Analysis, Design, and Logic Synthesis of Finite-state Machine-based Stochastic Computing

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

Peng Li

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

Advisor: David J. Lilja

June, 2013

© Peng Li 2013
ALL RIGHTS RESERVED

Acknowledgements

There are many people that have earned my gratitude for their contribution to my time in graduate school.

First and foremost, I want to thank my advisor Prof. David J. Lilja, who has been great mentor and teacher. It is my great honor to be his Ph.D. student. Throughout my graduate study, David has spent numerous amount of time to mentor me in my research. He also gives me a lot of guidance and advice on my career development. I appreciate his financial support which allows me to devote all of my attention to the research. As a student of him, I not only have developed expertise in several areas, but also have learned some important skills in research.

I would like to thank the other members of my dissertation committee, Dr. Kia Bazargan, Dr. Marc Riedel, and Dr. Antonia Zhai at the University of Minnesota, for their valuable suggestions and feedback on this project and the dissertation.

I am grateful to Dr. Weikang Qian at the University of Michigan-Shanghai Jiao Tong University Joint Institute, for his valuable feedback to my research and his collaboration in the stochastic computing project.

I would also like to thank Prof. David Du, Dr. Cory Devor, Dr. Weijun Xiao, Kevin Gomez, and David Tetzlaff, for their kind help throughout my graduate study.

Dedication

I would like to thank my mother Meiji Zhang and my father Jie Li, for their love and encouragement. Without their support in my education from the very beginning, I would not have earned my Ph.D. degree.

Abstract

Most digital systems operate on a positional representation of data, such as binary encoding. An alternative is to operate on random bit streams where the signal value is encoded by the probability of obtaining a one versus a zero. This representation is much less compact than the binary encoding. However, complex operations can be performed with very simple logic. Furthermore, since the representation is uniform, with all bits weighted equally, it is highly tolerant of soft errors (i.e., bit flips).

Complex algorithms, such as artificial neural networks (ANN), low-density parity-check (LPDC) error-correcting coding, and kernel density estimation (KDE)-based image segmentation, can be implemented using stochastic encoding with much lower hardware cost and higher fault-tolerance. For example, the hardware area of the stochastic implementation of the KDE-based image segmentation is only 1.2% of the corresponding deterministic implementation, and it can tolerate more than 30% soft errors. Compared to conventional fault-tolerant techniques, such as triple-module redundance (TMR), the stochastic implementations of the complex algorithms normally consumes equivalent or less energy and have better fault-tolerance. For example, the TMR implementation of the KDE-based implementation can only tolerate up to 10% soft errors, but consumes the same energy as the stochastic implementation. In addition, thanks to the simple construction of the stochastic computing elements, it makes the rounting much easier, which is a big issue for very large scale integrated circuit (VLSI) deterministic implementations of these complex algorithms.

Both combinational and sequential constructs have been proposed for operating on stochastic bit streams. Prior work has shown that combinational logic can implement multiplication and scaled addition effectively while finite-state machines (FSMs) can implement complex functions such as exponentiation and tanh effectively. Although the combinational logic-based stochastic computing elements had been well studied, they are inefficient for complex operations. The FSM-based stochastic computing elements are very efficient for complex operations. However, only three FSM-based stochastic computing elements were proposed by prior work, which limits the applications of

stochastic computing. To implement more applications and functions stochastically, this dissertation focuses on the FSM-based stochastic computing.

We first analyze the FSM-based stochastic computing elements proposed by prior work, which had largely been validated empirically. In this dissertation, we provide a rigorous mathematical treatment of the FSM-based stochastic computing elements. This gives us intuition about how to construct arbitrary functions stochastically using the FSM.

Then, based on the existing stochastic computing elements, we implement five digital image processing algorithms as case studies. So far as we know, this is the first time these digital image processing algorithms are implemented stochastically. For all the five algorithms, the stochastic implementation has much less hardware cost and better fault-tolerance than the corresponding deterministic implementation.

Last but not the least, we present a general method to synthesize a given target function stochastically using FSMs. We proposed three FSM topologies and discuss how to use these FSMs to synthesize the given target functions. The trade-offs among these different FSM topologies are introduced. Based on this synthesis method, more applications can be implemented stochastically to achieve lower hardware cost and better fault-tolerance.

