
5.17 Stochastic implementation corresponding to equation (5.46). 121
5.18 Stochastic implementation of second-order factor $1 - ax + bx^2$ using equation (5.47). 122
5.19 Stochastic implementation of the transformed second-order factor $(1 + bx^2) - ax$. 123
5.20 Simulation results of the stochastic implementation for $(x - 0.5)^2$. . . . 123
5.21 Stochastic unipolar implementation of $f(x)$ using subtractor (Method-I). 125
5.22 Stochastic unipolar implementation of $f(x)$ using factorization (Method-II). 126
5.23 Stochastic logic implementing the Bernstein polynomial (52) at $x = 0.5$. Stochastic bit streams x_1, x_2 and x_3 encode the value $x = 0.5$. Stochastic bit streams z_0, z_1, z_2 and z_3 encode the corresponding Bernstein coefficients. 127
5.24 Simulation results of different implementations for $f(x)$. 127
5.25 Simulation results of the proposed implementations for $g(x)$. 130

5.26 (a) The stochastic implementation using the factorization method and (b) simulation results for e^{-3x}. 132
5.27 The stochastic implementations of $p_1(x)$ using (a) our proposed factorization method and (b) the spectral transform approach. 133
5.28 The stochastic implementations of $f_1(x)$ using (a) unipolar format, and using (b) bipolar format. 135
5.29 Simulation results of unipolar and bipolar implementations for $f_1(x)$. . . 135
5.30 (a) The stochastic implementation of $f_2(x)$ based on the unipolar format, and (b) simulation results of unipolar and bipolar implementations for $f_2(x)$. 136
6.1 SVM classifier with linear kernel to maximize hyperplane margin. 140
6.2 An artificial neural network (ANN) model. 140
6.3 The thresholding using various functions. 141
6.4 The stochastic implementation of linear SVM kernel. 142
6.5 The whole procedure of seizure prediction using machine learning method. 144
6.6 (a) Computation kernels in a neuron implemented in conventional binary implementation and (b) in stochastic logic. 150
6.7 The state transition diagram of the FSM implementing the stochastic $\tanh(\frac{G}{2}x)$. 150
6.8 The ANN classifier for EEG signal classification. 151
6.9 The implementation of $\frac{||\boldsymbol{x}-\boldsymbol{x'}||^2}{16}$ in stochastic logic. 157
6.10 The state transition diagram of the FSM implementing e^{-2Gx} in stochastic logic. 158
6.11 The stochastic implementation of $\frac{(x_i-x'_i)^2}{4}$. 159
6.12 The circuit diagram of stochastic implementation of e^{-2x} using the 7th-order Maclaurin polynomial. 161
6.13 The computation of the final output for stochastic RBF kernel. 162

Chapter 1

Introduction

1.1 Introduction

Stochastic computing (SC), first proposed in 1960s [2] [3], has recently regained significant attention due to its fault-tolerance and extremely low-cost of arithmetic units [4] [5]. Despite these advantages, stochastic circuits suffer from long processing latency and degradation of accuracy. The energy-efficiency and accuracy of stochastic computing circuit were investigated in [6][7]. Stochastic computing has been exploited in the fields of neural networks [8], control [9], image processing [10], data mining [11] and error control coding applications [12][13][14][15]. SC has also been applied to the design of digital FIR filter [16][17], IIR filter [18][19][20][21] and Gabor filter [22].

The stochastic representation is based on the fraction of 1's in bit streams. Gaines proposed stochastic representation in two formats, unipolar and bipolar [3]. In the unipolar format, a real number x is represented by a stochastic bit stream X, where

$$x = p(X = 1) = p(X).$$

Since x corresponds to a probability value, the unipolar representation must satisfy $0 \leqslant x \leqslant 1$. In the bipolar format,

$$x = 2p(X = 1) - 1 = 2p(X) - 1,$$

where $-1 \leqslant x \leqslant 1$.

To convert a digital value x to a stochastic bit stream X, a stochastic number generator (SNG) is necessary. Fig. 1.1(a) shows an SNG circuit consisting of a comparator and a linear-feedback-shift-register (LFSR) which corresponds to a pseudo-random source [4]. This SNG generates one bit of a stochastic sequence (X_{SC}) every clock cycle. Fig. 1.1(b) shows the symbol which is used in the rest of the paper to represent an SNG. The random number generator (RNG) corresponds to the LFSR in Fig. 1.1(a). The stochastic bit stream is generated by comparing random numbers with the binary input x.

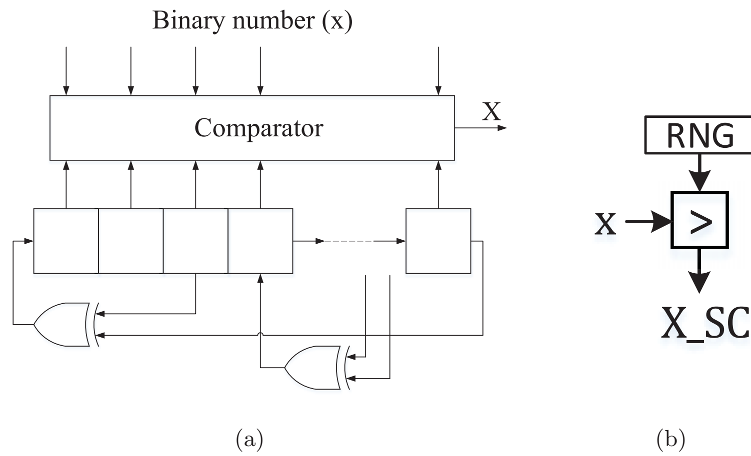


Figure 1.1: (a) The circuit diagram of a basic stochastic number generator (SNG) and (b) the symbol representing an SNG, where RNG stands for random number generator.

Stochastic computational elements can be implemented based on simple combinational logic [2]. Fig. 1.2 illustrates fundamental stochastic combinational logic blocks. The NOT gate is used to implement $1-x$ in unipolar format and $-x$ in bipolar format. The AND gate implements the multiplication in unipolar format. The scaled addition for both unipolar and bipolar formats is implemented using a multiplexer (MUX). Assume that the target function is $T = ax_1 + bx_2$. The computational result from the MUX is scaled by $|a| + |b|$. Signed multiplication is implemented using an XNOR gate in bipolar format. The details of stochastic computational elements can be found in [2] and [8][23].

One advantage of stochastic computing is that complex computations on stochastic

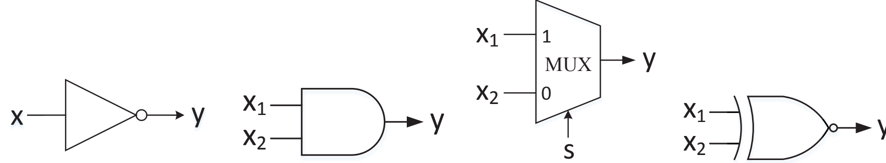


Figure 1.2: Fundamental stochastic computational elements. (a) $y = 1 - x$ in unipolar format or $y = -x$ in bipolar format. (b) Unsigned multiplication in unipolar format: $y = x_1 \cdot x_2$. (c) Scaled addition in unipolar/bipolar format: $y = \frac{a}{|a|+|b|} \cdot x_1 + \frac{b}{|a|+|b|} \cdot x_2$. (d) Signed multiplication in bipolar format: $y = x_1 \cdot x_2$.

bit streams can be realized using extremely low-cost designs in terms of hardware complexity [10]. Brown and Card [8] have proposed stochastic implementations of hyperbolic tangent and exponential functions using finite state machines (FSMs). Recently, several FSM-based implementations for stochastic arithmetic have been proposed to synthesize more sophisticated functions [1]. It has also been illustrated in [24] that complex functions can be approximated in stochastic logic by using Bernstein polynomials [25].

1.2 Summary of Contributions

Our major contributions lie in five categories: FIR digital filter design in stochastic logic, IIR digital filter design in stochastic logic, computing complex function in unipolar format, polynomial computation in unipolar stochastic logic, and machine learning system design in stochastic logic.

1.2.1 FIR Digital Filter Design in Stochastic Logic

It is well known that FIR digital filters can be implemented using lattice structures. FIR lattice [26] structures play a central role in the theory of autoregressive signal modeling [27] and are well suited for implementation of adaptive filters. Although, in general, $2N$ multipliers and adders are required for implementation of N-tap FIR lattice filter, *linear-phase* FIR lattice filters require about N multipliers and N adders. Therefore, linear-phase lattice filters can be implemented using approximately the same computation complexity as direct-form linear-phase structures using stochastic computing [28].

Various lattice structures are investigated for stochastic implementations of FIR digital filters. We propose two stochastic lattice implementations for *linear-phase* FIR filters. It is shown that it can achieve almost equivalent performance as stochastic implementation of direct-form structures. Fault tolerance properties of stochastic FIR digital filters due to random bit-flips at all internal nodes are demonstrated for both direct-form and lattice implementations using speech signals from ICA '99 Synthetic Benchmarks as input [29]. Comprehensive comparisons and analysis of simulation and synthesis results for binary and stochastic implementations are presented.

1.2.2 IIR Digital Filter Design in Stochastic Logic

It is shown that while direct-form IIR filters are not suitable for stochastic implementations [30], lattice IIR digital filters [26] are well suited for stochastic implementations. Intuitively, this is explained by the fact that the states in lattice IIR filters are orthogonal [27], and are inherently uncorrelated. The lattice structure can be described in a state-space form. The lattice structure for stochastic implementation can also be obtained by transforming the lattice IIR digital filter to an equivalent structure. To the best of our knowledge, this is the first feasibility demonstration of stochastic implementations for any arbitrary stable IIR digital filter.

It is also shown that scaling can be exploited to design new forms of lattice filters that suffer less from overflow problems; these structures correspond to scaled versions of well known normalized lattice filters [27]. The proposed structures require binary multipliers in addition to stochastic multipliers. To reduce the number of binary multipliers at the expense of some degradation in performance due to overflow, new types of stochastic lattice filters are proposed. Various stochastic recursive digital implementations based on basic and normalized lattice structures are proposed.

The stochastic IIR filter using a *modified* lattice structure is also presented. Our analysis shows that the stochastic number generators (SNGs) and stochastic-to-binary (S2B) converters are major sources of hardware complexity of stochastic IIR filter implementations. The modified lattice structure requires fewer S2B-SNG converting pairs and does not require any binary multiplier. Both area and power consumptions are reduced significantly.

1.2.3 Computing Arithmetic Functions using Stochastic Logic by Series Expansion

One advantage of stochastic computing is that complex computations on stochastic bit streams can be realized using extremely low-cost designs in terms of hardware complexity. Brown and Card [8] have proposed stochastic implementations of hyperbolic tangent and exponential functions using finite state machines (FSMs). Recently, several FSM-based implementations for stochastic arithmetic have been proposed to synthesize more sophisticated functions [1] [31] [32]. It has also been illustrated in [24] that complex functions can be approximated in stochastic logic by using Bernstein polynomials. However, for some functions, high-degree Bernstein polynomials are required for a precise approximation, and this requires higher hardware complexity. This thesis, for the first time, demonstrates that instead of using FSMs or Bernstein polynomials, the stochastic computation of arithmetic functions, such as trigonometric, exponential, logarithmic and sigmoid, can be implemented by using Maclaurin series expansion and/or factorization.

First, it is shown that a polynomial can be implemented using multiple levels of NAND gates based on Horner's rule, if the coefficients are alternatively positive and negative and their magnitudes are monotonically decreasing. Truncated Maclaurin series expansions of arithmetic functions are used to generate polynomials which satisfy these constraints. The input and output in these functions are represented by unipolar representation. Second, for a polynomial that does not satisfy these constraints, it still can be implemented based on Horner's rule if each factor of the polynomial satisfies these constraints by using factoring and factor-combining. Third, format conversion is proposed for arithmetic functions with input and output represented in different formats. Polynomials are transformed to equivalent forms that naturally exploit format conversions.

1.2.4 Computing Polynomials using Unipolar Stochastic Logic

We consider implementation of polynomials that map the interval [0,1] to itself or negative of itself, i.e., [-1,0]. It has also been illustrated in [24] that polynomials can be implemented in stochastic unipolar representation by using Bernstein polynomials

[25]. However, for some polynomials, high-degree Bernstein polynomials are required for a precise approximation, and this requires higher hardware complexity. This thesis, for the first time, demonstrates that instead of using Bernstein polynomials, stochastic computation of polynomials can be implemented by using a stochastic subtractor and factorization.

First, two approaches are presented to compute stochastic subtraction in unipolar format. Unipolar subtraction can be approximated using multi-levels of combinational logic, including OR and AND gates. It is also shown that a stochastic subtractor can be implemented based on a unipolar divider. Second, computation of polynomials in stochastic unipolar format is proposed using scaled addition and proposed stochastic subtraction. Third, we propose stochastic computation of polynomials using factorization. Different implementations are considered for distinct locations of polynomial roots.

1.2.5 Machine Learning Systems in Stochastic Logic

Stochastic implementations of machine learning classifiers are proposed in this thesis. First, it is shown that the linear support vector machine (SVM) can be implemented using stochastic inner-product. The number of stochastic number generator (SNG) is minimized to reduce the hardware complexity. The artificial neural network (ANN) classifier is implemented using stochastic inner-product and hyperbolic tangent function based on finite-state machine (FSM) based architectures. Second, a *data-oriented linear transform* for input data is proposed to improve the accuracy of classification using stochastic logic. This approach leads to full utilization of the range of bipolar stochastic representation ([-1,1]). The performance of stochastic linear SVM can be improved by the proposed method while it is not always true for ANN classifier due to its *multiple layers* and *non-linearity*. Third, the proposed methods are validated using classifiers for seizure prediction from electroencephalogram (EEG) signals for two subjects from the Kaggle seizure prediction contest [33]. Comparisons in terms of accuracy and synthesis results are presented for conventional binary implementation and proposed stochastic designs.

The stochastic implementation of RBF kernel for SVM classifier is also addressed in

this thesis. First, an architecture with both input and output in bipolar format is proposed. The computation of RBF kernel is comprised of the squared Euclidean distance and the exponential function. In this proposed design, both components are implemented in bipolar format. The squared Euclidean distance is computed using multiple levels of multiplexers, where the number of SNGs is minimized. The bipolar exponential function is designed based on the finite state machine (FSM) method. Second, we propose an implementation of RBF kernel with bipolar input and unipolar output. In this implementation, the squared Euclidean distance is computed with bipolar input and unipolar output. The exponential function is implemented in unipolar format, where factorization and Horner's rule are performed for the Maclaurin expansion of exponential function. The proposed designs are simulated using electroencephalogram (EEG) signals for one subject from the Kaggle seizure prediction contest [33]. Comparisons in terms of accuracy are presented for two proposed architectures.

1.3 Outline of the Thesis

The dissertation is outlined as follows.

The stochastic implementation of FIR digital filters is presented in Chapter 2. Afterwards, we present various experimental results including performance simulation, hardware synthesis and fault tolerance test.

The stochastic implementation of IIR digital filters is presented in Chapter 3. Then, we present various experimental results including performance simulation, hardware synthesis and fault tolerance test.

Chapter 4 presents the design of complex functions in stochastic logic based on series expansion. Then the proposed designs are compared with previous designs including the finite state machine method and the Bernstein polynomial method, in terms of accuracy, hardware complexity, power and latency.

Chapter 5 presents polynomial computation using subtractors and factorization. Several cases are studies to compare the proposed method with the Bernstein polynomial based method.

In Chapter 6, we propose our designs for linear SVM classifier, ANN classifier and

RBF kernel in stochastic logic. The proposed designs are simulated using electroencephalogram (EEG) data for seizure prediction.

Finally, Chapter 7 concludes with a summary of total contributions of this dissertation and future research directions.

Chapter 2

FIR Digital Filters in Stochastic Logic

It is well known that FIR digital filters can be implemented using lattice structures. FIR lattice [26] structures play a central role in the theory of autoregressive signal modeling [27] and are well suited for implementation of adaptive filters. Although, in general, $2N$ multipliers and adders are required for implementation of N-tap FIR lattice filter, *linear-phase* FIR lattice filters require about N multipliers and N adders. Therefore, linear-phase lattice filters can be implemented using approximately the same computation complexity as direct-form linear-phase structures using stochastic computing [28].

2.1 Background

2.1.1 SC Inner-Product

Inner-product modules are critical components of stochastic FIR and IIR digital filters. The design of an SC inner-product module involves stochastic multiplication and addition mentioned in the introduction part. Fig. 2.1 describes a straightforward implementation of SC inner-product which consists of XNOR gates and a multiplexer. The result from this inner-product is $\frac{1}{2}(ax(n) + by(n))$ with (a, b) and $(x(n), y(n))$ as input vectors. We assume that all the binary inputs are converted into stochastic

sequences. This implementation of stochastic inner-product module implies a scale factor of 1/2 such that output result is scaled down to prevent overflow. Unless the magnitude of the inner-product result approaches two, it can be expected that the sum of the stochastic numbers will become smaller and smaller after passing through levels of SC inner-products. Moreover, when the value of a stochastic number becomes smaller, its variance may increase due to the imprecision of SC caused by signal correlation and limited resolution of stochastic sequence with fixed number of bits.

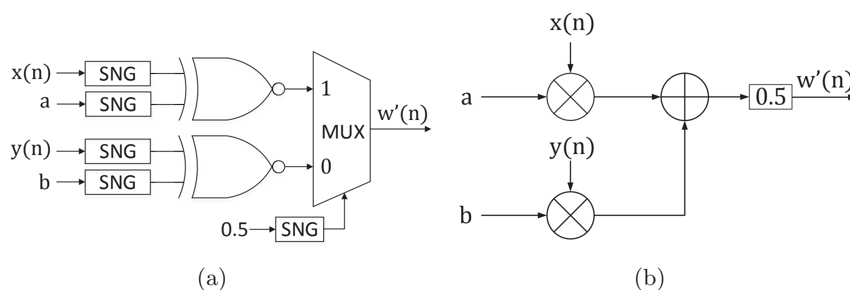


Figure 2.1: (a) circuit diagram, and (b) structure description of traditional stochastic inner-product scaled by 1/2.

To overcome this problem, an implementation of inner-product which is well-suited for the situation that one input vector (a, b) is constant was presented in [30]. This is always the case in filter design where the tap coefficients are constant. Fig. 2.2 shows the design based on an uneven weighted multiplexer. Instead of fixed

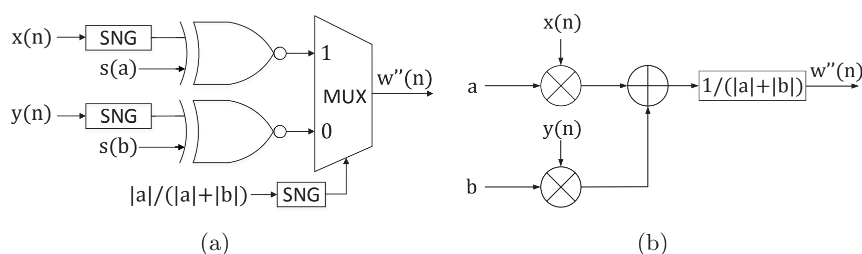


Figure 2.2: (a) Circuit diagram, and (b) structure description of proposed implementation of stochastic inner-product.

probability of 0.5, selecting signal of the multiplexer is set to $\frac{|a|}{|a|+|b|}$. Additionally, $s(a)$ and $s(b)$ indicate signs of coefficients a and b. If a is negative ($sign(a) = -1$), $s(a)$ will be set to 0. Otherwise, $sign(a) = 1$ and $s(a)$ is set to 1. The same holds true for $s(b)$.

The output result of the proposed inner-product is

$$w''(n) = \frac{|a|}{|a|+|b|} sign(a)x(n) + (1 - \frac{|a|}{|a|+|b|}) sign(b)y(n)$$
$$= \frac{1}{|a|+|b|}(ax(n) + by(n)).$$

Since in stochastic representation (both unipolar and bipolar formats) $|a|$ and $|b|$ are less than or equal 1,

$$\frac{1}{|a|+|b|} \geqslant \frac{1}{2}.$$

The scaling factor in proposed implementation is always greater than the fixed factor of 1/2 in conventional implementation. When $|a|+|b|$ is less than one, it will even scale-up the result. Besides better signal scaling, another advantage of proposed design is the reduced number of SNGs required. In this sense, the hardware cost of new inner-product decreases.

If the input vector size is extended, inner-product module can be implemented with tree structure. Fig. 2.3 presents the architecture of an SC inner-product whose input vector size is greater than 2.

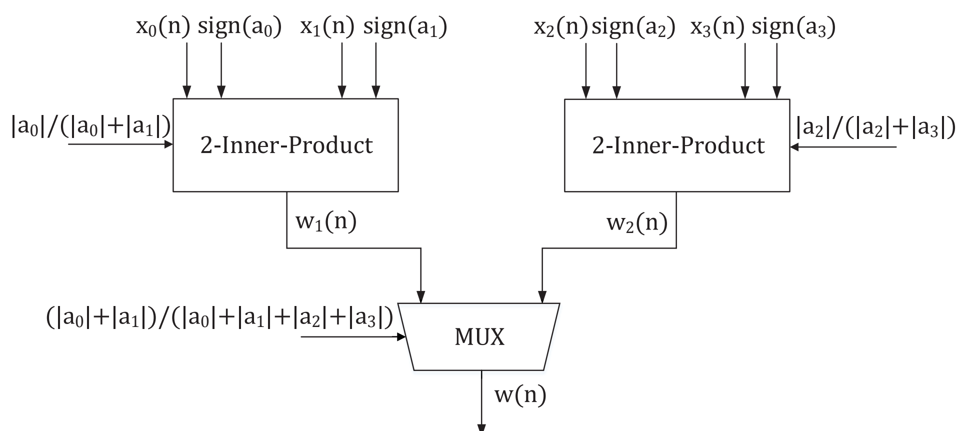


Figure 2.3: The architecture of a stochastic inner-product with input vector size of 4.

Consider the computation of the inner-product $< (a_0, a_1, a_2, a_3) \cdot (x_0(n), x_1(n), x_2(n), x_3(n)) >$. The internal nodes are described by:

$$\begin{cases} w_1(n) = \frac{1}{|a_0|+|a_1|}(a_0x_0(n) + a_1x_1(n)) \\ w_2(n) = \frac{1}{|a_2|+|a_3|}(a_2x_2(n) + a_3x_3(n)) \end{cases}$$

The final output is given by

$$w(n) = \frac{a_0x_0(n) + a_1x_1(n) + a_2x_2(n) + a_3x_3(n)}{|a_0| + |a_1| + |a_2| + |a_3|}$$

Notice that the output result is scaled by $\frac{1}{|a_0|+|a_1|+|a_2|+|a_3|}$.

In the second level of tree structure, we need to compute $((|a_0| + |a_1|), (|a_2| + |a_3|)) \cdot (w_1(n), w_2(n))$. Since the coefficients $(|a_0| + |a_1|)$ and $(|a_2| + |a_3|)$ are positive, the XNOR gates in the 2-input inner-product are not necessary. Therefore, only nodes at the first level of the tree require full implementation of 2-input inner-products. Other nodes can be implemented using single multiplexers.

2.1.2 Implementation Considerations for Stochastic Digital Filters

Trade-off in delay element implementations

In [30], stochastic FIR filters in direct-form were implemented using SC inner-product module based on two approaches. Fig. 2.4(a) shows one approach where the input signal $x(n)$ is first converted into a stochastic bit-stream, and then is passed through the delay line. In Fig. 2.4(b), the input signal first passes through the delay line, and then each signal from the delay line is converted separately to a stochastic bit sequence.

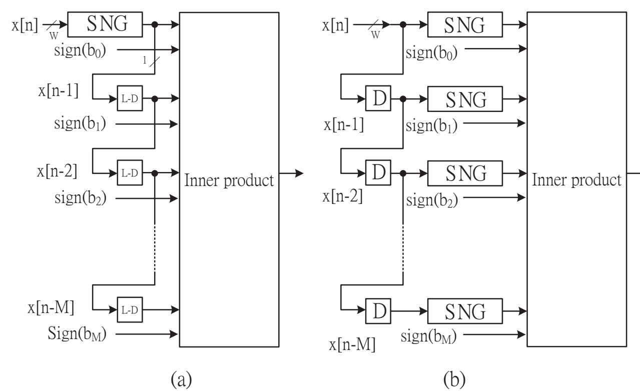


Figure 2.4: Two approaches to delaying input signals in stochastic digital filters: the input samples are delayed in (a) stochastic representation, (b) binary representation.

Table 2.1 shows synthesis results of two implementations for stochastic direct-form FIR filters. We assume that binary word-length is 10, whereas the length of stochastic

Table 2.1: Area consumption comparison of two implementations for stochastic direct-form FIR filters in terms of equivalent 2-input NAND gates.

Type of Implementations	Filter Order		
	2	4	6
2's complement	3243	6575	9147
Type-I	25761	51107	76450
Type-II	1453	2445	3762

sequence is 1024. The consumptions of area are given in terms of equivalent two input NAND gates in 65nm libraries. Type-I corresponds to the architecture in Fig. 2.4(a) and Type-II represents the architecture shown in Fig. 2.4(b). Type-I architecture leads to 10-fold increase in hardware complexity, compared to Type-II architecture, and it is even greater than traditional 2's complement filters. This fact suggests that in a feasible architecture of any kind of stochastic digital filters, signals should be stored in delay elements in 2's complement format even though more SNGs are required.

Hardware efficiency of stochastic digital signal processing system

In stochastic DSP implementations, the complexity of an addition, that is, the cost of a multiplexer containing SNGs, is significantly higher than that of an XNOR gate which implements a multiplication. Therefore, the optimization of stochastic filter architectures should focus on reducing the number of additions in a system.

2.2 Stochastic Lattice Implementation of Linear-phase FIR Filters

2.2.1 Lattice structure for FIR filters

Lattice structure for FIR filters can be derived using the Schur algorithm [34]. The Schur algorithm was originally used to test if a power series is analytic and bounded in the unit disk. If an Nth-order polynomial $\Psi_N(z)$ has all zeros inside the unit circle, $N+1$ polynomials $\Psi_i(z), i = N, N-1, \cdots, 0$ can be generated by the Schur algorithm. One of the most important properties of the Schur algorithm is that these $N+1$ polynomials

are orthogonal to each other and can be used as orthogonal basis functions to expand any N^{th} order polynomial. This orthogonality of the Schur algorithm has been exploited to synthesize various types of lattice filters. More details on using Schur algorithm to derive lattice structures can be found in Chapter 12 of [27].

A typical **basic** lattice stage for FIR filters is shown in Fig. 2.5. The

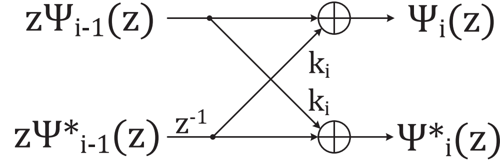


Figure 2.5: A typical basic lattice stage for FIR filters.

corresponding Schur polynomials are obtained by using the degree reduction procedure:

$$\Psi_{i-1}(z) = \frac{z^{-1}\{\Psi_i(z) - k_i\Psi_i^*(z)\}}{s_i}. \tag{2.1}$$

In equation (2.1), lattice coefficient k_i is given by $\Psi(0)/\Psi^*(0)$ and $\Psi_i^*(z)$ is the reverse polynomial of $\Psi_i(z)$. For basic lattice structure, $s_i = 1 - k_i^2$. The reverse polynomial of (2.1) is given by:

$$\Psi_{i-1}^*(z) = z^{i-1}\Psi_{i-1}(z^{-1}) = \frac{\Psi_i^*(z) - k_i\Psi_i(z)}{1 - k_i^2} \tag{2.2}$$

From equations (2.1) and (2.2), equations describing lattice stage shown in Fig. 2.5 are described as follows:

$$\begin{cases} \Psi_i(z) = z\Psi_{i-1} + k_i\Psi_{i-1}^*(z) \\ \Psi_i^*(z) = \Psi_{i-1}^*(z) + k_i z\Psi_{i-1}(z) \end{cases} \tag{2.3}$$

Equations (2.3) can also be transformed to the following equations based on equations (1) and (2):

$$\begin{cases} \Psi_i(z) = z\Psi_{i-1} + k_i\Psi_{i-1}^*(z) \\ \Psi_i^*(z) = (1 - k_i^2)\Psi_{i-1}^*(z) + k_i\Psi_i(z) \end{cases} \tag{2.4}$$

Then the basic lattice structure can be designed using an alternative implementation based on equations above as shown in Fig. 2.6.

A **normalized** lattice structure for FIR filters is derived by choosing s_i to be $\sqrt{1 - k_i^2}$ in Schur algorithm (2.1). Schur polynomials are denoted by Φ in normalized lattice structure. The Schur polynomial $\Phi_{i-1}(z)$ and reverse polynomial $\Phi_{i-1}^*(z)$

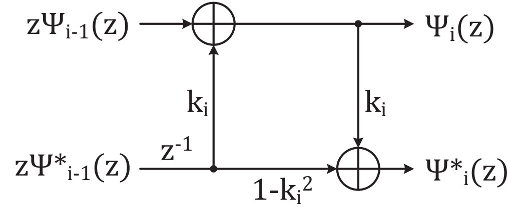


Figure 2.6: An alternative implementation of basic lattice stage for FIR filters.

are derived by degree reduction procedure as follows:

$$\Phi_{i-1}(z) = \frac{z^{-1}\{\Phi_i(z) - k_i\Phi_i^*(z)\}}{\sqrt{1-k_i^2}} \tag{2.5}$$

and

$$\Phi_{i-1}^*(z) = z^{i-1}\Phi_{i-1}(z^{-1}) = \frac{\Phi_i^*(z) - k_i\Phi_i(z)}{\sqrt{1-k_i^2}}. \tag{2.6}$$

Based on equations (2.5) and (2.6), equations to describe normalized lattice structures are derived in two formats. The first implementation is described by:

$$\begin{cases} \Phi_i(z) = \frac{1}{\sqrt{1-k_i^2}}(z\Phi_{i-1} + k_i\Phi_{i-1}^*(z)) \\ \Phi_i^*(z) = \frac{1}{\sqrt{1-k_i^2}}(\Phi_{i-1}^*(z) + k_i z\Phi_{i-1}(z)) \end{cases} \tag{2.7}$$

The corresponding lattice stage is shown in Fig. 2.7. The second implementation

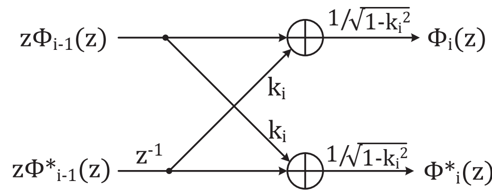


Figure 2.7: The implementation of normalized lattice stage corresponding to equations (2.7).

of normalized lattice structure is described as follows:

$$\begin{cases} \Phi_i(z) = \frac{1}{\sqrt{1-k_i^2}}(z\Phi_{i-1} + k_i\Phi_{i-1}^*(z)) \\ \Phi_i^*(z) = \sqrt{1-k_i^2}\Phi_{i-1}^*(z) + k_i\Phi_i(z) \end{cases} \tag{2.8}$$

The lattice stage corresponding to equations (2.8) is shown in Fig. 2.8.

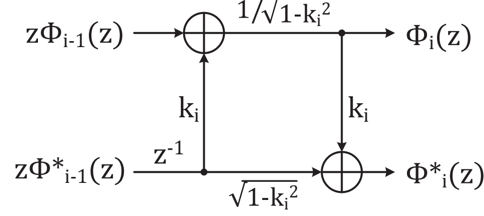


Figure 2.8: The implementation of normalized lattice stage corresponding to equations (2.8).

A **modified** lattice structure for FIR filters is derived by choosing s_i to be $1 - k_i$ in Schur algorithm (1) [35]. Schur polynomials are denoted by Θ in the modified lattice structure. The Schur polynomial $\Theta_{i-1}(z)$ and reverse polynomial $\Theta^*_{i-1}(z)$ are derived by degree reduction procedure as follows:

$$\Theta_{i-1}(z) = \frac{z^{-1}\{\Theta_i(z) - k_i\Theta^*_i(z)\}}{1 - k_i} \tag{2.9}$$

and

$$\Theta^*_{i-1}(z) = z^{i-1}\Theta_{i-1}(z^{-1}) = \frac{\Theta^*_i(z) - k_i\Theta_i(z)}{1 - k_i}. \tag{2.10}$$

Based on equations (2.9) and (2.10), equations to describe modified lattice structures are derived in two formats. The first implementation is described by:

$$\begin{cases} \Theta_i(z) = \frac{1}{1+k_i}(z\Theta_{i-1} + k_i\Theta^*_{i-1}(z)) \\ \Theta^*_i(z) = \frac{1}{1+k_i}(\Theta^*_{i-1}(z) + k_i z\Theta_{i-1}(z)) \end{cases} \tag{2.11}$$

The corresponding lattice stage is shown in Fig. 2.9.

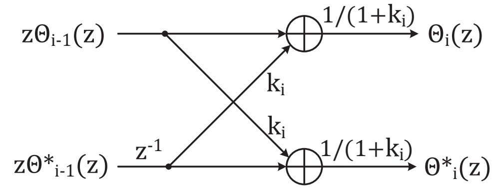


Figure 2.9: The implementation of modified lattice stage corresponding to equations (2.11).

The second implementation of modified lattice structure is described as follows:

$$\begin{cases} \Theta_i(z) = \frac{1}{1+k_i}(z\Theta_{i-1} + k_i\Theta^*_{i-1}(z)) \\ \Theta^*_i(z) = (1 - k_i)\Theta^*_{i-1}(z) + k_i\Theta_i(z) \end{cases} \tag{2.12}$$

The lattice stage corresponding to equations (2.12) is shown in Fig. 2.10.

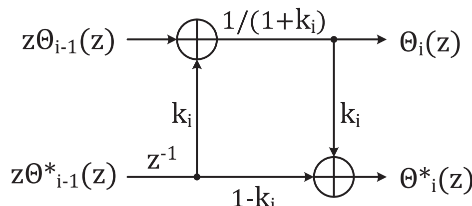


Figure 2.10: The implementation of modified lattice stage corresponding to equations (2.12).

2.2.2 Linear-phase lattice FIR filters

A typical basic lattice FIR filter is shown in Fig. 2.11. Notice that $2N$ adders are required for N-tap FIR lattice filter while a direct-form FIR filter with the same order has only N adders. It means the number of inner-products in stochastic lattice implementations is twice as that of stochastic direct-form FIR filters. Thus, hardware complexity and noise will increase due to the increase in the number of computations.

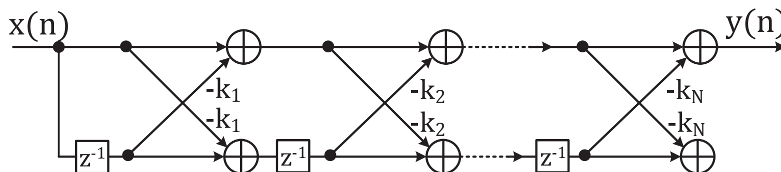


Figure 2.11: The block diagram of an N-tap FIR lattice filter.

However, lattice structure of linear phase FIR filters can be implemented with same number of computations as direct-form structure. Assume that k_i's represent coefficients in lattice structure. Directly applying coefficients-to-k-parameter algorithm [36] for linear-phase FIR filters leads to singularity which is caused by the symmetry of linear-phase FIR coefficients [28]. Fig. 2.12 shows an alternative approach to implementing lattice structure for an N-tap linear-phase FIR filter, where $L = \lfloor \frac{N+1}{2} \rfloor$ and $M = \lfloor \frac{N}{2} \rfloor$.

Assume the linear-phase FIR filter is described as $y(n) = b_0 x(n) + b_1 x(n-1) + \cdots + b_N x(n-N)$, where $b_i = b_{N-i}$. The key idea is applying Schur algorithm [34] only for $[b_0, b_1, \cdots, b_{\lfloor \frac{N+1}{2} \rfloor}]$ rather than all $N+1$ coefficients to avoid the singularity where $k_i = \pm 1$ (see [28] for detailed derivation). In Fig. 2.12, the basic lattice stage is

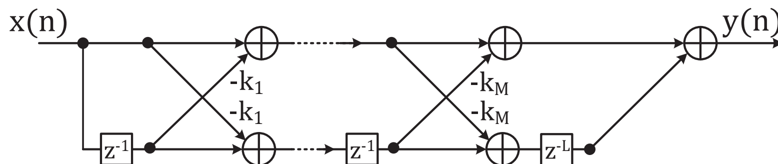


Figure 2.12: The block diagram of an N-tap linear-phase FIR lattice filter using *basic* lattice structure.

implemented using the circuit diagram shown in Fig. 2.5. Notice that multiple lattice stages have been proposed in the previous section. Therefore, as shown in Fig. 2.13, various types of lattice FIR filters can be implemented by replacing the basic lattice stage using structures from Fig. 2.6 to Fig. 2.10.

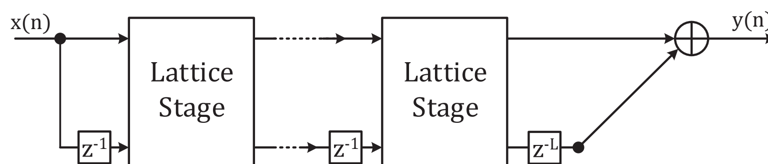


Figure 2.13: The block diagram of an N-tap linear-phase FIR lattice filter using arbitrary proposed lattice stage.

For linear-phase FIR filters, lattice coefficients k_i's may be greater than one since there are zeros located outside of the unit circle in z-plane as shown in Fig. 2.14. Notice that coefficients $\sqrt{1-k_i^2}$ and $1/\sqrt{1-k_i^2}$ in normalized lattice structure shown in Fig. 2.8 are complex numbers if $k_i > 1$. Therefore, this lattice structure is not suited for the implementation of linear-phase FIR filters.

2.2.3 Stochastic implementation of linear-phase lattice FIR filters

Consider implementing linear-phase lattice FIR filters using stochastic logic. Note that the output of a stochastic inner-product is implicitly scaled. A stochastic inner-product is implemented as shown in Fig. 2.15(a). Multiplications are performed by XNOR gates and a multiplexer is used to compute scaled addition. The select signal of the multiplexer is set to $\frac{|a|}{|a|+|b|}$. Additionally, $s(a)$ and $s(b)$ indicate signs of coefficients a and b. If a is negative, $s(a)$ will be set to 0. Otherwise, $s(a)$ is set to 1. The same holds true for $s(b)$. The output of the inner-product is given by

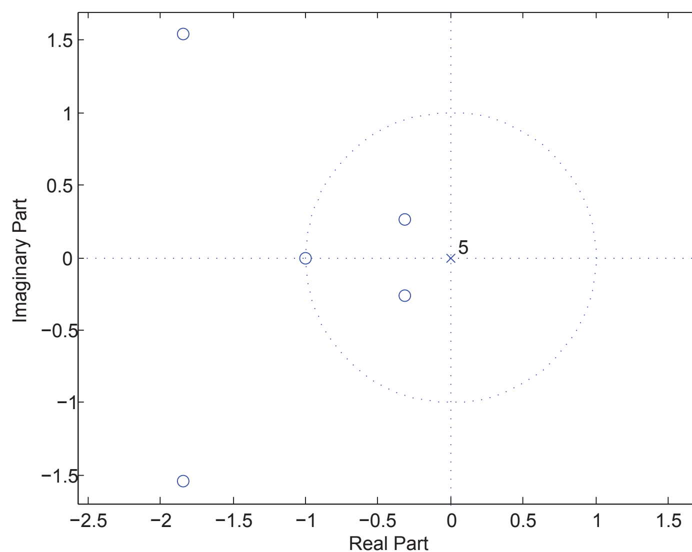


Figure 2.14: Zeros of $H(z) = 0.0264 + 0.1405z^{-1} + 0.3331z^{-2} + 0.3331z^{-3} + 0.1405z^{-4} + 0.0264z^{-5}$, which describes a linear-phase 5th-order FIR filter with cut-off frequency at 0.1π. The lattice coefficients are given by $k = [0.3909, \mathbf{12.6123}, 0, 0, 1]$.

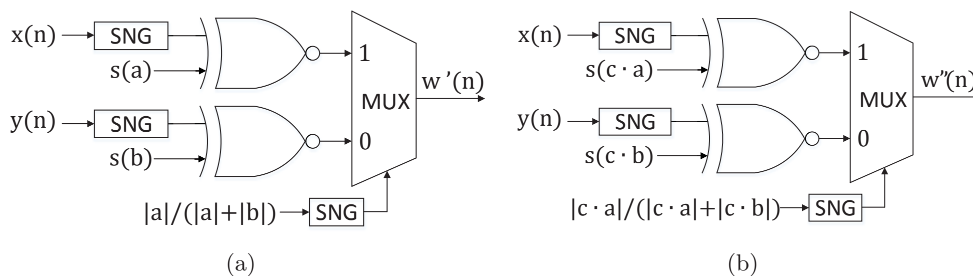


Figure 2.15: Stochastic inner-products to compute (a) $w_1(n) = ax(n) + by(n)$ and (b) $w_2(n) = c \cdot ax(n) + c \cdot by(n)$

$w'(n) = (ax(n) + by(n))/(|a| + |b|)$. Compared to the ideal output $w_1(n)$, the stochastic output is scaled by $|a| + |b|$. In Fig. 2.15(b), assume that the objective computation is $w_2(n) = c \cdot ax(n) + c \cdot by(n)$. The output of stochastic inner-product is given by

$$\begin{aligned} w''(n) &= \frac{c \cdot ax(n) + c \cdot by(n)}{|c \cdot a| + |c \cdot b|} \\ &= \frac{ax(n) + by(n)}{|a| + |b|} = w'(n). \end{aligned} \tag{2.13}$$

Therefore, equivalent scalings of two inputs for stochastic inner-product do not alter the computational result since only a *fractional* coefficient is required to determine the select signal of the multiplexer.

Comparing Fig. 2.5, Fig. 2.7 and Fig. 2.9, we observe that these three implementations of basic, normalized and modified lattice structures are similar, where the only difference is the scaling of outputs. Notice that top and bottom paths of these three lattice structures are equivalently scaled. Scaled results are used for inner-product computations at next level lattice stage. As compared in Fig. 2.15, equivalent scalings of two inputs for a stochastic inner-product do not alter the computational result. Therefore, lattice structures in Fig. 2.5, Fig. 2.7 and Fig. 2.9 lead to the same stochastic implementation of lattice FIR filter as shown Fig. 2.16. Notice that corresponding 2's complement implementation is shown in Fig. 2.12. XNOR gates perform multiplications and multiplexers perform scaled additions. The stochastic design computes a scaled result $y(n)/(2\prod_{i=1}^{m}(1+|k_i|))$. Coefficient $s(k_i)$ represents the sign of k_i. Full implementations of SC inner-products are not required since out of four coefficients in a lattice stage two are always unity. Stochastic-to-binary (S2B) modules [30] are used to convert stochastic bit-streams to binary numbers. The size of each delay element is determined by the word-length of 2's complement representation. All coefficients in the architecture are represented by stochastic sequences. Unlike stochastic lattice implementation of IIR filters [18], coefficients do not require extra scaling since computation results of top line and bottom line in a lattice stage are equivalently scaled by SC inner-product modules.

The normalized lattice structure shown in Fig. 2.8 is not suited for implementations of linear-phase FIR filters. We only consider lattice structures shown in Fig. 2.6 and Fig. 2.10. Compared to $\Psi_i(z)$ and $\Psi_i^*(z)$ of the basic lattice stage shown in Fig. 2.6,

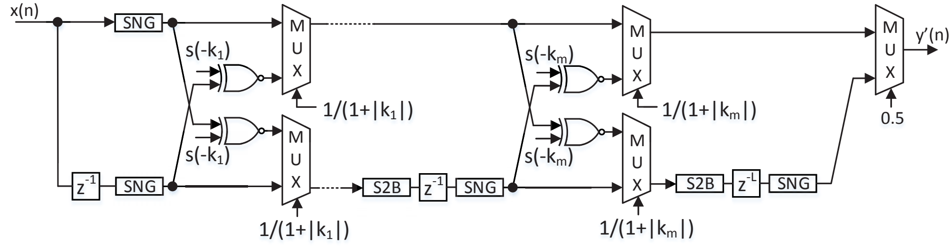


Figure 2.16: The architecture of a stochastic implementation for an N-tap linear-phase FIR lattice filter using lattice stages shown in Fig. 2.5, Fig. 2.7 and Fig. 2.9.

both $\Theta_i(z)$ and $\Theta_i^*(z)$ of the modified lattice stage shown in Fig. 2.10 are scaled by $1/(1+k_i)$. Therefore, these two lattice stage implementations lead to the same stochastic implementation of lattice FIR filter as shown in Fig. 2.17. The corresponding 2's complement implementations is shown in Fig. 2.18. Coefficient $s(k_i)$

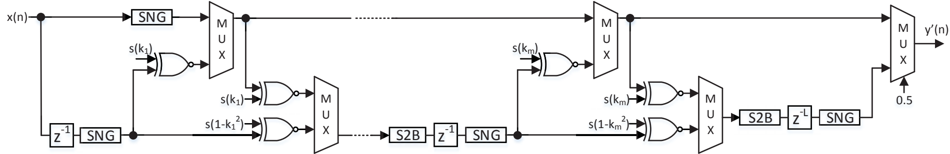


Figure 2.17: The architecture of a stochastic implementation for an N-tap linear-phase FIR lattice filter using lattice stages shown in Fig. 2.6 and Fig. 2.10.

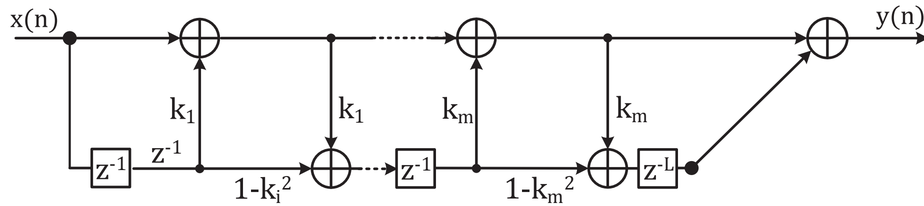


Figure 2.18: The 2's complement implementation corresponding to stochastic lattice FIR filter shown in Fig. 2.17.

represents the sign of k_i. The select signal of multiplexers in top path in the lattice stage is given by $1/(1+|k_i|)$ and the computational result is scaled by $1+|k_i|$. The select signal of multiplexers in bottom path is given by:

$$\frac{|k_i|}{\frac{|1-k_i^2|}{1+|k_i|}+|k_i|} \tag{2.14}$$

Notice that the first term of denominator is scaled by $1/(1+|k_i|)$ since the input generated from top path multiplexer is scaled by $1+|k_i|$. If $|k_i| \leq 1$, the computational result of bottom path multiplexer can be simplified based on (14) as follows:

$$\begin{aligned}\frac{\frac{1-k_i^2}{1+|k_i|}\Psi_{i-1}^* + k_i\frac{\Psi_i}{1+|k_i|}}{\frac{|1-k_i^2|}{1+|k_i|}+|k_i|} &= \frac{(1-k_i^2)\Psi_{i-1}^* + k_i\Psi_i}{|1-k_i^2|+|k_i|(1+|k_i|)} \\ &= \frac{(1-k_i^2)\Psi_{i-1}^* + k_i\Psi_i}{1+|k_i|}\end{aligned} \tag{2.15}$$

In this case, computational results of the top and bottom paths are equivalently scaled by $1/(1+|k_i|)$. If $|k_i| > 1$, the computational result of bottom path multiplexer is given by:

$$\begin{aligned}\frac{\frac{1-k_i^2}{1+|k_i|}\Psi_{i-1}^* + k_i\frac{\Psi_i}{1+|k_i|}}{\frac{|1-k_i^2|}{1+|k_i|}+|k_i|} &= \frac{(1-k_i^2)\Psi_{i-1}^* + k_i\Psi_i}{|1-k_i^2|+|k_i|(1+|k_i|)} \\ &= \frac{(1-k_i^2)\Psi_{i-1}^* + k_i\Psi_i}{2k_i^2+|k_i|-1}.\end{aligned} \tag{2.16}$$

The bottom path result is scaled by $1/(2k_i^2+|k_i|-1)$. Recall that the top path is scaled by $1/(1+|k_i|)$. To balance the scalings of two paths, an XNOR gate is required in the top path to perform a multiplication with $(1+|k_i|)/(2k_i^2+|k_i|-1)$. A large value of k_i leads to a large scaling, which degrades computation accuracy. Therefore, the implementation shown in Fig. 2.17 is suited for FIR lattice filters with small lattice coefficients.