Contents

Acknowledgements i

Dedication ii

Abstract iii

List of Tables viii

List of Figures x

1 Introduction 1

2 Background of Stochastic Computing 7

2.1 Conversion Approach 7

2.2 Coding Format 8

2.3 Combinational Logic-based Stochastic Computing Elements 9

2.3.1 Scaled Addition 9

2.3.2 Scaled subtraction 10

2.3.3 Multiplication 11

2.3.4 Synthesis Method Using Combinational Logic 12

2.4 FSM-based Stochastic Computing Elements 14

2.5 Other Types of Stochastic Computing 18

3 FSM-based Stochastic Computing Elements 20

3.1 Properties of the Single Input Linear FSM 20

3.1.1 Property 1 22

3.1.2 Property 2 23
3.1.3 Property 3 25
3.2 The FSM-based Stochastic Computing Elements 27
3.2.1 Stochastic Tanh Function 27
3.2.2 Stochastic Exponentiation Function 30
3.2.3 Stochastic Exponentiation Function base on an Absolute Value . 32
3.2.4 Stochastic Absolute Value Function 33
3.2.5 Stochastic Linear Gain Function 35
3.3 Error Analysis 40
3.3.1 Error Due to the Function Approximation 41
3.3.2 Quantization Error 43
3.3.3 Error Due to Random Fluctuations 44
3.3.4 Summary of Error Analysis 45
3.4 Experimental Evaluation 46
3.4.1 Comparison with Binary Radix 46

4 Digital Image Processing Case Studies **51**
4.1 Stochastic Implementations of Image Processing Algorithms 51
4.1.1 Edge Detection 52
4.1.2 Noise Reduction Based on The Median Filter 53
4.1.3 Contrast Stretching 56
4.1.4 Frame Difference-Based Image Segmentation 57
4.1.5 KDE-Based Image Segmentation 59
4.2 Experimental Results 62
4.2.1 Simulation Results 62
4.2.2 Hardware Resources Comparison 62
4.2.3 Energy Consumption Comparison 64
4.2.4 Fault-Tolerance Comparison 65
4.2.5 Fault-Tolerance Analysis 68

5 FSM-based Synthesis Methods **73**
5.1 Single Input FSM-Based Synthesis Approach 73
5.1.1 FSM Analysis 75

5.1.2 Synthesis Approach 75
5.2 Two-Input Linear FSM-Based Synthesis Approach 78
5.2.1 FSM Analysis 79
5.2.2 Synthesis Approach 80
5.2.3 Synthesis Examples 81
5.3 Two-Input Two-Dimension FSM-Based Synthesis Approach 85
5.3.1 FSM Analysis 87
5.3.2 Synthesis Approach 88
5.3.3 Synthesis Examples 90
5.4 Comparison of Different Synthesis Approaches 90
5.4.1 Hardware Cost of Different FSM Topologies 91
5.4.2 Comparison with the Bernstein Polynomial-based Approach . . . 92

6 Other Encoding Schemes **95**
6.1 Equally Weighted Non-stochastic Encoding 95
6.1.1 Computing Absolute Values 96
6.1.2 Making Comparisons 97
6.2 Overlapped Stochastic Encoding 99
6.2.1 Sharing Bits based on Fixed Length 100
6.2.2 Variable Overlap Bits Sharing 101

7 Conclusion and Future Work **104**

References **107**

List of Tables

3.1 Approximation errors (e_a) of the four FSM-based stochastic computing elements versus different numbers of states. 43

3.2 The maximum quantization errors (e_q) of the FSM-based stochastic computing elements. 44

3.3 The average overall errors (e) of the five FSM-based stochastic computing elements versus different lengths of stochastic bit streams. 46

3.4 The area and delay of the stochastic implementations and the deterministic implementations of the five functions. Delay values of the stochastic implementations stand for the overall delay. Critical path delay of the stochastic implementations can be obtained by dividing 2^M. 47

3.5 Relative errors of the stochastic implementations and the deterministic implementations of the five functions versus different error ratios ϵ in the input data. 49

4.1 Hardware resources comparison in terms of the equivalent two-input NAND gates. 64

4.2 Energy consumption comparison. 65

4.3 Area-delay product comparison. 65

4.4 Fault tolerance test results. 68

5.1 P_K and P_{w_t} $(0 \leq t \leq 7)$ for synthesizing the target function in (5.12) with $\delta = 2$ and $\mu = 0$. 82