2.3 Experimental Results

In this section, we present the experimental results for stochastic direct-form implementation and lattice implementation for linear-phase FIR filters. The metrics of measurement include accuracy, fault-tolerance performance, and hardware complexity.

2.3.1 Simulation Results

A truncated speech signal from ICA '99 Synthetic Benchmarks is used as the input signal. In our simulation, the length of the stochastic sequence is 1024 and the corresponding word-length of 2's complement representation is 10. A total of 5000 input

samples are used for simulation.

Table 2.2 shows the output signal-to-noise ratio (SNR) for various implementations of low-pass linear-phase FIR filters with different orders and cut-off frequencies. The conventional 2's complement implementations of linear FIR filters in direct-form and basic lattice structure are denoted by 2s-df and 2s-latc, respectively. The stochastic direct-form implementation is described by SC-df. The stochastic lattice implementation shown in Fig. 2.16 is denoted by SC-latc1 while SC-latc2 represents the design shown in Fig. 2.17. The SC-df is considered as a stochastic implementation in previous work [30] while SC-latc1 and SC-latc2 are the proposed stochastic implementations. From simulation results, we observe that the SC-df and SC-latc1 have equivalent performance. For several cut-off frequencies in high-order filters, the SC-df is slightly more accurate than the SC-latc1 since the output of SC-latc1 is implicitly scaled by $\prod_i(1+|k_i|)$ while no scaling is introduced for SC-df in our test. Notice that the outputs of a lattice stage in the SC-lat2 implementation is scaled by $1/(2k_i^2+|k_i|-1)$ for $|k_i|>1$, whereas the scaling of each lattice stage for the SC-latc1 implementation is given by $1/(1+|k_i|)$. The SC-latc1 outperforms the SC-latc2 implementation since SC-latc2 suffers from greater implicit scaling.

The traditional 2's complement implementations outperforms all stochastic implementations. This is not surprising, considering the random fluctuation in stochastic logic. Fig. 2.19 illustrates the spectrums of input and output signals obtained from stochastic and ideal implementations of filters. Stochastic lattice implementation-I corresponds to the design shown in Fig. 2.16 while stochastic lattice implementation-II corresponds to the design shown in Fig. 2.17.

It is known that accuracy of stochastic logic is influenced by the length of stochastic bit streams. In simulations above, the length is fixed at 10 bits. Fig. 2.20 shows output Mean Squared Error (MSE) and SNR of a specified filter with various sequence *lengths* for different stochastic implementations. The specified filter is a 3rd-order linear-phase FIR filters with cut-off frequency at 0.1π. We observe that filter performance is improved with the increase of stochastic sequence length. Compared to the stochastic lattice implementation I, MSE and SNR of lattice implementation II degrade faster with shorter sequence length. This is explained by the fact that the SC-latc2 implementation suffers more from implicit scaling of lattice stages than the SC-latc1 implementation.

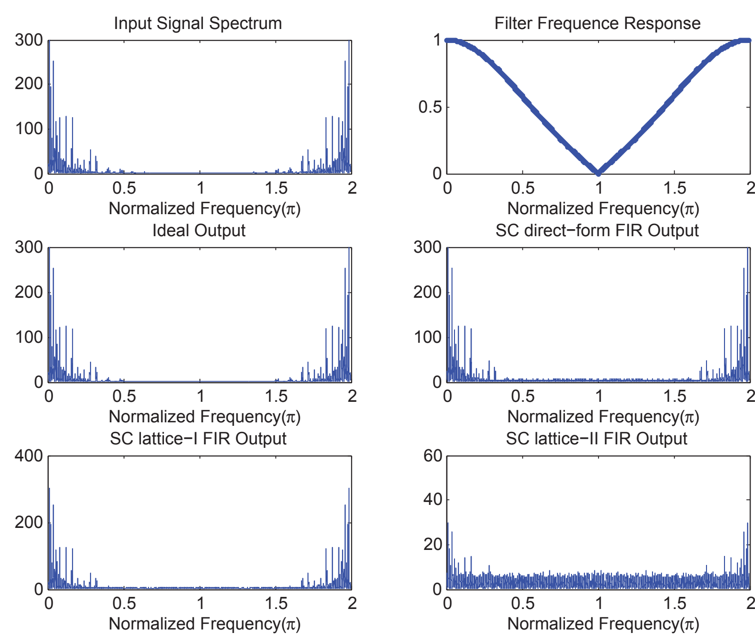


Figure 2.19: The spectrums of input signal, filter frequency response, ideal output, stochastic direct-form output, stochastic lattice-I output, and stochastic lattice-II output for filtering using a 3rd-order linear-phase FIR filters with cut-off frequency at 0.1π.

Table 2.2: The output SNR (dB) for linear-phase FIR filters of different orders and cut-off frequencies.

	Filter Order	Low-pass Cut-Off Frequency							
		0.1π	0.2π	0.3π	0.4π	0.5π	0.6π	0.7π	0.8π
2s-df	3	30.26	30.23	30.28	30.22	30.35	30.14	30.85	30.53
	5	26.69	26.89	26.74	26.92	27.04	26.98	27.27	26.83
	7	24.16	24.18	24.43	24.64	24.78	24.22	24.50	24.42
2s-latc	3	30.24	30.14	30.22	30.27	30.35	30.21	30.73	30.49
	5	26.52	26.68	26.72	26.87	27.04	26.65	27.13	26.62
	7	24.08	24.13	24.23	24.61	24.58	24.39	24.46	24.31
SC-df	3	15.80	15.80	16.02	16.24	16.38	16.48	16.56	16.03
	5	14.62	14.50	15.04	15.52	15.32	15.99	15.75	15.62
	7	14.12	14.28	14.61	14.66	14.35	14.58	14.41	14.44
SC-latc1	3	15.83	16.06	16.04	16.15	16.20	16.74	16.48	16.21
	5	14.71	14.91	14.96	15.54	15.44	15.41	15.39	15.21
	7	13.97	14.35	14.56	14.32	13.81	13.73	13.60	13.78
SC-latc2	3	14.74	14.88	14.59	14.81	14.78	14.80	14.65	14.33
	5	13.18	13.40	13.61	13.68	13.36	13.22	13.57	13.78
	7	12.34	13.29	13.57	13.17	13.57	13.78	13.77	13.81

Due to the extra scaling, the SC-latc2 implementation is more sensitive to the resolution of stochastic sequences, which is determined by the length of bit streams.

2.3.2 Synthesis Results

Synthesis results of stochastic FIR filters are evaluated using 65nm technology. The architectures are synthesized using Synopsys Design Compiler. We also compare hardware complexity between traditional binary implementations and stochastic implementations. In stochastic implementations, the length of stochastic sequences is 1024, and binary numbers in traditional implementations require 10 bits. Table 4.5 shows hardware complexity, power consumption and critical path of binary and stochastic implementations for linear phase FIR filters.

The results show that stochastic implementations require less hardware resources than traditional binary implementation due to the low cost of arithmetic units. Compared with the previous stochastic direct-form implementation, the proposed stochastic lattice implementations consume less hardware resources. Comparing Fig. 2.4(b)

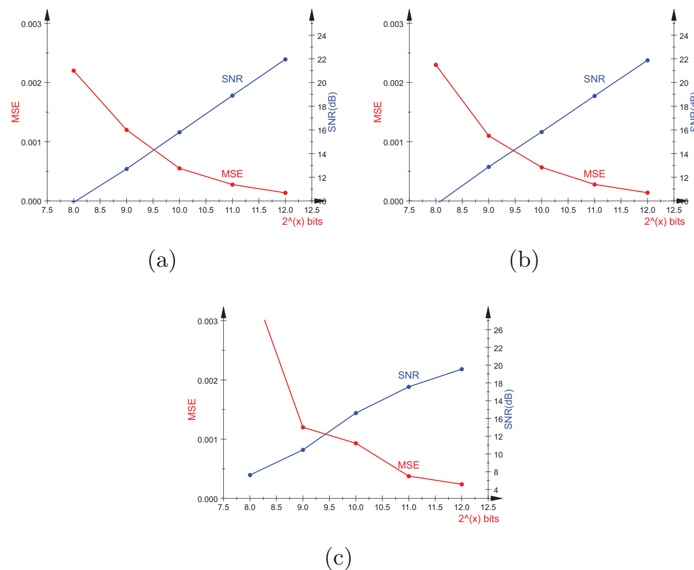


Figure 2.20: Output Mean Squared Error (MSE) and SNR of a specified filter with various sequence lengths for (a) stochastic direct-form implementation, (b) stochastic lattice implementation I, and (c) stochastic lattice implementation II.

and Fig. 2.16, we can observe that for an N-tap linear-phase FIR filter, a stochastic direct-form implementation requires N 2-input inner-products while a stochastic lattice implementation requires $(2 \cdot \lfloor \frac{N}{2} \rfloor + 1)$ 2-input inner-product. The hardware complexities of the additions in two implementations are about the same. However, there are N SNG modules in Type-II stochastic direct-form implementation, whereas stochastic lattice implementation requires $(\lfloor \frac{N}{2} \rfloor + 1)$ SNG modules. Compared to SNG modules, hardware complexity of a S2B module can be ignored. Therefore, the low hardware complexity of stochastic lattice implementation is explained by the reduction of the number of SNG modules. The SC-latc2 implementation slightly costs more hardware than the SC-latc1 due to several extra XNOR gates.

In general, the critical path delay of stochastic implementations is less than traditional 2's complement implementations. However, the proposed lattice implementations involve longer critical path than the stochastic direct-form implementation. The power consumption of the proposed stochastic lattice implementations is less than the previous stochastic direct-form implementation. The proposed stochastic lattice implementation also consumes less power than the traditional 2's complement lattice implementation.

Table 2.3: The hardware complexity (in terms of equivalent 2-input NAND gates), power consumption and critical path delay of various implementations of linear phase FIR filters.

		Filter Order		
		3	5	7
2s-df	Area	4573	7941	10593
	Power (μW)	9.55	13.65	17.40
	Critical Path (ns)	10.04	11.18	11.79
2s-latc	Area	3755	7295	10063
	Power (μW)	9.40	17.27	25.02
	Critical Path (ns)	9.30	13.76	14.78
SC-df	Area	2091	3193	4186
	Power (μW)	10.63	16.14	20.17
	Critical Path (ns)	2.87	3.30	4.15
SC-latc1	Area	1848	2716	3566
	Power (μW)	9.01	13.13	17.24
	Critical Path (ns)	5.34	6.03	6.30
SC-latc2	Area	1857	2735	3596
	Power (μW)	9.13	13.21	17.36
	Critical Path (ns)	5.36	6.04	6.32

However, in general, the improvement of power consumption from stochastic implementations is limited, compared to conventional 2's complement implementations. Considering the long latency of stochastic implementations (1024 clock cycles in our test), the stochastic logic is not an ideal low power/energy solution.

2.3.3 Fault Tolerance Analysis

We performed fault-tolerance test for both stochastic FIR filters by randomly injecting bit-flipping error at all internal nodes and measuring the corresponding output error-to-signal power ratio for each implementation. Real speech signals from ICA '99 Synthetic Benchmarks are used as the test inputs. The length of the stochastic sequence is 1024. A total of 5000 input samples are used. We control the level of injected soft error by flipping certain percent bits of all internal computational nodes in circuits. Flipped bits are selected at random.

A 7-tap linear-phase FIR filter with cut-off frequency at 0.1π is considered. The signals at marked nodes in Fig. 2.21 are flipped for a specified percent at random. A total of 14 internal nodes are considered in traditional binary and stochastic direct-form implementations. For stochastic lattice implementations I and II, 12 and 15 internal computational nodes are considered, respectively. Table 2.4 and Fig. 2.22 present output error-to-signal power ratios due to bit-flipping.

Table 2.4: The output SNR (dB) due to random bit-flipping for different implementations for a 7th-order low-pass linear-phase FIR filter with cut-off frequency 0.1π.

Type of Implementations	Percentage of Bit-flipping					
	0%	0.01%	0.05%	0.1%	0.5%	1%
traditional binary direct-form	24.20	9.82	2.64	-0.38	-7.06	-9.55
traditional binary binary lattice	24.08	9.68	2.51	-0.46	-7.11	-9.79
stochastic direct-form	13.33	13.33	13.32	13.31	13.12	12.68
stochastic lattice-I	13.28	13.26	13.22	13.21	12.49	10.86
stochastic lattice-II	13.19	13.16	13.09	12.72	11.87	10.15

In this simulation, it is shown that bit-flipping almost has no impact on the output accuracy of stochastic direct-form and lattice implementations when flipping percentage is under 0.5%. Starting with 0.01% bit-flipping, the performance of the traditional binary implementation is degraded significantly due to random bit-flippings while the performance of stochastic implementations remains stable. Notice that in our test the word-length of binary design is 10-bit and the length of stochastic bit streams is 2^{10}. The result of fault-tolerance test may vary for different word-lengths.

2.4 Conclusion

We investigate the implementation of linear-phase FIR digital filters in stochastic logic. Two novel architectures of stochastic linear-phase FIR filter based on lattice structures have been presented. Basic, normalized and modified lattice structures are considered for the stochastic implementation. Compared with the previous stochastic

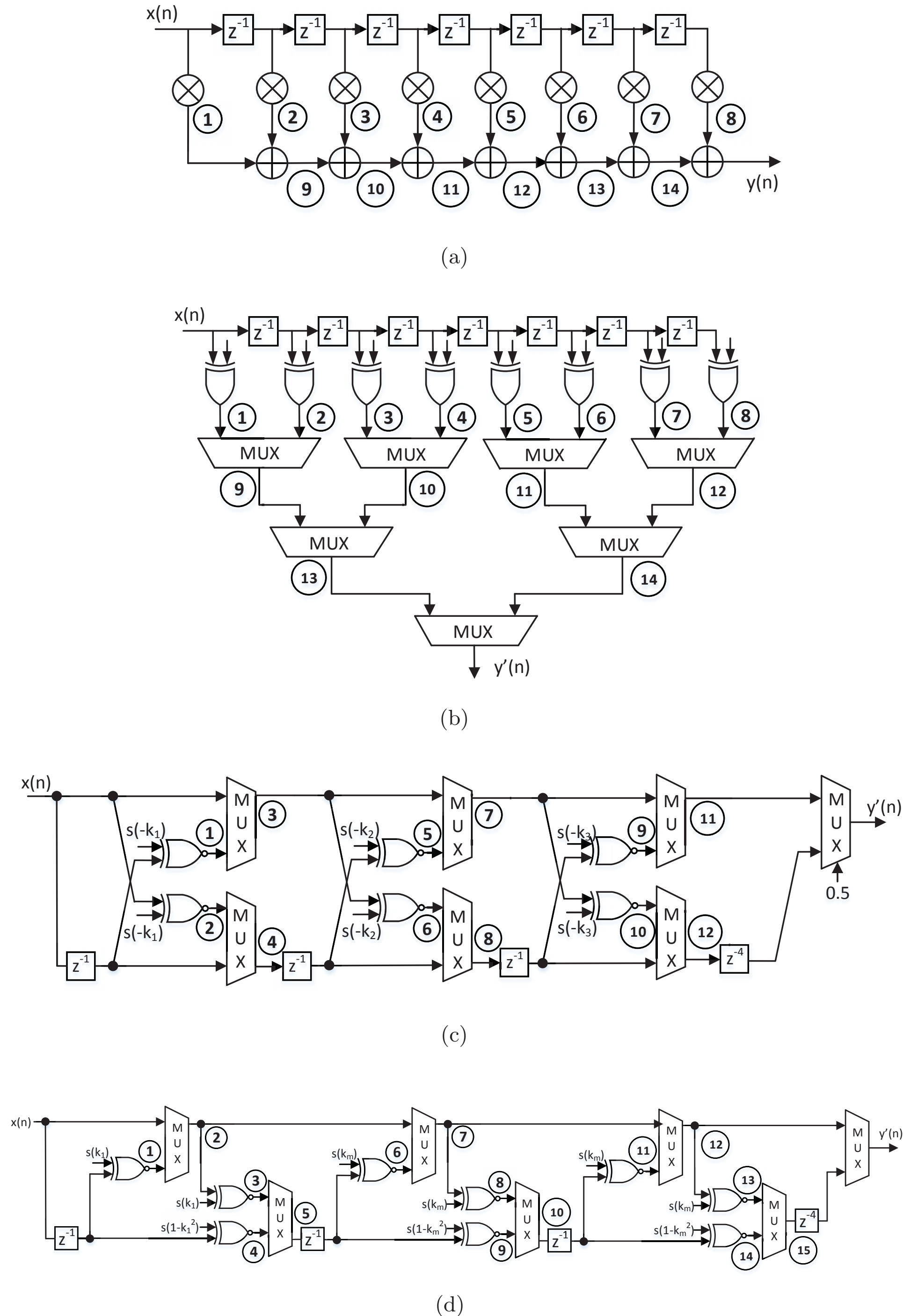


Figure 2.21: The architectures for (a) traditional binary FIR filter, (b) stochastic direct-form FIR filter, (c) stochastic lattice implementation-I, and (d) stochastic lattice implementation-II of linear-phase FIR filter, where random bit-flippings occur at the nodes marked (SNG and S2B modules are not shown in this figure).

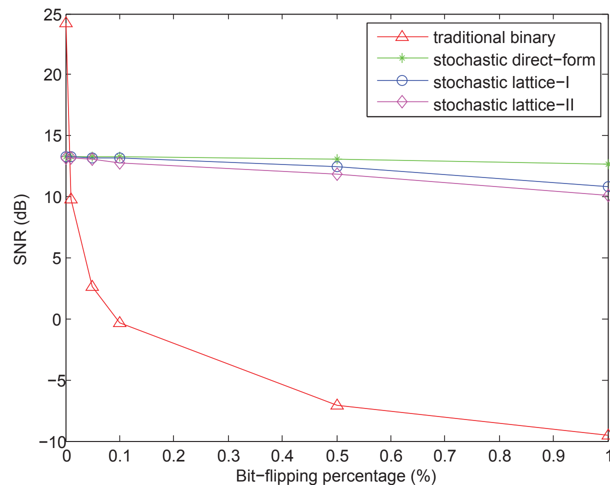


Figure 2.22: Fault-tolerance test results of different implementations for a 7th-order low-pass linear-phase FIR filters with cut-off frequency 0.1π.

implementation of FIR filters in direct-form, the proposed lattice implementations can obtain equivalent performance and involve less hardware complexity. The power consumption of stochastic implementation is also reduced by the proposed architectures. However, the critical path delay of the proposed implementations is greater than that of stochastic implementation in direct-form.

Compared stochastic implementations with conventional binary implementations, the hardware complexity and critical path delay are reduced. The stochastic implementations also show significantly better fault-tolerance than conventional binary implementations. The limitation of this work is consistent with inherent drawbacks of stochastic logic. Due to random fluctuation in computations, the performance of s-tochastic implementations is worse than that of conventional binary implementations. Additionally, the stochastic implementation is not an ideal low power/energy solution for FIR filter design, compared to the traditional binary implementation.

Chapter 3

IIR Digital Filters in Stochastic Logic

Stochastic computing implicitly assumes the independence of the input signals in arithmetic functions. As signals get correlated, the error in stochastic computing increases [37]. This limits the utility of stochastic computing for signal processing systems. The correlation in FIR digital filters increases with filter order; thus higher-order filters may require higher number of bits to represent numbers [20]. However, the feedback in IIR digital filters continuously propagates the signal correlation [38]. The error in stochastic computing implementations of these filters is so large that these filters compute functionally incorrect outputs, and fail to filter the input signal as desired. Implementation of simple first and second order IIR filters using stochastic logic was presented in [39]. However, the pole location was restricted to be very close to unity. Implementation of arbitrary IIR digital filters using stochastic logic with acceptable accuracy was considered impossible. In this chapter, we present the stochastic implementation of arbitrary IIR filters based on basic, normalized and modified lattice structures.

3.1 Stochastic State-space Implementation for IIR Digital Filters

3.1.1 Background of Lattice IIR Filter

Lattice digital filters have good numerical properties since the denominator of a given transfer function is synthesized in a robust way. Fig. 3.1 illustrates an example of the most commonly-used lattice structure, the basic lattice IIR filter. Lattice IIR filters can be derived using Schur algorithm [34]. The Schur algorithm was originally used to test if a power series in analytic and bounded in the unit disk. If an Nth-order polynomial $\Phi_N(z)$ has all zeros inside the unit circle, $N+1$ polynomials $\Phi_i(z), i = N, N-1, \cdots, 0$ can be generated by the Schur algorithm. One of the most important properties of the Schur algorithm is that these $N+1$ polynomials are orthogonal to each other and can be used as orthogonal basis functions to expand any N^{th} order polynomial. This orthogonality of the Schur algorithm has been exploited to synthesize various types of lattice filters. More details on using Schur algorithm to derive lattice structures can be found in Chapter 12 of [27].

Stochastic IIR filters can be implemented using inner-products similar to stochastic FIR filters. This approach leads to stochastic direct-form IIR filters [30]. However, states in direct-form IIR filters are highly correlated and the signal correlation is continuously propagated by the feedback. These filters compute functionally incorrect outputs, and fail to filter the input signal as desired. Therefore, we propose to implement stochastic IIR filters based on lattice structures, where states are orthogonal.

As it is mentioned in the introduction part, two approaches to implementing s-tochastic IIR digital filters are considered. This section describes the first approach to implementing the stochastic IIR filters by describing basic lattice structure in a state-space form.

3.1.2 An Example of 3rd-order Stochastic IIR Filter with State-Space Description

The IIR filter transfer function is first mapped to a basic lattice filter using the Schur algorithm [34] and the polynomial expansion algorithm. Then the lattice filter is

described by a state-space description.

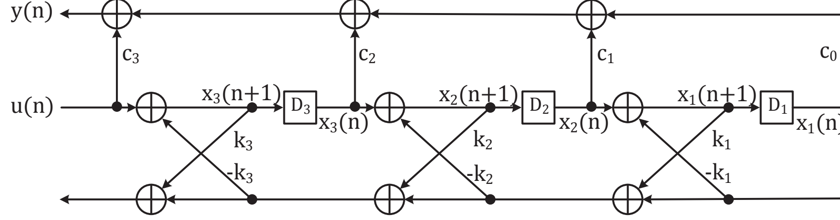


Figure 3.1: The 3rd-order basic lattice filter structure.

Consider the 3rd-order lattice IIR filter shown in Fig. 3.1. It can be described by the state-space description:

$$\boldsymbol{x}(n+1) = \boldsymbol{A}\boldsymbol{x}(n) + \boldsymbol{b}u(n), \tag{3.1}$$

$$y(n) = \boldsymbol{c}^T\boldsymbol{x}(n) + du(n). \tag{3.2}$$

The boldfaced letters imply a vector or a matrix. In the above representation, $\boldsymbol{x}$ is the state vector, u is the input and y is the output. From the filter diagram shown in Fig. 3.1, we can express $\boldsymbol{x}(n+1)$ and $y(n)$ in terms of $\boldsymbol{x}(n)$ and $u(n)$:

$$\begin{cases} x_1(n+1) = x_2(n) - k_1 \cdot x_1(n) \\ x_2(n+1) = x_3(n) - k_1k_2 \cdot x_2(n) - k_2(1-k_1^2) \cdot x_1(n) \\ x_3(n+1) = u(n) - k_2k_3 \cdot x_3(n) - k_1k_3(1-k_2^2) \cdot x_2(n) \\ \qquad\qquad +k_3(1-k_1^2)(1-k_2^2) \cdot x_1(n) \end{cases}$$

$$y(n) = c_0x_1(n) + c_1x_2(n) + c_2x_3(n) + c_3u(n).$$

Parameters $\boldsymbol{A}$, $\boldsymbol{b}$, $\boldsymbol{c}$ and d can be obtained by expressing above equations in a matrix form as:

$$\boldsymbol{A} = \begin{bmatrix} -k_1 & 1 & 0 \\ -k_2(1-k_1^2) & -k_1k_2 & 1 \\ k_3(1-k_1^2)(1-k_2^2) & -k_1k_3(1-k_2^2) & -k_2k_3 \end{bmatrix}$$

$$\boldsymbol{b} = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}, \qquad \boldsymbol{c} = \begin{bmatrix} c_0 \\ c_1 \\ c_2 \end{bmatrix}, \qquad d = c_3$$

The state-space description can be mapped to a stochastic logic architecture using stochastic inner-product modules to implement equations (3.1) and (3.2). The architecture is shown in Fig. 3.2.

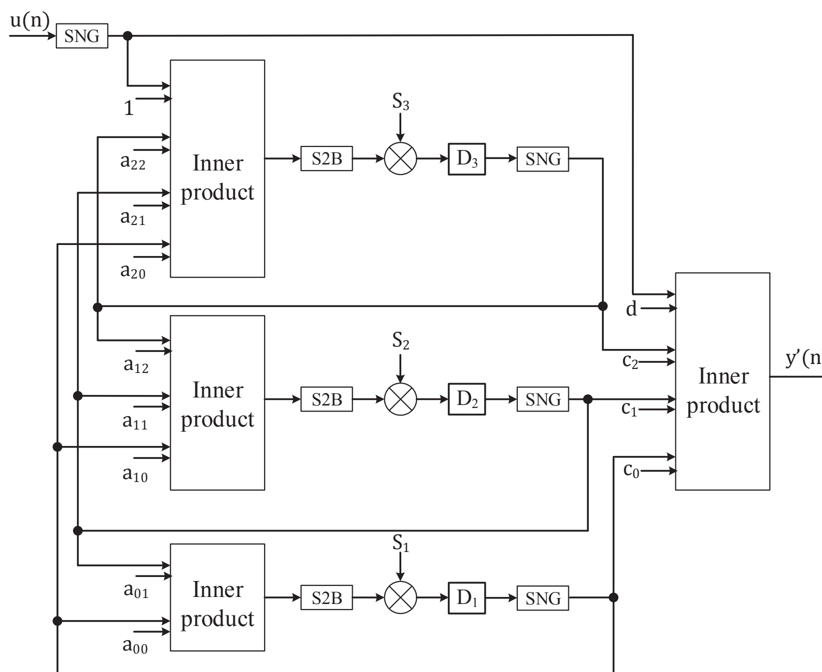


Figure 3.2: The circuit diagram of a 3rd-order stochastic IIR lattice filter with state space implementation.

The stochastic N^{th}-order filter computes the scaled result $y(n)/(\sum_{i=0}^{N-1}|c_i| + |d|)$. Stochastic number generators (SNG) are used to generate stochastic sequences which are input to stochastic inner-product modules. Coefficients a_{ij}, b_i and c_i represent entries in matrix $\boldsymbol{A}$, vectors $\boldsymbol{b}$ and $\boldsymbol{c}$, respectively. Notice that all internal states $\boldsymbol{x}(n)$ are fed-back for computing the output. These outputs are scaled by the reciprocal of the scale factor of the inner product module, denoted as S_i given by $\sum_{j=0}^{N-1}|a_{ij}| + |b_i|$ (see Fig. 3.2). Since S_i can be greater than one, and multiplication by a number larger than one cannot be implemented using stochastic logic, the three scale multipliers are implemented using binary multipliers. The inner product output needs to be converted back to a binary number using a stochastic-number-to binary-number converter (S2BC).

After binary scaling multiplication, the result is converted to a stochastic sequence again using an SNG.

3.1.3 The Lattice-to-State-Space Algorithm for Arbitrary Order IIR Filters

The Lattice-to-State-Space (L2SS) algorithm is an iterative procedure for transferring coefficients in lattice structure (k_i and c_i in Fig. 3.1) to parameters in state-space description ($\boldsymbol{A}$, $\boldsymbol{b}$, $\boldsymbol{c}$ and d in equations (3.1) and (3.2)). It is obvious that for N^{th}-order basic lattice IIR filters,

$$\boldsymbol{b} = \begin{bmatrix} 0 \\ 0 \\ \vdots \\ 1 \end{bmatrix}, \qquad \boldsymbol{c} = \begin{bmatrix} c_0 \\ c_1 \\ \vdots \\ c_{N-1} \end{bmatrix}, \qquad d = c_N$$

The L2SS algorithm computes $\boldsymbol{A}$ in two steps. In the first step, use $x_i(n+1)$ to represent the ith entry of $\boldsymbol{x}(n+1)$ in equation (3.1). For $i = 2, 3, \cdots, N-1$, $x_i(n+1)$ can be expressed in terms of $x_1(n+1), x_2(n+1), \cdots, x_{i-1}(n+1)$, $x_1(n)$ and $x_{i+1}(n)$ as follows,

$$x_i(n+1) = -k_i x_1(n) + x_{i+1}(n) - (\sum_{j=1}^{i-1} k_i k_j x_j(n+1)). \tag{3.3}$$

Besides,

$$x_1(n+1) = -k_1 x_1(n) + x_2(n) \tag{3.4}$$

and

$$x_N(n+1) = -k_N x_1(n) + u(n) - (\sum_{j=1}^{N-1} k_N k_j x_j(n+1)). \tag{3.5}$$

Notice that $x_i(n+1)$ is represented using $x_1(n+1), x_2(n+1), \cdots, x_{i-1}(n+1)$ in a recursive manner. In Fig. 3.3, the objective state $x_i(n+1)$ and the data path to compute $x_i(n+1)$ are labeled with red color. Coefficients in equation (3.3) can be written in vector form as:

$$\boldsymbol{B_i} = [-k_i, 1, -k_i k_1, -k_i k_2, \cdots, -k_i k_{i-1}],$$

where $i \in [2, N-1]$. The length of $\boldsymbol{B_i}$ is $i+1$. In order to place these $(N-2)$ $\boldsymbol{B_i}$'s in an $N \times N$ square matrix $\boldsymbol{B}$, pad with 0's at the end of each $\boldsymbol{B_i}$ whose length is less than N. Thus, length of $\boldsymbol{B_i}$ is extended to N and

$$\boldsymbol{B} = \begin{bmatrix} \boldsymbol{B_1} \\ \boldsymbol{B_2} \\ \vdots \\ \boldsymbol{B_N} \end{bmatrix},$$

where

$$\boldsymbol{B_1} = [-k_1, 1, 0, \cdots, 0],$$

and

$$\boldsymbol{B_N} = [-k_N, 1, -k_N k_1, -k_N k_2, \cdots, -k_N k_{N-2}].$$

Notice that $(-k_i k_{N-1})$, the last entry of $\boldsymbol{B_N}$, is removed to ensure the length of $\boldsymbol{B_N}$ to be N. It will be considered at the end of the algorithm (the last line labeled with * in **Algorithm 1**).

In the second step, starting with equation (3.4), recursively substitute $x_1(n+1), x_2(n+1), \cdots, x_{i-1}(n+1)$ on the right sides of equations (3.3) and (3.5) with $x_1(n), x_2(n), \cdots, x_N(n)$ from $i = 1$ to N. Thus, $\boldsymbol{B}$ will be converted to parameter $\boldsymbol{A}$, where

$$\boldsymbol{A} = \begin{bmatrix} \boldsymbol{A_1} \\ \boldsymbol{A_2} \\ \vdots \\ \boldsymbol{A_N} \end{bmatrix},$$

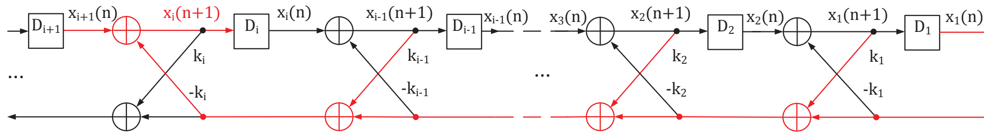


Figure 3.3: The data path to compute internal state $x_i(n+1)$.

The L2SS algorithm is described by the pseudo-code shown in **Algorithm 1**. Instead of calculating parameters in state-space description manually, the L2SS algorithm makes it possible to automatically calculate parameters for arbitrary filter order by a computer program.

Algorithm 1 The L2SS Algorithm.

Initialization:

  for $i = 1 \to N$ **and** $j = 1 \to N$ **do**
    $A_{ij} = 0$; $B_{ij} = 0$;
  end for
  $A_{11} = -k_1$, $A_{12} = 1$; $B_{11} = -k_1$, $B_{12} = 1$;

Step 1:

  for $i = 2 \to N$ **and** $j = 3 \to (i + 1)$ **do**
    $B_{i1} = -k_i$, $B_{i2} = 1$;
    if $j \leq N$ **then**
      $B_{ij} = -k_i k_{j-2}$;
    end if
  end for

Step 2:

  for $i = 2 \to N$ **do**
    for $j = 1 \to (i - 1)$ **do**
      if $(j + 2) \leq N$ **then**
        $\boldsymbol{A_i} = \boldsymbol{A_i} + \boldsymbol{A_j} B_{ij+2}$;
      end if
    end for
    $A_{i1} = A_{i1} + B_{i1}$;
    if $(i + 1) \leq N$ **then**
      $A_{ii+1} = 1$;
    end if
  end for
  $\boldsymbol{A_N} = \boldsymbol{A_N} - k_N k_{N-1} \boldsymbol{A_{N-1}}$; *

3.2 Stochastic Lattice Implementation

This section describes a second approach to implementing the stochastic IIR digital filter by transforming the basic lattice structure to an equivalent form. Consider the basic lattice filter shown in Fig. 3.1. This filter cannot be mapped directly to a stochastic implementation, as stochastic logic implicitly computes scaled inner product outputs.

A typical lattice stage is described in Fig. 3.4(a). Every inner-product (multiple multiply-accumulate) stage needs an implicit scale factor for stochastic implementation. The typical lattice stage is transformed into an equivalent stage shown in Fig. 3.4(b). Here the Schur polynomial in the top path is the same for both structures; however, the reverse Schur polynomial for the bottom path in the stochastic implementation is a scaled version of the original structure. This can be observed from the lattice equations below.

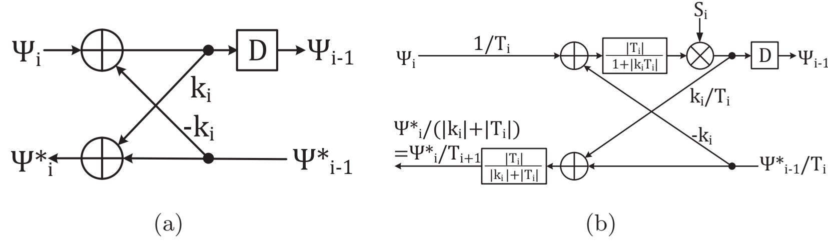


Figure 3.4: (a) Original 2's complement implementation of the basic lattice module, and (b) stochastic implementation of the basic lattice module.

The basic lattice stage is described by:

$$\begin{cases} \Psi_{i-1} = \Psi_i - k_i \Psi_{i-1}^* \\ \Psi_i^* = \Psi_{i-1}^* + k_i \Psi_{i-1} \end{cases}$$

The stochastic lattice stage is described by:

$$\begin{cases} \Psi_{i-1} = \frac{\Psi_i \frac{1}{T_i} - k_i \frac{\Psi_{i-1}^*}{T_i}}{(1+|k_i T_i|)/|T_i|} \cdot S_i \\ \frac{\Psi_i^*}{T_{i+1}} = \frac{\frac{\Psi_{i-1}^*}{T_i} + \frac{k_i}{T_i}\Psi_{i-1}}{1+\frac{k_i}{T_i}} = \frac{\Psi_i^*}{|T_i|+|k_i|} \end{cases}$$

From these equations, we obtain the scaling multiplier S_i and the reverse Schur polynomial scale factor T_i as:

$$\begin{cases} S_i = 1 + |k_i||T_i| \\ T_{i+1} = |T_i| + |k_i| \end{cases}$$

Notice that T_1 is always 1.

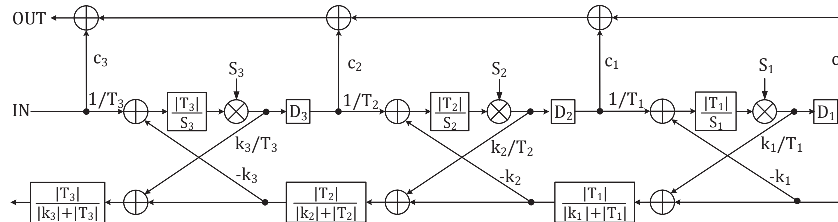


Figure 3.5: The transformed basic lattice filter structure to be used for stochastic implementation.

An example of a 3rd-order lattice IIR filter transformed for stochastic implementation is shown in Fig. 3.5. Notice that the lower left output is not used and thus there is no need to calculate it in a real implementation. This output is described in Fig. 3.5 to maintain consistency among lattice structures.

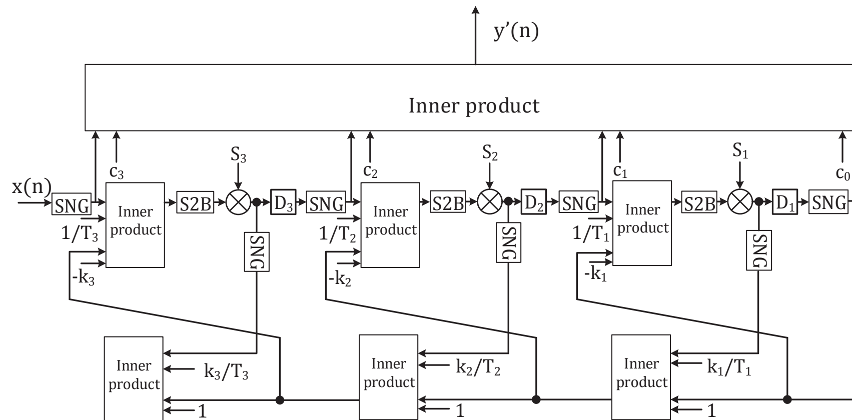


Figure 3.6: The circuit diagram of a 3rd-order stochastic IIR lattice filter with lattice module implementation.

Fig. 3.6 shows the architecture of a 3rd-order stochastic IIR lattice filter using s-tochastic inner products. These inner products implicitly compute scaled outputs. $x(n)$ is the input of the filter and the output of the Nth-order filter is the scaled result $y(n)/(\sum_{i=0}^{N} |c_i|)$. The numbers stored in delay elements are in 2's complement format. Similar to state-space implementation, the stochastic lattice implementation requires N binary multipliers.

3.3 Normalized Stochastic Lattice IIR Digital Filters

In previous two sections, two approaches to implementing stochastic lattice IIR filters were introduced. However, both of them are based on the *basic* lattice structure, where the power of states may be greater than unity power. Notice that in bipolar format, the stochastic representation is bounded by [-1, 1]. Thus, states of basic lattice structure may overflow. In this section, we introduce two approaches to implement *normalized* stochastic lattice IIR filters. First, we apply scaling operation to the state-space implementation for basic stochastic lattice IIR filters. Scaling operation constrains internal signal to unit power without altering filter transfer function by readjusting certain internal gain parameters. Note that scaling the state-space implementation for basic lattice IIR filters leads to normalized lattice structure for IIR filters [40]. This fact implicitly suggests the second implementation approach that we can directly start with normalized lattice IIR filters and transform them to equivalent stochastic structures that can exploit novel SC inner-products mentioned before. States in both approaches are orthonormal.

3.3.1 State-Space Implementation with Scaling

Scaling is a process of readjusting internal gain parameters to constrain internal signals to a desired range while maintaining the same filter transfer function. Fig. 3.7 illustrates scaling operation on a filter with transfer function

$$H(z) = D(z) + F(z)G(z).$$

Fig. 3.7(a) describes the original filter without scaling. To scale the node x, we divide $F(z)$ by T and multiply $G(z)$ by the same number as shown in Fig. 3.7(b). Although the transfer function does not change by this operation, the signal power at node x has been changed. We use l_2 scaling to achieve states with unity power. More details about scaling of state-space description can be found in Section 11.4 of [27]. The state-space description after scaling is described as

$$\begin{aligned} \boldsymbol{x_s}(n+1) &= \boldsymbol{A_s}\boldsymbol{x_s}(n) + \boldsymbol{b_s}u(n), & (3.6)\\ y(n) &= \boldsymbol{c_s}^T\boldsymbol{x_s}(n) + d_s u(n). & (3.7) \end{aligned}$$

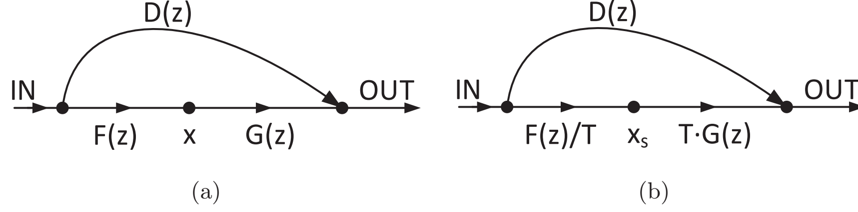


Figure 3.7: (a) A filter with unscaled node x, and (b) a filter with scaled node x_s.

The normalized stochastic lattice IIR filters are implemented by mapping scaled state-space description to a stochastic logic architecture using stochastic inner-product modules to implement equations (3.6) and (3.7). The architecture is the same as that shown in Fig. 3.2. The only difference is that coefficients a_{ij}, b_i and c_i represent entries in matrix $\boldsymbol{A_s}$, vectors $\boldsymbol{b_s}$ and $\boldsymbol{c_s}$, respectively, rather than entries in matrix $\boldsymbol{A}$, vectors $\boldsymbol{b}$ and $\boldsymbol{c}$.

3.3.2 Stochastic Lattice Implementation

The second approach to implementing normalized stochastic lattice IIR filters is to transform the normalized lattice structure to an equivalent stochastic version. It is similar to the method described in Section 3.2, where the differences involve different lattice structures and scaling factors introduced by stochastic inner-product modules. An example of a 3rd-order normalized lattice filter is shown in Fig. 3.8.

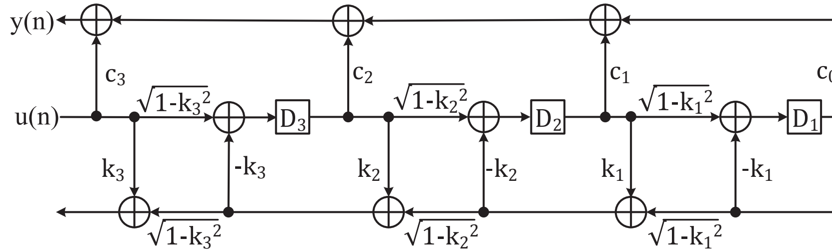


Figure 3.8: The 3rd-order normalized lattice filter structure.

When mapping a normalized lattice IIR filter to a stochastic logic, we need to consider that stochastic logic implicitly computes scaled inner product outputs. A typical normalized lattice stage is described in Fig. 3.9(a). Each inner-product stage requires an implicit scale factor for stochastic implementation. The typical lattice stage is transformed into an equivalent stage shown in Fig. 3.9(b). The Schur polynomial in

the top path is the same for both structures; however, the reverse Schur polynomial for the bottom path in the stochastic implementation is a scaled version of the original structure. This can be observed from the lattice equations below. The typical

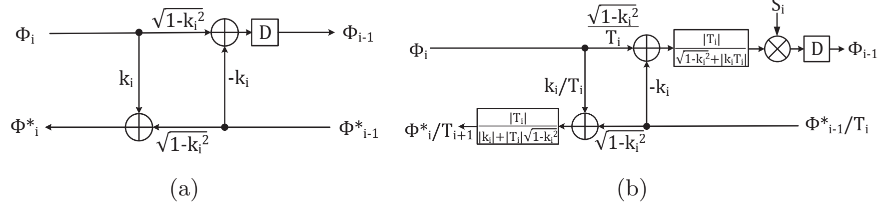


Figure 3.9: (a) Original 2's complement implementation of the normalized lattice structure, and (b) stochastic implementation of the normalized lattice structure.

normalized lattice stage is described by:

$$\begin{cases} \Phi_{i-1} = \sqrt{1-k_i^2}\Phi_i - k_i\Phi_{i-1}^* \\ \Phi_i^* = \sqrt{1-k_i^2}\Phi_{i-1}^* + k_i\Phi_i \end{cases}$$

The normalized stochastic lattice stage is described by:

$$\begin{cases} \Phi_{i-1} = \frac{\frac{\sqrt{1-k_i^2}}{T_i}\Phi_i - k_i\frac{\Phi_{i-1}^*}{T_i}}{(\sqrt{1-k_i^2}+|k_iT_i|)/|T_i|} \cdot S_i \\ \frac{\Phi_i^*}{T_{i+1}} = \frac{\sqrt{1-k_i^2}\frac{\Phi_{i-1}^*}{T_i} + \frac{k_i}{T_i}\Phi_i}{\sqrt{1-k_i^2}+\frac{k_i}{T_i}} = \frac{\Phi_i^*}{|T_i|\sqrt{1-k_i^2}+|k_i|} \end{cases}$$

From these equations, we obtain the scaling multiplier S_i and the reverse Schur polynomial scale factor T_i as:

$$\begin{cases} S_i = \sqrt{1-k_i^2} + |k_i||T_i| \\ T_{i+1} = |T_i|\sqrt{1-k_i^2} + |k_i| \end{cases}$$

Notice that T_1 is always 1.

Fig. 3.10 shows the architecture of a 3rd-order normalized stochastic lattice IIR filter using stochastic inner products. These inner products implicitly compute scaled outputs. $x(n)$ is the input of the filter and the output of the Nth-order filter is the scaled result $y(n)/(\sum_{i=0}^{N}|c_i|)$. Similar to previous implementations, the stochastic lattice implementation requires N binary multipliers. Compared to the stochastic implementation of the basic lattice IIR filter shown in Fig. 3.6, the normalized stochastic lattice IIR filter requires less SNG modules. In Fig. 3.6, outputs of binary multipliers and delay elements

are in 2s complement representation and all of them need to be converted to stochastic bit-streams for computations. However, in Fig. 3.10 only outputs of delay elements need to be converted to stochastic bit-streams using SNGs. It explains why the normalized stochastic lattice IIR filter requires less SNG modules.

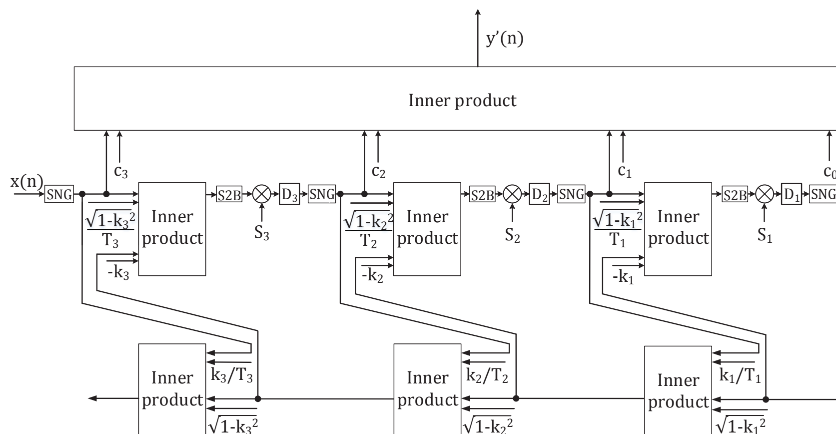


Figure 3.10: The circuit diagram of normalized 3rd-order stochastic IIR lattice filter with lattice module implementation.

3.4 Optimized Stochastic Lattice IIR Filters

Stochastic implementations of both basic and normalized N^{th}-order lattice IIR filters require N binary multipliers. In this section, we propose an approach to reduce the number of *binary* multipliers in *stochastic lattice implementation* for basic and normalized lattice IIR filters.

First we focus on the basic lattice structure. A stage of a basic stochastic lattice filter without binary multiplier is shown in Fig. 3.11. Compared to the basic stochastic lattice stage shown in Fig. 3.4(b), not only the reverse Schur polynomial for the bottom path, but also the Schur polynomial in the top path is a scaled version of the original structure, since no binary multiplier is used.

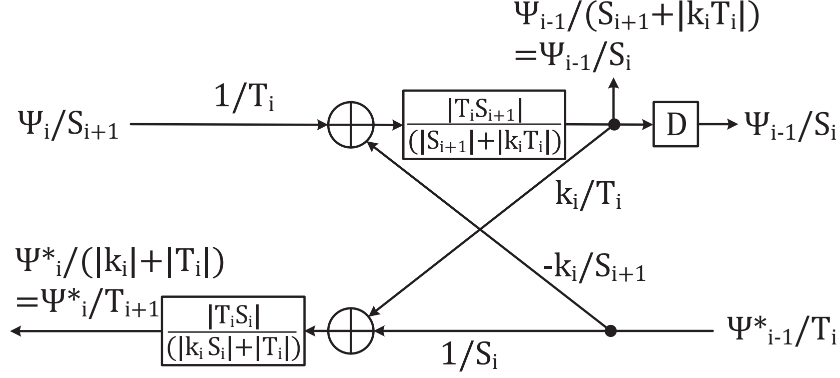


Figure 3.11: The implementation of stochastic basic lattice stage without binary multiplier.

The stochastic lattice stage shown in Fig. 3.11 is described by

$$\begin{cases} \frac{\Psi_{i-1}}{S_i} = \frac{\frac{1}{T_i}\frac{\Psi_i}{S_{i+1}} - \frac{k_i}{S_{i+1}}\frac{\Psi^*_{i-1}}{T_i}}{(|S_{i+1}|+|k_iT_i|)/|T_iS_{i+1}|} \\ \qquad = \frac{\Psi_{i-1}}{|S_{i+1}|+|k_iT_i|} \\ \frac{\Psi^*_i}{T_{i+1}} = \frac{\frac{1}{S_i}\frac{\Psi^*_{i-1}}{T_i} + \frac{k_i}{T_i}\frac{\Psi_{i-1}}{S_i}}{|T_i|+|k_iS_i|/|S_iT_i|} \\ \qquad = \frac{\Psi^*_i}{|T_i|+|k_iS_i|} \end{cases}$$

From these equations, we obtain the Schur polynomial scale factor S_i and the reverse Schur polynomial scale factor T_i as:

$$\begin{cases} S_i = |S_{i+1}| + |k_iT_i| \\ T_{i+1} = |T_i| + |k_iS_i| \end{cases} \tag{3.8}$$

Notice that $T_1 = S_1$ and $S_{N+1} = 1$, where N is the filter order.

An example of 3rd-order lattice IIR filter transformed for stochastic implementation with reduced number of binary multipliers is shown in Fig. 3.12. Notice that three binary multipliers are used in the structure in Fig. 3.5, whereas only one binary multiplier is required in the optimized implementation. The binary multiplier is denoted by $Scale_1$.

Filter coefficients in Fig. 3.12 are described by unfolding equations (3.8) as follows:

$$S_1 = 1 \tag{3.9}$$

$$S_2 = |S_3| + |k_2T_2| = S_3 + |k_2|T_2 \tag{3.10}$$

$$S_3 = 1 + |k_3T_3| = 1 + |k_3|T_3, \tag{3.11}$$

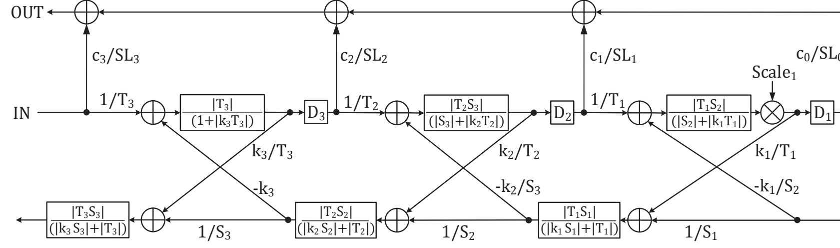


Figure 3.12: The transformed basic lattice filter structure using one binary multiplier for stochastic implementation.

and

$$T_1 = S_1 = 1 \quad (3.12)$$

$$T_2 = |T_1| + |k_1 S_1| = 1 + |k_1| \quad (3.13)$$

$$T_3 = |T_2| + |k_2 S_2| = T_2 + |k_2| S_2, \quad (3.14)$$

where S_i and T_i are all positive numbers. Since in this specified example, one binary multiplier is implemented in the first lattice stage to offset the scaling of output of stochastic inner-product, we obtain $S_1 = 1$ in equation (3.9). Solving equations (3.9) $\sim$ (3.14), we achieve values of S_i and T_i:

$$\begin{cases} S_1 = 1 \\ S_2 = \frac{c}{1-|k_2 k_3|} \\ S_3 = 1 + |k_3| + |k_1 k_3| + \frac{c|k_2 k_3|}{1-|k_2 k_3|} \end{cases}$$

and

$$\begin{cases} T_1 = 1 \\ T_2 = 1 + |k_1| \\ T_3 = 1 + |k_1| + \frac{c|k_2|}{1-|k_2 k_3|} \end{cases}$$

where $c = 1 + |k_2| + |k_3| + |k_1 k_2| + |k_1 k_3|$. The binary multiplier $Scale_1$ at the first lattice stage is given by:

$$Scale_1 = |S_2| + |k_1 T_1| = |S_2| + |k_1|.$$

The scale factors for the feed-forward path. SL_i's are given by:

$$\begin{cases} SL_i = \prod_{\substack{k=1\\k\neq i+1}}^{N} S_k, & 0 \leqslant i \leqslant N-1 \\ SL_i = \prod_{k=1}^{N} S_k, & i = N \end{cases}$$

where N is the filter order.

Fig. 3.13 shows the architecture of an optimized 3rd-order stochastic lattice IIR filter. $x(n)$ is the input of the filter and the output of the Nth-order filter is the scaled result:

$$\begin{aligned} y'(n) &= \frac{x(n)\frac{c_3}{SL_3} + \frac{D_3}{S_3}\frac{c_2}{SL_2} + \frac{D_2}{S_2}\frac{c_1}{SL_1} + \frac{D_1}{S_1}\frac{c_0}{SL_0}}{|\frac{c_3}{SL_3}| + |\frac{c_2}{SL_2}| + |\frac{c_1}{SL_1}| + |\frac{c_0}{SL_0}|} \\ &= \frac{x(n)\frac{c_3}{S_1S_2S_3} + \frac{D_3}{S_3}\frac{c_2}{S_1S_2} + \frac{D_2}{S_2}\frac{c_1}{S_1S_3} + \frac{D_1}{S_1}\frac{c_0}{S_2S_3}}{|\frac{c_3}{S_1S_2S_3}| + |\frac{c_2}{S_1S_2}| + |\frac{c_1}{S_1S_3}| + |\frac{c_0}{S_2S_3}|} \\ &= \frac{x(n)c_3 + D_3c_2 + D_2c_1 + D_1c_0}{|c_3| + |c_2S_3| + |c_1S_2| + |c_0S_1|} \\ &= \frac{y(n)}{|c_3| + |c_2S_3| + |c_1S_2| + |c_0S_1|} \end{aligned}$$

States stored in delay elements are converted to 2's complement representation using S2B modules to avoid long stochastic sequence.

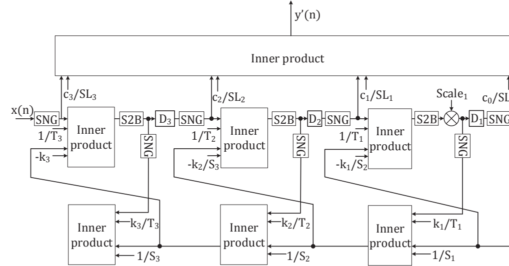


Figure 3.13: The circuit diagram of optimized implementation for 3rd-order stochastic IIR lattice filter.

Note that it is impossible to implement a stochastic basic lattice IIR filter without any binary multiplier. Consider the transformed 3rd-order basic lattice IIR filter shown in Fig. 3.12. Assuming that no binary multiplier is implemented, from equations (8), we obtain $S_1 = |S_2| + |k_1T_1|$ instead of $S_1 = 1$. Then equations (9) – (14) are changed

to:

$$
\begin{aligned}
S_1 &= |S_2| + |k_1 T_1| = S_2 + |k_1| T_1 &\quad (3.15)\\
S_2 &= |S_3| + |k_2 T_2| = S_3 + |k_2| T_2 &\quad (3.16)\\
S_3 &= 1 + |k_3 T_3| = 1 + |k_3| T_3, &\quad (3.17)
\end{aligned}
$$

and

$$
\begin{aligned}
T_1 &= S_1 = S_2 + |k_1| T_1 &\quad (3.18)\\
T_2 &= |T_1| + |k_1 S_1| = T_1 + |k_1| S_1 &\quad (3.19)\\
T_3 &= |T_2| + |k_2 S_2| = T_2 + |k_2| S_2, &\quad (3.20)
\end{aligned}
$$

where S_i and T_i are all positive numbers. Given an example of a 3rd-order IIR filter with transfer function:

$$H(z) = \frac{0.0029(1 + 3z^{-1} + 3z^{-2} + z^{-3})}{1 - 2.374z^{-1} + 1.929z^{-2} - 0.532z^{-3}}.$$

The k-parameters in this lattice filter are computed as

$$k_1 = -0.9743, k_2 = 0.9293, k_3 = -0.532. \quad (3.21)$$

Substitute k_i in equations (15) – (20) using (21) and solve these equations. We get:

$$
\begin{cases} S_1 = -0.3482 \\ S_2 = -0.0089 \\ S_3 = 0.6299 \end{cases}
\qquad
\begin{cases} T_1 = -0.3482 \\ T_2 = -0.6874 \\ T_3 = -0.6957 \end{cases}
$$

which contradict the fact that $S_i > 0$ and $T_i > 0$, as S_i and T_i are sums of absolute values as defined in equations (8). It indicates that the stochastic lattice filter without any binary multiplier is infeasible.

From our experiments, it is observed that every 3 lattice stages require *at least* one binary multiplier, which ensures the convergence of coefficients S_i and T_i. Therefore, for an Nth-order basic lattice IIR filter, the optimized stochastic implementation requires$\lceil \frac{N}{3} \rceil$ binary multipliers. For example, at least one binary multiplier is required for 3rd-order stochastic basic lattice IIR filters and two binary multipliers are necessary if the filter order is 6. This observation is proved as follows: An arbitrary N^{th}-order

lattice IIR filter can be considered as a cascade of $\lfloor \frac{N}{3} \rfloor$ 3-stage lattice blocks and one m-stage lattice block as shown in Fig. 3.14. Depending on the filter order, m may have different values from 1 to 3. Assume that one binary multiplier is implemented at first stage of each 3-stage lattice block, then $S_{3i+1} = 1$, where $0 \leq i \leq \lfloor \frac{N}{3} \rfloor$. Thus, each 3-stage lattice block is in the same situation as a 3rd-order optimized stochastic lattice IIR filter which has been implemented successfully. Since an Nth-order lattice IIR filter consists of $\lfloor \frac{N}{3} \rfloor$ 3-stage lattice blocks and one m-stage lattice block, we can demonstrate that $\lceil \frac{N}{3} \rceil$ binary multipliers are required for a feasible stochastic implementation.

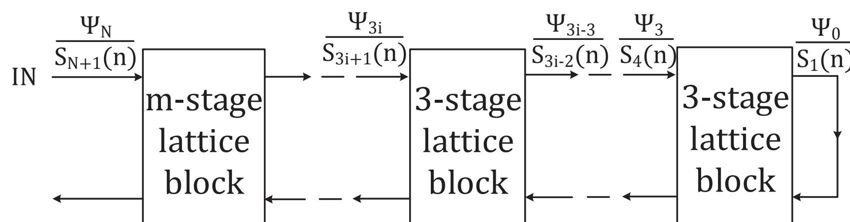


Figure 3.14: An Nth-order lattice IIR filter consisting of $\lfloor \frac{N}{3} \rfloor$ 3-stage lattice blocks and one m-stage lattice block.

Compared to the original stochastic implementation, the number of binary multipliers in the optimized implementation is reduced by 2/3.

The optimized stochastic *normalized* lattice IIR filters are derived by using the same procedure as we design the optimized stochastic *basic* lattice IIR filters. Starting with the architecture in Section 3.3.2, we can reduce the number of binary multipliers by two-third. For a feasible N^{th}-order stochastic normalized lattice IIR filters, also $\lceil \frac{N}{3} \rceil$ binary multipliers are required.

3.5 The Stochastic Implementation of Normalized Lattice IIR Filter Using Traditional SC Units

3.5.1 Inner-product for stochastic IIR filter design

Our previous implementations of stochastic IIR filters are based on the inner-product with a coefficients-based scaling method, where the scaling factor of the stochastic inner-product is determined by input coefficients. In a stochastic lattice IIR filter design using

this SC inner-product, binary multipliers are required to offset the scaling (see [30]). However, the scaling factor of a traditional stochastic adder is the fixed probability 0.5 for various input coefficients. If we implement the inner-product module in a stochastic lattice IIR filter using traditional stochastic multipliers and adders, the outputs of inner-products will be scaled by 0.5 independent of the filter coefficients. Thus, binary multipliers can be replaced by simple 1-bit left-shift operations. Fig. 3.15 shows the stochastic inner-products implemented using the coefficients-based scaling method and traditional SC units with offsets. The ideal output is represented by $w(n) = ax(n) + by(n)$. The $s(a)$ and $s(b)$ stand for the signs of coefficients a and b. $w'(n)$ and $w''(n)$ describe the scaled outputs of two stochastic inner-product implementations, respectively, where

$$w'(n) = \frac{ax(n) + by(n)}{2}$$

and

$$w''(n) = \frac{ax(n) + by(n)}{|a| + |b|}.$$

Note that to offset the implicit scaling introduced by stochastic adder, a 1-bit left-shift is required as shown in Fig. 3.15(a), whereas a binary multiplication of $(|a| + |b|)$ is required in Fig. 3.15(b). Before the binary multiplication or left-shift, the stochastic sequence needs to be converted to 2's complement number using a stochastic-to-binary (S2B) converter, which is not shown in this figure.

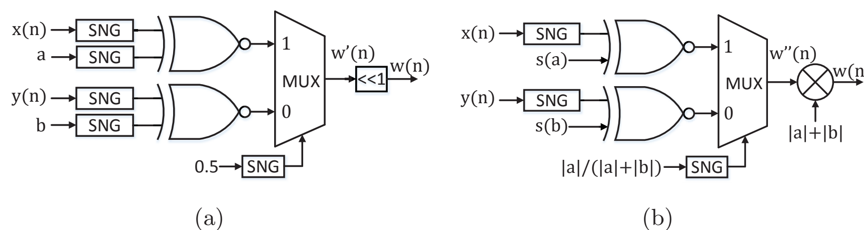


Figure 3.15: The stochastic inner-products implemented using (a) traditional SC units, and (b) the coefficients-based scaling method. (S2B modules are not shown in this figure.)

3.5.2 The Stochastic Implementation of Normalized Lattice IIR Filters

In our previous work [38], the stochastic IIR filter is designed using the *basic* lattice structure, where the power of states may be greater than unity. Notice that in the

bipolar format, the stochastic representation is bounded by $[-1, 1]$. Thus, the states of a basic lattice structure may overflow. However, the states of a *normalized* lattice IIR filter are *orthonormal*. The orthonormality guarantees the powers of all states are unity in the mean squared sense. It implicitly constrains the boundary of stochastic representation since the probability of state overflow is significantly reduced. Therefore, we implement stochastic IIR filters using the normalized lattice structure.

The mapping of a single lattice stage from 2's complement implementation to a stochastic implementation using traditional SC units is straightforward as shown in Fig. 3.16. The normalized lattice stage is described by:

$$\begin{cases} \Phi_{i-1} = \sqrt{1-k_i^2}\Phi_i - k_i\Phi^*_{i-1} \\ \Phi^*_i = \sqrt{1-k_i^2}\Phi^*_{i-1} + k_i\Phi_i \end{cases}$$

For both top and bottom paths, the outputs of stochastic inner-products are converted to 2's complement numbers using S2B modules, and then left-shifted by one bit to offset the scaling of stochastic adder. The SNG modules are used to convert 2's complement numbers back to stochastic sequences. The filter coefficients $\pm k_i$, $\sqrt{1-k_i^2}$ and select signal 0.5 are all represented as stochastic numbers. Compared to previous designs, no binary multiplier is required in this implementation while an extra S2B-SNG pair is required for the bottom path. The frequency rate between the clock generating the stochastic sequences and the clock controlling the delay element is 1024:1 since the length of stochastic bit streams in our test is 1024.

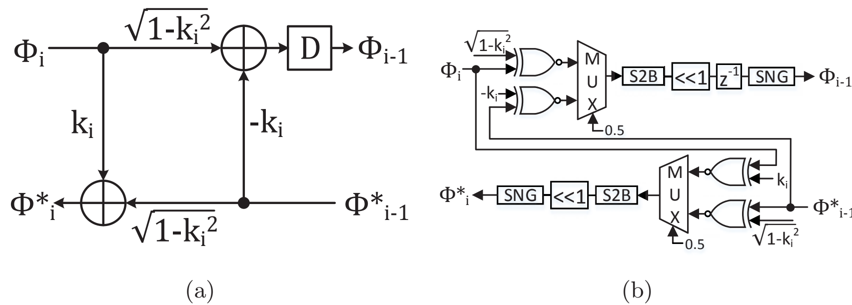


Figure 3.16: (a) The 2's complement implementation of a lattice stage, and (b) stochastic implementation of a lattice stage using traditional SC units.

Fig. 3.17 shows the architecture of a 3rd-order normalized stochastic IIR lattice filter using traditional SC units. $x(n)$ is the input of the filter. Notice that the feed-forward

part is implemented using the inner-product with scaling. Hence, the output for N^{th}-order filter is the scaled result $y(n)/(\sum_{i=0}^{N} c_i)$, where $y(n)$ is the filter output and c_i's represent the coefficients to implement the numerator of the transfer function as shown in Fig 3.8. The numbers stored in delay elements in Fig. 3.17 are in 2's complement format to reduce hardware complexity [20]. Note that no binary multiplier is required in this design.

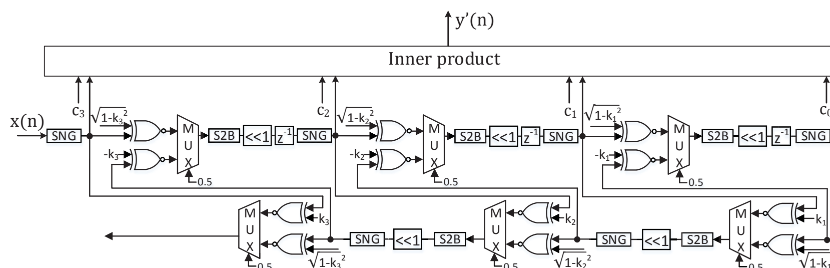


Figure 3.17: The circuit diagram of the stochastic implementation for a 3rd-order normalized lattice IIR filter using traditional SC units.

3.5.3 Analysis of Hardware Complexity

The area consumptions of proposed stochastic architectures for normalized lattice IIR filters using traditional SC units are evaluated using 65nm technology. The architectures are implemented using 65nm libraries and synthesized using Synopsys Design Compiler. In the stochastic implementation, the length of the stochastic sequence is 1024 and the corresponding 2's complement representation requires 10 bits, for same resolution. In this section, we analyze the area for different parts to optimize hardware complexity.

Table 3.1: The area of each single module in a stochastic lattice stage in terms of equivalent 2-input NAND gates.

Modules	S2B	SNG	Delay	MUX+XNOR
Area	149	187	130	12

As shown in Fig. 3.17, the area consumption of a stochastic lattice IIR filter comes from delay elements, computational units implemented with combinational logic and

S2B-SNG pairs. The area of each single module in a stochastic lattice stage shown in Fig. 3.16 is given in Table 3.1 in terms of equivalent 2-input NAND gates. Table 3.2 presents hardware composition of the architecture shown in Fig. 3.17.

Table 3.2: The area composition of the stochastic implementation for a 3rd-order normalized lattice IIR filter using traditional SC units in terms of equivalent 2-input NAND gates.

	Total	S2B-SNG pairs	Delay	Data path
Area	2342 (100%)	1874 (80.02%)	361 (15.41%)	107 (4.57%)

From the results, we can see that 80% hardware resources are consumed by S2B-SNG converting pairs. Since the numbers stored in delay elements are in 2's complement representation, the delay elements only require 15% area of the implementation. The others are computational units, which require extremely low hardware complexity, due to the implementation using simple combinational logic. Therefore, the primary objective of optimization towards hardware complexity for stochastic IIR filters is to reduce the number of S2B-SNG pairs.

3.6 The Stochastic Implementation of modified lattice IIR filters

In this section, we first derive a modified lattice filter structure and then present its stochastic implementation using fewer S2B-SNG converting pairs. We combine the traditional SC units and the stochastic inner-product using coefficients-based scaling to implement the modified lattice structure.

3.6.1 The Modified Lattice Structure

Consider the stochastic lattice structure shown in Fig. 3.16(b). The S2B module is used to convert a stochastic sequence to a 2's complement number for left-shift and the SNG module converts the shifted 2's complement number to a stochastic sequence. If there is no shifting operation, then we can remove S2B-SNG pairs. Notice that

shifting operation is required to offset the scaling of stochastic inner-product. The key to reducing the number of S2B-SNG pairs is to eliminate the implicit scaling of stochastic inner-product. In the inner-product implemented using traditional SC units (see Fig. 3.15(a)), the output is scaled by 1/2 which cannot be eliminated. In the stochastic inner-product implemented using coefficients-based scaling (see Fig. 3.15(b)), the output is scaled by $|a| + |b|$. The only way to remove scaling is if $|a| + |b| = 1$. Therefore, to reduce the number of S2B-SNG pairs, we propose the modified lattice structure which satisfies $|a| + |b| = 1$.

It is well known that lattice IIR filters can be derived using the Schur algorithm. The most commonly-used lattice structures are basic and normalized lattice structures. The Schur polynomials in the algorithm are obtained by using the degree reduction procedure:

$$\Phi_{i-1}(z) = \frac{z^{-1}\{\Phi_i(z) - k_i \Phi_i^*(z)\}}{s_i}, \tag{3.22}$$

where s_i is any nonzero scaling factor and

$$k_i = \Phi_i(0)/\Phi_i^*(0). \tag{3.23}$$

The basic lattice structure is designed by choosing $s_i = 1 - k_i^2$ while the normalized lattice structure is obtained by choosing $s_i = \sqrt{1 - k_i^2}$. If we choose $s_i = 1 \pm k_i$, the modified lattice structures can be derived as shown in Fig. 3.18.

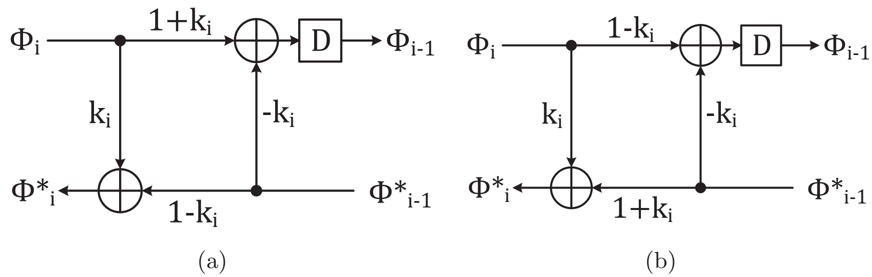


Figure 3.18: The lattice stages obtained by choosing (a) $s_i = 1 - k_i$, and (b) $s_i = 1 + k_i$.

First consider the case that $s_i = 1 - k_i$ shown in Fig. 3.18(a). The lattice stage is described by:

$$\begin{cases} \Phi_{i-1} = (1 + k_i)\Phi_i + (-k_i)\Phi_{i-1}^* \\ \Phi_i^* = k_i \Phi_i + (1 - k_i)\Phi_{i-1}^*. \end{cases} \tag{3.24}$$

To remove Φ^*_{i-1}, equations (3.24) are rewritten as

$$\begin{cases} (1-k_i)\Phi_{i-1} = (1-k_i^2)\Phi_i - k_i(1-k_i)\Phi^*_{i-1} \\ k_i\Phi^*_i = k_i^2\Phi_i + k_i(1-k_i)\Phi^*_{i-1}. \end{cases} \tag{3.25}$$

Adding two equations in (3.25), we obtain

$$\Phi_{i-1} = \frac{\Phi_i - k_i\Phi^*_i}{1-k_i}. \tag{3.26}$$

Notice that z^{-1} represents the delay element in the lattice structure. Then it is exactly same as the equation (3.22) where $s_i = 1 - k_i$.

It is known that in a stable lattice IIR filter, $|k_i| \leq 1$ [41]. The stochastic implementation of this modified lattice structure should be considered for two cases of k_i: $k_i > 0$ and $k_i < 0$.

Case-I: $k_i > 0$. When $0 < k_i \leq 1$, for the bottom path, the scaling factor of stochastic inner-product is given by:

$$|k_i| + |1-k_i| = 1, \tag{3.27}$$

which satisfies the ideal situation where $|a|+|b| = 1$; this eliminates the binary multiplier and S2B-SNG pair. However, for the top path, the scaling factor is

$$|-k_i| + |1+k_i| = 1 + 2k_i \neq 1. \tag{3.28}$$

If it is implemented using stochastic inner-product with coefficients-based scaling, the binary multiplier is still required. Therefore, we implement the top path using traditional SC units and the binary multiplier is replaced by a left-shift operation. The bottom path is implemented using the stochastic inner-product with coefficients-based scaling and no binary multiplier or S2B-SNG pair is needed.

Fig. 3.19 illustrates the stochastic implementation of the modified lattice structure shown in Fig. 3.18(a) for the case of $0 < k_i \leq 1$. Note that all numbers in traditional SC units cannot exceed unity, whereas, in the top path,

$$1 + k_i > 1, \tag{3.29}$$

where k_i is a positive number. Thus, $(1+k_i)\Phi_i - k_i\Phi^*_{i-1}$ cannot be implemented with one level of scaled addition. We implement the top path using two levels of stochastic

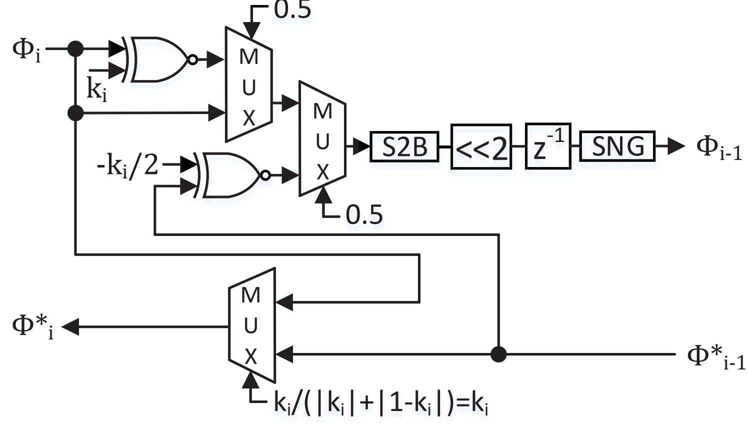


Figure 3.19: The circuit diagram of the stochastic modified lattice stage with $s_i = 1 - k_i$ for the case of $k_i > 0$.

additions. The first level computes

$$\frac{(1+k_i)\Phi_i}{2} \tag{3.30}$$

and the second level calculates

$$\frac{\frac{(1+k_i)\Phi_i}{2} - \frac{k_i\Phi^*_{i-1}}{2}}{2} = \frac{(1+k_i)\Phi_i - k_i\Phi^*_{i-1}}{4}. \tag{3.31}$$

Accordingly, a 2-bit left-shift is required since the scaling factor of 1/4 is introduced. Notice that the select signal in a traditional SC adder is fixed at 0.5. In the bottom path, two XNOR gates in the stochastic inner-product with coefficients-based scaling are removed because we already know $k_i > 0$ and $1 - k_i > 0$. The select signal is determined by the filter coefficients. The binary multiplier and the S2B-SNG pair are not required since the scaling factor is unity. All coefficients in the implementation are represented using stochastic sequences. In this design, we combine traditional SC units and the stochastic inner-product using coefficients-based scaling. No binary multiplier is required and the number of S2B-SNG pairs is reduced.

Case-II: $k_i < 0$. Consider the case $-1 \leq k_i < 0$. If the stochastic inner-product with coefficients-based scaling is used for implementation, the scaling of the bottom path is

$$|k_i| + |1 - k_i| = 1 - 2k_i \neq 1, \tag{3.32}$$

while the scaling of the top path is

$$|-k_i| + |1 + k_i| = 1. \tag{3.33}$$

Hence, the top path is implemented using the stochastic inner-product with coefficients-based scaling, whereas the bottom path is implemented using two-level traditional SC units as shown in Fig. 3.20. Notice that we still have two pairs of S2B-SNG converting units in this implementation. The S2B-SNG pair and a 2-bit left-shift operation in the bottom path are used to offset the scaling introduced by the traditional stochastic addition as in the top path of Fig. 3.19. However, the top path in Fig. 3.20 cannot be implemented without a S2B-SNG pair similar to the bottom path in Fig. 3.19, even though the scaling of the inner-product is unity. This is because the desired format of the number stored in delay element is 2's complement representation rather than a long stochastic sequence which would consume significantly more hardware resources.

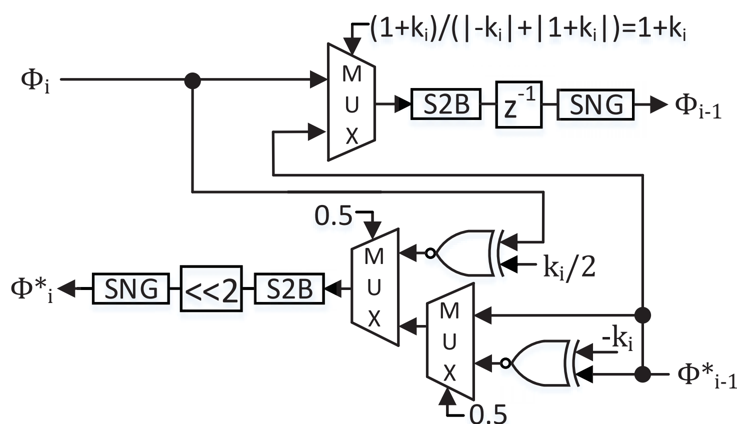


Figure 3.20: The circuit diagram of the stochastic modified lattice stage with $s_i = 1 - k_i$ for the case of $k_i < 0$.

From architectures shown in Fig. 3.19 and Fig. 3.20, we conclude that when $s_i = 1 - k_i$, $k_i > 0$ is the ideal case for a stochastic modified lattice stage while no significant improvement on hardware efficiency is achieved in the case of $k_i < 0$.

The stochastic implementation of modified lattice stage with $s_i = 1 + k_i$ shown in Fig. 3.18(b) is similar to the case of $s_i = 1 - k_i$. However, for $s_i = 1 + k_i$, $k_i < 0$ is the ideal case while there is no great improvement on hardware efficiency if $k_i > 0$. Notice that it is opposite to $s_i = 1 - k_i$.

3.6.2 The design of stochastic modified lattice IIR filters

The structure of stochastic modified lattice IIR filters is highly dependent on the signs of lattice coefficients k_i's. In our implementation, the choice of s_i depends on signs of k_i's. If there are more positive k_i's in a modified lattice IIR filter than negative k_i's, we choose $s_i = 1 - k_i$ in the Schur algorithm. Notice that $k_i > 0$ is the ideal case for stochastic modified lattice stage with $s_i = 1 - k_i$. Thus, More S2B-SNG pairs can be eliminated compared to the selection of $s_i = 1 + k_i$. In contrast, if most k_i's in a modified lattice IIR filter are negative, then we choose $s_i = 1 + k_i$ in the Schur algorithm, where $k_i < 0$ is the ideal case. By choosing different s_i for stochastic implementation of modified lattice IIR filters, we ensure at least $\lceil \frac{N}{2} \rceil$ S2B-SNG pairs are removed from the stochastic implementation of normalized lattice IIR filters using traditional SC units. Therefore, in the best case of stochastic modified lattice IIR filters the number of S2B-SNG pairs is reduced by N while $\lceil \frac{N}{2} \rceil$ S2B-SNG pairs are eliminated in the worst case.

Fig. 3.21 shows the stochastic implementation of a 3rd-order high-pass modified lattice Butterworth IIR filter with cut-off frequency $\omega_c = 0.8\pi$. The transfer function of the filter is given by:

$$H(z) = \frac{0.0181(1 - 3z^{-1} + 3z^{-2} - z^{-3})}{1 + 1.7600z^{-1} + 1.1829z^{-2} + 0.2781z^{-3}}$$

All lattice coefficients k_i's are positive and $s_i = 1 - k_i$ is selected in Schur algorithm. The coefficients in modified lattice structure are described as follows:

$$k = [0.8855,\ 0.7516,\ 0.2781]$$
$$c = [-0.0251,\ 0.0744,\ -0.0674,\ 0.0181]$$

The input of the filter is $x(n)$ in Fig. 3.21. The output for N^{th}-order filter is the scaled result $y(n)/(\sum_{i=0}^{N} c_i)$. The numbers stored in delay elements are in 2's complement format. No S2B-SNG pair exists in the bottom path and no binary multipliers is required in this design.

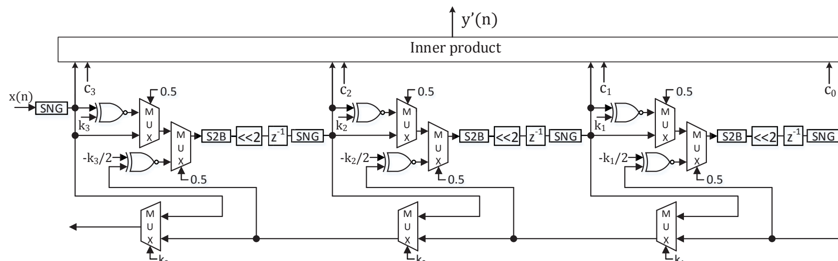


Figure 3.21: The architecture of the stochastic implementation of a 3rd-order high-pass modified lattice Butterworth IIR filter with cut-off frequency $\omega_c = 0.8\pi$.

3.6.3 State overflow and input scaling for the modified lattice structure

Compared to the stochastic normalized lattice IIR filters, the performance of the stochastic implementation of the modified lattice IIR filters is degraded due to the overflow of internal states. Notice that the range of bipolar stochastic number representation is $[-1, 1]$. States of normalized lattice IIR filters are bounded by unity power such that they are represented in stochastic sequences without overflow, whereas orthonormality is not guaranteed in the modified lattice structure for an arbitrary input signal. However, we can scale the input signal to prevent the overflow of states.

Consider the state-space description of a modified lattice IIR filter shown in Fig. 3.22:

$$\boldsymbol{x}(n+1) = \boldsymbol{A}\boldsymbol{x}(n+1) + \boldsymbol{b}u(n), \tag{3.34}$$

$$y(n) = \boldsymbol{c}^T\boldsymbol{x}(n) + du(n). \tag{3.35}$$

The boldfaced letters imply a vector or a matrix. In the above representation, $\boldsymbol{x}$

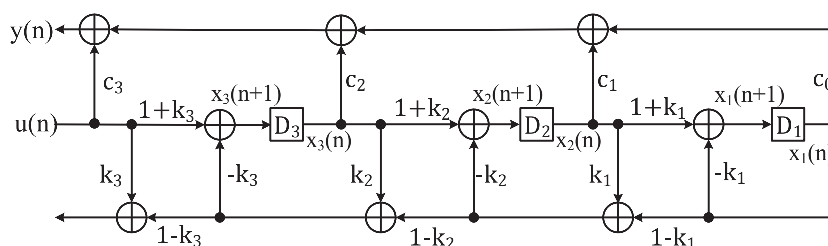


Figure 3.22: The 3rd-order modified lattice filter structure.

is the states vector, u is the input and y is the output. The state covariance matrix $\boldsymbol{K}$

is defined as

$$\boldsymbol{K} = E[\boldsymbol{x}(n)\boldsymbol{x}^T(n)], \tag{3.36}$$

where the diagonal elements K_{ii}'s describe the signal power of internal states x_i's. From the Section 11.3 of [27], it is known that the $\boldsymbol{K}$-matrix is obtained by solving the following *Lyapunov equation*:

$$\boldsymbol{K} = \boldsymbol{b}\boldsymbol{b}^T + \boldsymbol{A}\boldsymbol{K}\boldsymbol{A}^T \tag{3.37}$$

for unit-variance white noise input. Now assume that to satisfy the requirement of no overflow, the input signal power is scaled by p^2. Then equation (3.37) is transformed to:

$$\boldsymbol{K} = p^2\boldsymbol{b}\boldsymbol{b}^T + \boldsymbol{A}\boldsymbol{K}\boldsymbol{A}^T. \tag{3.38}$$

Solving the equation above for $\boldsymbol{K}$ [42], we achieve expressions of K_{ii} in terms of p. Then we can obtain maximum p by solving $|K_{ii}| \leq 1$. The value of p determines the maximum signal power of filter input to guarantee no overflow of internal states in the modified lattice structure.

3.7 Experimental Results

In this section, we present the experimental results for stochastic IIR filters. These experiments include performance test, comparisons in terms of hardware resources and fault tolerance.

3.7.1 Simulation Results

Several simulations were performed to test the performance of stochastic IIR filters. An input test signal consisting of a mixture of five sinusoids of different frequencies and random noise is used. In our simulation, the length of the stochastic sequence is 1024. A total of 512 input samples are used for simulation.

We simulated low-pass and high-pass IIR filters with nine different stochastic implementations.

- BSS: Stochastic implementation for **Basic State-Space** lattice IIR filters.

- BLI: **Basic** lattice IIR filters using stochastic **Lattice Implementation**.
- NSS: Stochastic implementation for **Normalized State-Space** lattice IIR filters.
- NLI: **Normalized** lattice IIR filters using stochastic **Lattice Implementation**.
- OBLI: **Optimized Basic** lattice IIR filters using stochastic **Lattice Implementation**.
- ONLI: **Optimized Normalized** lattice IIR filters using stochastic **Lattice Implementation**.
- NTD: Stochastic implementation of **Normalized** lattice IIR filters using **Traditional** SC units.
- MOD1: Stochastic implementation of **Modified** lattice IIR filters. No S2B-SNG pair is in the bottom path, which is the best case in terms of hardware complexity.
- MOD2: Stochastic implementation of **Modified** lattice IIR filters. $\frac{N}{2}$ S2B-SNG pairs are in the bottom path, which is the worst case in terms of hardware complexity.

Fig. 3.23 shows an example of filtering results achieved from stochastic and ideal implementations for low-pass and high-pass IIR filters. Table 3.3 shows SER of output for 3rd-order low-pass and high-pass stochastic IIR filters for different implementations. Table 3.4 shows SER of output for 6th-order low-pass and high-pass stochastic IIR filters for different implementations.

From above test results, we can observe that normalized lattice stochastic implementations for IIR filters always have less error than basic lattice stochastic implementations, especially for narrow-band filters. Notice that for the low-pass filter with cut-off frequency at 0.2π and the high-pass filter with cut-off frequency at 0.8π, NSS and NLI implementations outperform BSS and BLI implementations significantly. It is explained by the fact that states of normalized lattice structure are *orthonormal* while states of basic lattice structure are only *orthogonal*. The overflow of the outputs of binary multipliers in BSS and BLI implementations leads to the degrading of performance.

Notice that the performance of BLI is worse than BSS. Original coefficients are used for the BSS implementation while scaled versions of coefficients are used for the BLI

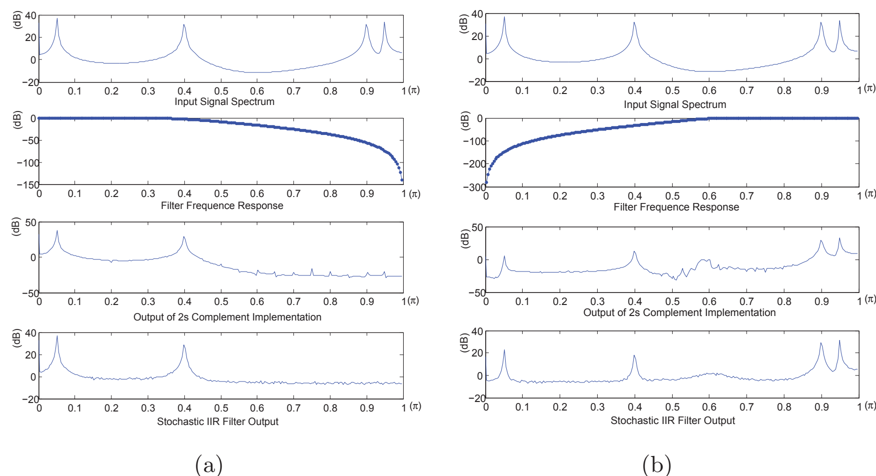


Figure 3.23: The filtering results of (a) a low-pass 3rd-order IIR filter with cutoff-frequency 0.4π (stochastic implementation: NSS), and (b) a high-pass 6th-order IIR filter with cutoff-frequency 0.6π (stochastic implementation: NLI).

implementation, where scaling factors may be less than one. It reduces the precision of stochastic computational results of the BLI implementation.

The declining performance of the OBLI and optimized ONLI implementations is explained by the reduced number of binary multipliers. First, binary multiplications can be considered as more accurate compared to approximation using stochastic computing. In the optimized BLI and optimized NLI implementations, accurate computations are replaced by approximated stochastic computing and thus the performance is degraded. Second, reducing the number of binary multipliers leads to more scaled filter coefficients in the optimized BLI and optimized NLI implementations. This leads to precision loss in the stochastic computing architectures.

Compare all implementations for different cut-off frequencies. For NSS and NLI implementations, there is no significant performance degradation with the change of cut-off frequency from 0.2π to 0.4π for low-pass filter or from 0.8π to 0.2π for high-pass filter, whereas the performance of all other implementations improve significantly. This is explained by the state overflow of different lattice structures for the given input signal. The normalized lattice structure leads to the state power less than one. Hence, there is almost no change of accuracy of NSS and NLI implementations for different cut-off frequencies. Fig. 3.24 illustrates state overflow for the basic lattice structure (BSS and

Table 3.3: The output SER (dB) of 3rd-order stochastic IIR filters for different implementations.

Low-pass Cut-off Frequency	BSS	BLI	NSS	NLI	OBLI	ONLI
0.2π	12.03	8.23	16.56	16.66	7.98	7.16
0.4π	17.52	17.26	16.90	16.92	15.58	15.06
High-pass Cut-off Frequency	**BSS**	**BLI**	**NSS**	**NLI**	**OBLI**	**ONLI**
0.8π	11.17	7.75	15.30	15.42	7.06	6.37
0.6π	16.02	15.67	15.31	15.35	14.10	13.59

Table 3.4: The output SER (dB) of 6th-order stochastic IIR filters for different implementations.

Low-pass Cut-off Frequency	BSS	BLI	NSS	NLI	OBLI	ONLI
0.2π	5.00	4.02	12.86	13.32	3.47	6.07
0.4π	16.20	15.13	14.84	14.93	13.17	13.33
High-pass Cut-off Frequency	**BSS**	**BLI**	**NSS**	**NLI**	**OBLI**	**ONLI**
0.8π	4.51	3.70	11.86	12.44	3.07	5.01
0.6π	14.16	13.19	12.58	12.72	11.08	11.11

BLI) for 3rd-order low-pass IIR filter. The x-axis indicates the cut-off frequency and the y-axis indicates the count of state overflow during 512 clock cycles. With the increase of cut-off frequency for low-pass filters or decrease of cut-off frequency of high-pass filters, the number of state overflow is reduced. Therefore, the performance of BSS and BLI improves.

The performance of implementations for 6th-order filter is worse than that of implementations for 3rd-order filters, since there are more state overflow and more errors are introduced by more stochastic computation units for higher order implementations.

Simulation results of NTD and MOD stochastic lattice IIR filters are considered. Notice that Butterworth IIR filters are used in our test. For high-pass Butterworth IIR filters with cut-off frequencies greater than 0.5π, all lattice coefficients k_i's are

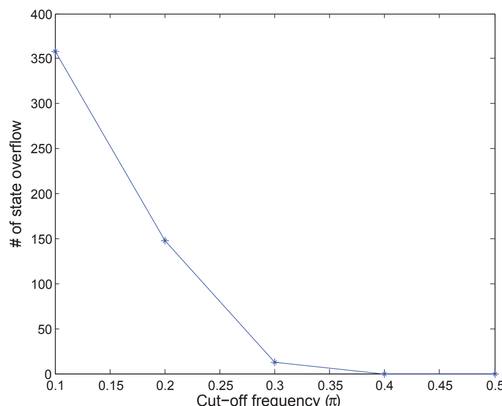


Figure 3.24: The counts of state overflow for the basic lattice structure (BSS and BLI) for the 3rd-order low-pass IIR filter.

positive. Therefore, they are implemented using the MOD1 implementation with $s_i = 1-k_i$. For low-pass Butterworth IIR filters with cut-off frequencies less than 0.5π, lattice coefficients k_i's are alternately positive and negative. Hence, they are implemented using the MOD2 implementation. Table 3.5 presents the output mean squre error (MSE) and signal-to-noise ratio (SNR) of specified filters for various implementations. The accuracy results are calculated compared to the output of IIR filters using floating-point precision.

Fig. 3.25 illustrates spectrums of input and output signals obtained from stochastic and ideal filters for specified high-pass filter.