5.2 P_K and P_{w_t} for synthesizing the target function in (5.13). 84

5.3 P_K and P_{w_t} for synthesizing the target function in (5.20) with $\alpha = 30$. . 90

5.4 Hardware area (um^2) of three different FSMs. 91

5.5 The number of the fan-in two logic gates for computing Bernstein polynomials of degree 3, 4, 5, and 6 [1]. 93
5.6 Relative error of the FSM-based and the Bernstein polynomial-based implementations of target function computation versus the error ratio γ in the input data. 94

List of Figures

1.1 Stochastic implementation of arithmetic operations [2]. Note that for real applications, the length of the bit stream for representing a value is normally greater than 100 for reducing the output errors due to random fluctuations. In this example, we use the 8-bit streams to simplify the discussion. 2

1.2 A comparison of the fault tolerance capabilities of different hardware implementations for the frame difference based image segmentation algorithm. The images in the second row are generated by a conventional implementation. The images in the third row are generated using a stochastic implementation. Soft errors are injected at a rate of (a) 0%; (b) 1%; (c) 2%; (d) 5%; (e) 10%; (f) 15%; (g) 30%. 3

2.1 The Randomizer Unit [3]. 8

2.2 The scaled addition implemented with a MUX. 10

2.3 The scaled subtraction implemented with a MUX and a NOT gate for the bipolar coding format. 10

2.4 Multiplication in stochastic computing. (a) AND Gate for multiplication using unipolar coding format. (b) XOR Gate for multiplication using bipolar coding format. 11

2.5 A reconfigurable stochastic computing architecture based on combinational logic (i.e., the *ReSC Unit* is implemented using an adder and a multiplexer) [3]. 12

2.6 *ReSC Unit* implementing the Bernstein polynomial $f(x) = \frac{2}{8}B_{0,3}(x) + \frac{5}{8}B_{1,3}(x) + \frac{3}{8}B_{2,3}(x) + \frac{6}{8}B_{3,3}(x)$ 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 [3]. 13
2.7 State transition diagram of the FSM-based stochastic exponentiation function. 14
2.8 Simulation result of the FSM-based stochastic exponentiation function. 15
2.9 State transition diagram of the FSM-based stochastic tanh function. . . 15
2.10 Simulation result of the FSM-based stochastic tanh function. 16
2.11 A generic linear state transition diagram. 16
2.12 State transition diagram of the FSM implementing the stochastic linear gain function. 17
2.13 Simulation result of the FSM-based stochastic linear gain function. . . . 17
3.1 A generic linear state transition diagram. 21
3.2 State transition diagram of the FSM-based stochastic tanh function. . . 27
3.3 The stochastic comparator. 30
3.4 State transition diagram of the FSM-based stochastic exponentiation function. 30
3.5 State transition diagram of the stochastic exponentiation function base on an absolute value. 32
3.6 State transition diagram of the stochastic absolute value function. . . . 33
3.7 State transition diagram of the FSM implementing the stochastic linear gain function. 35
3.8 A two-parameter stochastic linear gain function. 39
3.9 Simulation results of the stochastic exponentiation function based on an absolute value ($G = 2$ in (3.25)). 41
3.10 Simulation results of the stochastic absolute value function. 42
3.11 The average of the relative errors of the four functions shown in Table 3.5. 50
4.1 Robert's cross operator for edge detection. 52
4.2 The conventional implementation of the Robert's cross operator based edge detection. 52

4.3 The stochastic implementation of the Robert's cross operator based edge detection. 53
4.4 Hardware implementation of the 3×3 median filter based on a sorting network. 54
4.5 The basic sorting unit. 54
4.6 The conventional implementation of the basic sorting unit. It requires 379 logic gates in total. 55
4.7 The stochastic implementation of the basic sorting unit, which can be implemented using only 66 logic gates. 55
4.8 A piecewise linear gain function used in image contrast stretching. . . . 56
4.9 The conventional implementation of the piecewise linear gain function used in image contrast stretching, where $d = \frac{255}{b-a}$ and $c = \frac{255a}{b-a}$ based on equation (4.1). 57
4.10 The conventional implementation of the frame difference-based image segmentation. The total number of logic gates of this implementation is 486. 58
4.11 The stochastic implementation of the frame difference-based image segmentation. The total number of logic gates of this implementation is 107. 58
4.12 The stochastic implementation of the KDE-based image segmentation algorithm. 60
4.13 The conventional implementation of the KDE-based image segmentation algorithm. 61
4.14 Simulation results of the conventional and the stochastic implementations of the image processing algorithms. 63
4.15 Test circuit for the error injection. 66
4.16 The circuit for simulating injected soft errors. A, B, C, and S are the original signals. $Error\ A$, $Error\ B$, $Error\ C$, and $Error\ S$ are the soft error signals generated by the random soft error source. A', B', C', and S' are the signals corrupted by the soft errors. 67