From the test results, we observe that for both low-pass and high-pass IIR filters, proposed stochastic normalized lattice implementations (NTD) and modified lattice implementations (MOD) have less error than previous implementations with narrow-band. However, with the increase of filter passband, BSS and BLI implementations outperform NTD and MOD implementations. Consider the change of accuracy of each stochastic implementation from narrow-band to broad-band filters. The performance of the NTD implementation remains same for various cut-off frequencies of both low-pass and high-pass IIR filters. The accuracies of BSS, BLI and MOD implementations are all improved with the increase of filter passband, whereas the improvement of BSS and BLI implementations is greater than the MOD implementation.

The performance can be improved by increasing the length of stochastic bit streams.

Table 3.5: The output MSE and SNR (dB) of (a) 3rd-order low-pass stochastic IIR filters, (b) 3rd-order high-pass stochastic IIR filters for various implementations.

(a) 3rd-order low-pass stochastic IIR filters

		Low-pass Cut-off Frequency				
		0.1π	0.2π	0.3π	0.4π	0.5π
BSS	MSE	0.0261	0.0024	8.113e-04	9.955e-04	8.086e-04
	SNR	2.22	12.72	17.53	17.07	18.69
BLI	MSE	0.0311	0.0060	0.0012	0.0011	7.971e-04
	SNR	1.46	8.74	15.89	16.83	18.75
NTD	MSE	0.0023	0.0027	0.0029	0.0029	0.0028
	SNR	12.84	12.28	12.03	12.44	13.29
MOD2	MSE	0.0100	0.0018	0.0025	0.0028	0.0023
	SNR	6.40	13.95	12.58	12.65	14.15

(b) 3rd-order high-pass stochastic IIR filters

		High-pass Cut-off Frequency				
		0.9π	0.8π	0.7π	0.6π	0.5π
BSS	MSE	0.0099	0.0018	7.930e-04	9.862e-04	8.068e-04
	SNR	2.94	11.26	16.53	16.14	17.53
BLI	MSE	0.0136	0.0042	0.0012	0.0011	8.103e-04
	SNR	1.55	7.47	14.89	15.80	17.51
NTD	MSE	0.0026	0.0027	0.0028	0.0029	0.0028
	SNR	8.77	9.52	11.12	11.40	12.06
MOD1	MSE	0.0063	0.0092	0.0050	0.0065	0.0087
	SNR	4.89	4.08	8.54	7.98	7.21

Fig. 3.26 shows the increase of SNR and the decrease of MSE for the 3rd-order low-pass MOD implementation with the cut-off frequency at 0.3π. The x-axis indicates the length of stochastic bit streams which varies from 2^{10} to 2^{16}.

3.7.2 Synthesis Results

Hardware complexity

The area consumptions of proposed stochastic architectures for IIR filters are evaluated using 65nm technology. The architectures are implemented using 65nm libraries and synthesized using Synopsys Design Compiler. We also compare hardware cost

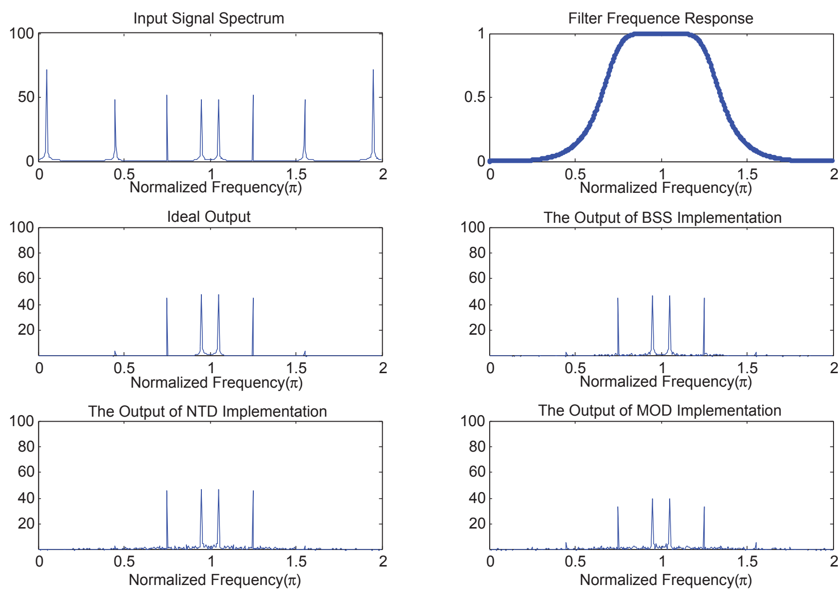


Figure 3.25: The filtering results of different stochastic implementations for the high-pass 3rd-order IIR filter with cut-off frequency at 0.7π.

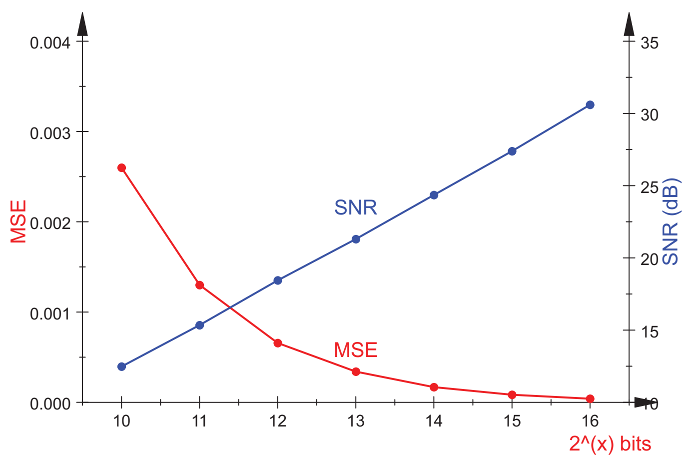


Figure 3.26: The output MSE and SNR for the 3rd-order low-pass MOD implementation with the cut-off frequency at 0.3π.

between traditional binary implementations and proposed stochastic implementations. The length of the stochastic sequence is 1024, whereas the binary numbers in corresponding traditional implementations require 10 bits. One clock cycle is required to compute one addition or one multiplication in 2's complement representation. However, a stochastic computing implementation is a bit-serial system. Therefore, 1024 clock cycles are required for the proposed filter architectures to compute one sample.

Table 3.6 shows the area and power cost of implementations for IIR filters. The consumptions of area are given in terms of equivalent two input NAND gates. From the table, we can see that in traditional binary implementations, lattice structures require more hardware resources than direct-form IIR since there are more multipliers in lattice IIR. Stochastic implementations cost substantially less hardware and power resources than traditional binary implementations, especially OBLI and ONLI implementations.

Table 3.6: The power consumption comparison and area consumption comparison in terms of equivalent 2-NAND gates for different implementations of IIR filters.

Filter order		Type of Implementations								
		Binary Direct-form	Binary Basic Lattice	Binary Normalized Lattice	BSS	BLI	NSS	NLI	OBLI	ONLI
3	Area	7311 (100%)	9077 (124.16%)	13301 (181.93%)	4371 (59.79%)	4813 (65.83%)	4371 (59.79%)	4441 (60.74%)	2844 (38.90%)	2652 (36.27%)
	Power	28.23μW (100%)	23.19μW (82.15%)	38.05μW (134.79%)	10.68μW (37.83%)	14.75μW (52.25%)	10.68μW (37.83%)	12.40μW (43.92%)	11.47μW (40.63%)	10.17μW (36.03%)
6	Area	14056 (100%)	19923 (141.74%)	28197 (200.60%)	8673 (61.70%)	9641 (68.59%)	8673 (61.70%)	8645 (61.50%)	5285 (37.60%)	5141 (36.58%)
	Power	65.17μW (100%)	37.46μW (57.48%)	67.84μW (104.10%)	20.45μW (31.38%)	29.57μW (45.37%)	20.45μW (31.38%)	23.43μW (35.95%)	20.44μW (31.36%)	19.20μW (29.46%)

Table 3.7 presents the hardware complexity comparison in terms of equivalent 2-NAND gates for different implementations of IIR filters. The results show that proposed stochastic implementations cost substantially less hardware than the previous designs. For the best case, the proposed stochastic implementation of modified lattice IIR filters reduces the hardware complexity by 60% compared to the previous BLI implementation and saves 90% area compared to the 2's complement lattice IIR filter.

Table 3.7: The hardware complexity comparison in terms of equivalent 2-NAND gates for different implementations of IIR filters.

Filter order	Type of Implementations					
	Binary Direct-form	Binary Lattice	NTD	MOD1	MOD2	BLI*
3	7311 (100%)	13301 (181.93%)	2342 (32.03%)	1561 (21.35%)	2016 (27.57%)	4813 (65.83%)
6	14056 (100%)	28197 (200.60%)	4819 (34.28%)	2924 (20.80%)	4117 (29.29%)	9641 (68.59%)

* previous design in [38].

Timing

Table 3.8 presents the delay of critical path for different implementations of IIR filters. The operating conditions for each implementation are specified by a supply voltage of 1.00 V and a temperature at 25°C. Each implementation is operated at clock frequency 10 MHz.

Table 3.8: The delay (ns) of critical path for different implementations of IIR filters.

Filter order	Type of Implementations					
	Binary Direct-form	Binary Lattice	NTD	MOD1	MOD2	BLI*
3	13.76 (100%)	27.52 (200%)	5.65 (41.06%)	5.69 (41.35%)	6.20 (45.06%)	11.92 (86.63%)
6	38.87 (100%)	65.48 (168.46%)	5.65 (14.54%)	7.10 (18.27%)	6.28 (16.16%)	14.20 (36.53%)

* previous design in [38].

It is shown in the table that the delays of critical paths of the proposed NTD and MOD implementations are 50% less than the previous BLI implementation. The decrease of delay can lead to higher clock frequency.

Consider the comparison of timing between stochastic implementation and 2's complement implementation of lattice IIR filters. For the 3rd-order IIR filter, the delays of critical paths of proposed NTD and MOD implementations are around 80% less than the 2's complement lattice IIR filter. For the 6th-order IIR lattice filter, the delays of critical paths of proposed NTD and MOD implementations are reduced by 90% compared to

the 2's complement lattice IIR filter. The results in Table 3.8 also show better timing scalability of proposed stochastic implementations of lattice IIR filter compared to 2's complement implementations. With increase in the filter order from 3 to 6, the delays of critical paths of 2's complement direct-form and lattice implementations increase by 182.49% and 137.94%, respectively, whereas the delay of the NTD implementation stay constant and the delays of MOD1 and MOD2 implementations increase by only 24.78% and 1.29%, respectively.

3.7.3 Fault Tolerance Analysis

We performed fault-tolerance test for stochastic filters by randomly injecting bit-flipping error at all internal nodes and measuring the corresponding output SER for each implementation. A signal consisting of a mixture of five sinusoids of different frequencies and random noise is used as the test inputs. The length of the stochastic sequence is 1024. A total of 512 input samples are used. We control the level of injected soft error by randomly flipping certain percent bits of *all* internal computational nodes in circuits.

A 3rd-order low-pass butterworth IIR filter with cut-off frequency at 0.3π is considered. We test fault-tolerance for different implementations of the filter, including traditional binary implementation and proposed stochastic implementations. Bit-flipping errors are injected in all internal computational nodes including *binary multipliers* in stochastic implementations. Table 3.9 presents the output SER due to random bit-flipping for each implementation. Fig. 3.27 illustrates the output SER due to random bit-flipping for different implementations. The X-axis of Fig. 3.27 is in logarithm s-cale for better visualization of the results. For binary implementations, the SER of the direct-form implementation is worse than the normalized lattice and the basic lattice because of the overflow. For very low bit-flipping rates, the stochastic implementation-s have worse SER; this is because the stochastic implementations suffer from certain minimum error.

It is shown that the proposed stochastic implementations suffer less from bit-flipping errors than traditional binary implementations. For the OBLI and ONLI implementations, bit-flipping almost has no impact on the output accuracy when flipping percentage

Table 3.9: The output SER (dB) with random bit-flipping for different implementations of a 3rd-order low-pass butterworth IIR filter with cut-off frequency 0.3π.

Filter Types	Percentage of Bit-flipping						
	0%	0.005%	0.01%	0.05%	0.1%	0.5%	1%
binary	28.53	28.46	11.38	6.70	4.26	-2.14	-4.13
BSS	17.70	17.35	17.15	15.59	14.63	10.34	7.95
BLI	17.40	17.15	16.82	15.68	15.05	10.26	7.91
NSS	17.10	16.77	16.48	15.21	13.78	9.82	7.45
NLI	17.03	16.78	16.51	15.30	13.97	9.65	7.53
OBLI	15.82	15.63	15.91	15.91	15.63	15.55	15.09
ONLI	15.36	15.05	15.23	15.41	15.07	14.98	14.78

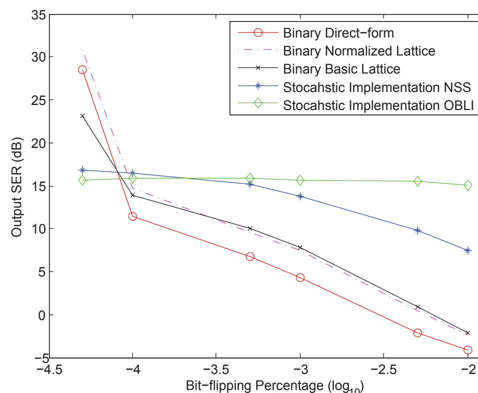


Figure 3.27: Fault-tolerance test results of traditional binary direct-form, normalized lattice, basic lattice, stochastic NSS and stochastic OBLI implementations for a 3rd-order low-pass butterworth IIR filter with cut-off frequency 0.3π.

is under 0.5%. Starting with 0.01% bit-flipping, the performance of the traditional binary implementation is degraded significantly due to random bit-flippings. For a very low rate of bit-flipping, the traditional binary implementation has 66.84% more output SER than stochastic implementations. Also OBLI and ONLI are slightly outperformed by the other 4 stochastic implementations since less binary multipliers are used in OBLI and ONLI implementations. However, at a high bit-flipping rate, performance of the tradition binary implementation is degraded more significantly compared to stochastic implementations. Notice that bit-flipping errors are also injected in binary multipliers for implementation. Since less binary multipliers are used in OBLI and ONLI implementations than the other 4 stochastic implementations, these two implementations

outperform the remaining stochastic designs at a high bit-flipping rate.

3.8 Conclusion

This section has presented nine novel structures for stochastic logic implementation of recursive digital filters. These structures are based on state-space and lattice forms. Out of these nine structures, two are optimized with respect to the number of binary multiplications; these structures require one-third of the number of binary multiplications compared to their non-optimized versions. It is shown that the normalized state-space and normalized lattice filters have the highest SER among all six proposed stochastic filters. The last three implementations are based on the normalized lattice structure and the modified lattice structure, respectively. Compared with previous designs, the proposed architectures improve the performance for narrow-band stochastic IIR filter and reduce the hardware complexity significantly. The analysis of state power in the modified lattice structure and input scaling method are also presented.

Chapter 4

Computing Arithmetic Functions using Stochastic Logic

In this chapter, we present the stochastic computation of arithmetic functions, such as trigonometric, exponential, logarithmic and sigmoid, can be implemented by using Maclaurin series expansion and/or factorization. First, it is shown that in ***unipolar*** stochastic logic, a polynomial can be implemented using multiple levels of NAND gates based on Horner's rule, if the coefficients are alternatively positive and negative and their magnitudes are monotonically decreasing. Truncated Maclaurin series expansions of arithmetic functions are used to generate polynomials which satisfy these constraints. The input and output in these functions are represented by unipolar representation. Second, for a polynomial that does not satisfy these constraints, it still can be implemented based on Horner's rule if each factor of the polynomial satisfies these constraints by using factoring and factor-combining. Third, format conversion is proposed for arithmetic functions with input and output represented in different formats. Polynomials are transformed to equivalent forms that naturally exploit format conversions. Table 4.1 lists the functions implemented in this paper using Horner's rule.

As seen from Table 4.1, most functions are implemented using unipolar stochastic logic where both inputs and outputs are in unipolar format. These functions do not require any scaling and are implemented without loss of precision. Functions that require factorization or format conversion require some scaling and suffer from some loss of

Table 4.1: The arithmetic functions implemented in this paper.

Function	Domain	Range	Representation	
			Input	Output
$\sin x$	$[0, 1]$	$[0, 0.8415]$	Unipolar	Unipolar
$\cos x$	$[0, 1]$	$[0.5403, 1]$	Unipolar	Unipolar
$\tanh ax^{*}$	$[0, 1]$	$[0, \tanh a]$	Unipolar	Unipolar
$\log(1 + ax)^{*}$	$[0, 1]$	$[0, \log(1 + a)]$	Unipolar	Unipolar
$\mathrm{sigmoid}(x)$	$[0, 1]$	$[0.5, 0.7311]$	Unipolar	Unipolar
e^{-ax**}	$[0, 1]$	$[e^{-a}, 1]$	Unipolar	Unipolar
$\sin \pi x$	$[0, 1]$	$[0, 1]$	Unipolar	Unipolar
$\cos \pi x$	$[0, 1]$	$[-1, 1]$	Unipolar	Bipolar
$\mathrm{sigmoid}(x)$	$[-1, 1]$	$[0.2689, 0.7311]$	Bipolar	Unipolar

*$0 < a \leq 1$. **$a > 0$ is required; $0 < a \leq 1$ and $a > 1$ are considered as two separate cases.

precision. Furthermore, most circuits presented in this paper contain feed-forward logic (except the delay elements which are inherently sequential), and thus can be pipelined at gate-level for low-power applications using subthreshold techniques. Only $\tanh ax$ and $\mathrm{sigmoid}(ax)$ contain feedback, for large values of a, as these require stochastic dividers that contain feed-back.

4.1 Theoretical Foundations for Stochastic logic

In this section, three theoretical foundations are proposed for stochastic implementations of functions. These foundations include stochastic logic implementation of polynomials expanded using: Horner's rule, factorization and format conversion principle.

4.1.1 Implementing Polynomials using Horner's Rule

Consider the following polynomials:

$$p_1(x) = 1 - a_1 x \tag{4.1}$$

$$p_2(x) = 1 - a_1 x + a_2 x^2 = 1 - a_1 x(1 - \frac{a_2}{a_1} x) \tag{4.2}$$

$$\begin{aligned} p(x) &= 1 - a_1 x + a_2 x^2 - a_3 x^3 + \cdots \\ &= 1 - a_1 x(1 - \frac{a_2}{a_1} x(1 - \frac{a_3}{a_2} x(1 - \cdots)))). \end{aligned} \tag{4.3}$$

In general form, a polynomial of degree n is expressed as $p(x) = \sum_{i=0}^{n}(-1)^i a_i x^i$, where $a_0 = 1$. Fig. 4.1 illustrates the unipolar implementation of these three polynomials.

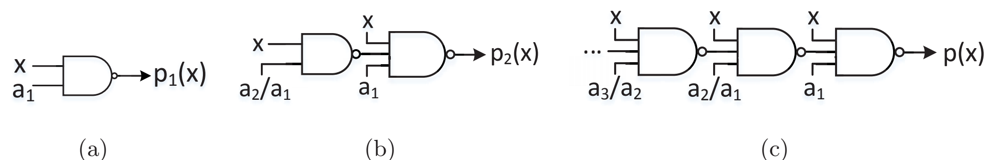


Figure 4.1: Fundamental blocks for stochastic unipolar with no scaling: (a) $p_1(x)$ is implemented using NAND gates, (b) $p_2(x)$ is implemented using two levels of NAND gates, (c) $p(x)$ is implemented using multiple levels of NAND gates.

Notice that $0 < a_1 \leq 1$, $0 < (a_2/a_1) \leq 1$ and $0 < (a_3/a_2) \leq 1$ must be guaranteed for feasible designs shown in Fig. 4.1. This method involves no scaling and does not require multiplexers which are prone to computational error. The implicit scale factor of multiplexers may lead to precision loss if the sum of magnitudes of the coefficients is greater than one. All polynomials that satisfy two constraints, denoted as C1 and C2 below, can be implemented using stochastic unipolar logic using Horner's rule:

1. C1: All terms in a polynomial are alternately positive and negative: $a_i > 0$.
2. C2: The magnitudes of all polynomial coefficients are less than one and decrease with the increase of term order: $a_{i+1} \leq a_i \leq 1$.

4.1.2 Implementation using Factoring and Factor-Combining

Consider a polynomial expressed in product form as:

$$p(x) = 1 - a_1x + a_2x^2 - a_3x^3 + \cdots \tag{4.4}$$

$$= \prod_{i=0}^{n}(1 - b_{1i}x + b_{2i}x^2). \tag{4.5}$$

Assume that in equation (4.4), a_i's do not satisfy constraints presented in Section 4.1.1. However, $p(x)$ can still be implemented using simple NAND gates and an AND gate without scaling if the condition $0 < b_{2i} \leq b_{1i} \leq 1$ is satisfied for all factors. Such an

example is illustrated as follows:

$$\begin{aligned}&(1-0.7x+0.5x^2)(1-0.8x+0.6x^2)\\&=1-1.5x^2+1.66x^2-0.82x^3+0.3x^4.\end{aligned}$$

Although coefficients of the expanded polynomial do not satisfy C1 and C2, both factors satisfy these constraints. Therefore, the polynomial can be implemented using an AND gate to perform multiplication of factors. Factors are implemented using multiple levels of NAND gates as shown Fig. 4.1. The final output is computed without scaling. Although the factors shown in (5) are all second-order, the factors can be of any arbitrary order as long as these satisfy the constraints in Section 4.1.1.

Consider a general polynomial given by:

$$\begin{aligned}p(x)&=a_0+a_1x+a_2x^2+\cdots+a_nx^n\\&=a_0\cdot\prod_{i=0}^{n_1}(1-c_ix)\cdot\prod_{j=0}^{n_2}(1-b_{1j}x+b_{2j}x^2)\end{aligned}\tag{4.6}$$

Since all coefficients are real, roots of $p(x)=0$ are either real or complex conjugates. Therefore, $p(x)$ can be represented by a product of first-order and second-order factors. The scaling of stochastic implementation is represented by a_0. Assume that the number of first-order and second-order factors are n_1+1 and n_2+1, respectively. Then the degree of the polynomial n equals $2n_2+n_1+3$. In stochastic unipolar representation, $p(x)$ can be implemented by multiplying all factors using AND gates.

First, consider the coefficient a_0 in equation (4.6). Note that $a_0>0$ since $p(0)>0$ for stochastic unipolar format. If $a_0\leq 1$, then it can be represented in unipolar format, where no scaling is introduced. If $a_0>1$, then we can only implement $p(x)/a_0$, where a scale factor of a_0 is introduced.

Second, consider the first-order factor in (4.6). The value of c_i is determined by a corresponding real root (r) of $p(x)=0$. Possible locations of real roots are shown in Fig. 5.12. In Fig. 5.12(a), we have $r\leq 0$ and then $c=1/r<0$. The first-order factor $1-cx$ is rewritten as $1+c'x$ where $c'=-c$ and $c'>0$. The first-order factor can be implemented using a multiplexer as described in Fig. 1.2(c). A fractional coefficient is calculated for the select signal of MUX. The implementation is still feasible for $c'>1$ since the computed result is a *scaled* version, where the scaling is $1+c'$. In Fig. 5.12(b),

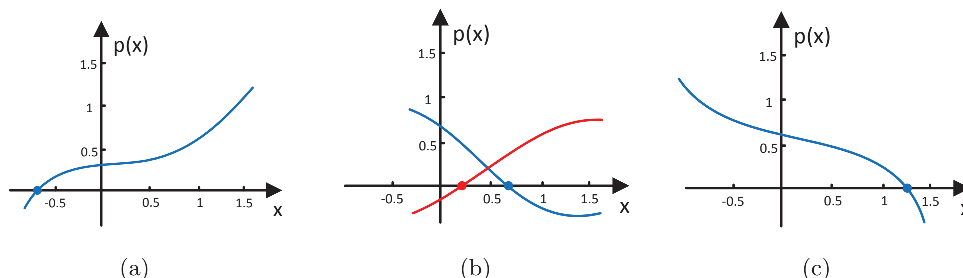


Figure 4.2: Three possible locations of a real root (r) of $p(x)$: (a) $r \leq 0$, (b) $0 < r \leq 1$ and (c) $1 \leq r$

$0 < r \leq 1$ and then $c > 1$. In this case, $1 - cx$ is infeasible in stochastic unipolar format. However, notice that two possible curves of $p(x)$ shown in Fig. 5.12(b) violate the constraint $0 \leq p(x) \leq 1$ given $x \in [0, 1]$, unless the multiplicity of the real root is an even number. These root locations lead to negative polynomial values. Thus, we assume that no real root is present between 0 and 1. Implementation of polynomials with real roots of even degree is beyond the scope of this paper. For implementation of these polynomials, the reader is referred to [43]. In Fig. 5.12(c), $r > 1$ and $0 < c = 1/r < 1$. The first-order factor $1-cx$ is implemented using a NAND gate as shown in Fig. 4.1(a).

Third, consider the product of second-order factors $1 - b_1x + b_2x^2$. If the condition $0 < b_1 \leq b_2 \leq 1$ is satisfied, the product can be implemented as illustrated in equation (5) with no scaling. However, it is not guaranteed for an arbitrary function that such a product of second-order factors satisfying C1 and C2 always exists. If the constraints are not satisfied, the second-order factors cannot be implemented without scaling. In some cases, two factors can be combined such that the product satisfies constraints C1 and C2. If it is not possible, then a scaled version of factors can be implemented using other methods that are beyond the scope of this paper. The reader is referred to [43] for implementation of such polynomials.

Another application of factoring involves factoring exponential functions that can be used in the implementation of exponential functions. Examples are shown as follows:

$$
\begin{aligned}
e^{-(a+b)x} =& e^{-ax}e^{-bx} \\
e^{-abx} =& (e^{-ax})^b
\end{aligned}
$$

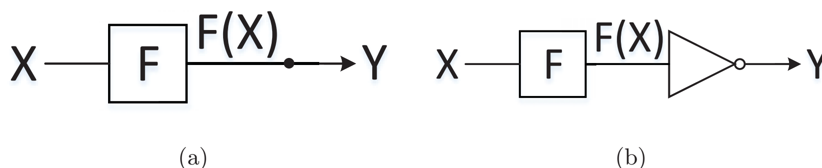
These factoring techniques are used to implement exponential, tangent hyperbolic and

sigmoid functions.

4.1.3 Format Conversion

Given a target polynomial $T(x)$, if $x \in [0,1]$ and $T(x) \in [-1,1]$, or $x \in [-1,1]$ and $T(x) \in [0,1]$, the input and output are represented using different formats. The format conversion method can be used to improve the precision since the ranges of representation formats (unipolar/bipolar) for input and output are fully exploited. This method breaks the limitation of unipolar implementation and improves the performance of bipolar implementation.

Consider the case where $x \in [0,1]$ and $T(x) \in [-1,1]$. The input can be represented in unipolar format while the output must be in bipolar format. Fig. 4.3 illustrates fundamental building blocks of format converters, where X is a unipolar input and Y is a bipolar output. F denotes arbitrary stochastic *unipolar* logic. Assume

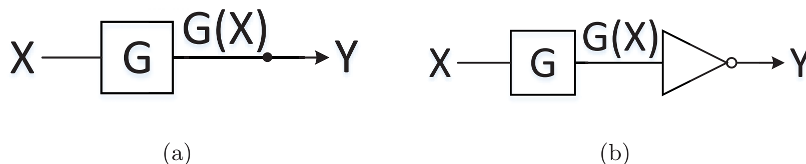


(a) (b)

Figure 4.3: Fundamental stochastic computational elements with unipolar input and bipolar output, where (a) $y = 2f(x) - 1$ and (b) $y = 1 - 2f(x)$.

$p_{F(X)}$ and p_Y represent the probability of ones in stochastic bit streams $F(X)$ and Y, whereas $f(x)$ and y stand for the values of $F(X)$ and Y. From the definition of unipolar and bipolar formats, we know that $f(x) = p_{F(X)}$ and $y = 2p_Y - 1$. Notice that no logic gate is implemented in Fig. 4.3(a) for conversion. Therefore, we obtain $p_Y = p_{F(X)}$ and then $y = 2f(x) - 1$. In Fig. 4.3(b), we obtain $p_Y = 1 - p_{F(X)}$ using a NOT gate and thus $y = 1 - 2f(x)$. Both of these can be considered as basic building blocks for format conversion from unipolar to bipolar format.

Consider the case where $x \in [-1,1]$ and $T_2(x) \in [0,1]$. The input is represented in bipolar format while the output is represented in unipolar format. The fundamental building blocks with bipolar input and unipolar output are illustrated in Fig. 4.4. Let X represent the bipolar input and Y represent the unipolar output. G denotes arbitrary stochastic *bipolar* logic. Assume $p_{G(X)}$ and p_Y represent the probability of ones



(a) (b)

Figure 4.4: Fundamental stochastic computational elements with bipolar input and unipolar output, where (a) $y = \frac{1}{2}g(x) + \frac{1}{2}$ and (b) $y = \frac{1}{2} - \frac{1}{2}g(x)$.

in stochastic bit streams $G(X)$ and Y, whereas $g(x)$ and y stand for the values represented by $G(X)$ and Y. It is known that $g(x) = 2p_{G(X)} - 1$ and $y = p_Y$. In Fig. 4.4(a), $p_Y = p_{G(X)}$ and then $y = (g(x) + 1)/2$. In Fig. 4.4(b), we obtain $p_Y = 1 - p_{G(X)}$ using an inverter and thus $y = 1/2 - g(x)/2$. Both of these can be considered as basic building blocks for format conversion from bipolar to unipolar format.

If the input and output are in different formats, the polynomials can be expressed in one of the above forms so that no overhead is required for format conversion.

4.2 Horner's rule for Maclaurin expansions

The stochastic implementation of functions in this section is based on the theoretical foundation presented in Section 4.1.1. These functions include $\sin x$, $\cos x$, $\tanh ax$, e^{-ax}, $\log(1 + ax)$ $(0 < a \leq 1)$ and sigmoid(x).

The Taylor series of a function $f(x)$ that is infinitely differentiable at a number a is represented by the power series

$$\sum_{n=0}^{\infty} \frac{f^{(n)}(a)}{n!}(x - a)^n.$$

If the Taylor series is centered at zero ($a = 0$), then that series is called a Maclaurin series. Several important Maclaurin series expansions are shown as follows:
Trigonometric functions:

$$\sin x = \sum_{n=0}^{\infty} \frac{(-1)^n}{(2n+1)!}x^{2n+1} = x - \frac{x^3}{3!} + \frac{x^5}{5!} \cdots$$

$$\cos x = \sum_{n=0}^{\infty} \frac{(-1)^n}{(2n)!}x^{2n} = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} \cdots$$

Exponential function:

$$e^{-x} = \sum_{n=0}^{\infty} \frac{(-x)^n}{n!} = 1 - x + \frac{x^2}{2!} - \frac{x^3}{3!} \cdots$$

Natural logarithm ($|x| < 1$):

$$\log(1+x) = \sum_{n=0}^{\infty} (-1)^{n+1} \frac{x^n}{n} = x - \frac{x^2}{2} + \frac{x^3}{3} \cdots$$

Hyperbolic function:

$$\begin{aligned} \tanh x &= \sum_{n=1}^{\infty} \frac{B_{2n} 4^n (4^n - 1)}{(2n)!} x^{2n-1} \\ &= x - \frac{1}{3}x^3 + \frac{2}{15}x^5 - \frac{17}{315}x^7 \cdots, \end{aligned}$$

where B_i's represent the Bernoulli numbers. A function can be approximated by a finite number of initial terms of its Maclaurin series.

Consider the implementation of $\sin x$. The Horner's rule for the 7th-order Maclaurin polynomial of $\sin x$ is given by:

$$\begin{aligned} \sin x &\approx x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} \\ &= x(1 - \frac{x^2}{6}(1 - \frac{x^2}{20}(1 - \frac{x^2}{42}))). \end{aligned} \tag{4.7}$$

Consider the stochastic implementation of $\sin x$ for $x \in [0, 1]$. The unipolar format of stochastic representation is used since $\sin x \in [0, 0.8415]$ for $x \in [0, 1]$. Three types of stochastic computation units are required in the implementation of equation (4.7). First, the AND gate is used to implement unipolar SC multiplication. Second, the NOT gate is used to implement $(1 - x)$. Third, x^2 is implemented using a one-bit delay element and an AND gate as shown in Fig. 4.5 [3]. The delay element is used for the decorrelation of inputs to the AND gate.

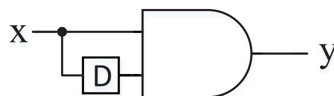


Figure 4.5: The SC square operation: $y = x^2$.

Fig. 4.6 shows the circuit diagram of stochastic $\sin x$ for $x \in [0, 1]$. This implementation consists of multiple levels of combinational logic and several one-bit delay elements for decorrelation. The delay elements are added at locations such that each path from input to output contains different number of delay elements; this leads to decorrelation of all paths. This decorrelation approach is adopted in all circuits presented in this paper. Note that using one delay instead of three delays in Fig. 4.6 achieves satisfactory decorrelation. Effect of correlation in stochastic logic circuits has been addressed in [37]. An approach to optimizing the delays for decorrelation has been proposed in [44]. Notice that the input signal and all coefficients are represented by unipolar stochastic bit streams. The outputs of internal nodes and final output are described as

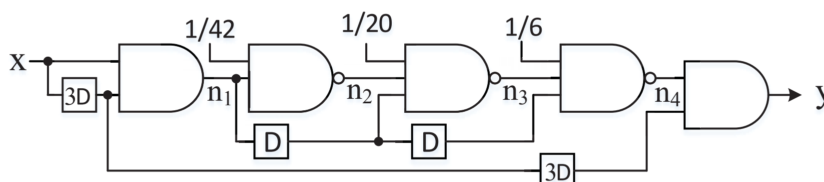


Figure 4.6: The circuit diagram of stochastic implementation of $\sin x$ using the 7th-order Maclaurin polynomial (4.7).

follows:

$$n_1 = x^2,\ n_2 = 1 - \frac{1}{42}n_1,\ n_3 = 1 - \frac{x^2}{20}n_2$$
$$n_4 = 1 - \frac{x^2}{6}n_3,\ y = n_4 \cdot x.$$

Applying Horner's rule to the 8th-order Maclaurin polynomial of $\cos x$, we obtain similar expressions to equation (4.7):

$$\begin{aligned} \cos x &\approx 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \frac{x^8}{8!} \\ &= 1 - \frac{x^2}{2}(1 - \frac{x^2}{12}(1 - \frac{x^2}{30}(1 - \frac{x^2}{56}))). \end{aligned} \tag{4.8}$$

Fig. 4.7 shows the stochastic implementation of $\cos x$ using equations (4.8).

The stochastic implementations for $\tanh ax$ and $\log(1 + ax)$ are considered for $x \in [0, 1]$. The requirement of $0 < a \leq 1$ satisfies the constraints for the domain of convergence for Maclaurin expansions of $\tanh x$ and $\log(1 + x)$. The 9th-order and 5th-order Maclaurin polynomials of $\tanh ax$ and $\log(1 + ax)$ transformed for the stochastic

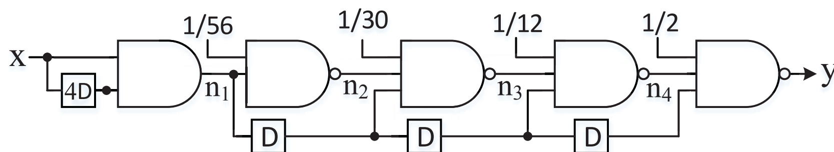


Figure 4.7: The circuit diagram of stochastic implementation of $\cos x$ using the 8th-order Maclaurin polynomial (4.8). Replacing 4 delays by one delay for the input would also satisfy decorrelation.

implementation are, respectively, given by:

$$\begin{aligned}\tanh ax &\approx ax - \frac{a^3x^3}{3} + \frac{2a^5x^5}{15} - \frac{17a^7x^7}{315} + \frac{62a^9x^9}{2835} \\ &= ax(1 - \frac{a^2x^2}{3}(1 - \frac{2a^2}{5}x^2(1 - \frac{17a^2}{42}x^2 \\ &\quad (1 - \frac{62a^2}{153}x^2))))\end{aligned} \tag{4.9}$$

and

$$\begin{aligned}\log(1+ax) &\approx ax - \frac{a^2x^2}{2} + \frac{a^3x^3}{3} - \frac{a^4x^4}{4} + \frac{a^5x^5}{5} \\ &= ax(1 - \frac{ax}{2}(1 - \frac{2ax}{3}(1 - \frac{3ax}{4}(1 - \frac{4ax}{5})))).\end{aligned} \tag{4.10}$$

Fig. 4.8 and Fig. 4.9 illustrate stochastic implementations of $\tanh ax$ and $\log(1+ax)$, where $a = 1$, by using equations (4.9) and (4.10), respectively.

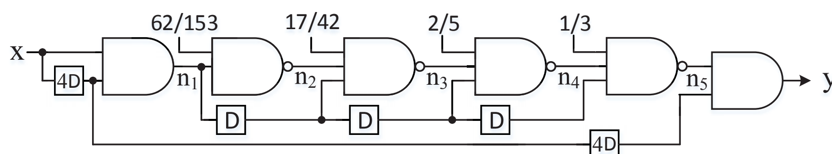


Figure 4.8: The circuit diagram of stochastic implementation of $\tanh x$ using the 9th-order Maclaurin polynomial (4.9).

Notice that this proposed method is not suited for the implementation of $\tanh ax$ in the domain x belonging to $[0, 1]$ with $a \gg 1$ since the Maclaurin expansion of $\tanh x$ only converges for $|x| < (\pi/2)$.

The 5th-order Maclaurin polynomial of $e^{-ax}(0 < a \leq 1)$ transformed for stochastic

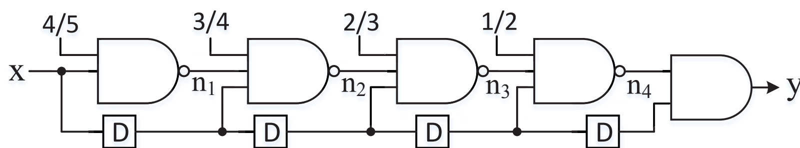


Figure 4.9: The circuit diagram of stochastic implementation of $\log(1 + x)$ using the 5th-order Maclaurin polynomial (10).

implementation is given by:

$$\begin{aligned} e^{-ax} &\approx 1 - ax + \frac{a^2x^2}{2!} - \frac{a^3x^3}{3!} + \frac{a^4x^4}{4!} - \frac{a^5x^5}{5!} \\ &= 1 - ax(1 - \frac{ax}{2}(1 - \frac{ax}{3}(1 - \frac{ax}{4}(1 - \frac{ax}{5})))). \end{aligned} \tag{4.11}$$

Fig. 4.10 illustrates the stochastic implementation of e^{-ax} where $a = 1$ by using equation (4.11).

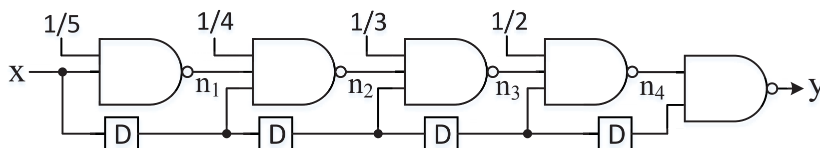


Figure 4.10: The circuit diagram of stochastic implementation of e^{-x} using the 5th-order Maclaurin polynomial (4.11).

Consider the sigmoid function described by a 5th-order Maclaurin polynomial as follows:

$$\begin{aligned} \text{sigmoid}(x) &= \frac{1}{2} + \frac{x}{4} - \frac{x^3}{48} + \frac{x^5}{480} \\ &= 1 - \frac{1}{2} + \frac{x}{4} - \frac{x^3}{48} + \frac{x^5}{480} \\ &= 1 - \frac{1}{2}(1 - \frac{x}{2}(1 - \frac{x^2}{12}(1 - \frac{x^2}{10}))) \end{aligned}$$

Fig. 4.11 shows the stochastic implementation of $\text{sigmoid}(x)$ given the input $x \in [0, 1]$.

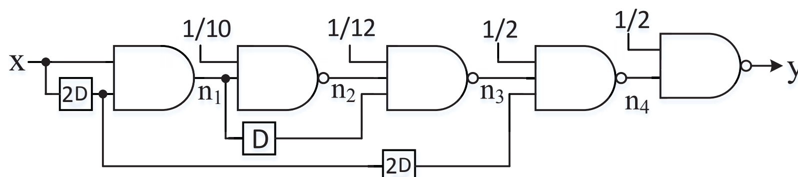


Figure 4.11: The circuit diagram of stochastic implementation of sigmoid(x) for $x \in [0, 1]$ using the 5th-order Maclaurin polynomial.

4.3 Implementations using Factorization

Stochastic unipolar implementations of functions in this section correspond to the theoretical foundation proposed in Section 4.1.2. Implementations of $\sin \pi x$ and e^{-ax} ($a > 1$) are considered in this section.

4.3.1 The Implementation of $\sin \pi x$

The stochastic implementation of $\sin x$ for $x \in [0, 1]$ is straightforward by using Maclaurin expansion and Horner's rule, whereas it cannot cover a full period of $\sin x$. Therefore, we consider the stochastic implementation of $\sin \pi x$ for $x \in [0, 1]$.

Substituting x in equation (4.7) by πx, we obtain:

$$\begin{aligned}\sin \pi x &\approx \pi x - \frac{\pi^3 x^3}{3!} + \frac{\pi^5 x^5}{5!} - \frac{\pi^7 x^7}{7!} + \frac{\pi^9 x^9}{9!} \\ &= \pi x(1 - \frac{\pi^2 x^2}{6}(1 - \frac{\pi^2 x^2}{20}(1 - \frac{\pi^2 x^2}{42} \\ &\quad (1 - \frac{\pi^2 x^2}{72})))) \\ \Rightarrow \frac{\sin \pi x}{\pi} &= x(1 - \frac{\pi^2 x^2}{6}(1 - \frac{\pi^2 x^2}{20}(1 - \frac{\pi^2 x^2}{42}(1 - \frac{\pi^2 x^2}{72})))) \end{aligned} \tag{4.12}$$

As discussed in Section 4.1.1, since π is greater than one and cannot be represented by a stochastic number, we implement the scaled function $\sin \pi x/\pi$. Notice that $\pi^2/6$ in (4.12) is greater than 1, which violates the constraint C2 described in Section 4.1.1. Hence, it is impossible to implement stochastic $\sin \pi x/\pi$ directly using Horner's rule.

The factorization method is considered. Factorize Maclaurin polynomials of $\sin \pi x/\pi$ with different orders over real numbers using polynomial roots and apply Horner's rule

to the 4th-order factors:

$$\begin{aligned}
\frac{\sin \pi x}{\pi} &\approx x - \frac{\pi^2 x^3}{3!} + \frac{\pi^4 x^5}{5!} \\
&= x(1 - 1.6449x^2 + 0.8117x^4) \\
&= x(1 - 1.6449x^2(1 - 0.4935x^2)) \qquad (4.13) \\
\frac{\sin \pi x}{\pi} &\approx x - \frac{\pi^2 x^3}{3!} + \frac{\pi^4 x^5}{5!} - \frac{\pi^6 x^7}{7!} \\
&= x(1 - 1.04x^2)(1 - 0.6036x^2 + 0.1832x^4) \\
&= x(1 - 1.04x^2)(1 - 0.6036x^2(1 - 0.3035x^2)) \qquad (4.14)
\end{aligned}$$

Factorizations (4.13) and (4.14) are not suitable for stochastic implementation as these contain coefficients greater than one. Factorization of 9th and 11th order polynomials are given by:

$$\begin{aligned}
\frac{\sin \pi x}{\pi} &\approx x - \frac{\pi^2 x^3}{3!} + \frac{\pi^4 x^5}{5!} - \frac{\pi^6 x^7}{7!} + \frac{\pi^8 x^9}{9!} \\
&= x(1 - x^2)(1 - 0.4x^2)(1 - 0.2488x^2 + 0.0656x^4) \\
&= x(1 - x^2)(1 - 0.4x^2)(1 - 0.2488x^2(1 \\
&\quad - 0.2637x^2)) \qquad (4.15) \\
\frac{\sin \pi x}{\pi} &\approx x - \frac{\pi^2 x^3}{3!} + \frac{\pi^4 x^5}{5!} - \frac{\pi^6 x^7}{7!} + \frac{\pi^8 x^9}{9!} - \frac{\pi^{10} x^{11}}{11!} \\
&= x(1 - 0.5424x^2 + 0.0833x^4)(1 - 0.1023x^2 \\
&\quad + 0.0282x^4)(1 - x^2) \\
&= x(1 - 0.5424x^2(1 - 0.1535x^2))(1 - 0.1023x^2 \\
&\quad (1 - 0.2754x^2))(1 - x^2) \qquad (4.16)
\end{aligned}$$

From equations (4.15) and (4.16), we observe that all coefficients are less than one in the 9th-order and 11th-order polynomials. The selection of required order for a feasible stochastic implementation is addressed in Appendix A. The 4th-order factor contains complex roots of x^2. To satisfy the structure of $1 - ax$, Horner's rule is applied to the 4th-order factor.

The stochastic implementation of $\sin \pi x/\pi$ using the 9th-order factorized Maclaurin polynomial (4.15) is shown in Fig. 4.12.

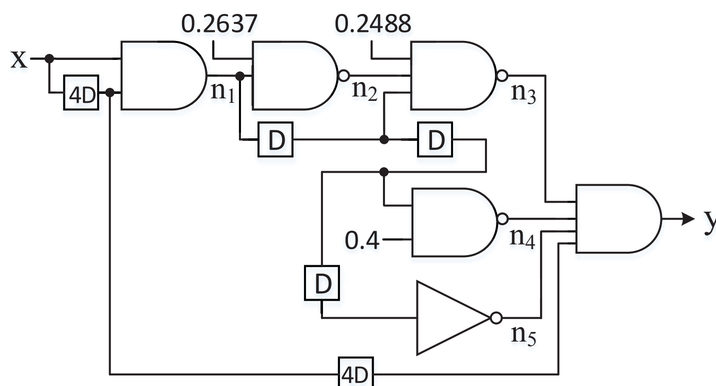


Figure 4.12: The circuit diagram of stochastic implementation of $\sin \pi x$ using the 9th-order Maclaurin polynomial.

Note that the input and all coefficients are represented in stochastic unipolar format. The internal nodes and final output are described by:

$$n_1 = x^2,\ n_2 = 1 - 0.2637n_1,\ n_3 = 1 - 0.2488n_2 \cdot x^2$$
$$n_4 = 1 - 0.4x^2,\ n_5 = 1 - x^2,\ y = n_3 n_4 n_5 \cdot x$$

4.3.2 The Implementation of e^{-ax} $(a > 1)$

Another example that exploits factorization and Horner's rule to implement a stochastic function is e^{-ax} where $a > 1$ and $x \in [0, 1]$. Assuming that $a = 1.9$, the 7th-order Maclaurin polynomial of $e^{-1.9x}$ is transformed for stochastic implementation:

$$e^{-1.9x} \approx 1 - 1.9x + \frac{(1.9x)^2}{2!} - \frac{(1.9x)^3}{3!} + \frac{(1.9x)^4}{4!} \tag{4.17}$$
$$- \frac{(1.9x)^5}{5!} + \frac{(1.9x)^6}{6!} - \frac{(1.9x)^7}{7!} \tag{4.18}$$
$$\approx (1 - 0.689x)(1 + 0.269x + 0.182x^2)$$
$$(1 - 0.394x + 0.326x^2)(1 - 1.087x + 0.434x^2) \tag{4.19}$$
$$= (1 - 0.689x)(1 - 0.394x + 0.326x^2)$$
$$(1 - 0.818x + 0.323x^2 - 0.081x^3 + 0.079x^4) \tag{4.20}$$

Notice that all coefficients in $(1 + 0.269x + 0.182x^2)$ are positive and coefficient 1.087 is greater than one in the factor $(1 - 1.087x + 0.434x^2)$. These two factors are combined to

build a 4th-order factor which satisfies given constraints. The stochastic implementation of $e^{-1.9x}$ is shown in Fig. 4.13.

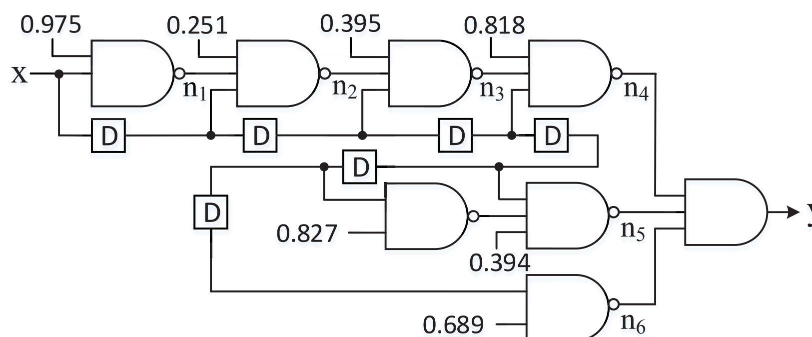


Figure 4.13: The circuit diagram of stochastic implementation of $e^{-1.9x}$.

The primary goal is to implement e^{-ax} $(a > 1)$ without scaling for which three conditions must be satisfied. First, for an odd degree polynomial, the real root must be greater than one. Second, for factors with orders equal to or greater than two, magnitudes of all coefficients must be less than or equal to one. Third, these coefficients must be alternately positive and negative with decreasing magnitudes. The Maclaurin expansion of e^{-ax} is given by: $e^{-ax} \approx 1 - ax + \frac{a^2x^2}{2} - \frac{a^3x^3}{6} + \frac{a^4x^4}{24} + \cdots$. If the factorized Maclaurin polynomial has a second-order factor $f_1(x) = 1 + d_1x + d_2x^2$, where $d_1 > 0$ and $d_2 > 0$, $f_1(x)$ may be combined with another factor with the form of $1 - b_1x + b_2x^2$, where $b_1,\ b_2 > 0$, to avoid scaling. The advantage of this approach is that if $b_i > 1$, the product of two factors may lead to coefficients of x less than one.

It is important to note that e^{-ax} with large a can not be easily implemented by directly factoring the Maclaurin polynomials. It has been shown [45][46] that for large a, the real root of an odd degree polynomial of large degree approaches 0.278465, which is less than one. For a small coefficient a, e^{-ax} is implementable using the proposed method *without scaling* for $a \leq 1.94$.

4.3.3 Generalization of Stochastic Implementation for e^{-ax} with $a > 1$

Notice that it is difficult to implement e^{-ax} with large a in stochastic logic by directly factoring the Maclaurin polynomials. However, for large a, e^{-ax} can be implemented based on e^{-bx} with small b. Consider the stochastic implementation of e^{-2x}, which can

be written as follows:

$$e^{-2x} = e^{-x} \cdot e^{-x}.$$

Then the e^{-2x} can be implemented as shown in Fig. 4.14. The e^{-x} in Fig. 4.14 is implemented using the circuit shown in Fig. 4.10. The one-bit delay element is used for decorrelation. A complete decorrelation of all paths would require 5 delays. However, the error due to correlation using only one AND gate is small in this example. Another implementation of e^{-2x} using Horner's rule, factorization and approximation is illustrated in [47].

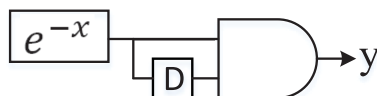


Figure 4.14: The circuit diagram of stochastic implementation of e^{-2x}.

For any arbitrary a ($a > 1$), e^{-ax} can be described as follows:

$$e^{-ax} = e^{(-\frac{a}{n}x)n} = e^{(-bx)n}, \ b = \frac{a}{n}.$$

where $0 < b \leq 1$ and n is an integer. Since $b \leq 1$, e^{-bx} can be easily implemented using Horner's rule. Then e^{-ax} can be implemented as shown in Fig. 4.15 by using e^{-bx} and $n-1$ cascaded AND gates.

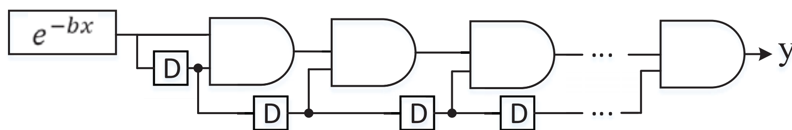
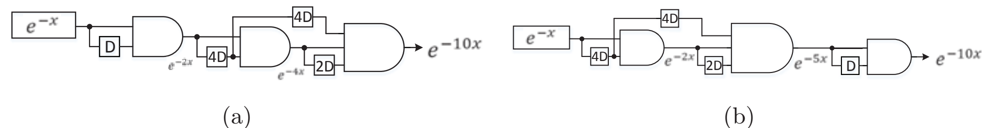


Figure 4.15: The circuit diagram of stochastic implementation of e^{-ax} ($a > 1$) by using e^{-bx} ($b \leq 1$) and $n-1$ cascaded AND gates.

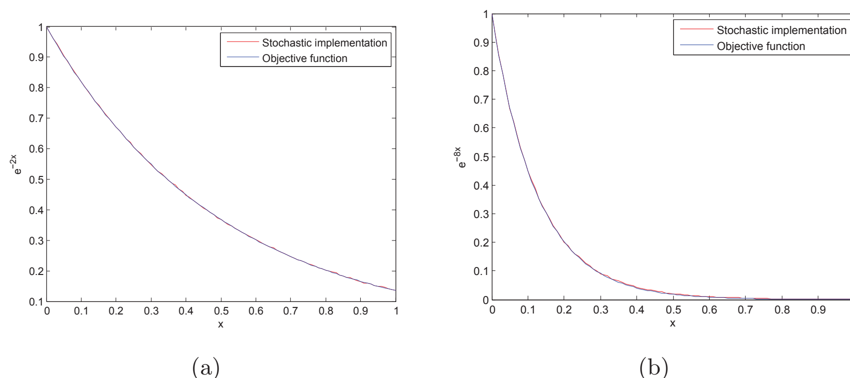
Notice that the method of decomposing e^{-ax} for $a > 1$ is not unique. Consider another example e^{-10x}, which can be decomposed either as $e^{-10x} = (e^{-2x})^5$ or as $e^{-10x} = (e^{-5x})^2$. Implementations of these two forms are shown in Fig. 4.16.

Simulation results of e^{-2x} using one AND gate and e^{-8x} using 3 AND gates from e^{-x} are shown in Fig. 4.17. The length of stochastic sequence is given by 1024. The degree of Maclaurin polynomial of e^{-x} is 5.



(a) (b)

Figure 4.16: The two stochastic implementations of e^{-10x}. If one decorrelating delay is used everywhere, this circuit will not function correctly.



(a) (b)

Figure 4.17: Simulation results of (a) e^{-2x} and (b) e^{-8x} using the proposed method.

4.3.4 The Implementation of $\tanh ax$ and sigmoid(ax) for $a > 1$

The stochastic implementation of $\tanh ax$ directly using Maclaurin expansion is not suited for large a, since the Maclaurin expansion of $\tanh x$ only converges for $|x| < (\pi/2)$. However, stochastic $\tanh ax$ $(a > 1)$ can be implemented based on e^{-2ax} and unipolar division.

The $\tanh ax$ is defined as:

$$\tanh ax = \frac{e^{ax} - e^{-ax}}{e^{ax} + e^{-ax}} = \frac{1 - e^{-2ax}}{1 + e^{-2ax}} = \frac{\frac{1-e^{-2ax}}{2}}{\frac{1-e^{-2ax}}{2} + e^{-2ax}}.$$

As shown in Fig. 4.18(a), the unipolar $\tanh ax$ can be implemented based on e^{-2ax} and a JK flip-flop.

The coefficient 0.5 is represented by a stochastic bit stream. The JK flip-flop is used to implement $y = x_1/(x_1 + x_2)$ as shown in Fig. 4.18(b). The design of the unipolar division using a JK flip-flop can be found in [3].

An alternative design of $\tanh ax$ can be implemented using the following equation:

$$\tanh ax = \frac{1}{1 + e^{-2ax}} \cdot (1 - e^{-2ax}).$$

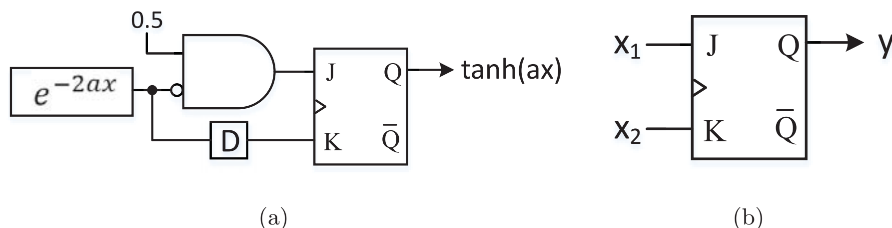


Figure 4.18: (a) The circuit diagram of stochastic implementation of $\tanh ax$ $(a > 1)$ using e^{-2ax} and a JK flip-flop. (b) The function $y = \frac{x_1}{x_1+x_2}$ implemented using a JK flip-flop.

The circuit is shown in Fig. 4.19. Notice that sigmoid$(2ax)$ is computed at the output of the JK flip-flop, where sigmoid$(2ax) = 1/(1 + e^{-2ax})$.

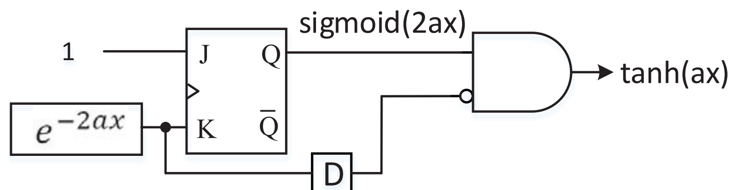


Figure 4.19: An alternative design of $\tanh ax$ in stochastic logic, with sigmoid$(2ax)$ computed at an internal node.

Simulation results of an example $\tanh 4x$ is shown in Fig. 4.20. The length of s-tochastic bit streams is 1024. The degree of Maclaurin polynomial of e^{-x} is 5.

4.4 Implementation of Functions with Input and Output Represented using Different Formats

Stochastic implementations of functions in this section correspond to the theoretical foundation presented in Section 4.1.3. Implementations of $\cos \pi x$ and sigmoid(x) are considered in this section.

4.4.1 Unipolar Input and Bipolar Output

The stochastic implementation of $\cos \pi x$ instead of $\cos x$ expands the coverage to half a period. As discussed in Section 4.1.3, any polynomial expressed as $1 - 2f(x)$ or $2f(x) - 1$ can be implemented using unipolar input and bipolar out without format

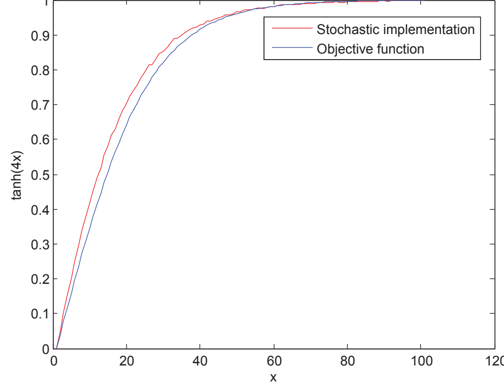


Figure 4.20: The simulation result of stochastic implementation of $\tanh 4x$ using the proposed method.

conversion overhead. This principle is exploited in the implementation of $\cos \pi x$ given $x \in [-1, 1]$.

The 10th-order Maclaurin polynomial of $\cos \pi x$ is transformed for stochastic implementation as follows:

$$\cos \pi x \approx 1 - \frac{\pi^2 x^2}{2!} + \frac{\pi^4 x^4}{4!} - \frac{\pi^6 x^6}{6!} + \frac{\pi^8 x^8}{8!} - \frac{\pi^{10} x^{10}}{10!} \tag{4.21}$$

$$= \underbrace{\frac{\pi^4 x^4}{4!} - \frac{\pi^2 x^2}{2!}}_{P(x)} + \underbrace{1 - \frac{\pi^6 x^6}{6!} + \frac{\pi^8 x^8}{8!} - \frac{\pi^{10} x^{10}}{10!}}_{Q(x)} \tag{4.22}$$

$$P(x) = \frac{\pi^2 x^2}{2!}(2 \cdot \frac{\pi^2 x^2}{24} - 1)$$
$$= 4.9348x^2(2 \cdot 0.4112x^2 - 1) = 4.9348 \cdot R(x) \tag{4.23}$$

$$Q(x) = 1 - \frac{\pi^6 x^6}{6!} + \frac{\pi^8 x^8}{8!} - \frac{\pi^{10} x^{10}}{10!}$$
$$= 1 - 1.3353x^6 + 0.2353x^8 - 0.0258x^{10}$$
$$= 1 - 2 \cdot 0.6676x^6(1 - 0.1762x^2(1 - 0.1097x^2)) \tag{4.24}$$

$$\cos \pi x \approx P(x) + Q(x) = 4.9348 \cdot R(x) + Q(x) \tag{4.25}$$

The coefficients of 2nd-order and 4th-order terms in the Maclaurin polynomial of $\cos \pi x$ are $\pi^2/2!$ and $\pi^4/4!$, which are much greater than one. Directly factorizing (21) would generate a factor of $1 - 4x$ which cannot be implemented in stochastic representation.

Therefore, we associate 2nd-order and 4th-order terms as polynomial $P(x)$ and the remaining terms as $Q(x)$ in (22).

For $P(x)$, factor out $\pi^2 x^2/2!$ and rewrite the polynomial with decimal representation in (23). The $(2 \cdot 0.4112x^2 - 1)$ in $R(x)$ corresponds to a format conversion as shown in Fig. 4.3(a) by using $0.4112x^2$ as the unipolar input. Then $R(x)$ is implemented using an XNOR gate as the stochastic multiplication where x^2 and $(2 \cdot 0.4112x^2 - 1)$ are in bipolar format.

In (24), apply Horner's rule for $Q(x)$ and rewrite it in decimal representation. Three levels of NAND gates are required to implement (24). The first two levels compute $(1 - 0.1762x^2(1 - 0.1097x^2))$ with unipolar input and *unipolar* output while the last NOT gate takes $0.6676x^6(1-0.1762x^2(1-0.1097x^2))$ as the unipolar input and generates $Q(x)$ as the *bipolar* output, where the functionality is $1 - 2x$ as shown in Fig. 4.3(b).

Finally, both $R(x)$ and $Q(x)$ are in stochastic bipolar format. The equation (4.25) is implemented using a multiplexer with two bipolar inputs. The select signal has a probability of $\frac{4.9348}{4.9348+1}$. The final output of the stochastic implementation of $\cos \pi x$ is scaled by 5.9348. No matter what order of the Maclaurin polynomial of $\cos \pi x$ is used, the scaling of the final output for this stochastic implementation is fixed and the performance is not degraded.

The stochastic implementation of $\cos \pi x$ using the 10th-order Maclaurin polynomial is shown in Fig. 4.21. Notice that x is a binary number. The unipolar

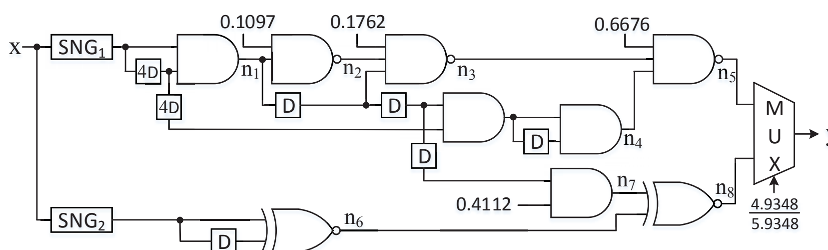


Figure 4.21: The circuit diagram of stochastic implementation of $\cos \pi x$ using the 10th-order Maclaurin polynomial.

stochastic bit stream of x is generated by SNG_1 while SNG_2 generates the bipolar bit stream. Table 4.2 describes the computational results and formats of internal nodes in Fig. 4.21. It is shown in the table that unipolar signals are converted to bipolar

Table 4.2: The computational results and formats of internal nodes in Fig. 4.21.

Internal nodes	n_1	n_2	n_3	n_4	n_5 ($Q(x)$)	n_6	n_7	n_8 ($R(x)$)
Result	x^2	$1-0.1097n_1$	$1-0.1762x^2n_2$	x^6	$1-2\cdot 0.6676n_3n_4$	x^2	$2\cdot 0.4112x^2-1$	n_6n_7
Format	unipolar	unipolar	unipolar	unipolar	bipolar	bipolar	bipolar	bipolar

format at nodes n_5 and n_7. $Q(x)$ is computed at node n_5 while $R(x)$ is generated at node n_8.

Generally, for the format conversion design with unipolar input and bipolar output, the Maclaurin polynomial is divided into two parts. The first part is $Q(x) = 1 - q_1x^j + q_2x^{j+1} + \cdots$, where $a < q_i < q_{i-1} \leq 2$. $Q(x)$ can be implemented using Horner's rule. Notice that the upper bound is 2 rather than 1, since the format conversion is applied for the outermost computation as follows:

$$Q(x) = 1 - 2 \cdot \frac{q_1}{2} \cdot x^j(1 - \frac{q_2}{q_1}x(1 - \cdots)) = 1 - 2f(x),$$

where $1 - 2f(x)$ is computed at the last stage using an inverter with unipolar input and bipolar output. The second part is $P(x)$, which contains all terms with coefficients greater than 2. The final output is generated by performing scaled addition using a multiplexer and, therefore, implicit scaling is introduced.

4.4.2 Bipolar Input and Unipolar Output

Consider the stochastic implementation of sigmoid(x) for $x \in [-1, 1]$. The corresponding output range is $[0.2689, 0.7311]$. Therefore, the input is represented in stochastic bipolar format while the output is in unipolar format. Expressions of the form $(1 + g(x))/2$ or $(1 - g(x))/2$ can be implemented without requiring any additional circuit for format conversion overhead, as explained in Section 4.1.3. This principle is exploited in this section.

The 5th-order Maclaurin polynomial of sigmoid(x) is transformed for stochastic implementation as follows:

$$\begin{aligned}\text{sigmoid}(x) &= \frac{1}{1+e^{-x}} \\ &\approx \frac{1}{2} + \frac{x}{4} - \frac{x^3}{48} + \frac{x^5}{480}\end{aligned} \quad (4.26)$$

$$= \frac{1}{2} - \frac{1}{2} \cdot \underbrace{x(-\frac{1}{2} + \frac{x^2}{24} - \frac{x^4}{240})}_{P(x)} \tag{4.27}$$

$$P(x) = x(-\frac{1}{2} + \frac{x^2}{24} - \frac{x^4}{240})$$

$$= x(\frac{1}{2}(-1 + \frac{x^2}{12} - \frac{x^4}{120})) \tag{4.28}$$

$$= x(\frac{1}{2}(-1 + x^2\frac{1}{2}(\frac{1}{6} - \frac{x^2}{60}))). \tag{4.29}$$

We observe that in (26) x exists starting from the second term. Notice that x must be represented in stochastic bipolar format. Therefore the conversion from bipolar to unipolar format can only be implemented in the last stage. The objective of transforming $P(x)$ is to construct the scaled addition of $\frac{1}{2}(a + b)$ which is implemented using a multiplexer with the select signal of 1/2. $P(x)$ is first implemented in stochastic bipolar representation using polynomial (29). Then sigmoid function (27) is implemented using a NOT gate with $P(x)$ as the input. The NOT gate is also considered as the converter of stochastic representation formats as shown in Fig. 4.4(b). The final output is given in unipolar format.

The stochastic implementation of sigmoid(x) using the 5th-order Maclaurin polynomial is shown in Fig. 4.22.

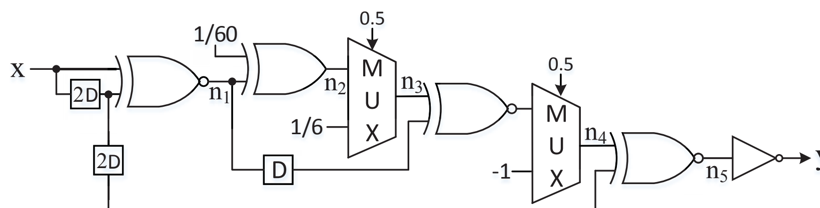


Figure 4.22: The circuit diagram of stochastic implementation of sigmoid(x) using the 5th-order Maclaurin polynomial.

Notice that x is a bipolar stochastic bit stream. XNOR gates are used to implement bipolar multiplications. The scaled addition is implemented using a multiplexer. The negation of a bipolar stochastic number is obtained by using a NOT gate. Table 4.3 describes the computational results and formats of internal nodes in Fig. 4.22. It is shown in the table that bipolar signals are converted to unipolar format at the last stage. $P(x)$ is computed at node n_5.

Table 4.3: The computational results and formats of internal nodes and the output in Fig. 4.22.

Internal nodes	n_1	n_2	n_3	n_4	n_5 $(P(x))$	y
Result	x^2	$-\frac{1}{60}n_1$	$\frac{1}{2}(\frac{1}{6}+n_2)$	$\frac{1}{2}(-1+x^2n_3)$	$x \cdot n_4$	$\frac{1}{2}-\frac{1}{2}n_5$
Format	bipolar	bipolar	bipolar	bipolar	bipolar	unipolar

Generally, for the format conversion design with bipolar input and unipolar output, the format conversion is performed at the last stage (i.e., the outermost computation of the Maclaurin polynomial). The form $\frac{1}{2}-\frac{1}{2}P(x)$ is applied for stochastic implementation. $P(x)$ is implemented by using XNOR gates and multiplexers with bipolar input and output. The final result is generated by performing bipolar to unipolar conversion using an inverter for $P(x)$.

4.5 Experimental Results

In this section, we present the experimental results of performance test and synthesis results for stochastic implementations of arithmetic functions using Maclaurin polynomials. Only one decorrelating delay in each distinct edge is considered for all circuits evaluated in this section. Results of previous work including implementations using Bernstein polynomial method [5] and FSM [8] [1] are also presented for comparisons.

4.5.1 Previous Work

Implementation using Bernstein polynomial

A function $f(x) \in [0,1]$ given $x \in [0,1]$ can be implemented using Bernstein polynomial method in stochastic unipolar logic. The target function can be described based on Bernstein polynomials as follows:

$$f(x) = \sum_{i=0}^{n} \beta_i B_{i,n}(x),$$

where β_i's are Bernstein coefficients and the Bernstein basis polynomial $B_{i,n}(x)$ is given as follows:

$$B_{i,n}(x) = \binom{n}{i} x^i (1-x)^{n-i}.$$

Fig. 5.23 illustrates an example of stochastic implementation based on 3rd-order Bernstein polynomials.

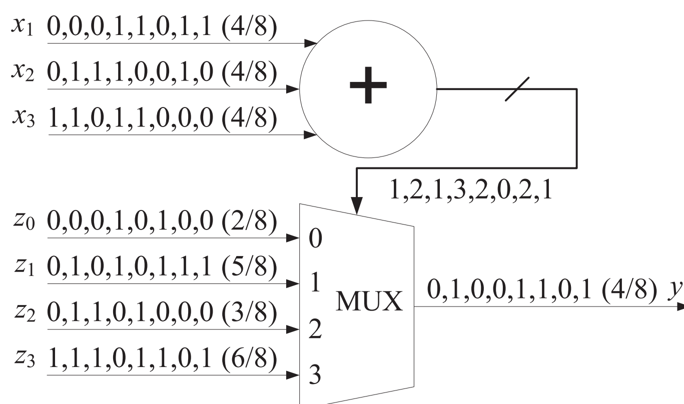


Figure 4.23: An example of stochastic implementation based on 3rd-order Bernstein polynomials. Stochastic bit streams x_1, x_2 and x_3 encode the input value x. Stochastic bit streams z_0, z_1, z_2 and z_3 encode the corresponding Bernstein coefficients.

Notice that the generation of bit streams x_i's and z_i's required SNGs, which increase hardware complexity of this implementation. More details of stochastic implementation using Bernstein polynomials can be found in [5].

Implementation using FSM

The finite-state-machine approach to implementing arithmetic functions was proposed by Brown and Card in [8]. A typical state transition diagram of FSM is shown in Fig. 6.7, where the function $\tanh \frac{G}{2}x$ is implemented.

In Fig. 6.7, X is the stochastic input while y is the output stochastic bit stream. Such an FSM is implemented using an up and down saturate counter. This FSM-based design is used to implement tangent hyperbolic and exponential functions. However, quoted from [1], "the FSM topology proposed by Brown and Card cannot be used to synthesize more sophisticated functions, such as high order polynomials and other non-polynomials." Therefore, the FSM topology shown in Fig. 4.25 was proposed

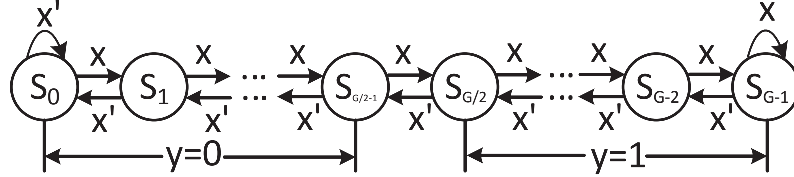


Figure 4.24: The state transition diagram of the FSM implementing the stochastic $\tanh(\frac{G}{2}x)$, where G is the number of states.

in [1][48][49] to implement arbitrary computations including trigonometric functions.

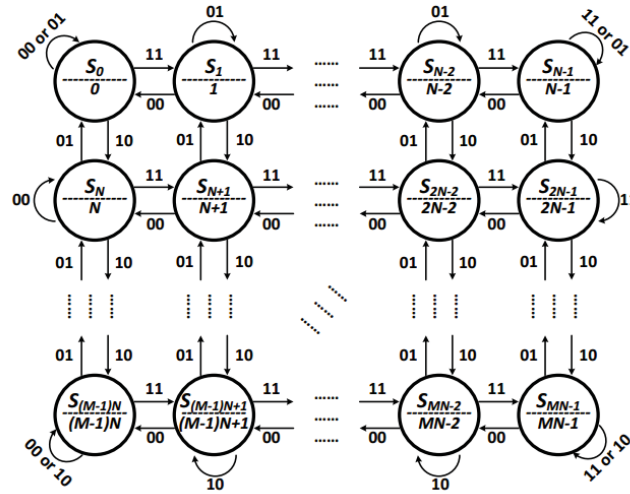


Figure 4.25: The state transition diagram of the FSM topology proposed in [1].

In Fig. 4.25, there are two inputs X and K. The numbers on each arrow represent the transition condition, with the first corresponding to the input X and the second corresponding to the input K. The binary output of FSM is encoded using $\log_2\lceil MN\rceil$ bits. The number below each state S_t $(0 \leq t \leq MN-1)$ represents the value encoded by the outputs of the FSM when the current state is S_t. Notice that the output of the FSM is not a stochastic sequence. The complete architecture for the implementation of stochastic functions using this FSM topology is shown in Fig. 4.26.

The FSM corresponds to the topology shown in Fig. 4.25, where X represents the input stochastic bit stream. The output of FSM is used as the select signal of the multiplexer (MUX). The final output bit stream is generated by the MUX. Parameters

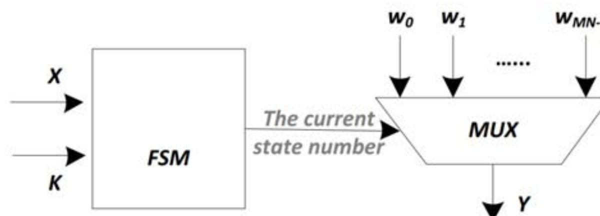


Figure 4.26: The complete circuit for implementation of stochastic functions [1].

K and w_i's are described using stochastic representation [50]. Notice that probabilities of ones in K and w_i's are calculated to minimize the difference of outputs from this implementation and the target function. More details of computing these parameters can be found in [1]. The authors of [1] demonstrate that additional inputs K and w_i enable the proposed FSM-based architecture more degree of design freedom to synthesize more sophisticated functions. However, these extra inputs may represent any probability values in $[0, 1]$. It means that compared to the original FSM-based design proposed in [8], more SNGs are required to generate bit streams for these coefficients and the hardware complexity increases significantly.

4.5.2 Performance Comparison

We present test results of different functions implemented using the proposed method, the FSM-based method and the Bernstein polynomial method with different orders. In our simulations, the inputs of target functions are given by 0:0.01:1. The output results are obtained using Monte Carlo experiments for different inputs. 1000 Monte Carlo runs were performed for each input. The length of stochastic bit streams is 1024. Table 4.4 presents the mean absolute error (MAE) of outputs of different stochastic implementations. Outputs of conventional implementations using floating-point precision are considered as the baseline.

It is shown in Table 4.4 that the proposed implementations for all functions outperform the Bernstein polynomial method. Notice that the sigmoid function using the Bernstein polynomial method is not presented since the bipolar input does not satisfy the unipolar constraint. For the FSM-based implementation, functions $\sin x$, $\cos x$, $\sin \pi x$, $\cos \pi x$, $\log(1 + x)$ and e^{-x} need to be implemented using 2-dimensional FSM [1]. Functions $\tanh x$, e^{-2x} and sigmoid(x) are implemented using the FSM-based

Table 4.4: The output mean absolute error (MAE) of stochastic implementations for different functions using the proposed method, the FSM-based method and the Bernstein polynomial method with different orders.

		Proposed			Bernstein Polynomial			FSM-based		
$\sin x$	Order	3	5	7	3	5	7	8-state		
	Error	0.0016	0.0033	0.0034	0.0136	0.0088	0.0066	0.0025		
$\cos x$	Order	2	4	6	2	4	6	8-state		
	Error	0.0082	0.0025	0.0023	0.0356	0.0178	0.0120	0.0053		
$\sin \pi x$	Order	9	11	13	9	11	13	8-state		
	Error	0.0514	0.0487	0.0451	0.0693	0.0569	0.0480	0.4716		
$\cos \pi x$	Order	10	12	14	10	12	14	8-state		
	Error	0.0537	0.0546	0.0560	0.0579	0.0724	0.0716	0.0174		
$\log(1+x)$	Order	5	6	7	5	6	7	8-state		
	Error	0.0141	0.0109	0.0081	0.0090	0.0076	0.0066	0.0186		
$\tanh x$	Order	3	5	7	3	5	7	4	8	16
	Error	0.0178	0.0175	0.0140	0.0182	0.0110	0.0082	0.0210	0.0351	0.0804
$\tanh 4x$	Order	4	5	6	4	5	6	8		
	Error	0.0199	0.0192	0.0191	0.0836	0.0667	0.0554	0.0046		
e^{-x}	Order	4	5	6	4	5	6	8-state		
	Error	0.0018	0.0008	0.0008	0.0130	0.0103	0.0086	0.0154		
e^{-2x}	Order	5	6	7	6	7	8	4	8	16
	Error	0.0019	0.0011	0.0009	0.0195	0.0170	0.0875	0.0508	0.0423	0.0368
sigmoid(x)	Order	5			–			4	8	16
	Error	0.0046			–			0.0091	0.0198	0.0414

method proposed in [8], which requires less hardware complexity. From Table 4.4, we can observe that except for $\cos \pi x$ and $\tanh 4x$, the proposed method outperforms the FSM-based method. Especially for $\sin \pi x$, the FSM-based method basically fails with respect to functionality.

Consider the 1-dimensional FSM-method [8]. The FSM based implementation realizes $\tanh(\frac{N}{2}t)$, where N is the total number of states and t is the stochastic bipolar input. To compare the FSM based implementation with our proposed method, t is chosen to be $\frac{2x}{N}$ such that $\frac{N}{2}t = x$. Additionally, the exponential function implemented using FSM-based method is e^{-2Gx}, where G represents the number of states with output equal to one and $G \geq 1$. Therefore the 1-dimensional FSM-based method is not suited for the implementation of e^{-ax}, where $0 < a < 2$. In this case, e^{-ax} (i.e., e^{-x}) must be implemented using 2-dimensional FSM.

4.5.3 Hardware Complexity and Critical Path Delay comparisons

In this subsection, synthesis results are presented for stochastic implementations of different complex functions using various approaches. The architectures are implemented using 65nm libraries and synthesized using Synopsys Design Compiler. The length of the stochastic sequence is 1024 and all required SNGs including 10-bit LFSRs as random number generators are considered in our synthesis. The length of stochastic sequences are consistent with tests for accuracy. The operating conditions for each implementation are specified by a supply voltage of 1.05 V and a temperature of 25 degree Celsius.

Notice that coefficients in the proposed design do not require different SNGs to generate bit streams. All coefficient can share one 10-bit LFSR and 1-bit delay elements are used for decorrelation of bit-streams which represent different coefficients. In this case, the hardware complexity can be reduced significantly. In our comparisons of performance and synthesis results, this optimization technique is also applied for the Bernstein polynomial and FSM designs for fair comparisons. Table 4.5 shows synthesis results of various implementations of complex functions. The hardware complexity results are given in terms of equivalent 2-input NAND gates. The numbers of SNGs in Bernstein polynomial and FSM-based methods are also optimized using the LFSR-sharing technique for fair comparisons. All SNGs are included in synthesis for all implementations.

Same orders as implementations for performance test are considered for synthesis. The proposed designs require less hardware complexity than the Bernstein polynomial based implementations except for $\tanh 4x$. For FSM-based implementation, the proposed approach requires less hardware complexity than 2-dimensional FSM-based method, which is used to implement $\sin x$, $\cos x$, $\sin \pi x$, $\cos \pi x$, $\log(1+x)$ and e^{-x}. The 1-dimensional FSM-based method involves less overhead than the proposed approach, since less SNGs are required. The FSM method requires less hardware complexity than the proposed method for $\tanh x$, $\tanh 4x$, e^{-2x} and sigmoid(x). The critical path delays of the proposed circuits and the FSM-based circuits are similar as observed from Table 4.5. However, the proposed circuits contain only feed-forward logic (except the delay elements which contain feedback and circuits for $\tanh 4x$) while the FSM based

implementations contain longer computations inside feedback loops. Therefore the critical paths of the proposed stochastic logic implementations can be further reduced by pipelining to the critical path of a single NAND gate. Using gate-level pipelining, the critical path of the proposed circuits will be reduced by a factor of 5-8, except for $\tanh 4x$ where the critical path with pipelining is limited by the critical path of the JK flip-flop. Therefore, the proposed circuits are better suited for low-power applications as these can be operated in sub-threshold mode.

4.6 Conclusion

Stochastic logic based implementations of complex arithmetic functions using truncated Maclaurin series polynomials have been presented in this chapter. The methods based on Horner's rule, factorization and format conversion are proposed. The generalized methods for stochastic unipolar implementations of e^{-ax}, $\tanh ax$ and $\text{sigmoid}(ax)$ for arbitrary $a > 0$ have also been presented. However, the proposed approach for $\tanh ax$ and $\text{sigmoid}(ax)$, where a is greater than 1, requires more hardware and leads to more error than FSM based implementations.

Table 4.5: The hardware complexity and critical path delay (ns) of stochastic implementations for different functions using the proposed method, the FSM-based method and the Bernstein polynomial method with different orders.

		Proposed			Bernstein Polynomial			FSM-based		
$\sin x$	Order	3	5	7	3	5	7	8-state		
	Area	411.8	469.0	528.8	569.4	759.7	998.9	1144.0		
	Delay	2.39	2.48	2.60	2.96	3.73	3.78	2.62		
$\cos x$	Order	2	4	6	2	4	6	8-state		
	Area	400.9	455.0	515.3	496.1	664.0	872.0	1144.0		
	Delay	2.21	2.38	2.60	2.59	2.90	4.01	2.76		
$\sin \pi x$	Order	9	11	13	9	11	13	8-state		
	Area	596.9	608.9	652.1	1410.8	1844.4	2277.1	1144.0		
	Delay	2.55	2.68	2.71	4.10	4.09	4.19	2.76		
$\cos \pi x$	Order	10	12	14	10	12	14	8-state		
	Area	885.6	975.0	1076.9	1636.4	2065.4	2277.1	1144.0		
	Delay	3.16	3.31	3.33	4.03	3.93	4.31	2.76		
$\log(1+x)$	Order	5	6	7	5	6	7	8-state		
	Area	577.2	655.7	746.2	759.7	872.0	998.9	1144.0		
	Delay	3.05	3.35	3.59	3.73	4.01	3.78	2.76		
$\tanh x$	Order	3	5	7	3	5	7	4	8	16
	Area	411.8	469.0	528.8	569.4	759.7	998.9	226.7	253.8	287.5
	Delay	2.39	2.48	2.60	2.96	3.73	3.78	2.37	2.54	3.11
$\tanh 4x$	Order	4	5	6	4	5	6	8		
	Area	583.4	649.5	728.5	664.0	759.7	872.0	253.8		
	Delay	3.39	3.63	4.07	2.90	3.73	4.01	2.54		
e^{-x}	Order	4	5	6	4	5	6	8-state		
	Area	512.2	576.7	655.2	664.0	759.7	872.0	1144.0		
	Delay	2.54	2.98	3.29	2.90	3.73	4.01	2.76		
e^{-2x}	Order	5	6	7	5	6	7	4	8	16
	Area	662.1	674.5	681.7	759.7	872.0	998.9	228.7	256.4	292.2
	Delay	3.19	3.40	3.62	3.73	4.01	3.78	2.37	2.54	3.10
sigmoid(x)	Order	5			–			4	8	16
	Area	512.7			–			226.7	253.8	287.5
	Delay	3.47			–			2.37	2.54	3.11

Chapter 5

Polynomial Computation in Unipolar Stochastic Logic

In this chapter we consider implementation of polynomials that map the interval [0,1] to itself or negative of itself, i.e., [-1,0]. We demonstrate that stochastic computation of polynomials can be implemented by using a stochastic subtractor and factorization. Two approaches are presented to compute stochastic subtraction in unipolar format. Unipolar subtraction can be approximated using multi-levels of combinational logic, including OR and AND gates. It is also shown that a stochastic subtractor can be implemented based on a unipolar divider. Computation of polynomials in stochastic unipolar format is proposed using scaled addition and proposed stochastic subtraction. We also propose stochastic computation of polynomials using factorization. Different implementations are considered for distinct locations of polynomial roots.

5.1 Subtraction in Stochastic Unipolar Representation

For the stochastic implementation of arbitrary polynomials in unipolar format, the unipolar subtraction is required for certain polynomials. In this section, two approaches to computing subtraction in stochastic unipolar logic are presented. The first method implements subtraction using multi-level combinational logic. More accurate approximations can be achieved with the increase in the number of levels. We also present a second approach to computing subtraction based on unipolar division. The stochastic

division is implemented using a counter and an LFSR. Comparisons of accuracy and hardware complexity of two proposed implementations are presented.

5.1.1 Unipolar Subtraction using Multi-Level Logic

It is known that combinational logic can be used to implement fundamental computations in stochastic unipolar representation. Table 5.1 lists corresponding Boolean and arithmetic operations for basic combinational logic, including AND, OR and NOT gates.

Table 5.1: The corresponding Boolean and arithmetic operations for AND, OR and NOT gates.

Logic Gate	AND	OR	NOT
Boolean Operation	$x \wedge y$	$x \vee y$	$\neg x$
Arithmetic Operation	xy	$x + y - xy$	$1 - x$

The design of subtraction module starts from the OR operation since we observe that the subtraction of two operands appears in the arithmetic operation represented by an OR gate. Consider the implementation of $y = x_1 - x_2$. The requirement of $x_1 > x_2$ needs to be satisfied since the range of unipolar output y is $[0, 1]$.

Fig. 5.1 presents implementations of stochastic subtraction using combinational logic. The NOR is used to perform imperfect subtraction as shown in Fig. 5.1(a). The inputs x_1 and x_2 are uncorrelated stochastic bit streams. The computation result is given by:

$$\begin{aligned} y &= 1 - [(1 - x_1) + x_2 - (1 - x_1)x_2] \\ &= x_1 - x_2 + (1 - x_1)x_2, \end{aligned} \tag{5.1}$$

which is a reasonable approximation of $x_1 - x_2$ when x_1 is near one and x_2 is near zero. However, the error increases significantly when the value of x_2 is close to x_1. The performance of approximate subtraction can be improved by introducing the enhancement unit as shown in Fig. 5.1(b). A one-bit delay element is used to decorrelate the signals generated from the same stochastic bit-streams $1 - x_1$ and x_2. The arithmetic results

of internal nodes and the output are described as follows:

$$n_1 = (1 - x_1) + x_2 - (1 - x_1)x_2 \tag{5.2}$$

$$m_1 = (1 - x_1)x_2 \tag{5.3}$$

$$\begin{aligned} y &= 1 - (n_1 + m_1 - n_1 m_1) \\ &= x_1 - x_2 + (1 - x_1)x_2(1 - x_1(1 - x_2)). \end{aligned} \tag{5.4}$$

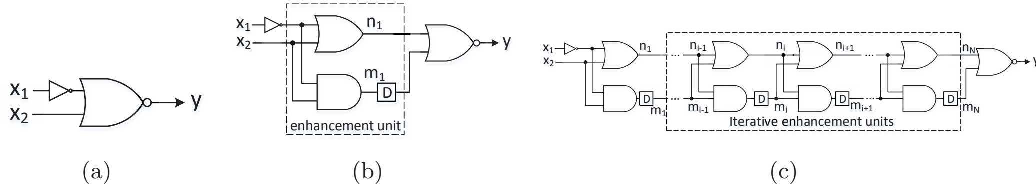
Notice that the AND gate is used for error correction. The approximation error $(1 - x_1)x_2$ at n_1 is offset by the result of AND gate at the next level, whereas an error with higher order $(n_1 m_1)$ is introduced. Compare the approximations of subtraction $x_1 - x_2$ using equations (5.1) and (5.4). The computation error of equation (5.1) is given by

$$\epsilon_1 = (1 - x_1)x_2. \tag{5.5}$$

The computation error in equation (5.4) is described as:

$$\epsilon_2 = (1 - x_1)x_2(1 - x_1(1 - x_2)). \tag{5.6}$$

Since the factor $(1 - x_1(1 - x_2))$ in ϵ_2 is less than one for $x_1, x_2 \in [0, 1]$, we obtain $\epsilon_2 < \epsilon_1$. Therefore, the enhancement unit reduces the error in the subtraction.



(a) (b) (c)

Figure 5.1: The implementations of stochastic subtraction using (a) a NOR gate, (b) the enhancement unit and a NOR gate, and (c) iterative enhancement units and a NOR gate.

To further improve the approximation accuracy, iterative enhancement units are implemented as shown in Fig. 5.1(c). By duplicating the enhancement units for multiple levels, we can further reduce the computation error. The results of internal nodes at any arbitrary i-th stage are given as follows:

$$n_i = n_{i-1} + m_{i-1} - n_{i-1} m_{i-1} \tag{5.7}$$

$$m_i = n_{i-1} m_{i-1} \tag{5.8}$$

where n_i and m_i represent internal results of the OR gate and AND gate at the i-th stage, respectively. Substituting n_i and m_i using (5.7) and (5.8), we obtain arithmetic results of internal nodes at the $(i+1)$-st stage:

$$\begin{aligned} n_{i+1} &= n_i + m_i - n_i m_i && (5.9)\\ &= n_{i-1} + m_{i-1} - n_i m_i && (5.10)\\ &= n_{i-1} + m_{i-1} - n_i n_{i-1} m_{i-1} && (5.11)\\ m_{i+1} &= n_i m_i && (5.12) \end{aligned}$$

As pointed out in the discussion of Fig. 5.1(b), AND gates perform error-correction for the implementation of subtraction. Comparing equation (5.11) to equation (5.7), we find that the error term has changed from $n_{i-1}m_{i-1}$ to $n_i n_{i-1} m_{i-1}$ after the enhancement. Since it is known that $0 < n_i < 1$ as n_i represents a probability value, the error is reduced with each added enhancement unit. With more levels of enhancement units, the subtraction is approximated more accurately. Assume there are total N-stages of iterative enhancement units. Repeatedly substituting n_{i+1} and m_{i+1} by n_i and m_i similar to the derivation from equations (5.9) to (5.10) for the final stage, we obtain:

$$\begin{aligned} y &= 1 - (n_N + m_N - n_N m_N)\\ &\vdots\\ &= 1 - (n_1 + m_1 - n_N m_N) && (5.13)\\ \Rightarrow y &= 1 - [(1 - x_1) + x_2 - n_N m_N]\\ &= x_1 - x_2 + n_N m_N && (5.14) \end{aligned}$$

where n_1 and m_1 are given by equations (5.2) and (5.3). The final computation error is $\epsilon_N = n_N m_N$.

Notice that in Fig. 5.1(c), one delay for each enhancing stage is not enough to completely decorrelate internal signals n_i and m_i. Consider two 3-stage versions of Fig. 5.1(c) shown in Fig. 5.2. The circuit illustrated in Fig. 5.2(a) includes only one delay element for each stage while the number of delay elements of the circuit illustrated in Fig. 5.2(b) increases exponentially for complete decorrelation for internal signals.

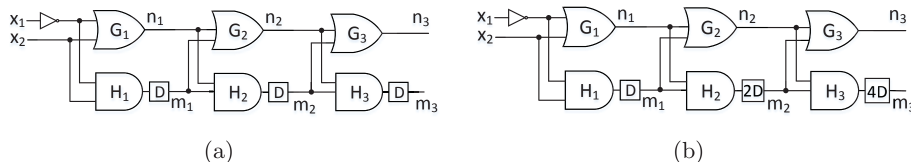


Figure 5.2: 3-stage versions of subtraction with (a) one delay for each stage, and (b) increasing delays for each stage.

In Fig. 5.2(a), n_1 depends on non-delayed version of inputs X_1 and X_2, which are denoted by $X_1(0)$ and $X_2(0)$. The internal node m_1 depends on one-delayed versions of X_1 and X_2, denoted by $X_1(1)$ and $X_2(1)$. Hence n_2 depends on $\{X_1(0), X_2(0), X_1(1), X_2(1)\}$, while m_2 depends on $\{X_1(1), X_2(1), X_1(2), X_2(2)\}$ due to the second delay at the output of the AND gate H_2. Both n_2 and m_2 depend on $X_1(1)$ and $X_2(1)$ such that they are correlated. Therefore, the accuracy of n_3 and m_3 may be degraded.

In Fig. 5.2(b), due to two delays at the output of H_2, m_2 depends on $\{X_1(2), X_2(2), X_1(3), X_2(3)\}$. Therefore, n_2 and m_2 are completely decorrelated with the increasing number of delays. We can observe that n_2 and m_2 depend on two versions of $X_i(k)$, where $k = 0$ and 1, while n_3 and m_3 depend on four versions of $X_i(k)$, where $0 \leq k \leq 3$. For internal state n_4 and m_4 at the next stage, the number of different delayed-version of inputs increases to eight. The number of different delayed-versions increases exponentially with the increase of stage. Accordingly, the number of delays should also increase exponentially for complete decorrelation.