4.17 A comparison of the fault tolerance capabilities of different hardware implementations for the KDE-based image segmentation algorithm. The images in the top row are generated by a conventional deterministic implementation. The images in the second row are generated by the conventional implementation with a TMR-based approach. The images in the bottom row are generated using a stochastic implementation [4]. Soft errors are injected at a rate of (a) 0%; (b) 1%; (c) 2%; (d) 5%; (e) 10%; (f) 15%; (g) 30%. 69
5.1 The circuit for synthesizing target functions based on the single input linear FSM. 74
5.2 The FSM has a single input X. The numbers on each arrow represent the transition condition. The FSM has $log_2\lceil N\rceil$ outputs, encoding a value in binary radix. In the figure, the number below each state S_t $(0 \leq t \leq N-1)$ represents the value encoded by the output of the FSM when the current state is S_t. 74
5.3 The state transition diagram of an N-state Moore style FSM. It has 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. This FSM has $\lceil log_2 N\rceil$ outputs, encoding a value in binary radix. In the figure, the number below each state S_i $(0 \leq i \leq N-1)$ represents the output of the FSM when the current state is S_i. 78
5.4 The circuit for synthesizing target functions. 80
5.5 Synthesis result of the target function in (5.12) with $\delta = 2$ and $\mu = 0$. . 82
5.6 An example about how the circuit shown in Fig. 5.4 works (the FSM is implemented with the state transition diagram shown in Fig. 5.3). . . . 83
5.7 Synthesis result of the target function in (5.13). 85

5.8 The FSM has 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 FSM has $log_2\lceil MN \rceil$ outputs, encoding a value in binary radix. In the figure, 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. 86
5.9 An example of the state transition digram of the proposed FSM with 8 states. Here, $M = 2$ and $N = 4$. 87
5.10 The circuit for synthesizing target functions. 89
5.11 Synthesis result of the target function in (5.20) with $\alpha = 30$. 91
5.12 Logical computation on stochastic bit streams implementing the Bernstein polynomial $f(x) = \frac{2}{8}B_{0,3}(x) + \frac{5}{8}B_{1,3}(x) + \frac{3}{8}B_{2,3}(x) + \frac{6}{8}B_{3,3}(x)$ 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 [1]. 92
6.1 Representing the value $\frac{L}{N}$ using the proposed encoding scheme. 96
6.2 Computing absolute values using a single XOR gate based on the proposed encoding. 96
6.3 Computing absolute values using the stochastic encoding. 96
6.4 Computing the minimum value using an n-input AND gate. 97
6.5 Computing the maximum value using an n-input OR gate. 98
6.6 Comparator based on the proposed encoding. 98
6.7 An example of sharing 50% of bits in the consecutive bit stream to represent the deterministic value stochastically. 99
6.8 An example of sharing a variable number of bits based on the arithmetic difference of the adjacent values. 102

Chapter 1

Introduction

Future integrated circuits are expected to be made of emerging nano-scale devices and interconnects. The expected higher probability of failure as well as the higher sensitivities to noise and variations could make future integrated circuits prohibitively unreliable. In a paradigm first advocated by Gaines [5], logical computations are performed on stochastic bit streams: each real-valued number x ($0 \leq x \leq 1$) is represented by a sequence of random bits, each of which has probability x of being one and probability $1 - x$ of being zero. For example, the 10-bit stream "0110001000" represent a real-valued number 0.3. Note that each bit in this bitstream has a probability 0.3 of being one and probability 0.7 of being zero. Thus, we can use the total number of ones in this bit stream to evaluate the real-valued number. Compared to binary encoding, this stochastic encoding is not very compact. However, it leads to remarkably simple hardware for complex functions; it also provides very high tolerance to soft errors.

For example, as shown in Fig. 1.1(a), with the unipolar coding format, multiplication can be implemented using an AND gate. Assuming that the two input stochastic bit streams A and B are independent, then we have,

$$\begin{aligned} c &= P(C = 1) = P(A = 1 \text{ and } B = 1) \\ &= P(A = 1)P(B = 1) = a \cdot b. \end{aligned} \tag{1.1}$$

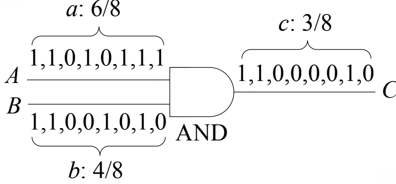
Thus, the number represented by the output stochastic bit stream C is $c = a \cdot b$. The AND gate multiplies the two values represented by the stochastic bit streams.