The circuit with completely decorrelated internal signals is shown in Fig. 5.3. Notice that the number of one-bit delay elements increases exponentially. In this design, $2^i D$ indicates that 2^i delays are required for the ith stage.

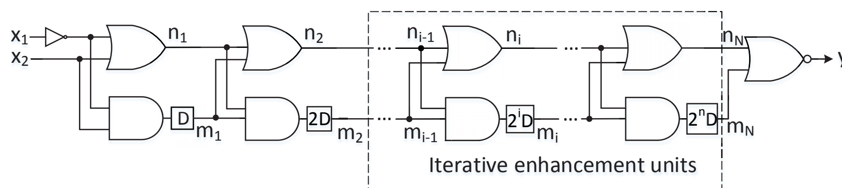


Figure 5.3: The unipolar subtractor using multiple levels of combinational logic with one delay for each stage.

Simulations were performed to test accuracies of the proposed subtraction shown

in Fig. 5.3. The length of stochastic bit streams is assumed to be 1024. Consider the subtraction $y = x_1 - x_2$, where $x_1 > x_2$. Fig. 5.4 shows simulation results of proposed designs as a function of x_2 for different values of x_1. In Fig. 5.4(a) the value of x_1 is given by 0.9 and x_2 is given by 0:0.01:0.9. In Fig. 5.4(b), x_1 is fixed at 0.7 and x_2 is given by 0:0.01:0.7. 1000 Monte Carlo runs were performed for each data point. The stochastic subtractor using iterative enhancement units (red) has three more enhancement stages than the implementation using one stage of enhancement unit (green). From simulation results, it is shown that the implementation with iterative enhancement units has better accuracy.

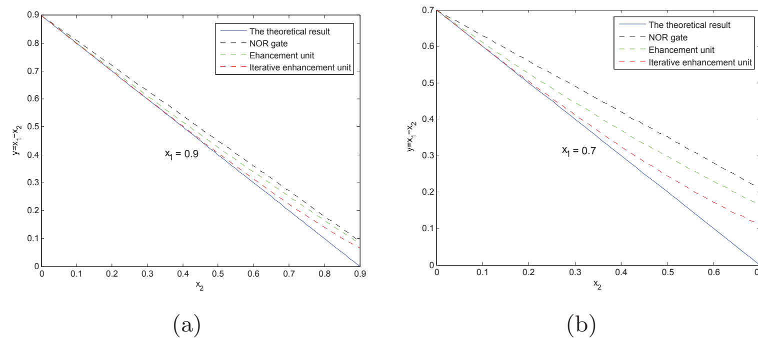


Figure 5.4: Simulation results of proposed stochastic subtractors using multi-level combinational logic gates are given as functions of x_2 for different values of x_1, where (a) $x_1 = 0.9$ and (b) $x_1 = 0.7$.

5.1.2 Computing Subtraction Based on Stochastic Unipolar Division

The polynomial computation is widely used in applications involving stochastic logic, such as image processing and machine learning system. Notice that in these fields, various complex functions such as trigonometric, exponential, logarithm and hyperbolic functions are used for computational kernels. The first step of the stochastic implementation of these functions is to describe these functions based on polynomials generated using techniques including Taylor expansion and Lagrange interpolation. Then the stochastic implementation of complex arithmetic functions is converted to *polynomial computations*.

Consider the stochastic implementation of $y = x_1 - x_2$, where $0 < x_2 < x_1 < 1$. A

subtractor can be implemented based on an arithmetic expression shown below:

$$y = x_1 - x_2 = x_1(1 - x_2/x_1), \tag{5.15}$$

which corresponds to the Boolean expression:

$$y = x_1 \wedge (\overline{x_2/x_1}). \tag{5.16}$$

Fig. 5.5 shows the implementation of stochastic subtraction using equation (5.15).

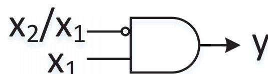


Figure 5.5: The implementation of stochastic subtraction using equation (5.15).

In Fig. 5.5, generating stochastic bit stream x_2/x_1 requires stochastic unipolar division, which can be implemented using the divider proposed by Gaines [3] as shown in Fig. 5.6(a).

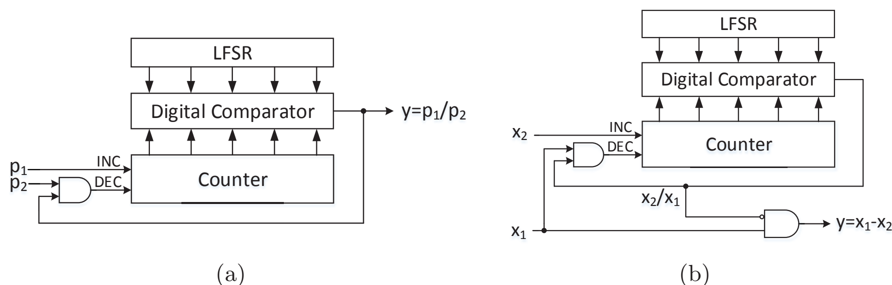


Figure 5.6: (a) Stochastic divider in unipolar format where $p_1 < p_2$, (b) Stochastic subtractor using unipolar divider.

Consider the implementation of the division p_1/p_2. The divider is implemented based on a binary counter, which may be incremented or decremented by unit count. The count increases by unity at a clock pulse if the INCREMENT (INC) line is ON and DECREMENT (DEC) line is OFF, whereas it decreases by unity at a clock pulse if the converse situation holds. If the lines are both OFF or both ON, then the count remains unchanged. As shown in Fig. 5.6(a) the input p_1 is fed to the INC line of the counter. The stochastic output, represented as p_o, is fed back through an AND gate,

together with the input p_2, into the DEC line of the counter, to form the term $p_o p_2$. The stochastic output p_o is generated by comparing the count to the value of the LFSR, which generates a random number. If the random value is greater than the count, the output p_o is one; otherwise, the value of p_o is zero. A simple explanation of how division occurs was given by Gaines. Consider that in equilibrium the probability that the count will increase must equal the probability that it will decrease, so that $p_o p_2 = p_1$, and then $p_o = p_1/p_2$. More details can be found in Section 4 of [3].

Fig. 5.6(b) shows the overall circuit diagram of subtraction using stochastic unipolar divider. The division module is used to generate the stochastic bit stream representing x_2/x_1.

Simulations were performed to test accuracies of proposed stochastic subtractions based on the unipolar divider. The length of stochastic bit streams is 1024. Consider the subtraction $y = x_1 - x_2$, where $x_1 > x_2$. Fig. 5.7 shows simulation results of proposed designs as a function of x_2 for different values of x_1. The simulation results of approximate subtraction using multi-level combinational logic are also illustrated for comparison. In Fig. 5.7, the value of x_1 is given by 0.3, 0.5, 0.7 and 0.9, respectively. The value of x_2 is given by 0:0.01:0.3, 0:0.01:0.5, 0:0.01:0.7 and 0:0.01:0.9. 1000 Monte Carlo runs were performed for each data point. The stochastic subtractor using iterative enhancement units has four enhancement stages. From simulation results, it is shown that the subtractor using stochastic divider has better accuracy than the approximation using multi-level combinational logic. Table 5.2 presents the output Mean Absolute Error (MAE) of stochastic subtractions $x_1 - x_2$ for different values of x_1, where SDiv represents the implementation using a divider and SIter denotes the implementation using iterative enhance units. Computational errors decrease with the increase of the output value.

Table 5.2: The output Mean Absolute Error (MAE) of stochastic subtractions $x_1 - x_2$ for different values of x_1.

x_1	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9
SDiv	0.0020	0.0024	0.0025	0.0022	0.0028	0.0020	0.0018	0.0015	0.0016
SIter	0.0315	0.0423	0.0460	0.0450	0.0420	0.0371	0.0302	0.0230	0.0134

Although the subtractor based on stochastic divider has better accuracy than the

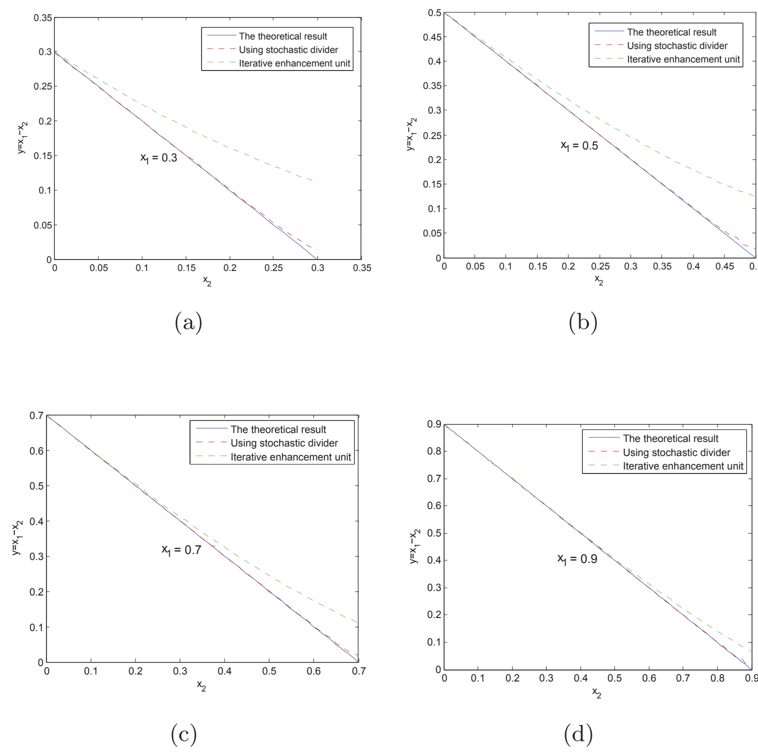


Figure 5.7: Simulation results of proposed stochastic subtractors based on division and using iterative enhancement units are given as functions of x_2 for different values of x_1, where (a) $x_1 = 0.3$ (b) $x_1 = 0.5$ (c) $x_1 = 0.7$ and (b) $x_1 = 0.9$.

approximate subtractor using iterative enhancement units, the latter implementation requires less hardware complexity. Table 5.3 presents the hardware complexity of the subtractor based on stochastic divider (Subtractor 1). The synthesis result of the subtractor using 4-stage enhancement units (Subtractor 2) is also presented. We also list the hardware complexity of a stochastic scaled adder shown in Fig. 1.2(c) and a 10-bit LFSR as references. The scaled adder includes a multiplexer and a stochastic number generator, which is not shown in Fig. 1.2(c), to generate the select signal s. The area results include all computational logic and required SNGs for generating stochastic sequences.

Table 5.3: Synthesis results of the subtractor based on stochastic divider (subtractor 1), the subtractor using 4-stage enhancement units (subtractor 2) and stochastic scaled adder.

Implementation	Subtractor 1	Subtractor 2	Scaled adder	LFSR
Area (μm^2)	459.68 (238.92%)	71.24 (37.03%)	192.40 (100%)	168.68 (87.67%)

From the table, we observe that hardware complexity of the stochastic subtractor using 4-stage enhancement units (Subtractor 2) is 84.50% less than the subtractor based on a stochastic divider (Subtractor 1). Therefore, the proposed two designs provide a trade-off between accuracy and hardware efficiency.

5.2 Polynomial Computation Using Unipolar Subtraction

One straightforward approach to computing polynomials in unipolar stochastic representation is directly using subtraction and scaled addition. This method divides all polynomials into two groups. One group contains all terms with positive coefficients and the other with all negative. Polynomials must satisfy the requirement that computational results are in unit interval. Consider computing a polynomial:

$$p(x) = a_0 + a_1 x + a_2 x^2 + \cdots + a_n x^n \tag{5.17}$$

in stochastic unipolar representation, where $0 \leq p(x) \leq 1$ for any given $x \in [0, 1]$. Assume that all coefficients a_i's are positive. The polynomial can be simply implemented

using scaled additions with multiple inputs. If negative coefficients exist in this polynomial, unipolar subtraction is required for implementation. Assume that a negative coefficient is described as $a_i = -b_i$, where $b_i > 0$. The polynomial (17) is rewritten as follows:

$$p(x) = a_0 + a_1 x + a_2 x^2 + \cdots + a_n x^n = \underbrace{\sum_{i=0}^{n_1} a_i x_i^{\alpha}}_{Q(x)} - \underbrace{\sum_{j=0}^{n_2} b_j x_j^{\beta}}_{R(x)}, \tag{5.18}$$

where $a_i > 0$ and $b_j > 0$. The number of positive coefficients is $n_1 + 1$ while the number of negative coefficients is given by $n_2 + 1$. Notice that $n = n_1 + n_2 + 2$. $Q(x)$ computes the sum of all terms with positive coefficients, whereas $R(x)$ adds additive inverses of all terms with negative coefficients. Both of them are implemented using stochastic scaled additions. Then a unipolar subtractor is used to compute the final result of the polynomial. Since the final result is guaranteed in unit interval, the requirement $0 \leq R(x) \leq Q(x) \leq 1$ for the unipolar subtraction $Q(x) - R(x)$ is satisfied.

Two examples are illustrated to present implementations of polynomial computation in stochastic unipolar format for two categories. The first example is given by $p_1(x) = \frac{1}{8} + \frac{1}{8}x + \frac{1}{4}x^2 + \frac{3}{8}x^3$, where all coefficients are positive. The second example is given by $p_2(x) = \frac{2}{3} - \frac{1}{2}x + \frac{1}{2}x^2 - \frac{1}{4}x^3$, where negative coefficients exist.

5.2.1 Case-I: all positive coefficients

In polynomial

$$p_1(x) = \frac{1}{8} + \frac{1}{8}x + \frac{1}{4}x^2 + \frac{3}{8}x^3, \tag{5.19}$$

all coefficients are positive. The polynomial is implemented using multi-levels of combinational logic as shown in Fig. 5.8 [30].

All coefficients and the input are represented in stochastic unipolar format. One-bit delay elements are used for decorrelation. x^i is calculated using AND gates which perform unipolar multiplications. In the first level, inputs of two multiplexers are stochastic bit streams denoting different powers of x: $\{1, x, x^2, x^3\}$. Notice that both coefficients for inputs 1 and x are $1/8$. Thus, the select signal s_1 is given by

$$s_1 = \frac{1/8}{1/8 + 1/8} = 0.5. \tag{5.20}$$

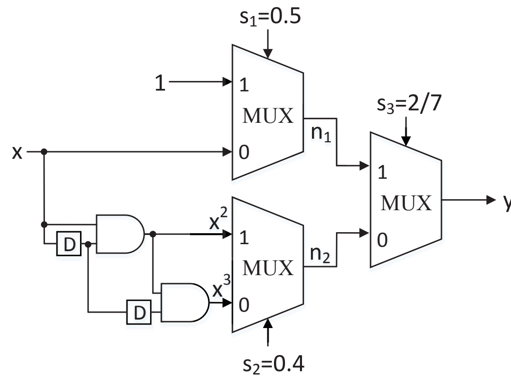


Figure 5.8: Stochastic implementation of polynomial (19) using multi-levels of multiplexers.

The computational result of node n_1 is described as:

$$n_1 = 0.5 \cdot 1 + (1 - 0.5)x = 0.5(1 + x) = 4 \cdot (\frac{1}{8} + \frac{1}{8}x), \quad (5.21)$$

which is a scaled version of original result and the scaling factor is 4. Coefficients for power bases x^2 and x^3 are 1/4 and 3/8, respectively. Therefore the select signal s_2 is given by:

$$s_2 = \frac{1/4}{1/4 + 3/8} = 0.4. \quad (5.22)$$

The computational result of node n_2 is given by:

$$n_2 = 0.4x^2 + (1 - 0.4)x^3 = 0.4x^2 + 0.6x^3 = \frac{8}{5} \cdot (\frac{1}{4}x^2 + \frac{3}{8}x^3), \quad (5.23)$$

where the result is scaled by 8/5 compared to the original result. The select signal s_3 is determined by all coefficients as follows:

$$s_3 = \frac{1/8 + 1/8}{1/8 + 1/8 + 1/4 + 3/8} = \frac{2}{7}. \quad (5.24)$$

The final output is given by:

$$\begin{aligned} y &= \frac{2}{7}n_1 + (1 - \frac{2}{7})n_2 = \frac{8}{7} \cdot (\frac{1}{8} + \frac{1}{8}x) + \frac{8}{7} \cdot (\frac{1}{4}x^2 + \frac{3}{8}x^3) \\ &= \frac{8}{7} \cdot (\frac{1}{8} + \frac{1}{8}x + \frac{1}{4}x^2 + \frac{3}{8}x^3) = \frac{8}{7} \cdot p_1(x) \end{aligned} \quad (5.25)$$

The implementation computes a scaled result for $p_1(x)$, where the scaling factor is determined by the sum of all coefficients. Notice that it is possible to use this approach

to implement polynomials with coefficients **greater** than 1, since we only need to use **fractions** of coefficients to determine select signals. Simulation results of the proposed stochastic polynomial computation are shown in Fig. 5.9. The desired polynomial is $\frac{8}{7} \cdot p_1(x)$. In our simulations, x is given by 0:0.01:1. 1000 Monte Carlo runs were performed for each data point. The length of stochastic bit streams is 1024.

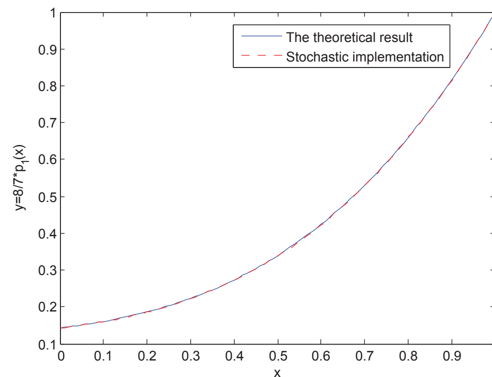


Figure 5.9: Comparison of simulation results of the proposed stochastic implementation for polynomial (19) and theoretical results.

5.2.2 Case-II: positive and negative coefficients

Consider the polynomial:

$$p_2(x) = \frac{2}{3} - \frac{1}{2}x + \frac{1}{2}x^2 - \frac{1}{4}x^3 = (\frac{2}{3} + \frac{1}{2}x^2) - (\frac{1}{2}x + \frac{1}{4}x^3), \tag{5.26}$$

where negative coefficients exist. The polynomial is implemented using a stochastic subtractor as shown in Fig. 5.10.

The select signal s_1 is given by:

$$s_1 = \frac{2/3}{2/3 + 1/2} = \frac{4}{7}. \tag{5.27}$$

The computational result of n_1 is described as:

$$n_1 = \frac{4}{7} \cdot 1 + (1 - \frac{4}{7})x^2 = \frac{4}{7} + \frac{3}{7}x^2 = \frac{6}{7} \cdot (\frac{2}{3} + \frac{1}{2}x^2). \tag{5.28}$$

Compared to the original value, the calculated result is scaled by 6/7. The select signal s_2 is given by:

$$s_2 = \frac{1/2}{1/2 + 1/4} = \frac{2}{3}. \tag{5.29}$$

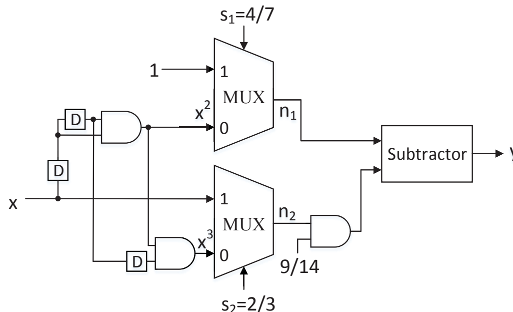


Figure 5.10: Stochastic implementation of polynomial (26) based on unipolar subtractor.

The computational result is described as:

$$n_2 = \frac{2}{3}x + (1 - \frac{2}{3})x^3 = \frac{2}{3}x + \frac{1}{3}x^3 = \frac{4}{3} \cdot (\frac{1}{2}x + \frac{1}{4}x^3), \quad (5.30)$$

which is scaled by 4/3. To guarantee the correctness of the subtraction, two inputs of the subtractor must be equivalently scaled. Therefore, n_2 is multiplied by 9/14 and scaling factor is modified as $\frac{4}{3} \cdot \frac{9}{14} = \frac{6}{7}$, which is same as n_1. The final output is given by:

$$\begin{aligned} y =& n_1 - \frac{9}{14}n_2 = \frac{6}{7} \cdot (\frac{2}{3} + \frac{1}{2}x^2) - \frac{9}{14} \cdot \frac{4}{3} \cdot (\frac{1}{2}x + \frac{1}{4}x^3) \\ &= \frac{6}{7} \cdot (\frac{2}{3} - \frac{1}{2}x + \frac{1}{2}x^2 - \frac{1}{4}x^3) = \frac{6}{7} \cdot p_2(x). \end{aligned} \quad (5.31)$$

In this implementation the desired result is scaled by 6/7. The subtractor can be realized using either the structure based on iterative enhancement units or the design using stochastic divider. Simulation results of the proposed stochastic polynomial computation are shown in Fig. 5.11. The desired polynomial is $\frac{6}{7} \cdot p_2(x)$. The subtractor in stochastic implementation is based on unipolar divider. In our simulations, x is given by 0:0.01:1. 1000 Monte Carlo runs were performed for each data point. The length of stochastic bit streams is 1024.

5.3 Polynomial Computation Using Factorization

In the previous section, the proposed approach for polynomial computation is based on stochastic scaled addition and subtraction. Notice that accurate implementation of unipolar subtractor leads to increase in hardware complexity. In this section, we present

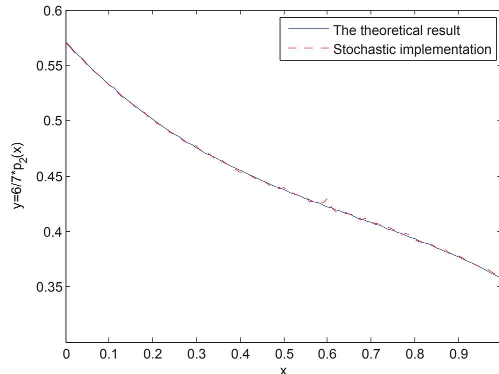


Figure 5.11: Comparison of simulation results of the proposed stochastic implementation for polynomial (26) and theoretical results.

stochastic polynomial computation using factorization. This method leads to various implementations depending on different locations of polynomial roots. For certain locations of polynomial roots, stochastic subtractors are not required although there are negative coefficients in polynomials.

Consider the polynomial in equation (5.17). Since all coefficients are real, roots of $p(x) = 0$ are either real or complex conjugates. Therefore, $p(x)$ can be represented by a product of first-order and second-order factors as follows:

$$p(x) = a_0 + a_1x + a_2x^2 + \cdots + a_nx^n = d \cdot \prod_{i=0}^{n_1}(1 - c_ix) \cdot \prod_{j=0}^{n_2}(1 - a_jx + b_jx^2), \quad (5.32)$$

where d denotes the scaling of stochastic implementation. Assume that the number of first-order and second-order factors are $n_1 + 1$ and $n_2 + 1$, respectively. In stochastic unipolar representation, $p(x)$ can be implemented by multiplying all factors using AND gates.

Consider the first-order factor $1-cx$. The value of c is determined by a corresponding real root (r) of $p(x) = 0$. Possible locations of real roots are shown in Fig. 5.12.

In Fig. 5.12(a), we have $r \leq 0$ and then $c = 1/r < 0$. The first-order factor $1 - cx$ is rewritten as $1 + c'x$ where $c' = -c$ and $c' > 0$. The first-order factor can be implemented using a multiplexer as described in Section 5.2.1. Since a fractional coefficient is calculated for the select signal of MUX, the implementation is still feasible for $c' > 1$ since the computed result is a scaled version.

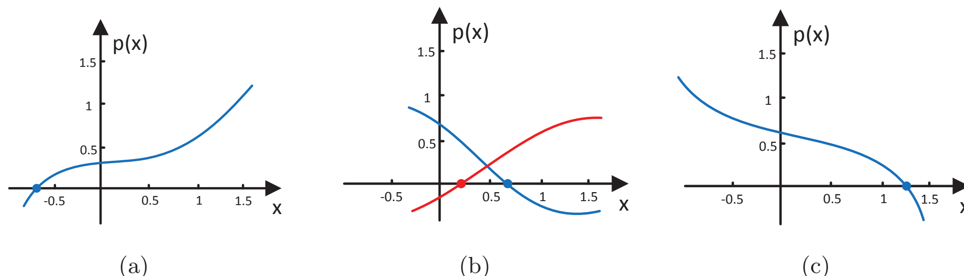


Figure 5.12: Three possible locations of a real root (r) of $p(x)$: (a) $r \leq 0$, (b) $0 < r \leq 1$ and (c) $1 \leq r$

In Fig. 5.12(b), $0 < r \leq 1$ and then $c > 1$. In this case, $1 - cx$ is infeasible in stochastic unipolar format. However, notice that two possible curves of $p(x)$ shown in Fig. 5.12(b) violate the constraint $0 \leq p(x) \leq 1$ given $x \in [0, 1]$. These root locations lead to negative polynomial values. Thus, we assume that no real root is present between 0 and 1.

In Fig. 5.12(c), $r > 1$ and $0 < c = 1/r < 1$. The first-order factor $1 - cx$ is implemented using a NAND gate, where inputs are stochastic bit streams representing c and x.

From the discussion above, we can see that the implementation of first-order factor is straightforward. However, the implementation of second-order factors in $p(x)$ is non-trivial. In Sections 5.3.1 to 5.3.6, implementations of second-order factors are presented in detail depending on various locations of complex conjugate roots. Consider $1-ax+bx^2$ in (32). Assume that the second-order factor is introduced by complex conjugate roots $x_1 = u + iv$ and $x_2 = u - iv$. Then coefficients a and b are expressed as follows:

$$\begin{cases} a = \frac{1}{x_1} + \frac{1}{x_2} = \frac{2u}{u^2+v^2} \\ b = \frac{1}{x_1} \cdot \frac{1}{x_2} = \frac{1}{u^2+v^2} \end{cases} \tag{5.33}$$

Different locations of roots are specified by different ranges of u and v.

5.3.1 Location of complex roots: $u < 0$

In this case, r_1 and r_2 are on the left side of imaginary axis as shown in Fig. 5.13(a) by the red area. We obtain $a = \frac{2u}{u^2+v^2} < 0$ and $b = \frac{1}{u^2+v^2} > 0$. All coefficients in $1-ax+bx^2$ are positive. Therefore, second-order factors introduced by complex conjugate roots on

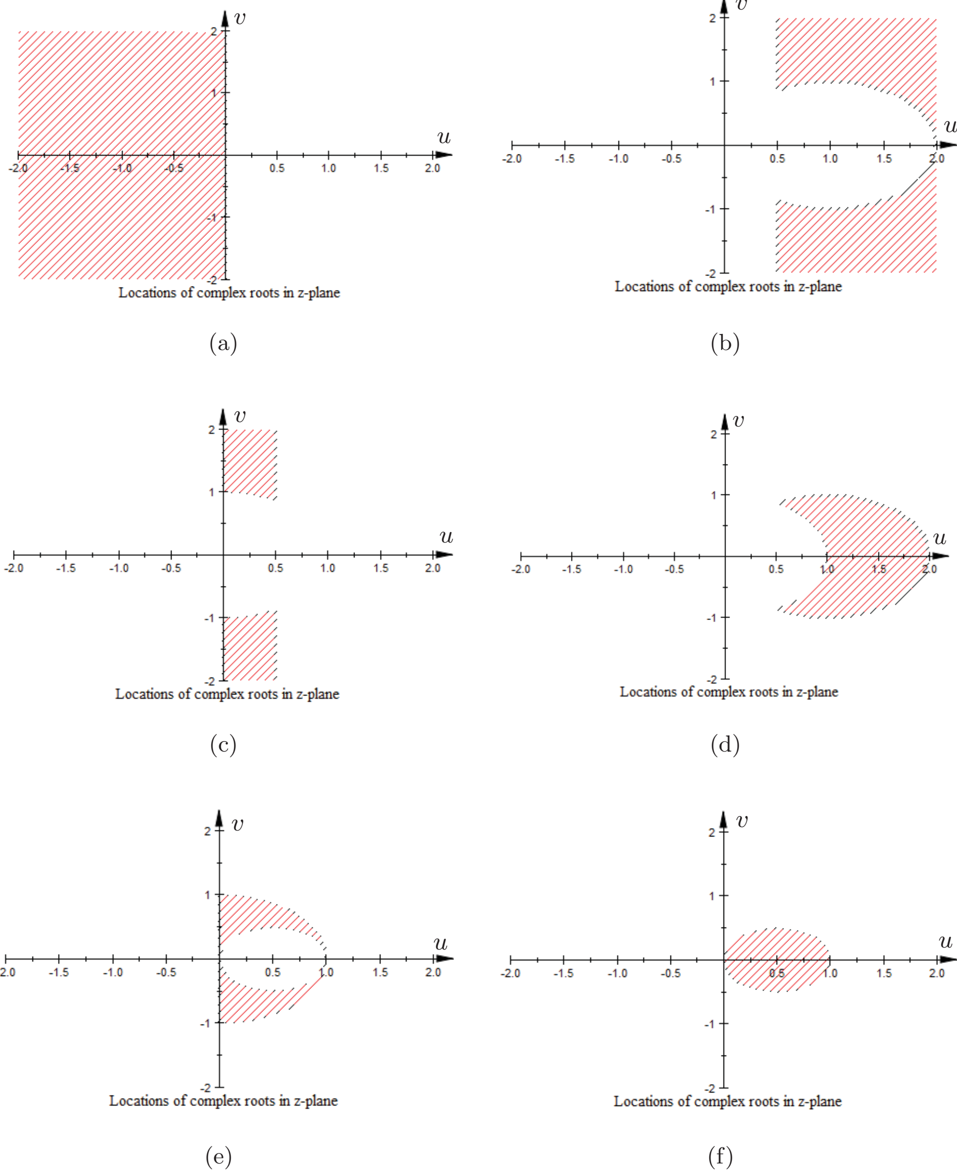


Figure 5.13: Various locations of complex conjugate roots, which are determined by constraints of u and v: (a) $u < 0$, (b) $(u^2 + v^2 - 2u \geq 0)\&\&(u \geq 0.5)$, (c) $(0 < u < 0.5)\&\&(u^2 + v^2 \geq 1)$, (d) $(u^2 + v^2 - 2u < 0)\&\&(u^2 + v^2 > 1)\&\&(u \geq 0.5)$, (e) $((u - 0.5)^2 + v^2 \geq 0.25)\&\&(u^2 + v^2 < 1)\&\&(u > 0)$, and (f) $(u - 0.5)^2 + v^2 < 0.25$.

the left side of imaginary axis can be implemented using multiplexers as presented in Section 5.2.1.

5.3.2 Location of complex roots: $(u^2 + v^2 - 2u \geq 0)$ and $(u \geq 0.5)$

Possible locations of complex roots under these constrains are indicated by red area in Fig. 5.13(b). Coefficients a and b are determined by u and v as shown below:

$$u^2 + v^2 - 2u \geq 0 \Rightarrow a = \frac{2u}{u^2 + v^2} \leq 1$$
$$u \geq 0.5 \Rightarrow b = \frac{1}{u^2 + v^2} \leq \frac{2u}{u^2 + v^2} = a$$

Therefore, we obtain $0 < b \leq a \leq 1$. The second-order factor is transformed using **Horner's rule** as follows:

$$1 - ax + bx^2 = 1 - ax(1 - \frac{b}{a}x), \tag{5.34}$$

where $a \leq 1$ and $b/a \leq 1$. Then both a and b/a can be represented in stochastic unipolar format. The second-order factor is implemented based on equation (5.34) by using simple combinational logic as shown in Fig. 5.14. The implementation includes two NAND gates. The coefficients and the input are in stochastic unipolar format.

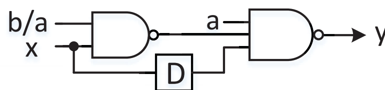


Figure 5.14: Stochastic implementation of second-order factor $1-ax+bx^2$ using equation (5.34).

5.3.3 Location of complex roots: $(0 < u < 0.5)$ and $(u^2 + v^2 \geq 1)$

In this section, consider complex conjugate roots located in the red area shown in Fig. 5.13(c). To implement $1 - ax + bx^2$ using simple combinational logic, the variable

x in the second-order factor is substituted by $1-t$ as follows:

$$\begin{aligned}1-ax+bx^2 &= 1-a(1-t)+b(1-t)^2 \\ &= 1-a+b-(2b-a)t+bt^2 \\ &= (1-a+b)(1-\frac{2b-a}{1-a+b}t+\frac{b}{1-a+b}t^2),\end{aligned} \tag{5.35}$$

where $t=1-x$. Note that roots of $1-ax+bx^2=0$ are given by $x_1=u+iv$ and $x_2=u-iv$. Then roots of $1-a+b+(a-2b)t+bt^2=0$ are described as follows:

$$\begin{cases} t_1 = 1-x_1 = 1-(u+iv) = (1-u)-iv = u'+iv' \\ t_2 = 1-x_2 = 1-(u-iv) = (1-u)+iv = u'-iv' \end{cases} \tag{5.36}$$

where u' and v' represent real and imaginary parts of t_1 and t_2. The relation between complex roots x_i and t_i is given by $u'=1-u$ and $v'=-v$. Possible locations of t_1 and t_2 in complex-plane are derived from constraints of u and v as follows:

$$\begin{cases} u^2+v^2 \geq 1 \\ u < 0.5 \end{cases} \Rightarrow \begin{cases} (1-u')^2+(v')^2 \geq 1 \\ 1-u' < 0.5 \end{cases} \Rightarrow \begin{cases} (u')^2+(v')^2-2u' \geq 0 \\ u' > 0.5 \end{cases} \tag{5.37}$$

We observe that ranges of u' and v' are same as constraints of u and v after the substitution of $x=1-t$. Constraints of coefficients in equation (35) are derived as follows:

$$(u')^2+(v')^2-2u' \geq 0 \Rightarrow \frac{2b-a}{1-a+b} \leq 1 \tag{5.38}$$

$$u' > 0.5 \Rightarrow \frac{b}{1-a+b} < \frac{2b-a}{1-a+b} \tag{5.39}$$

and we obtain:

$$0 < \frac{b}{1-a+b} < \frac{2b-a}{1-a+b} \leq 1. \tag{5.40}$$

Then equation (35) can be further transformed using Horner's rule as follows:

$$(1-a+b)(1-\frac{2b-a}{1-a+b}t+\frac{b}{1-a+b}t^2) = (1-a+b)(1-\frac{2b-a}{1-a+b}t(1-\frac{b}{2b-a}t)). \tag{5.41}$$

In the equation above, coefficients $(2b-a)/(1-a+b)$ and $b/(2b-a)$ are in the range of $[0,1]$. The coefficient $(1-a+b)$ is also guaranteed in the required range of unipolar format since it is the value of second-order factor $1-ax+bx^2$ at $x=1$. The stochastic unipolar implementation of $1-ax+bx^2$ with roots in this region is shown in Fig. 5.15. All coefficients and input are represented in unipolar format.

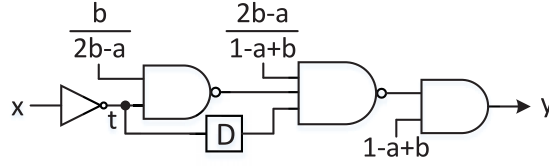


Figure 5.15: Stochastic implementation of second-order factor $1-ax+bx^2$ using equation (5.41).

5.3.4 Location of complex roots: $(u^2+v^2-2u<0)$ and $(u^2+v^2>1)$ and $(u \geq 0.5)$

Consider complex conjugate roots located in the red area shown in Fig. 5.13(d). The implementation of $1-ax+bx^2$ is similar to that in Section 5.3.3. Substitute x by $1-t$ as shown in equation (5.35). However, only one out of two constraints in (5.37) holds true, that is:

$$u^2+v^2 \geq 1 \Rightarrow (u')^2+(v')^2-2u' \geq 0, \tag{5.42}$$

whereas $u' < 0.5$, which contradicts (5.37). Therefore we obtain $\frac{2b-a}{1-a+b} \leq 1$ which is same as (5.38) but $\frac{b}{1-a+b} > \frac{2b-a}{1-a+b}$ which is opposite to (5.39). Then equation (5.41) is modified as follows:

$$\begin{aligned}
&(1-a+b)(1-\frac{2b-a}{1-a+b}t+\frac{b}{1-a+b}t^2) \\
&=(1-a+b)(1-\frac{2b-a}{1-a+b}t+\frac{2b-a}{1-a+b}t^2+\frac{a-b}{1-a+b}t^2) \\
&=(1-a+b)(1-\frac{2b-a}{1-a+b}t(1-t)+\frac{a-b}{1-a+b}t^2) \\
&=(1-a+b)(1-\frac{2b-a}{1-a+b}t(1-t))+(a-b)t^2,
\end{aligned} \tag{5.43}$$

where $\frac{a-b}{1-a+b}$ is a positive number since $\frac{b}{1-a+b} > \frac{2b-a}{1-a+b}$. The implementation of a second-order factor with roots in this region is shown in Fig. 5.16. One more multiplexer is required to add the extra positive term in (5.43), compared to Fig. 5.15. In this implementation, all coefficients and input are in unipolar format. The internal node n_1 is computed by $(1-\frac{2b-a}{1-a+b}t(1-t))$ and $1-t$ represents x. The select signal of the multiplexer is given by

$$\frac{1-a+b}{(1-a+b)+(a-b)} = 1-a+b. \tag{5.44}$$

The scaling of the multiplexer output is $(1-a+b)+(a-b)=1$.

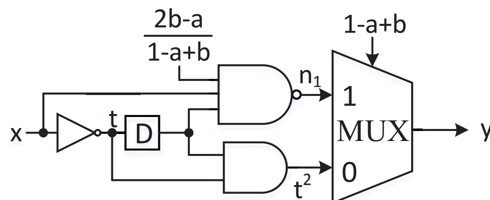


Figure 5.16: Stochastic implementation of second-order factor $1-ax+bx^2$ using equation (5.43).

5.3.5 Location of complex roots: $((u-0.5)^2+v^2 \geq 0.25)$ and $(u^2+v^2 < 1)$ and $(u > 0)$

Locations of complex conjugate roots are given by the red area as shown in Fig. 5.13(e). Coefficients a and b in second-order factor $1 - ax + bx^2$ are determined by constraints on u and v as follows:

$$\begin{cases} (u-0.5)^2 + v^2 \geq 0.25 \Rightarrow a = \frac{2u}{u^2+v^2} \leq 2 \\ u^2 + v^2 < 1 \Rightarrow b = \frac{1}{u^2+v^2} > 1 \end{cases} \tag{5.45}$$

Consider the implementation shown in Fig. 5.17.

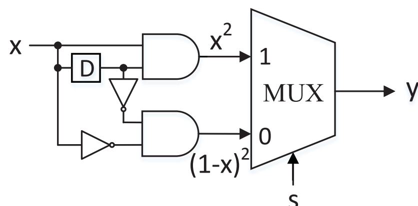


Figure 5.17: Stochastic implementation corresponding to equation (5.46).

The output is given by:

$$y = sx^2 + (1-s)(1-x)^2 = x^2 - 2(1-s)x + (1-s) \tag{5.46}$$

The second-order factor with roots in this area can be implemented based on this multiplexer. The transformation is given by:

$$\begin{aligned} 1 - ax^2 + bx^2 &= 1 - \frac{a}{2} + \frac{a}{2} - ax + x^2 + (b-1)x^2 \\ &= 1 - \frac{a}{2} + \underbrace{(1-s) - 2(1-s)x + x^2}_{T(x)} + (b-1)x^2, \end{aligned} \tag{5.47}$$

where $s = 1 - a/2$. $T(x)$ is implemented using the structure shown in Fig. 5.17. Since $0 < a < 2$, we obtain $0 < 1 - a/2 < 1$. Additionally, $b - 1$ is positive since $b > 1$. Then $(1 - a/2) + T(x) + (b - 1)x^2$ can be implemented using multiplexers to perform scaled addition. The overall circuit to implement $1 - ax + bx^2$ using equation (5.47) is shown in Fig. 5.18. $T(x)$ is computed at node n_1. Two remaining terms are added using multiplexers. All coefficients and input are given in stochastic unipolar bit-streams.

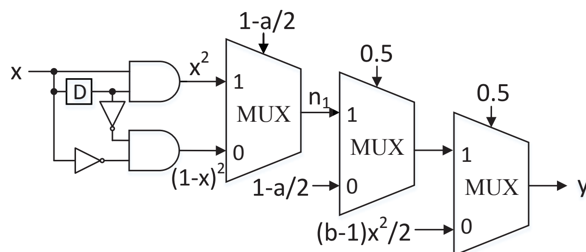


Figure 5.18: Stochastic implementation of second-order factor $1-ax+bx^2$ using equation (5.47).

5.3.6 Location of complex roots: $(u - 0.5)^2 + v^2 < 0.25$

Possible locations of complex roots under these constrains are indicated by red area in Fig. 5.13(f). In Section 5.3.5, $(u - 0.5)^2 + v^2 \geq 0.25$ is required to guarantee $a \leq 2$ in second-order factor. Otherwise, in equation (5.47), the coefficient $1 - a/2$ is negative. In this case, a stochastic subtractor is required for the implementation. According to equation (5.45), we obtain $a > 2$ and $b > 1$. Factor $1 - ax + bx^2$ can not be implemented with simple combinational logic. However, as discussed in Section 5.2.2, it can be implemented using unipolar subtractor based on transformed polynomial $(1 + bx^2) - ax$ as shown in Fig. 5.19.

An example of $(x - 0.5)^2$ is considered as a special case, which has two real roots at 0.5. The second-order polynomial is transformed as: $(x - 0.5)^2 = x^2 - x + 0.25 = (x^2 + 0.25) - x$. Simulation results are shown in Fig. 5.20. The subtractor is based on a stochastic divider.

Actually, if there are more than one second-order factors with roots in this area, the best design strategy is to implement the overall polynomial using the approach proposed

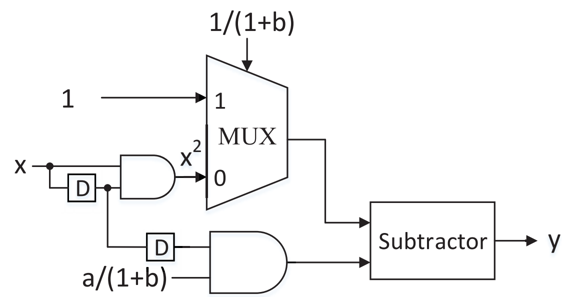


Figure 5.19: Stochastic implementation of the transformed second-order factor $(1 + bx^2) - ax$.

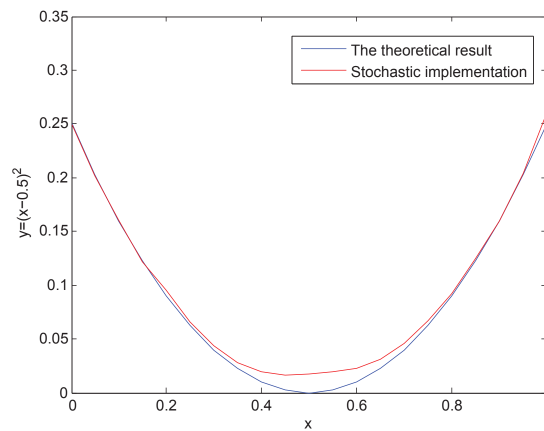


Figure 5.20: Simulation results of the stochastic implementation for $(x - 0.5)^2$.

in Section 5.2.2 without factorization. It ensures that fewest stochastic subtractors are used in the implementation to reduce hardware complexity.

In the factorization method for the stochastic implementation of arbitrary polynomials, we try to avoid the usage of SC adders by implementing the factor $1 - cx$ using NAND gates without scaling. Notice that SC adders are prone to precision loss due to implicit scaling introduced from the select signal. Also, hardware complexity is slightly reduced by using less MUXs. The proposed factorization method is well suited for the stochastic implementation of high order polynomials. Since all coefficients in polynomials are real values, an objective polynomial can be described as a product of first and second order factors, which can be implemented based on root locations.

5.4 Comparison of simulation and synthesis results

We performed experiments to test performance and generate synthesis results for proposed polynomial computations in stochastic unipolar representation. Comparisons of accuracy and hardware complexity for proposed designs and the implementation using Bernstein polynomials are presented in Sections 5.4.1 and 5.4.2, where objective functions are polynomials and complex arithmetic functions, respectively. In Section 5.4.3, we present comparisons of the proposed design and the implementation using spectral transform (STRAUSS) [51] [52]. Since this paper focuses on synthesis methods using combinational logic, the finite state machine (FSM) method is not considered for comparisons.

For the factorization method, no subtractor is required for factors with complex roots located in regions shown in Fig. 5.13(a)-(e) and therefore, the accuracy of the factorization method has no significant difference for these cases. However, if there are complex roots located in the region shown in Fig. 5.13(f), the stochastic subtractor is required and the performance is degraded especially when results are near zero. In Section 5.4.1, an example is given to illustrate the implementation without subtractors while the example presented in Section 5.4.2 is used to illustrate the implementation involving subtractors.

5.4.1 Stochastic polynomial computations

Consider a 3th-order polynomial (from [5]):

$$f(x) = \frac{1}{4} + \frac{9}{8}x - \frac{15}{8}x^2 + \frac{5}{4}x^3, \tag{5.48}$$

where $0 \leq f(x) \leq 1$ given $x \in [0,1]$. The stochastic computation of the polynomial can be implemented using two proposed approaches.

The **Method-I** proposed in Section 5.2 requires one stochastic subtractor and it is based on transformed polynomial:

$$f(x) = (\frac{1}{4} + \frac{9}{8}x + \frac{5}{4}x^3) - \frac{15}{8}x^2. \tag{5.49}$$

Fig. 5.21 shows circuit diagram of unipolar $f(x)$ computation using the transformed polynomial above. The select signal s_1 is given by $\frac{1/4}{1/4+9/8} = \frac{2}{11}$ and s_2 is given

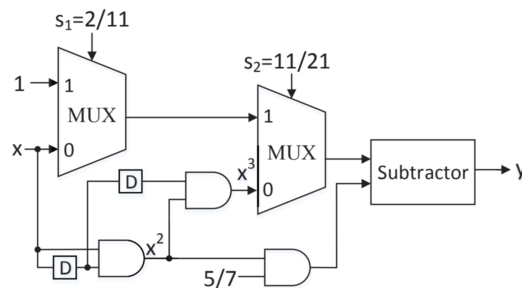


Figure 5.21: Stochastic unipolar implementation of $f(x)$ using subtractor (Method-I).

by $\frac{1/4+9/8}{1/4+9/8+5/4} = \frac{11}{21}$. The final result is scaled by $\frac{1}{1/4+9/8+5/4} = \frac{8}{21}$.