We can also implement scaled addition with a multiplexer, as shown in Fig. 1.1(b).

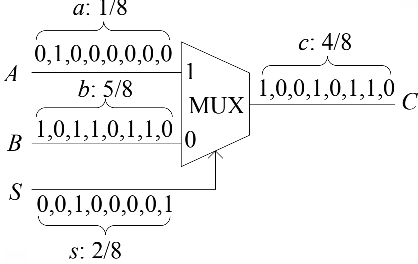
With the assumption that the three input stochastic bit streams A, B, and S are independent, we have,

$$\begin{aligned} c &= P(C = 1) \\ &= P(S = 1 \text{ and } A = 1) + P(S = 0 \text{ and } B = 1) \\ &= P(S = 1)P(A = 1) + P(S = 0)P(B = 1) \\ &= s \cdot a + (1 - s) \cdot b. \end{aligned} \tag{1.2}$$

Thus, the number represented by the output stochastic bit stream C is $c = s{\cdot}a{+}(1{-}s){\cdot}b$. With the unipolar coding format, the computation performed by a multiplexer is the scaled addition of the two input values a and b, with a scaling factor of s for a and $1-s$ for b. Note that it is not feasible to add two probability values directly; this could result in a value greater than one, which cannot be represented as a probability value.



(a) Multiplication with the unipolar coding. Here the inputs are 6/8 and 4/8. The output is $6/8 \times 4/8 = 3/8$.



(b) Scaled addition with the unipolar coding. Here the inputs are $1/8, 5/8$, and $2/8$. The output is $2/8 \times 1/8 + (1-2/8) \times 5/8 = 4/8$.

Figure 1.1: Stochastic implementation of arithmetic operations [2]. Note that for real applications, the length of the bit stream for representing a value is normally greater than 100 for reducing the output errors due to random fluctuations. In this example, we use the 8-bit streams to simplify the discussion.

The images in Fig. 1.2 illustrate the fault tolerance capability of stochastic computing for the frame difference-based image segmentation algorithm as opposed to a conventional implementation. As the soft error injection rate increases, a conventional implementation of the image segmentation algorithm rapidly degrades until the output image is no longer recognizable. However, the second row shows that an implementation using stochastic bit streams to encode the data is able to produce the correct output at even very high error rates [6].

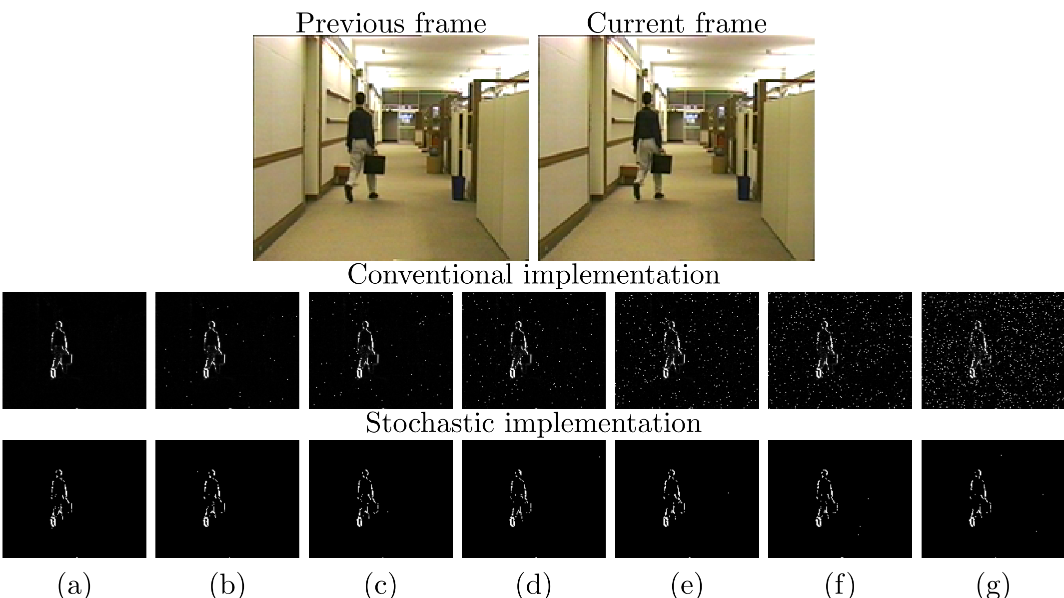


Figure 1.2: A comparison of the fault tolerance capabilities of different hardware implementations for the frame difference based image segmentation algorithm. The images in the second row are generated by a conventional implementation. The images in the third row are generated using a stochastic implementation. Soft errors are injected at a rate of (a) 0%; (b) 1%; (c) 2%; (d) 5%; (e) 10%; (f) 15%; (g) 30%.