Method-II corresponds to the proposed implementation in Section 5.3 using factorization. The polynomial is factorized as follows:

$$f(x) = \frac{1}{4} + \frac{9}{8}x - \frac{15}{8}x^2 + \frac{5}{4}x^3 = (\frac{1}{4} + 1.4775x)(1 - 1.4117x + 0.8458x^2), \tag{5.50}$$

where one real root is $-0.25/1.4775 = -0.1692$ and complex conjugate roots are $0.8345 \pm 0.6970i$. Notice that complex roots are located in the area described in Section 5.3.4. Depending on the discussion in Section 5.3.4, the second-order factor is transformed as follows:

$$\begin{aligned} 1 - 1.4117x + 0.8458x^2 &= 1 - 1.4117(1-t) + 0.8458(1-t)^2 \\ &= 0.4341(1 - 0.6455t + 1.9484t^2) \\ &= 0.4341(1 - 0.6455t(1-t)) + 0.5656t^2, \end{aligned} \tag{5.51}$$

which corresponds to equation (5.43). The stochastic unipolar implementation of the second-order factor is shown in Fig. 5.16. The circuit diagram of the overall polynomial $f(x)$ is shown in Fig. 5.22. The second-order factor is computed at node n_1 while node n_2 calculate first-order factor $0.25+1.4775x$. The select signal is given by $1.4775/(1.4775+0.25)) = 0.8553$. The final output is scaled by $1/(1.4775 + 0.25) = 0.5789$.

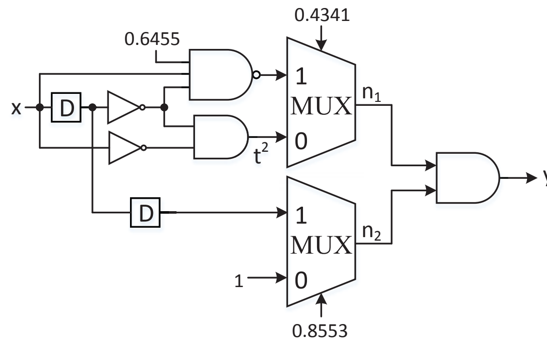


Figure 5.22: Stochastic unipolar implementation of $f(x)$ using factorization (Method-II).

The stochastic implementation of $f(x)$ using Bernstein polynomials [5] is based on the following transformed polynomial:

$$f(x) = \frac{2}{8}B_{0,3}(x) + \frac{5}{8}B_{1,3}(x) + \frac{3}{8}B_{2,3}(x) + \frac{6}{8}B_{3,3}(x), \tag{5.52}$$

where $B_{i,n}(x)$ is a Bernstein basis polynomial with the form:

$$B_{i,n}(x) = \binom{n}{i} x^i (1-x)^{n-i}. \tag{5.53}$$

Details of using Bernstein polynomials to compute polynomials in stochastic unipolar logic can be found in [5]. Fig. 5.23 shows stochastic logic implementation of $f(x)$ using Bernstein polynomials.

In our simulations, x is given by 0:0.01:1. 1000 Monte Carlo runs were performed for each data point. The length of stochastic bit streams is 1024. Since all constants are represented using stochastic sequences with 1024 bits, SNG blocks with 10-bit LFSR are used to generate bit streams for constants. This is consistent with generating stochastic bit streams for the input signal. Simulation results of the proposed stochastic polynomial computations (Method-I and Method-II) and previous design using Bernstein

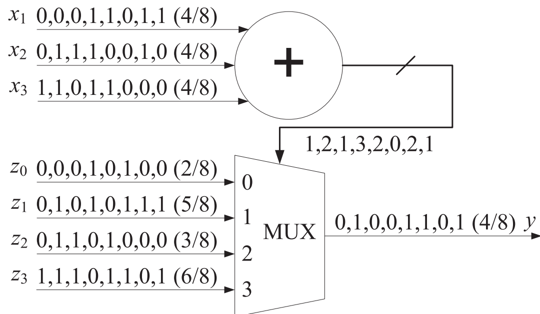


Figure 5.23: Stochastic logic implementing the Bernstein polynomial (52) at $x = 0.5$. Stochastic bit streams x_1, x_2 and x_3 encode the value $x = 0.5$. Stochastic bit streams z_0, z_1, z_2 and z_3 encode the corresponding Bernstein coefficients.

polynomials are shown in Fig. 5.24. Table 5.4 presents mean absolute error (MAE) of output of proposed implementations and the implementation using Bernstein polynomials to compute $f(x)$, where MAE is a quantity used to measure how close forecasts or predictions are to the eventual outcomes in median sense. From simulation results, we observe that the proposed Method-I has almost same accuracy as previous design using Bernstein polynomials while the proposed Method-II achieves better performance.

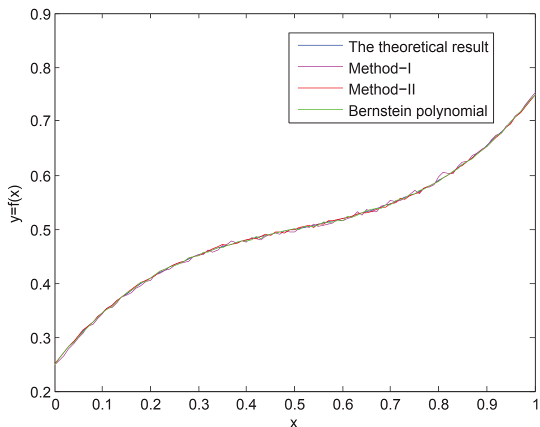


Figure 5.24: Simulation results of different implementations for $f(x)$.

Table 5.5 presents hardware complexity comparison for different stochastic implementations of $f(x)$. Architectures are implemented using 65nm libraries and synthesized using Synopsys Design Compiler. The operating conditions for each implementation are

Table 5.4: Output mean absolute error (MAE) of two proposed implementations and previous implementation using Bernstein polynomials for $f(x)$.

Implementation	Method-I	Method-II	Bernstein polynomials [5]
MAE	0.0118	0.0089	0.0106

specified by a supply voltage of 1.00 V and a temperature of 25 degree Celsius. The clock frequency is given by 100MHz. All SNGs which are used to generate the input signal and stochastic constants are included in our synthesis for these proposed designs and the implementation using Bernstein polynomials. It is shown that the first method requires more hardware complexity than the two other implementations since a unipolar subtractor is used in this design. Compared to the implementation using Bernstein polynomials, the computation of $f(x)$ based on factorization leads to less hardware complexity, power consumption and shorter critical path.

Table 5.5: Synthesis results for different stochastic implementations of $f(x)$.

Implementation	Method-I	Method-II	Bernstein polynomials [5]
Area (μm^2)	1272	807	1003
Critical Path (ns)	3.5485	2.7357	2.8885
Power (μW)	7.1406	4.9817	6.1410

5.4.2 Complex arithmetic functions based on stochastic polynomial computations

This section illustrates implementation of complex functions using polynomial approximations based on Taylor series expansion. It is known that a complex arithmetic function $g(x)$ can be described by Taylor series as follows:

$$g(x) = \sum_{n=0}^{\infty} \frac{g^{(n)}(a)}{n!}(x-a)^n. \tag{5.54}$$

Stochastic complex arithmetic functions are implemented based on Taylor series by using proposed approaches for polynomial computation. Consider the following function (from [53]):

$$g(x) = 4x^2 \log(x+0.1) + 0.53 \tag{5.55}$$

The 5th-order Taylor polynomial at $x = 0.5$ is given by:

$$\begin{aligned} g(x) &\approx 0.0192 - 0.3766(x-0.5) + 3.2345(x-0.5)^2 + 2.6543(x-0.5)^3 - 1.3117(x-0.5)^4 \\ &\quad + 1.0288(x-0.5)^5 \\ &= 0.5701 - 0.6430x - 4.0006x^2 + 7.8498x^3 - 3.8838x^4 + 1.0288x^5 \end{aligned} \quad (5.56)$$

A stochastic implementation of $g(x)$ can be obtained by using Method-I to compute the Taylor polynomial. The transformed polynomial is given by:

$$g(x) \approx (0.5701 + 7.8498x^3 + 1.0288x^5) - (0.6430x + 4.0006x^2 + 3.8838x^4). \quad (5.57)$$

Two sums are computed using multiplexers and the final result is generated from a stochastic subtractor.

The stochastic $g(x)$ can also be implemented using Method-II with factorization. The factorized $g(x)$ is described as follows:

$$g(x) \approx (1+3x)(1-0.5594x+0.1867x^2)(1-3.5736x+3.2172x^2). \quad (5.58)$$

Polynomial Roots are given by $r_1 = -0.3328$, $r_{2,3} = 1.4984 \pm 1.7640i$ and $r_{4,5} = 0.5554 \pm 0.0486i$. The first order factor is implemented using a multiplexer. Then the corresponding factor $(1-0.5594x+0.1867x^2)$ can be simply implemented using two levels of NAND gates as shown in Fig. 5.14. Complex roots r_4 and r_5 are located in the region discussed in Section 5.3.6. The corresponding factor $(1-3.5736x+3.2172x^2)$ is implemented as shown in Fig. 5.19. Note that a stochastic subtractor is required. The final result is generated by a 3-input AND gate performing multiplication.

Simulation results of the proposed implementations are shown in Fig. 5.25. The output MAE is given in Table 5.6 for proposed implementations and previous implementations using Bernstein polynomials. x is given by 0:0.01:1. 1000 Monte Carlo runs were performed for each data point. The length of stochastic bit streams is 1024. Implementations using Bernstein polynomials are considered as references and error results are obtained from [53]. The proposed implementations have same accuracy as the Bernstein-II implementation and significantly better performance compared to the Bernsterin-I implementation.

Table 5.7 presents hardware complexity comparison for different stochastic implementations of $g(x)$. We used 65nm library for synthesis while FreePDK45 library was

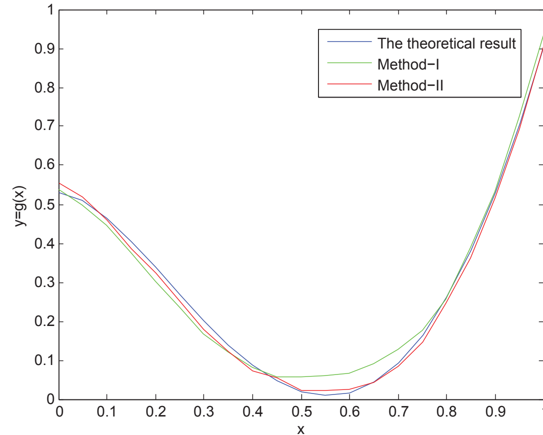


Figure 5.25: Simulation results of the proposed implementations for $g(x)$.

Table 5.6: Output mean absolute error (MAE) of different stochastic implementations of $g(x)$.

Implementation	Method-I	Method-II	Bernstein-I [5]	Bernstein-II [53]
MAE	0.0163	0.0101	0.0251	0.0103

used for Bernstein polynomial implementations in [53]. Therefore, results of Bernstein-I and Bernstein-II from [53] are scaled by K^2, where K is given by $65nm/45nm$. Two proposed implementations have same hardware complexity since a unipolar subtractor is required for both designs. Compared to implementations using Bernstein polynomials, the hardware complexity of proposed designs is reduced by more than 50%. This is explained by the fact that proposed designs only need a **5th-order** Taylor polynomial for stochastic implementation of $g(x)$ whereas Bernstein polynomials with **degree-12** and **degree-6** are required for Bernstein-I and Bernstein-II implementations, respectively.

Table 5.7: Hardware complexity for different stochastic implementations of $g(x)$.

Implementation	Method-I	Method-II	Bernstein-I [5]	Bernstein-II [53]
Area (μm^2)	1651	1627	2370	1655

Consider the stochastic implementation of the exponential function e^{-3x}. The 9th-order Maclaurin polynomial is factorized as follows:

$$\begin{aligned} e^{-3x} \approx & 1 - 3x + \frac{9x^2}{2} - \frac{9x^3}{2} + \frac{27x^4}{8} - \frac{81x^5}{40} + \frac{81x^6}{80} - \frac{243x^7}{560} + \frac{729x^8}{4480} - \frac{243x^9}{4480} \\ = & (1 - 0.9x)(1 + 0.474x + 0.264x^2)(1 - 0.118x + 0.465x^2) \\ & (1 - 0.904x + 0.643x^2)(1 - 1.552x + 0.766x^2). \end{aligned}$$

The stochastic implementation is shown in Fig. 5.26(a). In this figure, n_1 computes $(1 - 0.9x)$. The node n_2 computes $(1 + 0.474x + 0.264x^2)/(1 + 0.474 + 0.264)$. The node n_3 computes $(1 - 0.904x(1 - 0.7113x)) = (1 - 0.904x + 0.643x^2)$. The node n_4 computes $(1 - 1.552x + 0.766x^2)$, where $a_1 = 1.552$ and $b_1 = 0.766$. The node n_5 computes $(1 - 0.118x + 0.465x^2)/(1 - 0.118 + 0.465)$, where $a_2 = 0.118$ and $b_2 = 0.465$. The final result is scaled by $1/(1 + 0.474 + 0.264)(1 - 0.118 + 0.465) = 0.4272$.

Alternatively, the stochastic e^{-3x} can be implemented using the method proposed in Section 5.2.2, where all positive parts and negative parts are added and then subtracted using the unipolar subtractor based on division. The 9th-order Maclaurin polynomial is transformed as follows:

$$e^{-3x} \approx \underbrace{(1 + \frac{9x^2}{2} + \frac{27x^4}{8} + \frac{81x^6}{80} + \frac{729x^8}{4480})}_{Q(x)} - \underbrace{(3x + \frac{9x^3}{2} + \frac{81x^5}{40} + \frac{243x^7}{560} + \frac{243x^9}{4480})}_{R(x)}.$$

$Q(x)$ and $R(x)$ are implemented using multiplexers, where $Q(x)$ is scaled by $1/(1+9/2+27/8+81/80+729/4480) \approx 1/10$ and $R(x)$ is scaled by $1/(3+9/2+81/40+243/560+243/4480) \approx 1/10$. Notice that no scaling is involved for the unipolar subtraction. Therefore, the final computational result is scaled by $1/10$.

The simulation results of both implementations are shown in Fig. 5.26(b), where results are scaled up for the comparison with the objective function. The input is given by 0:0.01:1. The length of bit streams in the simulation is 1024 and 1000 runs were performed for each data point. It is shown that the factorization method obtains better performance than the method using subtraction.

5.4.3 Comparison with the STRAUSS implementation

In [51] and [52], a spectral transform approach (STRAUSS) was proposed to synthesize stochastic circuits. In this section, we conducted experiments in performance and

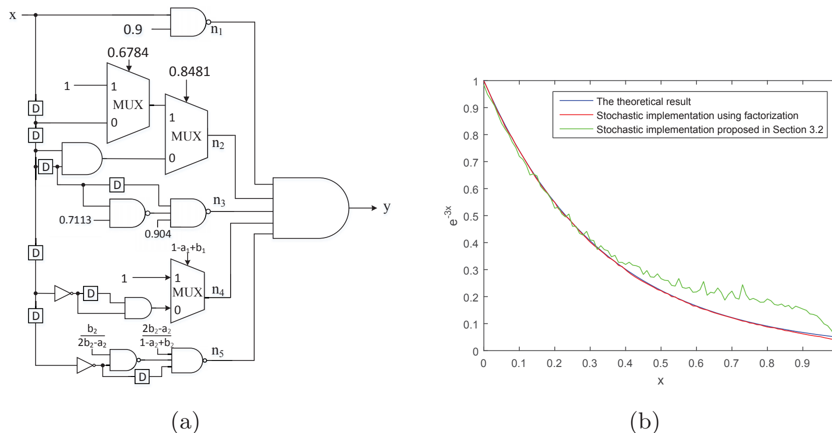


Figure 5.26: (a) The stochastic implementation using the factorization method and (b) simulation results for e^{-3x}.

hardware complexity test for our proposed method and the spectral transform approach.

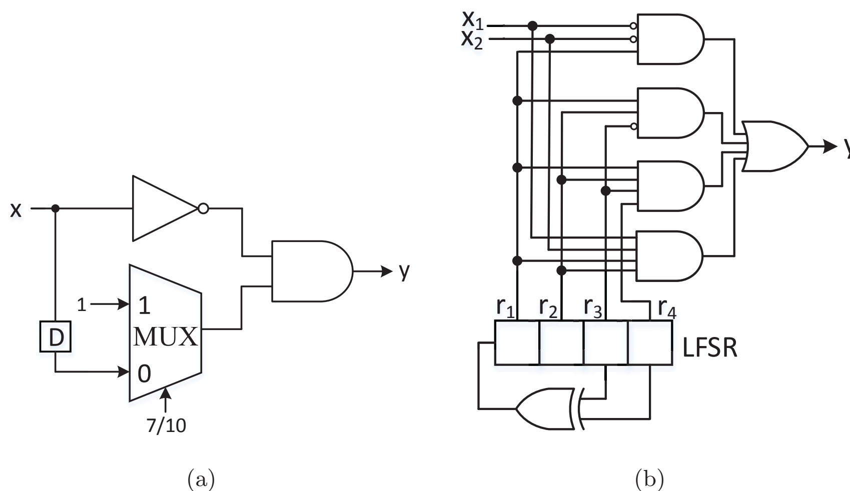
Consider the objective polynomial $p_1(x) = 0.4375 - 0.25x - 0.1875x^2$, where $0 \leq p_1(x) \leq 1$ given $x \in [0, 1]$. The stochastic implementation based on our proposed factorization method is shown in Fig. 5.27(a), where the polynomial $p_1(x)$ is factorized as follows:

$$p_1(x) = 0.4375 - 0.25x - 0.1875x^2 = 0.4375(1 - x)(1 + \frac{3}{7}x) \tag{5.59}$$

The stochastic implementation using the spectral transform is shown in Fig. 5.27(b). In this implementation, the non-linear function $p_1(x)$ is first converted into a multi-linear polynomial $\hat{p}_1(x) = 0.4375 - 0.125(x_1 + x_2) - 0.1875x_1x_2$. Since four iterations are required to in the step 4 of the algorithm proposed in [51], 4 auxiliary inputs r_1, r_2, r_3 and r_4 are needed for the implementation. Those four auxiliary inputs are generated by a 4-bit LFSR as shown in Fig. 5.27(b). More details of the implementation method can be found in [51] [52].

In our simulations, x is given by 0:0.01:1. 1000 Monte Carlo runs were performed for each data point. The length of stochastic bit streams is 1024. SNG blocks with 10-bit LFSR are used to generate bit streams for constants and the input. Simulation results of stochastic implementations for $p_1(x)$ using the proposed factorization method and the spectral transform approach are presented in Table 5.8.

In Table 5.8, the STRAUSS (4-bit LFSR) corresponds to the implementation with



(a) (b)

Figure 5.27: The stochastic implementations of $p_1(x)$ using (a) our proposed factorization method and (b) the spectral transform approach.

Table 5.8: Output mean absolute error (MAE) of different stochastic implementations of $p_1(x)$.

Implementation	Factorization	STRAUSS (4-bit LFSR)	STRAUSS (10-bit LFSR)
MAE	0.0025	0.0964	0.0104

4-bit LFSR as shown in Fig. 5.27(b). To make a fair comparison, the simulation result of the STRAUSS implementation with 10-bit LFSR is also presented in Table 5.8. Notice that in the 10-bit implementation, only four bits of the 10-bit LFSR are used. It is shown that the proposed method outperforms the STRUSS with 4-bit LFSR significantly and is slightly more accurate than the STRUSS with 10-bit LFSR. Moreover, notice that in the given example $p_1(x)$, coefficients are all in the format of a fraction $\frac{m}{16}$, where m is an integer number and $0 < m \leq 16$. If these coefficients were arbitrary values rather than in the specific format, the performance of the spectral transform approach would be further degraded. More rounding error would be introduced since the resolution of polynomial coefficients is $\frac{1}{2^m}$ given m auxiliary inputs ($m = 4$ in this example). The accuracy is improved by increasing the length of LFSR while the design complexity also increases.

Table 5.9 shows hardware complexity of different implementations of $p_1(x)$. The area of the Bernstein implementation is presented as a reference. Architectures are

implemented using 65nm libraries and synthesized using Synopsys Design Compiler. It is shown from the table that the STRAUSS implementation with 4-bit LFSR requires less hardware resources than our proposed factorization method while the STRAUSS 10-bit LFSR introduces more hardware complexity.

Table 5.9: Hardware complexity for different stochastic implementations of $p_1(x)$.

Implementation	Factorization	STRAUSS (4-bit)	STRAUSS (10-bit)	Bernstein
Area (μm^2)	402	356	464	786

5.5 Comparison of polynomial computations using unipolar and bipolar format

Given the same bit stream length, the precision of the unipolar format is twice that of the bipolar format, since the range of bipolar format ($[-1, 1]$) is twice that of the unipolar format ($[0, 1]$). Consider a simple example illustrated in Section 5.4.2 of the manuscript:

$$f_1(x) = \frac{1}{4} + \frac{9}{8}x - \frac{15}{8}x^2 + \frac{5}{4}x^3.$$

By using the factorization method, the unipolar implementation does not require any subtraction since no root of $f_1(x) = 0$ is located in the area shown in Fig. 5.13(f) in the manuscript. The unipolar implementation is shown in the Fig. 5.28(a) below. The bipolar implementation can be obtained using three multiplexers as shown in the Fig. 5.28(b) below. S_1, S_2 and S_3 are given by $S_1 = \frac{1/4}{1/4+9/8}$, $S_2 = S_2 = \frac{15/8}{15/8+5/4}$ and $S_3 = \frac{1/4+9/8}{1/4+9/8+15/8+5/4}$. We can observe that the unipolar implementation is less complex than the bipolar implementation since less multiplexers are required and the multiplication is implemented using AND gates in unipolar unlike XNOR gates in bipolar. All constant coefficients in two designs shown in Fig. 5.28 are generated using different SNGs. Notice that there are both three coefficients in these two implementations. The overheads in terms of hardware complexity for coefficient generation are the same. The performance of two implementations is described by simulation results shown in the Fig. 5.29 and Table 5.10. In our simulations, inputs were given as 0:0.01:1 and 1000 simulations were performed for each sample. Notice that for all simulations in this section, stochastic bit streams are generated using SNGs with a 10-bit

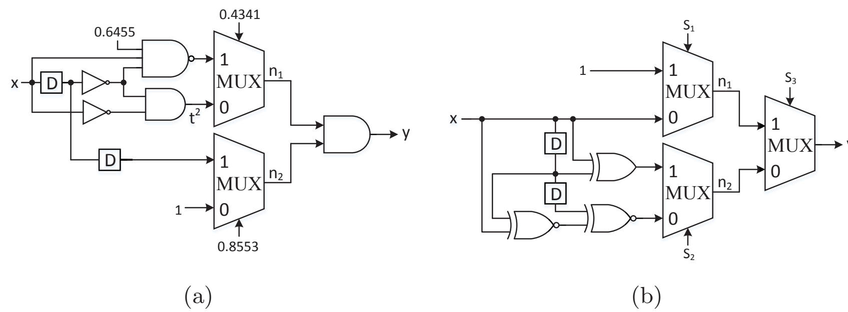


Figure 5.28: The stochastic implementations of $f_1(x)$ using (a) unipolar format, and using (b) bipolar format.

LFSR. It is shown the performance of the unipolar implementation is better than the bipolar implementation.

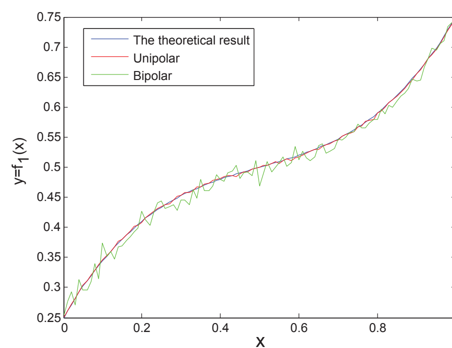


Figure 5.29: Simulation results of unipolar and bipolar implementations for $f_1(x)$.

Table 5.10: The signal-to-error ratio (SER) in dB for unipolar and bipolar implementations of $f_1(x)$.

Implementation	Unipolar	Bipolar
SER (dB)	49.50	33.39

Consider another example which is extremely difficult for both unipolar and bipolar implementations:

$$f_2(x) = 59.2x^4 - 118.7x^3 + 74.9x^2 - 15.4x + 1.$$

The unipolar implementation is shown in Fig. 5.30(a), where subtraction is required since after factorization there are complex roots located in the region shown in Fig. 5.13(f)

in the manuscript. The factorized polynomial is given by $f_2(x) = 59.2(x^2 - 0.3044x + 0.0233)(x^2 - 1.7x + 0.7236)$. Two second-order factors $(x^2 - 0.3044x + 0.0233)$ and $(x^2 - 1.7x + 0.7236)$ can be implemented using unipolar format without scaling. Therefore, the scaling of unipolar implementation is 1/59.2. The bipolar implementation is similar to the circuit diagram shown in Fig. 5.28(b) except one more multiplexer is needed. The scaling of bipolar implementation is given by 1/(59.2+118.7+74.9+15.4+1)=1/269.2. Notice that 1024-bit sequences are used in our simulation. The scaling of bipolar implementation implies that only 4 out of 1024 bits are effective and this leads to functional failure as shown in Fig. 5.30(b).

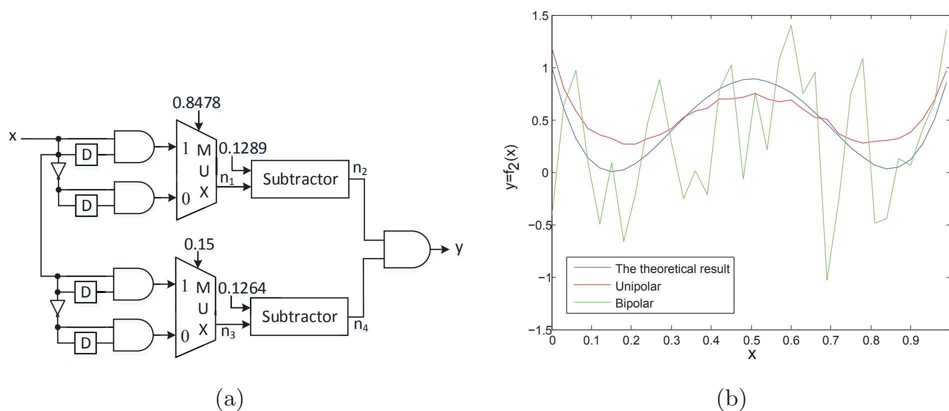


Figure 5.30: (a) The stochastic implementation of $f_2(x)$ based on the unipolar format, and (b) simulation results of unipolar and bipolar implementations for $f_2(x)$.

Therefore, we make several conclusions from these examples above for the unipolar and bipolar implementation of polynomials:

(1) As shown in example 1, by using the factorization method, the unipolar implementation is less complex than the bipolar implementation in some cases. This shows that although the bipolar implementation is straightforward, it does not necessarily lead to less complex design than the unipolar implementation.

(2) Only for some special cases, where complex roots are located in the region shown in Fig. 5.13(f) in the manuscript, the unipolar subtraction is required for the unipolar implementation. Although in this situation the bipolar implementation is less complex than the unipolar implementation, the bipolar implementation functionally fails due to

the scaling issue, as illustrated in the example 2. However, the reasonable approximation can be obtained by the unipolar implementation.
(3) For all situations, the unipolar implementation outperforms the bipolar implementation.

5.6 Conclusions

This chapter presents two approaches to compute polynomials in unipolar stochastic logic. The first implementation requires a stochastic subtractor. The second design is based on factorization. Moreover, stochastic implementations of complex arithmetic functions based on Taylor series and proposed polynomial computations are presented. Compared to previous designs using Bernstein polynomials, the proposed architectures achieve better accuracy and less hardware complexity. Various applications of the techniques presented in this paper to implement computation of polynomials have been presented in [54]. Another approach to computing a restricted class of polynomials using Horner's rule has been presented in [55].

Chapter 6

Machine Learning

In this chapter, we consider the stochastic implementation of machine learning classifiers. First, it is shown that the linear support vector machine (SVM) can be implemented using stochastic inner-product. The number of stochastic number generator (SNG) is minimized to reduce the hardware complexity. The artificial neural network (ANN) classifier is implemented using stochastic inner-product and hyperbolic tangent function based on finite-state machine (FSM) based architectures. Second, a *data-oriented linear transform* for input data is proposed to improve the accuracy of classification using stochastic logic. This approach leads to full utilization of the range of bipolar stochastic representation ([-1,1]). The performance of stochastic linear SVM can be improved by the proposed method while it is not always true for ANN classifier due to its *multiple layers* and *non-linearity*. Third, the proposed methods are validated using classifiers for seizure prediction from electroencephalogram (EEG) signals for two subjects from the Kaggle seizure prediction contest [33]. Comparisons in terms of accuracy and synthesis results are presented for conventional binary implementation and proposed stochastic designs.

We also discuss the stochastic implementation of RBF kernel in this chapter. First, an architecture with both input and output in bipolar format is proposed. The computation of RBF kernel is comprised of the squared Euclidean distance and the exponential function. In this proposed design, both components are implemented in bipolar format. The squared Euclidean distance is computed using multiple levels of multiplexers, where the number of SNGs is minimized. The bipolar exponential function is designed

based on the finite state machine (FSM) method. Second, we propose an implementation of RBF kernel with bipolar input and unipolar output. In this implementation, the squared Euclidean distance is computed with bipolar input and unipolar output. The exponential function is implemented in unipolar format, where factorization and Horner's rule are performed for the Maclaurin expansion of exponential function. The proposed designs are simulated using electroencephalogram (EEG) signals for one subject from the Kaggle seizure prediction contest [33]. Comparisons in terms of accuracy are presented for two proposed architectures.

6.1 Background of Machine Learning Classifiers

In machine learning, a support vector machine (SVM) is a discriminative classifier formally defined by a separating hyperplane [56]. Based on the different kernel functions, SVM family can be divided into two categories: linear classification and non-linear classification. For the linear SVM, the decision is made based on a linear kernel function:

$$K(\boldsymbol{x}) = \boldsymbol{w} \cdot \boldsymbol{x} + b, \tag{6.1}$$

where $\boldsymbol{x}$ represents the input vector, which describes extracted features from original data. Parameters $\boldsymbol{w}$ and b are trained from training data, where $\boldsymbol{w}$ stands for weights and b represents the bias. As shown in Fig. 6.1, two classes of data are separated by two parallel hyperplanes: $\boldsymbol{w} \cdot \boldsymbol{x} + b = 1$ and $\boldsymbol{w} \cdot \boldsymbol{x} + b = -1$. The training objective is to maximize distance ($\frac{b}{||\boldsymbol{w}||}$) between the data separated by hyperplanes. In classification, decisions are made based on the comparison between computational results of the kernel and a threshold value.

For the non-linear SVM, the kernel function is changed from inner-product to non-linear functions by applying kernel trick [57] to maximum-margin hyperplanes. Some common kernels include:

Polynomial: $K(\boldsymbol{x_i}, \boldsymbol{x_j}) = (\boldsymbol{x_i} \cdot \boldsymbol{x_j})^d$,

Gaussian radial basis function: $K(\boldsymbol{x_i}, \boldsymbol{x_j}) = e^{-\gamma||\boldsymbol{x_i} - \boldsymbol{x_j}||^2}$,

Hyperbolic tangent: $K(\boldsymbol{x_i}, \boldsymbol{x_j}) = \tanh(\beta \boldsymbol{x_i} \cdot \boldsymbol{x_j} + c)$.

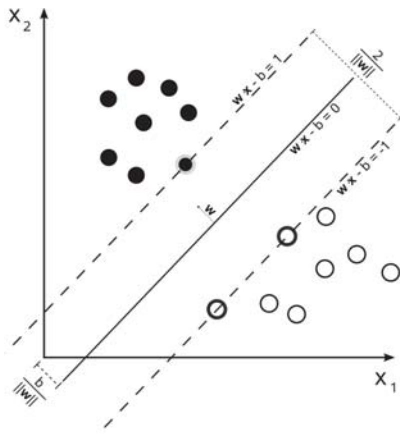


Figure 6.1: SVM classifier with linear kernel to maximize hyperplane margin.

In these equations above, K represents the kernel of SVM classifier. The input vector and support vectors are represented by $\boldsymbol{x_i}$ and $\boldsymbol{x_j}$, respectively. All other parameters including d, γ, β and c are obtained from the training process.

In machine learning, artificial neural networks (ANNs) are a family of models inspired by biological neural networks and are used to estimate or approximate functions that can depend on a large number of inputs and are generally unknown. Fig. 6.2

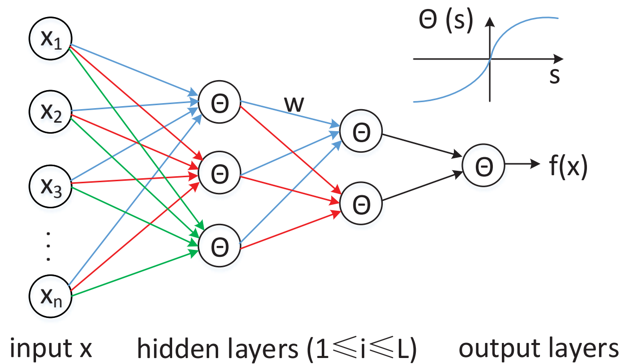


Figure 6.2: An artificial neural network (ANN) model.

shows a typical ANN classifier. The network is in layers, which are made up of a number of interconnected nodes (neurons) which contain an activation function (Θ). Patterns are presented to the network via the input layer, which communicates to one or more hidden layers where the actual processing is done via a system of weighted connections

($\boldsymbol{w}$). The hidden layers then link to an output layer where the answer is output. Assume that the overall function is described as $f(x)$ recursively, which is given by:

$$f(x) = \Theta(\sum_i w_i g_i(x)). \tag{6.2}$$

Notice that g_i represents results from the previous hidden layer and for the first hidden layer, g_0 simply corresponds to the input vector $\vec{x}$. The activation function Θ is a threshold function and is usually implemented using a non-linear hyperbolic tangent function ($\tanh x$) as shown in Fig. 6.3.

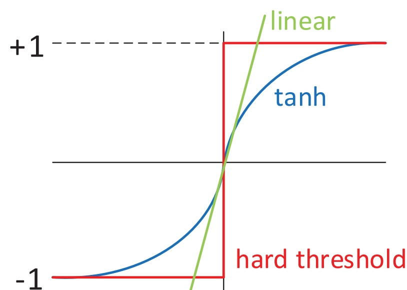


Figure 6.3: The thresholding using various functions.

6.2 Stochastic Implementation of linear SVM classifier

Machine learning classification includes two processes: training and testing. Our proposed stochastic implementation of classifiers is focused on the testing process. Therefore, we assume that all required parameters generated from the training process are already known. Given input data, our objective is to design stochastic classifiers which perform classification testing.

6.2.1 The Architecture of Stochastic Implementaion

Consider the linear SVM classifier. Given input testing data $\boldsymbol{x}$ and trained weight vector $\boldsymbol{w}$, the stochastic implementation of linear SVM classifier is shown in Fig. 6.4.

The computational kernel $K(\boldsymbol{x}) = \boldsymbol{w} \cdot \boldsymbol{x} + b$ is implemented using stochastic inner-product [30]. Assume that there are 4 features in given input data, namely the dimension

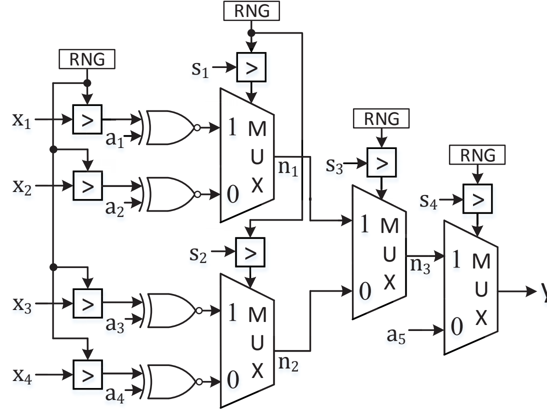


Figure 6.4: The stochastic implementation of linear SVM kernel.

of the input vector $\boldsymbol{x}$ is 4, where $\boldsymbol{x} = \{x_1, x_2, x_3, x_4\}$. Then the corresponding weight vector is described as $\boldsymbol{w} = \{w_1, w_2, w_3, w_4\}$. In this implementation, the multiplexer is used to perform addition and the XNOR gate is used to perform multiplication. Selecting signals of multiplexers are described as follows:

$$s_1 = \frac{|w_1|}{|w_1| + |w_2|} \tag{6.3}$$

$$s_2 = \frac{|w_3|}{|w_3| + |w_4|} \tag{6.4}$$

$$s_3 = \frac{|w_1| + |w_2|}{|w_1| + |w_2| + |w_3| + |w_4|} \tag{6.5}$$

$$s_5 = \frac{|w_1| + |w_2| + |w_3| + |w_4|}{|w_1| + |w_2| + |w_3| + |w_4| + |b|} \tag{6.6}$$

In Fig. 6.4, input features x_i and pre-calculated coefficients s_i are described in binary representation. Stochastic bit streams representing these signals are generated using SNGs, which are comprised of RNGs and binary comparators. Notice that all stochastic bit streams for input signal x_i's are generated using one single RNG. This leads to correlation among sequences representing input signals. However, it is known that unlike the correlation between the selecting signal and input signals, the correlation between input signals does not degrade the performance of stochastic additions [37]. Therefore, we only need to ensure different RNGs are used to generate bit streams for input signals and selecting signals. Moreover, note that the computational results of multiplexers in the same level are used as *inputs* for the next level of multiplexers. Then

these computational results can be correlated since the correlation does not affect the computation in the next level. Therefore, to reduce the hardware complexity, we use one RNG for the stochastic number generation for all selecting signals of multiplexers in the same level. We can conclude that the number of required RNG in our implementation is determined by the levels of multiplexers. Compared to the design using separated SNGs for multiplexers[30], the proposed design with sharing RNGs reduces hardware complexity significantly, since the SNG is the major source of area consumption (more than 90%) in stochastic logic.

In Fig. 6.4, a_i's ($1 \leq i \leq 4$) represent signs of w_i's. Given $w_i > 0$, $a_i = 1$. Otherwise, $a_i = 0$. The sign of b is represented by a_5. The computational results of internal nodes and the final output are given as follows:

$$n_1 = \frac{w_1x_1 + w_2x_2}{|w_1| + |w_2|} \tag{6.7}$$

$$n_2 = \frac{w_3x_3 + w_4x_4}{|w_3| + |w_4|} \tag{6.8}$$

$$n_3 = \frac{w_1x_1 + w_2x_2 + w_3x_3 + w_4x_4}{|w_1| + |w_2| + |w_3| + |w_4|} \tag{6.9}$$

$$y = \frac{w_1x_1 + w_2x_2 + w_3x_3 + w_4x_4 + b}{|w_1| + |w_2| + |w_3| + |w_4| + |b|} \tag{6.10}$$

Finally, to make a decision, the stochastic output y is converted to binary representation and is compared with a threshold value.

6.2.2 EEG Signal Classification using Stochastic Linear SVM

In this subsection, the stochastic linear SVM is tested based on the application of seizure diagnosis using EEG signals. A data-oriented optimization is proposed to improve the accuracy of the classification using stochastic linear SVM. We present comparisons in terms of accuracy and synthesis results between our proposed stochastic implementation and conventional binary implementation.

seizure prediction using EEG signal classification

An experiment is performed to predict seizures for epileptic patients [58] using s-tochastic logic. Given EEG signals of one patient, the objective is to predict occurrence of a seizure for the patient. Seizure prediction can be viewed as a binary classification

problem where one class consists of preictal signals corresponding to the signal right before an occurrence of the seizure, and the other class consists of normal EEG signals, also referred as interictal signals. The whole procedure of seizure prediction using machine learning method consists of three steps as shown in Fig. 6.5.

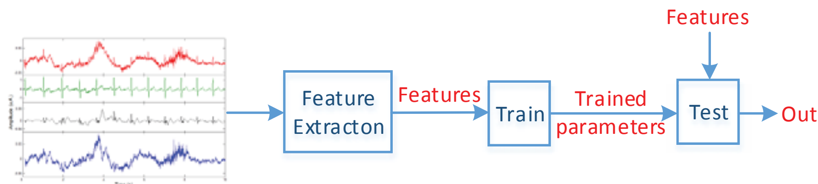


Figure 6.5: The whole procedure of seizure prediction using machine learning method.

First, features are extracted from original EEG signals. Second, classifier parameters are trained using features extracted from training data. Third, testing process is performed to predict whether a patient has a seizure or not based on trained classifiers and given testing data. The original EEG signals are taken from the dataset from the recent American Epilepsy Society Seizure Prediction Challenge database [33] [58]. In our test, two groups of data are considered. EEG signals of two patients were sampled from 16 electrodes at 400 Hz, and recorded voltages were referenced to the group average. Features used in our design are spectral power ratios of selected subbands of EEG signals captured from selected electrodes [59]. The details of feature extraction can be found in [60]. In our test, four features are extracted for the first patient while five features are extracted for the second patient. Assume that step 1 and step 2 of the seizure prediction procedure are done. It means that features of data and classifier parameters are known. We focus on the step 3 (testing) using the stochastic implementation of linear SVM.

For data from the first patient with 4 features, the testing data include 10244 samples and each sample is a vector with 4 elements ($\boldsymbol{x} = \{x_1, x_2, x_3, x_4\}$). Notice that the bipolar stochastic logic requires the range of $[-1, 1]$ for input signal. To this end, each sample needs to be scaled separately using l_1 scaling [27] as follows:

$$\boldsymbol{x} \Leftarrow \frac{\boldsymbol{x}}{x_{max}}, \tag{6.11}$$

where x_{max} represents the maximum magnitude of the input data. Accordingly, the bias also needs to be scaled as b/x_{max}. The linear SVM classifier is used to test whether there

is a seizure or not. The stochastic linear SVM is implemented using the circuit diagram shown in Fig. 6.4. The computational result is given in a scaled version: y/x_{max}. The threshold value for this application is zero. In our simulation, the length of stochastic bit stream is 1024 and the RNG is implemented using a 10-bit LFSR. The testing results are described using confusion matrices, which are presented in Table 6.1 and Table 6.2 for conventional binary implementation and the stochastic implementation, respectively. In a confusion matrix, TP, FN, FP and TN represent numbers of samples which are true positive, false negative, false positive and true negative, respectively. The metrics shown in the table to measure the performance of classification are defined as follows:

$$\text{True Positive Rate (TPR) or Sensitivity} = \frac{TP}{TP + FN}$$
$$\text{True Negative Rate (TNR) or Specificity} = \frac{TN}{FP + TN}$$
$$\text{Positive Predictive Value (PPV) or Precision} = \frac{TP}{TP + FP}$$
$$\text{Negative Predictive Value (NPV)} = \frac{TN}{TN + FN}$$
$$\text{Accuracy (ACC)} = \frac{TP + TN}{TP + FP + FN + TN}$$

Notice that the larger these values are, more accurate the classification is.

Table 6.1: The confusion matrix of classification for patient-1 (4-features) using *conventional binary* linear SVM (16-bit fixed point implementation).

		Predicted		
		Positive	**Negative**	
Actual Class	**Positive**	TP=4725	FN=488	TPR=0.9064
	Negative	FP=872	TN=4159	TNR=0.8266
	ACC=0.8672	PPV=0.8442	NPV=0.8950	

Table 6.2: The confusion matrix of classification for patient-1 (4-features) using *stochastic* linear SVM with l_1 scaling for input data.

		Predicted		
		Positive	**Negative**	
Actual Class	**Positive**	TP=4694	FN=519	TPR=0.9004
	Negative	FP=1272	TN=3759	TNR=0.7472
	ACC=0.8252	PPV=0.7868	NPV=0.8787	

In Table 6.1, the classification results, which are considered as ideal results in our test, are generated using a 2's complement SVM with 16-bit fixed point representation. Notice that the accuracy of the model is 0.8672 and can not classify the dataset perfectly. Comparing the accuracy results from Table 6.1 and Table 6.2, we can see that stochastic linear SVM has slight precision loss compared to the conventional binary implementation.

Tests were also performed for data with 5 features from the second patient. The stochastic implementation of the linear SVM is similar to the design for 4 features as shown in Fig. 6.4. The only difference is that one more multiplexer is required since the size of input vectors increases from 4 to 5. One RNG is used for each level of multiplexers. The l_1 scaling is also performed for both input data and the bias. There are 11381 samples in the dataset from patient-2. Confusion matrices of classifications for patient-2 are presented in Table 6.3 for conventional linear SVM and in Table 6.4 for stochastic linear SVM.

Table 6.3: The confusion matrix of classification for patient-2 (5-features) using *conventional binary* linear SVM (16-bit fixed point implementation).

		Predicted		
		Positive	**Negative**	
Actual Class	**Positive**	TP=5009	FN=381	TPR=0.9293
	Negative	FP=136	TN=5855	TNR=0.9773
	ACC=0.9546	PPV=0.9736	NPV= 0.9389	

Table 6.4: The confusion matrix of classification for patient-2 (5-features) using *stochastic* linear SVM with l_1 scaling for input data.

		Predicted		
		Positive	**Negative**	
Actual Class	**Positive**	TP=2541	FN=2850	TPR=0.4712
	Negative	FP=2756	TN=3234	TNR=0.5399
	ACC=0.5073	PPV=0.4796	NPV=0.5315	

As shown in Table 6.3, the classification accuracy for the ideal case is 0.9546. From the results in Table 6.3 and Table 6.4, it is observed that stochastic linear SVM leads to significant performance degradation in the classification using data from patient-2, compared to the conventional binary implementation.

Optimization with linear transform for input data

Consider the proposed EEG signal classification using stochastic linear SVM. Since the bipolar stochastic logic requires numbers in the range of $[-1, 1]$, the input data is scaled by the maximum magnitude. However, the scaling degrades the performance of stochastic classifier due to the loss of precision. For example, consider the input data of the first patient in our test. The maximum magnitude is 4.1836 and the range of input data is $[-0.6452, 4.1836]$. After scaling using the maximum magnitude, the range of input data is changed to $[-0.1542, 1]$, which means that 42% of the full range of bipolar format ($[-1, 1]$) is not occupied and the precision is lost by 42%.

The l_1 scaling of input data can be further optimized. First, we can scale each feature separately instead of scaling the whole input using one maximum magnitude. Second, the simple l_1 scaling can be replaced by a linear transformation. Consider the input data with 4 features from the first patient as a 10244-by-4 matrix $\boldsymbol{X}$. The number of rows (10244) corresponds to the number of input data samples while the number of features is represented by the column number. For the first feature X_{1j}, the linear transformation is performed fpr all samples as follows:

$$X_{1j} \Leftarrow \frac{2(X_{1j} - \min(\boldsymbol{X_1}))}{\max(\boldsymbol{X_1}) - \min(\boldsymbol{X_1})} - 1, \text{ for } 1 \leq j \leq 10244, \tag{6.12}$$

where $\min(\boldsymbol{X_1})$ and $\max(\boldsymbol{X_1})$ represent the minimum and maximum values of the first feature among all 10244 samples. After this linear transformation, the first column of the input matrix $\boldsymbol{X}$ is centered and the range of $[-1, 1]$ is fully occupied. Then similar linear transformations are performed for three other features. Recall that a data sample is tested by computing $y = X_{1j}w_1 + X_{2j}w_2 + X_{3j}w_3 + X_{4j}w_4 + b$. Therefore, elements of the weight vector and the bias need to be transformed as follows to guarantee a correct functionality:

$$w_i' \Leftarrow \frac{\max(\boldsymbol{X_i}) - \min(\boldsymbol{X_i})}{2} \cdot w_i, \text{ for } 1 \leq i \leq 4 \tag{6.13}$$

$$b' \Leftarrow \sum_{i=1}^{4} \frac{\max(\boldsymbol{X_i}) + \min(\boldsymbol{X_i})}{2} \cdot w_i + b \tag{6.14}$$

The optimized stochastic linear SVM testing is implemented based on the circuit diagram shown in Fig. 6.4 and the transformed $\boldsymbol{X}$, $\boldsymbol{w'}$ and b'. In our simulation, this

data-oriented optimization is applied for two datasets, which are used for the previous testing without optimization. The confusion matrix of the classification for patient-1 is presented in Table 6.5 while Table 6.6 shows the confusion matrix of the classification for patient-2. For patient-1, compare accuracy results in Table 6.5 with that in Table 6.1 and Table 6.2. For patient-2, compare accuracy results in Table 6.6 with that in Table 6.3 and Table 6.4. It is shown that for patient-1 the accuracy of EEG classification using *stochastic* linear SVM is improved by 3.88% by using the proposed data-oriented optimization, and for patient-2 the accuracy is improved by 85.49% by using the proposed data-oriented optimization. For both datasets, the performance of stochastic classification using linear transform for input data is close to the ideal result from conventional design.