In the decades since Gaines' original work, there have been numerous papers discussing the paradigm. For example, Keane and Atlas [7] proposed a stochastic implementation of a finite impulse response filter for digital signal processing. Gaudet and Rapley proposed a stochastic implementation of an iterative decoding algorithm [8]. Gross et al. proposed a stochastic implementation of a low-density parity-check decoder [9]. McNeill and Card proposed a refractory system for counting pulses in neural computation [10]. Li et al. proposed a stochastic implementation of a neural network

controller for small wind turbine systems [11]. Hori et al. proposed a stochastic implementation of a blind source separation system [12]. Onomi et al. proposed a stochastic implementation of a high-speed single flux-quantum up/down counter for neural computation [13]. Qian et al. presented a general synthesis method for logical computation on stochastic bit streams [3, 1, 14]. They showed that combinational logic can be synthesized to implement arbitrary polynomial functions, provided that such polynomials map the unit interval onto the unit interval. Their method is based on novel mathematics for manipulating polynomials in a form called Bernstein polynomials. In [1], Qian et al showed how to realize such a polynomial with a form of "generalized multiplexing." In [3], they demonstrated a reconfigurable architecture for computation on stochastic bit streams. In [14], they showed how to convert a general power-form polynomial into a Bernstein polynomial with coefficients in the unit interval. They analyzed cost as well as the sources of error: approximation, quantization, and random fluctuations. They also studied the effectiveness of the architecture on a collection of benchmarks for image processing.

Among them, most notable has been the work by Brown and Card [15, 16]. They demonstrated efficient constructs for a wide variety of basic functions, including multiplication, squaring, addition, subtraction, and division. Further, they provided elegant constructs for complex functions such as tanh, linear gain, and exponentiation. Such functions are of interest to the artificial neural networks community. The tanh function, in particular, performs a non-linear, sigmoidal mapping; this is used to model the activation function of a neuron. They used combinational logic to implement simple functions such as multiplication and scaled addition. They further used sequential logic in the form of finite-state machines (FSMs) to implement complex functions such as tanh. It was the first time that FSMs were used to construct sophisticated functions stochastically. Their contributions are significant: systems implemented using the FSM-based stochastic computing elements normally have better performance in terms of energy consumption, hardware cost, and fault-tolerance. However, many questions still need to be answered regarding the FSM-based stochastic computing elements. For example, how can we mathematically show how the behavior of the FSM relate to the computation of the tanh function? Can we implement more functions stochastically using the FSM, and how can we design the corresponding state transition diagrams? Since Brown

and Card's work had an empirical focus and complex constructs were validated by simulation only, we cannot obtain such answers directly from their prior work. As a result, the algorithms that can be implemented using the FSM-based stochastic computing elements are very limited. Most algorithms in digital image processing and artificial neural networks cannot benefit from this technique without additional research.

The first goal of this dissertation is to provide a rigorous mathematical treatment of the FSM-based stochastic computing elements proposed by Brown and Card [15], which can help us better understand the behavior of the FSM in stochastic computing and give us intuitions to implement more functions stochastically using the FSM.

The second goal of this dissertation is to demonstrate the applications of the FSM-based stochastic computing elements for specific digital image processing algorithms, which can be used as practical case studies to evaluate the trade-off between deterministic implementations and stochastic implementations of these algorithms.

The third goal of this dissertation is to develop a general synthesis approach to synthesize a given target function stochastically using the FSMs, so that more applications can be implemented stochastically.

The primary contributions of this dissertation are:

- The analysis of the three stochastic computing elements, including stochastic tanh, exponentiation, and linear gain functions [17].
- The analysis of three fundamental properties of the linear FSMs used to construct those computing elements [2].
- The development and analysis of two new stochastic computing elements, including stochastic absolute values and exponentiation based on absolute values [4, 6].
- The stochastic implementations of five digital image processing algorithms, which include image edge detection, median filter-based noise reduction, image contrast stretching, frame difference-based image segmentation, and KDE-based image segmentation [4, 6].
- The comparison between the stochastic implementations and the deterministic implementations of these five algorithms in terms of the hardware cost, latency and energy consumption [18, 19].

- The general synthesis method using three different FSM topologies [20, 21].
- The analysis of the trade-off among these FSM topologies [22, 23].
- The development of two new encoding schemes [24].

The remainder of this dissertation is organized as follows.

- Chapter 2 provides background information including the details of stochastic encoding and decoding, combinational logic for simple functions, and several FSM-based constructs proposed by Brown and Card [15].
- In Chapter 3, we present and analyze five FSM-based constructs: tanh, exponentiation, absolute values, exponentiation based on absolute values, and linear gain function. The sources of error in these five constructs are analyzed. Experimental results for the cost, the performance, and the error-tolerance of both stochastic and deterministic implementations of the five constructs are also demonstrated.
- In Chapter 4, we introduce the FSM-based stochastic implementations of the five digital image processing algorithms. We analyze the error tolerate of these FSM-based stochastic implementations and compare them to the implementations using the binary encoding. We also analyze and compare the hardware cost, latency and energy consumption.
- Chapter 5 introduces a general method for synthesizing the given target function stochastically using different FSMs, and discusses the trade-offs among these FSMs by comparing their hardware cost and limitations.
- Chapter 6 introduces two new encoding schemes.
- Chapter 7 presents a final discussion of the analyses presented in the dissertation.

Chapter 2

Background of Stochastic Computing

This chapter introduces the background information of stochastic computing. First, we introduce how to convert a number between the binary encoding and the stochastic encoding. Then, we introduce two encoding formats in the stochastic encoding. Next, we introduce the combinational logic-based stochastic computing elements, including the synthesis approach based on combinational logic proposed by Qian et al. [3, 1]. We also introduce the prior work on the FSM-based stochastic computing elements. Finally, we briefly introduce other types of stochastic computing.

2.1 Conversion Approach

In stochastic computing, logical operations are performed on the stochastic bit streams, and signal values are encoded by probabilities. To convert a deterministic value x_d ($x_d \in [a, b]$) into a stochastic bit stream X, we can generate a random number and compare it to x_d. The pseudocode for this operation is shown as follows,

==============================

$for\ (t = 0;\ t < L;\ t++)\{$

$\quad if\ rand() < \frac{x_d - a}{b - a}$

$\quad\quad X(t) = 1;$

$\quad else$

$X(t) = 0; \}$

============================

where L is the length of the stochastic bit stream X. The function $rand()$ is used to generate a random number in the range $[0, 1]$ based on a uniform distribution. The stochastic bit stream X generated by this code has the probability

$$P(X = 1) = \frac{x_d - a}{b - a}.$$

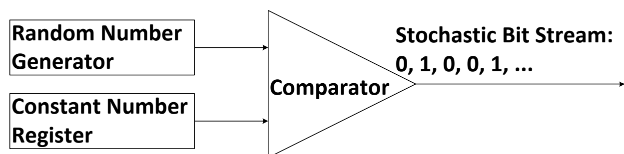


Figure 2.1: The Randomizer Unit [3].

This conversion can be implemented with the circuit shown in Fig. 2.1, which is called a *Randomizer Unit* [3]. The *Random Number Generator* is implemented with a linear feedback shift register (LFSR). By setting the constant value of the *Constant Number Register*, we can generate a stochastic bit stream with a desired probability.

By counting the number of ones in a stochastic bit stream, we can convert the stochastic bit stream back to the corresponding deterministic value as

$$x'_d = a + \frac{sum(X)}{L} \cdot (b - a) \approx x_d.$$

The error between x'_d and x_d stems from quantization effects and random fluctuations [3].

2.2 Coding Format

In the stochastic encoding, we have two coding formats: a unipolar coding format and a bipolar coding format [5]. These two coding formats are the same in essence, and can coexist in a single system.

In the unipolar coding format, a real number x in the unit interval (i.e., $0 \leq x \leq 1$) corresponds to a bit stream $X(t)$ of length L, where $t = 1, 2, ..., L$. The probability that each bit in the stream is one is

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

For example, the value $x = 0.3$ would be represented by a random stream of bits such as 0100010100, where approximately 30% of the bits are "1" and the remainder are "0."