Table 6.5: The confusion matrix of classification for patient-1 (4-features) using *stochastic* linear SVM with *linear transform* for input data.

		Predicted		
		Positive	**Negative**	
Actual Class	**Positive**	TP=4570	FN=643	TPR=0.8766
	Negative	FP=820	TN=4211	TNR=0.8370
	ACC=0.8572	PPV=0.8479	NPV=0.8675	

Table 6.6: The confusion matrix of classification for patient-2 (5-features) using *stochastic* linear SVM with *linear transform* for input data.

		Predicted		
		Positive	**Negative**	
Actual Class	**Positive**	TP=4902	FN=489	TPR=0.9093
	Negative	FP=181	TN=5809	TNR=0.9696
	ACC=0.9410	PPV=0.9643	NPV=0.9223	

Hardware Complexity

The architectures are implemented using 32nm libraries and synthesized using Synopsys Design Compiler. The length of the stochastic sequence is 1024 and all SNGs including 10-bit LFSRs as RNGs are considered in our synthesis. The conventional binary linear SVM is implemented using 16-bit fixed point representation. The bit-widths of both implementations are consistent with tests for accuracy in Section 6.2.2. For the

conventional binary design, the multiplication is implemented using the shift-and-add multiplier to minimize the hardware complexity. The operating conditions for each implementation are specified by a supply voltage of 1.05 V and a temperature of 25 degree Celsius. The clock frequency is given by 0.5 GHz.

Table 6.7 shows synthesis results of various implementations of linear SVM for dataset-1 with 4 features and dataset-2 with 5 features. Compared to conventional binary implementation, hardware complexity, power consumption and critical path of the proposed stochastic implementations are reduced significantly.

Table 6.7: Synthesis results of conventional binary and stochastic linear SVM classifiers for dataset-1 with 4 features and dataset-2 with 5 features.

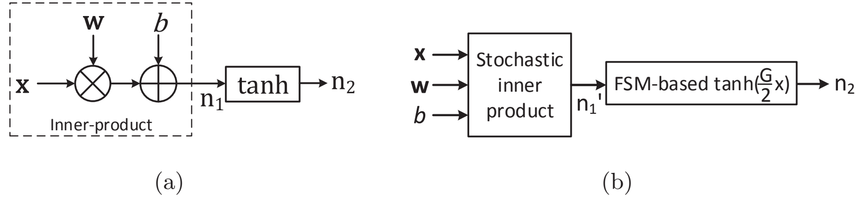
Dataset-1 with 4 features			
Implementations	Area	Power	Critial Path
Conventional Binary	8287 μm^2	203.0 μW	1.96 ns
Proposed Stochastic	1831 μm^2	52.87 μW	0.93 ns
Dataset-2 with 5 features			
Implementations	Area	Power	Critial Path
Conventional Binary	10271 μm^2	251.8 μW	1.99 ns
Proposed Stochastic	2230 μm^2	63.96 μW	0.93 ns

6.3 Stochastic Implementation of ANN classifier

6.3.1 The Architecture of Stochastic Implementation

Consider the general ANN module shown in Fig. 6.2. Assume that computational kernels of a neuron include an inner-product and a tangent hyperbolic function as shown in Fig. 6.6(a).

The inner-product $\boldsymbol{w} \cdot \boldsymbol{x} + b$ is computed at node n_1, where b is the bias. The $\tanh n_1$ is given at the node n_2. Assume that there are 4 features for given input data. The stochastic implementation of a neuron is shown in Fig. 6.6(b). The stochastic inner-product is the same as the circuit shown in Fig. 6.4. Due to the scaled addition, the



(a) (b)

Figure 6.6: (a) Computation kernels in a neuron implemented in conventional binary implementation and (b) in stochastic logic.

computational result of node n_1' is a scaled version of n_1:

$$n_1' = \frac{w_1x_1 + w_2x_2 + w_3x_3 + w_4x_4 + b}{|w_1| + |w_2| + |w_3| + |w_4| + |b|}. \tag{6.15}$$

In the stochastic implementation, the tangent hyperbolic function is implemented using the finite-state machine (FSM) method [1]. The state transition diagram of the FSM implementing the stochastic $\tanh(\frac{G}{2}x)$ function is shown in Fig. 6.7, where G is the number of states in the FSM. Such an FSM can be implemented using an

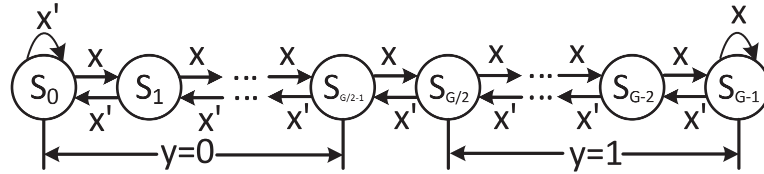


Figure 6.7: The state transition diagram of the FSM implementing the stochastic $\tanh(\frac{G}{2}x)$.

up and down saturate counter. The detail of the implementation and analysis can be found in [31].

Notice that the computational result generated at node n_1' is a scaled version of the original value at node n_1. However, the final output of a neuron implemented in stochastic logic is the same as the output of conventional implementation. This is because $\tanh(\frac{G}{2}x)$ is implemented in stochastic logic instead of the original $\tanh(x)$. The number of states in FSM is determined by the scaling at node n_1' as follows:

$$G = \text{round}(2(|w_1| + |w_2| + |w_3| + |w_4| + |b|)), \tag{6.16}$$

which is rounded to the nearest integer. In this case, considering the whole ANN, no

scaling from previous layer affects the computation of the next layer, since the output generated at node n_2 in stochastic computation is the same as the ideal case.

6.3.2 EEG Signal Classification using Stochastic ANN Classifier

In this subsection, the stochastic ANN classifier is tested based on the application of seizure diagnosis using EEG signals. The linear transform of input data is tested for classifications using ANN. We present comparisons in terms of accuracy and synthesis results between our proposed stochastic implementation and conventional binary implementation.

ANN for EEG signal classification

Same as the classification using linear SVM, two groups of data from two patients are considered. There are 4 features in the data from the first patient and 5 features in the data from the second patient. The ANN classifier for these two datasets is shown in Fig. 6.8.

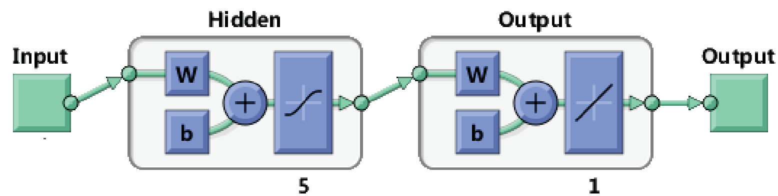


Figure 6.8: The ANN classifier for EEG signal classification.

For both datasets, there is one hidden layer which contains five neurons with tanh threshold function. The stochastic implementation of the neurons are shown in Fig. 6.6(b). The neuron in the output layer includes a linear threshold function $y = x$. Then the stochastic implementation of this neuron is just an inner-product. The only difference of ANN classifiers for two datasets is the size of the input vector, which leads to different numbers of multiplexers in inner-product modules.

In our test, l_1 scaling for input data is performed for the proposed stochastic implementation as described by equation (6.11). The threshold for the final classification is zero. The length of stochastic bit stream is $2^{13} = 8192$ and the RNG is implemented using a 13-bit LFSR. The length of sequences is increased from 1024-bit since more

precision is required for multiple layers and neurons in ANN compared to the linear SVM. For data from patient-1 with 10244 samples, simulation results are presented in confusion matrices as shown in Table 6.8 and Table 6.9 for conventional binary implementation and the stochastic implementation, respectively.

Table 6.8: The confusion matrix of classification for patient-1 (4-features) using *conventional binary* ANN with 16-bit fixed point implementation.

		Predicted		
		Positive	**Negative**	
Actual Class	**Positive**	TP=4652	FN=561	TPR=0.8923
	Negative	FP=1219	TN=3812	TNR=0.7577
	ACC=0.8262	PPV=0.7924	NPV=0.8717	

Table 6.9: The confusion matrix of classification for patient-1 (4-features) using *stochastic* ANN with l_1 scaling for input data.

		Predicted		
		Positive	**Negative**	
Actual Class	**Positive**	TP=4765	FN=448	TPR=0.9141
	Negative	FP=834	TN=4197	TNR=0.8343
	ACC=0.8749	PPV=0.8510	NPV=0.9036	

Table 6.10: The confusion matrix of classification for patient-1 (4-features) using *stochastic* ANN with *linear transform* for input data.

		Predicted		
		Positive	**Negative**	
Actual Class	**Positive**	TP=3307	FN=1906	TPR=0.6343
	Negative	FP=667	TN=4364	TNR=0.8674
	ACC=0.7487	PPV=0.8322	NPV=0.6960	

In Table 6.8, the classification results, which are considered as ideal results in our test, are generated using a 2's complement ANN with 16-bit fixed point representation. Notice that the accuracy for the ideal case is 0.8262, where the error is from the model. Comparing the accuracy results in Table 6.9 with that in Table 6.8, we can see that the ACC of the proposed stochastic ANN is 0.8749, which is 5.89% more than the ACC of conventional binary design. It can be explained by the fact that the error in the model is partially canceled out by random fluctuation from stochastic computing.

For patient-2, the dataset includes 11381 samples. Confusion matrices of classifications for patient-2 are presented in Table 6.11 for conventional binary ANN and in Table 6.12 for stochastic ANN classifier.

Table 6.11: The confusion matrix of classification for patient-2 (5-features) using *conventional binary* ANN with 16-bit fixed point implementation.

		Predicted		
		Positive	**Negative**	
Actual Class	**Positive**	TP=4484	FN=156	TPR=0.8319
	Negative	FP=907	TN=5834	TNR=0.9738
	ACC=0.9066	PPV=0.8317	NPV=0.9739	

Table 6.12: The confusion matrix of classification for patient-2 (5-features) using *s-tochastic* ANN with l_1 scaling for input data.

		Predicted		
		Positive	**Negative**	
Actual Class	**Positive**	TP=4499	FN=891	TPR=0.8346
	Negative	FP=1557	TN=4433	TNR=0.7399
	ACC=0.7848	PPV=0.7429	NPV=0.8421	

As shown in Table 6.11, the classification accuracy for the ideal case is 0.9066. From the results in Table 6.11 and Table 6.12, it is shown that stochastic ANN leads to performance degradation in the classification using data from patient-2, compared to the conventional binary implementation.

Table 6.13: The confusion matrix of classification for patient-2 (5-features) using *s-tochastic* ANN with *linear transform* for input data.

		Predicted		
		Positive	**Negative**	
Actual Class	**Positive**	TP=4444	FN=946	TPR=0.8245
	Negative	FP=780	TN=5210	TNR=0.8696
	ACC=0.8483	PPV=0.8507	NPV=0.8463	

Optimization with linear transform for input data

We performed the proposed linear transform for input data for stochastic implementation of ANN classifier. The method is the same as that described in Section 6.2.2 for

the classification using linear SVM. Consider the computation of each neuron shown in Fig. 6.6(b). $\boldsymbol{X}$, $\boldsymbol{w}$ and b are transformed as described in equations (6.12), (6.13) and (6.14). The number of states in the FSM-based tanh function (G) is determined by the transformed $\boldsymbol{w'}$ and b' as given in equation (6.16). The same two datasets used in previous tests are considered in our simulation. The difference of implementations for two datasets is the size of input and parameter vectors.

The confusion matrix of the classification for patient-1 is presented in Table 6.10 while Table 6.13 shows The confusion matrix of the classification for patient-2. For patient-1, compare accuracy results in Table 6.10 with that in Table 6.9. It is shown that the accuracy of EEG classification using stochastic ANN classifier is reduced by 14.42% by using the proposed linear transformation for input data. For patient-2, compare accuracy results in Table 6.13 with that in Table 6.12. It is shown that the accuracy of EEG classification using stochastic ANN is improved by 7.49% by using the proposed data-oriented optimization. Therefore, we concluded that, unlike the linear SVM, the improvement of performance for stochastic ANN classifier is not guaranteed by using the linear transform for input data. This is because the linear transform may lead to large weight coefficients (w_i) and bias (b). The ANN with multiple layers and neurons may suffer from precision loss due to large scaling. It is also explained by the non-linearity introduced by hyperbolic tangent function in ANN classifier.

Hardware complexity

The architectures are implemented using 32nm libraries and synthesized using Synopsys Design Compiler. The length of the stochastic sequence is 8192 and all SNGs including 13-bit LFSRs as RNGs are considered in out synthesis. The conventional binary ANN is implemented using 16-bit fixed point representation. The bit-widths of both implementations are consistent with tests for accuracy in Section 6.3.2. For the conventional binary design, the hyperbolic tangent function is implemented using Taylor series expansion. The multiplication is implemented using a shift-and-add multiplier to minimize the hardware complexity. The operating conditions for each implementation are specified by a supply voltage of 1.05 V and a temperature of 25 degree Celsius. The clock frequency is given by 0.5 GHz.

Table 6.14 shows synthesis results of various implementations of ANN classifier for

dataset-1 with 4 features and dataset-2 with 5 features. Compared to conventional binary implementation, hardware complexity, power consumption and critical path of the proposed stochastic implementation are reduced significantly.

Table 6.14: Synthesis results of conventional binary and stochastic ANN classifiers for dataset-1 with 4 features and dataset-2 with 5 features.

Dataset-1 with 4 features			
Implementations	Area	Power	Critial Path
Conventional Binary	78598 μm^2	1698 μW	1.98 ns
Proposed Stochastic	6651 μm^2	203.9 μW	1.05 ns
Dataset-2 with 5 features			
Implementations	Area	Power	Critial Path
Conventional Binary	89733 μm^2	2120 μW	1.98 ns
Proposed Stochastic	7479 μm^2	232.6 μW	1.74 ns

6.4 Computing RBF Kernel for SVM Classification using Stochastic Logic

Based on different kernel functions, SVM classification can be divided into two categories: linear and non-linear. For the linear SVM, the decision is made based on a linear kernel function:

$$K(\boldsymbol{x}) = \boldsymbol{w}^{\boldsymbol{T}}\boldsymbol{x} + b, \tag{6.17}$$

where $\boldsymbol{x}$ represents the input vector, which describes extracted features from original data. Parameters $\boldsymbol{w}$ and b are obtained from training data, where $\boldsymbol{w}$ stands for weights and b represents the bias. In classification, decisions are made based on the comparison between computational results of the kernel and a threshold value. For the non-linear SVM, the kernel function is changed from inner-product to non-linear functions by applying kernel trick [57] to maximum-margin hyperplanes. The Gaussian radial basis function (RBF) kernel is given by:

$$K(\boldsymbol{x}, \boldsymbol{x'}) = e^{-\gamma||\boldsymbol{x}-\boldsymbol{x'}||^2}. \tag{6.18}$$

where K represents the kernel of SVM classifier. The input vector and support vectors are represented by $\boldsymbol{x}$ and $\boldsymbol{x'}$, respectively. In this paper, the parameter γ is given by the reciprocal of the number of features.

6.4.1 Stochastic Implementation of RBF Kernel using Finite State Machine

Machine learning classification includes two processes: training and testing. Our proposed stochastic implementations are focused on kernel computation during the testing process. Therefore, we assume that all support vectors generated from the training process are already known.

Consider the RBF kernel of SVM classifier which is described by equation (6.18). The stochastic implementation consists of two parts: the squared Euclidean distance between input vectors and support vectors ($||\boldsymbol{x} - \boldsymbol{x'}||^2$) and the exponential function. Assume that the number of features for a dataset is 4. The objective kernel function is given by

$$e^{-\frac{||\boldsymbol{x}-\boldsymbol{x'}||^2}{4}}. \tag{6.19}$$

The $||\boldsymbol{x} - \boldsymbol{x'}||^2$ can be implemented using the architecture with multiple levels of multiplexers as shown in Fig. 6.9.

The kernel computation is based on stochastic inner-product proposed in [30]. Since there are 4 features in given input data, the dimension of the input vector $\boldsymbol{x}$ is 4, where $\boldsymbol{x} = \{x_1, x_2, x_3, x_4\}$. The support vector is described as $\boldsymbol{x'} = \{x'_1, x'_2, x'_3, x'_4\}$. In this implementation, the multiplexer is used to perform addition and the XNOR gate is used to perform multiplication. Notice that the probability of ones of select signals for all multiplexers are 0.5. The computational result from the first level multiplexer is $\frac{x_i - x'_i}{2}$. At node n_i, $\frac{(x_i - x'_i)^2}{4}$ is calculated. Then the final output is given by:

$$\begin{aligned} w &= \frac{(x_1 - x'_1)^2 + (x_2 - x'_2)^2 + (x_3 - x'_3)^2 + (x_4 - x'_4)^2}{16} \\ &= \frac{||\boldsymbol{x} - \boldsymbol{x'}||^2}{16} \end{aligned} \tag{6.20}$$

which is a scaled version of $||\boldsymbol{x} - \boldsymbol{x'}||^2$. Notice that one-bit delay elements are required to decorrelate stochastic sequences [3].

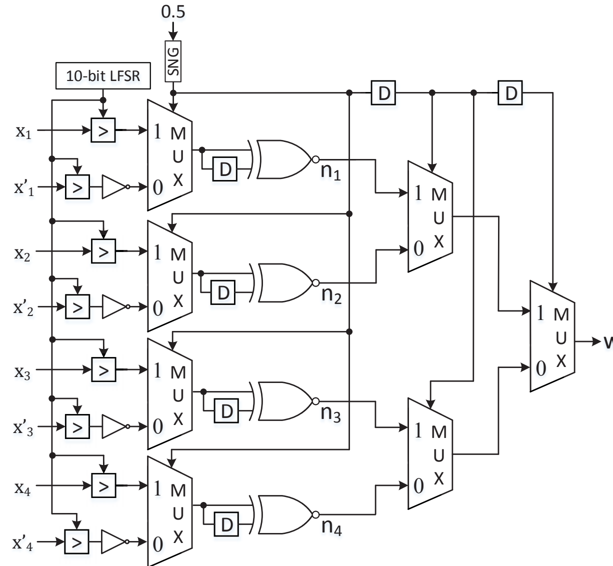


Figure 6.9: The implementation of $\frac{||\boldsymbol{x}-\boldsymbol{x'}||^2}{16}$ in stochastic logic.

In Fig. 6.9, input signals x_i's and x_i''s are described in binary representation. Stochastic bit streams representing these signals are generated using SNGs, which are comprised of RNGs and binary comparators. Notice that all stochastic bit streams for input signals are generated using one single RNG. This increases correlation among sequences representing input signals. However, it is known that unlike the correlation between the select signal and input signals, the correlation between input signals does not degrade the performance of stochastic additions [37]. Therefore, we only need to ensure different RNGs are used to generate bit streams for input signals and select signals.

Moreover, to reduce the hardware complexity, one SNG is used to generate select signals for all multiplexers since the probability of ones is fixed at 0.5. Note that the computational results of multiplexers in the same level are used as *inputs* for the next level of multiplexers. Then these computational results can be correlated since the correlation does not affect the computation in the next level. Therefore, all multiplexers in the same level share the same stochastic bit stream as select signals. One-bit delay elements are used to decorrelate select signals for multiplexers at different levels. Compared to the design using separated SNGs for multiplexers[30], the proposed design with RNG/SNG reusing reduces hardware complexity significantly, since the SNG is

the major source of area consumption (more than 90%) in stochastic logic.

The second part of the stochastic implementation of the objective kernel function (6.19) is the exponential function. Notice that in equation (6.20), $||\boldsymbol{x} - \boldsymbol{x'}||^2/16$ is computed by the stochastic implementation. Therefore, to compute $||\boldsymbol{x} - \boldsymbol{x'}||^2/4$ as required in equation (6.19), we consider the implementation of e^{-4x} using stochastic logic. The stochastic exponential function is implemented using the finite-state machine (FSM) method [8]. The state transition diagram of the FSM implementing e^{-2Gx} in stochastic logic is shown in Fig. 6.10, where the parameter G determines the number of states with different outputs. The output from state S_0 to state S_{N-G-1} is one while the output from S_{N-G} to S_{N-1} is 0, where N is the total number of states in the FSM. Notice that $G \ll N$ needs to be satisfied for an accurate computation. In our design, G is given by 2 since the objective function is e^{-4x}. The total number of states is given by $N = 32$. Such an FSM can be implemented using

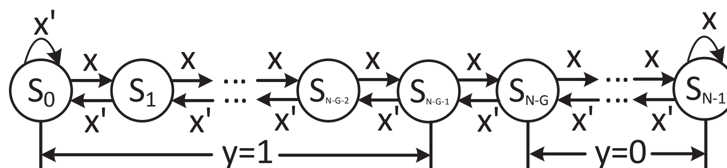


Figure 6.10: The state transition diagram of the FSM implementing e^{-2Gx} in stochastic logic.

an up and down saturate counter. The input sequence x determines the FSM state transition. The output sequence y is determined by the output of the FSM. The detail of the implementation and analysis can be found in [31].

The whole architecture of stochastic RBF kernel is implemented by cascading the circuit shown in Fig. 6.9 and the FSM shown in Fig. 6.10, where the output of multi-level multiplexers is given as the input of the FSM implementation for the exponential function. The final output of the system is given by:

$$y = e^{-4w} = e^{-4\frac{||\boldsymbol{x}-\boldsymbol{x'}||^2}{16}} = e^{-\frac{||\boldsymbol{x}-\boldsymbol{x'}||^2}{4}}. \tag{6.21}$$

6.4.2 Stochastic RBF Kernel based on format conversion

In the stochastic implementation of RBF kernel in Section 6.4.1, all stochastic bit streams are interpreted in bipolar format. However, notice that the range of $e^{-\frac{||\boldsymbol{x}-\boldsymbol{x'}||^2}{4}}$ is $[0, 1]$. It is possible to represent the kernel output in unipolar format, which is more accurate than the bipolar format. Given the same word length, the precision of the unipolar format is twice that of the bipolar format, since the range of bipolar format ([-1,1]) is twice that of the unipolar format ([0,1]). The design of RBF kernel in stochastic logic with unipolar output and implicit format conversion is proposed in this section.

The RBF kernel of SVM classifier in equation (6.19) can be rewritten as follows:

$$\begin{aligned} e^{-\frac{||\boldsymbol{x}-\boldsymbol{x'}||^2}{4}} &= e^{-\frac{(x_1-x_1')^2+(x_2-x_2')^2+(x_3-x_3')^2+(x_4-x_4')^2}{4}} \\ &= e^{-\frac{(x_1-x_1')^2}{4}} \cdot e^{-\frac{(x_2-x_2')^2}{4}} \cdot e^{-\frac{(x_3-x_3')^2}{4}} \cdot e^{-\frac{(x_4-x_4')^2}{4}} \end{aligned} \tag{6.22}$$

Consider the stochastic implementation of a scaled version of $(x_i - x_i')^2$ shown in Fig. 6.11.

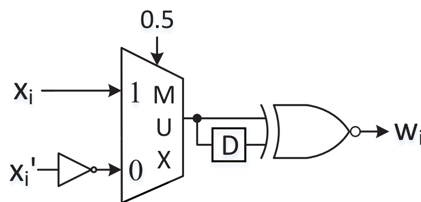


Figure 6.11: The stochastic implementation of $\frac{(x_i-x_i')^2}{4}$.

Since the input is scaled to the range of $[-1, 1]$, the bipolar format is required to represent input signals. The output bit stream can be explained using unipolar format. Assume that x_i, x_i' and w_i denote values represented by bit streams while P_{w_i} represents the probability of ones for the output bit stream. If w_i were defined in *bipolar* format, the computation result of the circuit shown in Fig. 6.11 would be given by:

$$w_i = \frac{(x_i - x_i')^2}{4} \tag{6.23}$$

$$\Rightarrow 2P_{w_i} - 1 = \frac{(x_i - x_i')^2}{4} \tag{6.24}$$

$$\Rightarrow P_{w_i} = \frac{(x_i - x_i')^2}{8} + \frac{1}{2} \tag{6.25}$$

However, in this case, note that the output value is implicitly represented in the *unipolar* format. Therefore, the output value is described as follows:

$$w_i = P_{w_i} = \frac{(x_i - x_i')^2}{8} + \frac{1}{2}, \tag{6.26}$$

which is a scaled and shifted version of $(x_i - x_i')^2$.

Consider the following computation of exponential function:

$$e^{-2w_i} = e^{-2(\frac{(x_i-x_i')^2}{8}+\frac{1}{2})} = \frac{1}{e} \cdot e^{-\frac{(x_i-x_i')^2}{4}}, \tag{6.27}$$

which is a scaled version of the target function $e^{-\frac{(x_i-x_i')^2}{4}}$. Notice that the 1/2 shifting from the equation (6.26) leads to the scaling. Since in equation (6.27), w_i is represented in unipolar format, the correct implementation of the scaled target function requires the stochastic implementation of e^{-2x} with unipolar input and output.

The unipolar e^{-2x} can be implemented based on the Maclaurin expansion [61]. The expression for stochastic implementation is derived by factorizing and applying Horner's rule for the Maclaurin polynomial. Factorization is used to satisfy the constraint that the magnitude of coefficients in stochastic implementation is less than one. The Horner's rule is used to construct the format $1 - ax$, which can be simply implemented using a NAND gate in unipolar format. Notice that $a \in [0, 1]$. The 7th-order Maclaurin polynomial is transformed as follows:

$$\begin{aligned} e^{-2x} \approx\ & 1 - 2x + \frac{4x^2}{2!} - \frac{8x^3}{3!} + \frac{16x^4}{4!} - \frac{32x^5}{5!} + \frac{64x^6}{6!} \\ & - \frac{128x^7}{7!} && (6.28) \\ =\ & (1 - 0.7249x)(1 - 0.4143x + 0.3612x^2) \\ & (1 - 1.1445x + 0.4810x^2)(1 + 0.2837x + 0.205x^2) && (6.29) \\ =\ & (1 - 0.7249x)(1 - 0.4143x + 0.3612x^2) \\ & (1 - 0.8608x + 0.3612x^2 - 0.0982x^3 + 0.0986x^4) && (6.30) \\ =\ & (1 - 0.7249x)(1 - 0.4143x(1 - 0.8718x)) \\ & (1 - 0.8608x(1 - 0.4196x(1 - 0.2719x(1 - x)))). && (6.31) \end{aligned}$$

Since not all coefficients in (12) are less than one, factorization is used to derive (13). The 2nd-order factors are generated due to complex roots of x. Notice that in polynomial

(13), the coefficient 1.1445 is greater than one and $1 + 0.2837x + 0.205x^2$ can not be implemented using NAND gates in stochastic unipolar representation. Therefore, we expand the last two 2nd-order factors to derive polynomial (14). The polynomial (15) is obtained by applying Horner's rule. Note that the coefficient 0.0986 in (14) was approximated to 0.0982 to avoid a coefficient greater than one in (15).

The stochastic implementation of e^{-2x} using the 7th-order Maclaurin polynomial (15) is shown in Fig. 6.12. All coefficients and the input are represented in stochastic unipolar format.

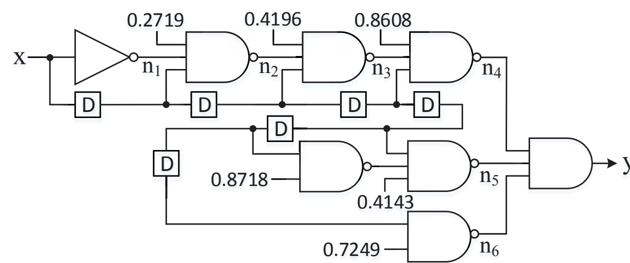


Figure 6.12: The circuit diagram of stochastic implementation of e^{-2x} using the 7th-order Maclaurin polynomial.

The output and internal nodes are given as follows:

$$n_1 = 1 - x$$
$$n_2 = 1 - 0.2719 n_1 x$$
$$n_3 = 1 - 0.4196 n_2 x$$
$$n_4 = 1 - 0.8608 n_3 x$$
$$n_5 = 1 - 0.4143 x (1 - 0.8718 x)$$
$$n_6 = 1 - 0.7249 x$$
$$y_i = n_4 n_5 n_6$$

Then, equation (6.27) is implemented by cascading the circuit shown in Fig. 6.11 with the architecture shown in Fig. 6.12. Notice that inputs are in bipolar format while the final output is in unipolar format. The intermediate signal w_i is used as the unipolar input of the implementation for exponential function.

As shown in Fig. 6.13, the whole stochastic RBF kernel is implemented by multiplying four outputs y_i from stochastic exponential function e^{-2w_i} using an AND gate.

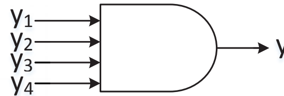


Figure 6.13: The computation of the final output for stochastic RBF kernel.

6.4.3 Test RBF Kernel of SVM classifier based on EEG Signals

In this section, the stochastic RBF kernel of SVM classifier is tested using features extracted from EEG signals. We present comparisons in terms of accuracy between two proposed designs.

The original EEG signals are taken from the dataset from the recent American Epilepsy Society Seizure Prediction Challenge database [33] [58]. In our test, EEG signals of one patient were sampled from 16 electrodes at 400 Hz, and recorded voltages were referenced to the group average. Features used in our design are spectral power ratios of selected subbands of EEG signals captured from selected electrodes [60]. In our test, four features are extracted for the patient. Assume that support vectors are already known from training process. We focus on the kernel computation in the testing process.

The testing data include 10244 samples and each sample is a vector with 4 elements ($\boldsymbol{x} = \{x_1, x_2, x_3, x_4\}$). Notice that the bipolar stochastic logic requires the range of $[-1, 1]$ for input signal. To this end, each sample needs to be scaled separately using l_1 scaling [27] as follows:

$$\boldsymbol{x} \Leftarrow \frac{\boldsymbol{x}}{x_{max}}, \tag{6.32}$$

where x_{max} represents the maximum magnitude of the input data. Support vectors are trained based on scaled input data. The stochastic RBF kernel is implemented using two approaches proposed in Section 6.4.1 and Section 6.4.2. In our simulation, the length of stochastic bit stream is 1024 and the RNG is implemented using a 10-bit LFSR. The output mean absolute error (MAE) of two implementations of stochastic RBF kernel are presented in Table 6.15. The MAE is computed by using the implementation with floating point precision as the ideal case. Five support vectors denoted by SV_i are selected in our test. 1000 simulations are performed for each test. It is shown in

Table 6.15 that the error of the implementation with bipolar input and unipolar output is reduced by 24.90%, compared to the implementation with bipolar input and output.

Table 6.15: The output mean absolute error (MAE) of two implementations of stochastic RBF kernel.

Support Vector	SV_1	SV_2	SV_3	SV_4	SV_5
Imp-1	0.0251	0.0242	0.0250	0.0251	0.0265
Imp-2	0.0195	0.0202	0.0208	0.0205	0.0198

6.5 conclusion

Novel stochastic implementations of machine learning classifiers including SVM and ANN have been presented in this chapter. These proposed architectures are validated using seizure prediction from EEG signals as an application example. Future work will be directed towards analyzing the proposed data-oriented optimization for non-linear classifiers using stochastic logic. The area and power consumption are included only for the classifiers and do not include the feature extraction. Future work will be also directed towards a complete implementation that includes computing features. We also presented novel stochastic implementations of RBF kernels for SVM classifiers. These proposed architectures are tested using EEG signals for seizure prediction. We have also presented a specific implementation of e^{-2x} using Horner's rule. Several other alternative architectures need to be exploited to compute e^{-2x}. In one approach, the factored form can be implemented using stochastic logic. In another implementation, all positive and negative terms can be grouped and computed separately, and the result can be computed by using a subtractor [62]. Future work will also be directed towards a complete implementation that includes computing features and processing final classification.

Chapter 7

Conclusion and Future Work

This dissertation has considered digital signal processing and machine learning system design using stochastic logic.

We investigate the implementation of linear-phase FIR digital filters in stochastic logic. Two novel architectures of stochastic linear-phase FIR filter based on lattice structures have been presented. Basic, normalized and modified lattice structures are considered for the stochastic implementation. Compared with the previous stochastic implementation of FIR filters in direct-form, the proposed lattice implementations can obtain equivalent performance and involve less hardware complexity. The power consumption of stochastic implementation is also reduced by the proposed architectures. However, the critical path delay of the proposed implementations is greater than that of stochastic implementation in direct-form. Compared stochastic implementations with conventional binary implementations, the hardware complexity and critical path delay are reduced. The stochastic implementations also show significantly better fault-tolerance than conventional binary implementations.

We have also presented novel structures for stochastic logic implementation of recursive digital filters. These structures are based on state-space and lattice forms. Out of these nine structures, two are optimized with respect to the number of binary multiplications; these structures require one-third of the number of binary multiplications compared to their non-optimized versions. It is shown that the normalized state-space and normalized lattice filters have the highest SER among all six proposed stochastic filters. The last three implementations are based on the normalized lattice structure

and the modified lattice structure, respectively. Compared with previous designs, the proposed architectures improve the performance for narrow-band stochastic IIR filter and reduce the hardware complexity significantly.

Stochastic logic based implementations of complex arithmetic functions using truncated Maclaurin series polynomials have been presented. The methods based on Horner's rule, factorization and format conversion are proposed. Future work will be directed towards stochastic logic implementations of different types of machine learning classifiers.

In this dissertation we present two approaches to compute polynomials in unipolar stochastic logic. The first implementation requires a stochastic subtractor. The second design is based on factorization. Moreover, stochastic implementations of complex arithmetic functions based on Taylor series and proposed polynomial computations are presented. Compared to previous designs using Bernstein polynomials, the proposed architectures achieve better accuracy and less hardware complexity.

Novel stochastic implementations of machine learning classifiers including SVM and ANN have been presented in this chapter. These proposed architectures are validated using seizure prediction from EEG signals as an application example. We also presented novel stochastic implementations of RBF kernels for SVM classifiers. These proposed architectures are tested using EEG signals for seizure prediction. This paper has presented a specific implementation of e^{-2x} using Horner's rule. Several other alternative architectures need to be exploited to compute e^{-2x}. In one approach, the factored form of (13) can be implemented using stochastic logic. In another implementation, all positive and negative terms can be grouped and computed separately, and the result can be computed by using a subtractor [62]. Future work will be directed towards analyzing the proposed data-oriented optimization for non-linear classifiers using stochastic logic. The area and power consumption are included only for the classifiers and do not include the feature extraction. Future work will be also directed towards a complete implementation that includes computing features.

References

[1] Peng Li, David J Lilja, Weikang Qian, Kia Bazargan, and Marc Riedel. The synthesis of complex arithmetic computation on stochastic bit streams using sequential logic. In *Proceedings of the International Conference on Computer-Aided Design*, pages 480–487. ACM, 2012.

[2] Brian R Gaines. Stochastic computing. In *Proceedings of AFIPS spring joint computer conference*, pages 149–156. ACM, 1967.

[3] Brian R Gaines. Stochastic computing systems. In *Advances in information systems science*, pages 37–172. Springer, 1969.

[4] Armin Alaghi and John P Hayes. Survey of stochastic computing. *ACM Transactions on Embedded computing systems (TECS)*, 12(2s):92, 2013.

[5] Weikang Qian, Xin Li, Marc D Riedel, Kia Bazargan, and David J Lilja. An architecture for fault-tolerant computation with stochastic logic. *IEEE Transactions on Computers*, 60(1):93–105, 2011.

[6] Bert Moons and Marian Verhelst. Energy-efficiency and accuracy of stochastic computing circuits in emerging technologies. *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, 4(4):475–486, 2014.

[7] Peng Li and David J Lilja. A low power fault-tolerance architecture for the kernel density estimation based image segmentation algorithm. In *Proceedings of 2011 IEEE International Conference on Application-Specific Systems, Architectures and Processors (ASAP)*, pages 161–168. IEEE, 2011.

[8] Bradley D Brown and Howard C Card. Stochastic neural computation. I. computational elements. *IEEE Transactions on Computers*, 50(9):891–905, 2001.

[9] A Dinu, MN Cirstea, and M McCormick. Stochastic implementation of motor controllers. In *Proceedings of the 2002 IEEE International Symposium on Industrial Electronics (ISIE)*, volume 2, pages 639–644, 2002.

[10] Weikang Qian, Marc D Riedel, Hongchao Zhou, and Jehoshua Bruck. Transforming probabilities with combinational logic. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 30(9):1279–1292, 2011.

[11] Antoni Morro, Vincent Canals, Antoni Oliver, Miquel L Alomar, and Josep L Rossello. Ultra-fast data-mining hardware architecture based on stochastic computing. *PLoS ONE 10(5): e0124176. doi:10.1371/journal. pone.0124176*, 2015.

[12] Ali Naderi, Shie Mannor, Mohamad Sawan, and Warren J Gross. Delayed stochastic decoding of LDPC codes. *IEEE Transactions on Signal Processing*, 59(11):5617–5626, 2011.

[13] Bo Yuan and Keshab K Parhi. Successive cancellation decoding of polar codes using stochastic computing. In *Proceedings of IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 3040–3043. IEEE, 2015.

[14] V Gaudet and A Rapley. Iterative decoding using stochastic computation. *Electronics Letters*, 39(3):299–301, 2003.

[15] Saeed Sharifi Tehrani, Ali Naderi, G-A Kamendje, Saied Hemati, Shie Mannor, and Warren J Gross. Majority-based tracking forecast memories for stochastic ldpc decoding. *IEEE Transactions on Signal Processing*, 58(9):4883–4896, 2010.

[16] Yin Liu and Keshab K Parhi. Lattice FIR digital filter architectures using stochastic computing. In *Proceedings of IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 1027–1031, 2015.

[17] Yin Liu and Keshab K Parhi. Linear-phase lattice fir digital filter architectures using stochastic logic. *Journal of Signal Processing Systems*, pages 1–13, 2017.

[18] Keshab K Parhi and Yin Liu. Architectures for IIR digital filters using stochastic computing. In *Proceedings of 2014 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 373–376, 2014.

[19] Yin Liu and Keshab K Parhi. Architectures for recursive digital filters using stochastic computing. *IEEE Transactions on Signal Processing*, 64(14):3705–3718, 2015.

[20] Yin Liu and Keshab K Parhi. Lattice FIR digital filters using stochastic computing. In *Proceedings of 2015 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP), Brisbane, Australia*, pages 1027–1031, April 2015.

[21] Naoya Onizawa, Shunsuke Koshita, and Takahiro Hanyu. Scaled iir filter based on stochastic computation. In *2015 IEEE 58th International Midwest Symposium on Circuits and Systems (MWSCAS)*, pages 1–4. IEEE, 2015.

[22] Naoya Onizawa, Daisaku Katagiri, Kazumichi Matsumiya, Warren J Gross, and Takahiro Hanyu. Gabor filter based on stochastic computation. *IEEE Signal Processing Letters*, 22(9):1224–1228, 2015.

[23] Weikang Qian, Marc D Riedel, Kia Bazargan, and David J Lilja. The synthesis of combinational logic to generate probabilities. In *Proceedings of the 2009 International Conference on Computer-Aided Design*, pages 367–374. ACM, 2009.

[24] Weikang Qian and Marc D Riedel. The synthesis of robust polynomial arithmetic with stochastic logic. In *45th ACM/IEEE Design Automation Conference (DAC), 2008.*, pages 648–653, 2008.

[25] George G Lorentz. *Bernstein polynomials*. American Mathematical Soc., 2012.

[26] A Gray Jr. and J Markel. Digital lattice and ladder filter synthesis. *IEEE Transactions on Audio and Electroacoustics*, 21(6):491–500, 1973.

[27] Keshab K. Parhi. *VLSI Digital Signal Processing Systems: Design and Implementation*. Hoboken, NJ: Wiley, Wiley 1999.

[28] Karl Schwarz. Linear phase FIR-filter in lattice structure. In *Proceedings of IEEE International Symposium on Circuits and Systems (ISCAS), 1993*, pages 347–350, 1993.

[29] ICA'99 synthetic benchmarks. `http://sound.media.mit.edu/ica-bench/`, September 2014.

[30] Yun-Nan Chang and Keshab K Parhi. Architectures for digital filters using stochastic computing. In *Proceedings of 2013 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 2697–2701, 2013.

[31] Peng Li, Weikang Qian, and David J Lilja. A stochastic reconfigurable architecture for fault-tolerant computation with sequential logic. In *Proceedings of the International Conference on Computer Design (ICCD)*, pages 303–308. IEEE, 2012.

[32] Peng Li, Weikang Qian, Marc D Riedel, Kia Bazargan, and David J Lilja. The synthesis of linear finite state machine-based stochastic computational elements. In *Proceedings of 2012 17th Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 757–762. IEEE, 2012.

[33] American epilepsy society seizure prediction challenge. `http://www.kaggle.com/c/seizure-prediction`.

[34] J Schur. Über potenzreihen, die im innern des einheitskreises beschränkt sind. *Journal für die reine und angewandte Mathematik*, 147:205–232, 1917.

[35] Yin Liu and Keshab K Parhi. Architectures for stochastic normalized and modified lattice IIR filters. In *Proceedings of 2015 Asilomar Conference on Signals, Systems and Computers, Pacific Grove, CA*, Nov. 2015.

[36] Alan V Oppenheim, Ronald W Schafer, John R Buck, et al. *Discrete-time signal processing*, volume 2. Prentice-hall Englewood Cliffs, 1989.

[37] Megha Parhi, Marc D Riedel, and Keshab K Parhi. Effect of bit-level correlation in stochastic computing. In *Proceddings of IEEE International Conference on Digital Signal Processing (DSP)*, pages 463–467, 2015.

[38] Keshab K Parhi and Yin Liu. Architectures for IIR digital filters using stochastic computing. In *Proceedings of IEEE International Symposium on Circuits and Systems (ISCAS)*. IEEE, 2014.

[39] Naman Saraf, Kia Bazargan, David J Lilja, and Marc D Riedel. IIR filters using stochastic arithmetic. In *Proceedings of Design, Automation and Test in Europe Conference and Exhibition (DATE), 2014*, pages 1–6. IEEE.

[40] Jin-Gyun Chung and Keshab K Parhi. Scaled normalized lattice digital filter structures. *IEEE Transactions on Circuits and Systems II: Analog and Digital Signal Processing*, 42(4):278–282, 1995.

[41] John E Markel and Augustine H Gray. *Linear prediction of speech*. Springer-Verlag New York, Inc., 1982.

[42] Thilo Penzl. Numerical solution of generalized Lyapunov equations. *Advances in Computational Mathematics*, 8(1-2):33–48, 1998.

[43] Yin Liu and Keshab K Parhi. Computing polynomials using unipolar stochastic logic. *ACM Journal on Emerging Technologies in Computing*, to appear.

[44] Pai-Shun Ting and John P Hayes. Isolation-based decorrelation of stochastic circuits. In *Proceedings of IEEE International Conference on Computer Design (ICCD)*, 2016.

[45] G Szego. Uber eine eigenschaft der exponentialreihe. In *Sitzungsberichte, Berliner Mathematische Gesellschaft*, volume 23, pages 50–64, 1924.

[46] T Kriecherbauer, ABJ Kuijlaars, KDTR McLaughlin, and PD Miller. Locating the zeros of partial sums of exp(z) with riemann-hilbert methods. In *Contemporary Mathematics*, volume 458, pages 183–196, 2008.

[47] Yin Liu and Keshab K Parhi. Computing RBF kernel for SVM classification using stochastic logic. In *Proceedings of 2016 IEEE Workshop on Signal Processing Systems*. Dallas.

[48] Peng Li and David J Lilja. Using stochastic computing to implement digital image processing algorithms. In *Proceedings of 2011 IEEE 29th International Conference on Computer Design (ICCD)*, pages 154–161. IEEE, 2011.

[49] Peng Li, David J Lilja, Weikang Qian, Marc D Riedel, and Kia Bazargan. Logical computation on stochastic bit streams with linear finite-state machines. *IEEE Transactions on Computers*, 63(6):1474–1486, 2014.

[50] Peng Li, David J Lilja, Weikang Qian, Kia Bazargan, and MD Riedel. Using a two-dimensional finite-state machine for stochastic computation. In *International Workshop on Logic and Synthesis, IWLS*, volume 12, 2012.

[51] Armin Alaghi and John P Hayes. A spectral transform approach to stochastic circuits. In *2012 IEEE 30th International Conference on Computer Design (ICCD)*, pages 315–321, 2012.

[52] Armin Alaghi and John P Hayes. STRAUSS: Spectral transform use in stochastic circuit synthesis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 34(11):1770–1783, 2015.

[53] Yi Wu, Chen Wang, and Weikang Qian. Minimizing error of stochastic computation through linear transformation. In *Proceedings of the 25th edition on Great Lakes Symposium on VLSI*, pages 349–354. ACM, 2015.

[54] Yin Liu and Keshab K Parhi. Computing complex functions using factorization in unipolar stochastic logic. In *Proceedings of the 26th edition on Great Lakes Symposium on VLSI*, pages 109–112. ACM, 2016.

[55] Keshab K Parhi and Yin Liu. Computing arithmetic functions using stochastic logic by series expansion. *IEEE Transactions on Emerging Topics in Computing*, page to appear, 2016.

[56] Manohar Ayinala and Keshab Parhi. Low-energy architectures for support vector machine computation. In *2013 Asilomar Conference on Signals, Systems and Computers*, pages 2167–2171. IEEE, 2013.

[57] A Aizerman, Emmanuel M Braverman, and LI Rozoner. Theoretical foundations of the potential function method in pattern recognition learning. *Automation and remote control*, 25:821–837, 1964.

[58] Zisheng Zhang and Keshab K Parhi. Seizure prediction using polynomial svm classification. In *Proceedings of 2015 IEEE Engineering in Medicine and Biology Society Conference (EMBC)*, pages 5748–5751.

[59] Yun Park, Lan Luo, Keshab K Parhi, and Theoden Netoff. Seizure prediction with spectral power of eeg using cost-sensitive support vector machines. *Epilepsia*, 52(10):1761–1770, 2011.

[60] Zisheng Zhang and Keshab K Parhi. Low-complexity seizure prediction from ieeg/seeg using spectral power and ratios of spectral power. *IEEE Transactions on Biomedical Circuits and Systems*, 10(3):693–706, 2016.

[61] Yin Liu and Keshab K Parhi. Computing complex functions using factorization in unipolar stochastic logic. In *Proceedings of the 26th edition on Great Lakes Symposium on VLSI*. ACM, 2016.

[62] Yin Liu and Keshab K Parhi. Computing subtraction and polynomial computation using unipolar stochastic logic. In *Proceedings of 2016 50th Asilomar Conference on Signals, Systems and Computers*. IEEE.