In the bipolar coding format, the range of a real number x is extended to $-1 \leq x \leq 1$. However, the probability that each bit in the stream is one is

$$P(X = 1) = \frac{x + 1}{2}.$$

The trade-off between these two coding formats is that the bipolar format can deal with negative numbers directly while, given the same bit stream length, L, the precision of the unipolar format is twice that of the bipolar format.

2.3 Combinational Logic-based Stochastic Computing Elements

Three basic arithmetic operations – scaled addition, scaled subtraction, and multiplication – can be implemented very simply with combinational logic in the stochastic paradigm [5, 15]. Scaled addition can be implemented with a multiplexer (MUX) for both the bipolar and the unipolar coding formats. Scaled subtraction can be implemented with a MUX and a NOT gate using the bipolar coding format. Multiplication can be implemented with a two-input AND gate using the unipolar coding format, and with a two-input XNOR gate using the bipolar coding format. We will briefly explain each of these constructs in this section.

2.3.1 Scaled Addition

This operation can be implemented with a multiplexer (MUX) as shown in Fig. 2.2 for both the bipolar and the unipolar coding formats. The function of the MUX is,

$$C = (S \wedge A) \vee (\bar{S} \wedge B),$$

where $\wedge$ represents logical AND, $\vee$ represents logical OR, and $\bar{S}$ represents the negation of S.

If we assume A, B, C, and S are stochastic bit streams, and P_A, P_B, P_C, and P_S are their corresponding probabilities, we obtain:

$$P_C = P_S \cdot P_A + (1 - P_S) \cdot P_B. \tag{2.1}$$

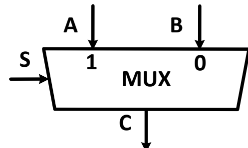


Figure 2.2: The scaled addition implemented with a MUX.

For the unipolar coding format, the values represented by the stochastic bit streams A, B, and C are $a = P_A$, $b = P_B$, and $c = P_C$, respectively. For the bipolar coding format, the values represented by the stochastic bit streams A, B, and C are $a = 2P_A - 1$, $b = 2P_B - 1$, and $c = 2P_C - 1$, respectively. Based on (2.1), for both of the two coding formats, we have

$$c = P_S \cdot a + (1 - P_S) \cdot b. \tag{2.2}$$

In (2.2), we normally set $P_S = 0.5$ to perform unbiased scaled addition.

2.3.2 Scaled subtraction

This operation can be implemented with a MUX and a NOT gate as shown in Fig. 2.3. It works only for the bipolar coding format. The function of the circuit is

$$C = (S \wedge A) \vee (\bar{S} \wedge \bar{B}).$$

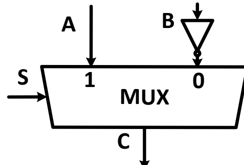


Figure 2.3: The scaled subtraction implemented with a MUX and a NOT gate for the bipolar coding format.

If we assume A, B, C, and S are stochastic bit streams, and P_A, P_B, P_C, and P_S are their corresponding probabilities, we obtain:

$$P_C = P_S \cdot P_A + (1 - P_S) \cdot (1 - P_B). \tag{2.3}$$

We define a, b, and c as the corresponding values encoded by the stochastic bit streams A, B, and C using the bipolar coding format, respectively, i.e., $a = 2P_A - 1$, $b = 2P_B - 1$, and $c = 2P_C - 1$. Then equation (2.3) can be rewritten as

$$c = P_S \cdot a - (1 - P_S) \cdot b. \tag{2.4}$$

If we set $P_S = 0.5$ in equation (2.4), we have $c = 0.5 \cdot (a - b)$.

2.3.3 Multiplication

This operation can be implemented with a two-input AND gate as shown in Fig. 2.4(a) for the unipolar coding format, and with a two-input XNOR gate as shown in Fig. 2.4(b) for the bipolar coding format. The multiplication based on the AND gate for unipolar coding format is straightforward. We will explain the one based on the XNOR gate for the bipolar coding format as follows. In Fig. 2.4(b), we have

$$C = (A \wedge B) \vee (\bar{A} \wedge \bar{B}). \tag{2.5}$$

A B C A B C

(a) (b)

Figure 2.4: Multiplication in stochastic computing. (a) AND Gate for multiplication using unipolar coding format. (b) XOR Gate for multiplication using bipolar coding format.

If we define P_A, P_B, and P_C as the probabilities of the streams A, B, and C, respectively, then based on the Boolean function of the XNOR gate, we have

$$P_C = P_A \cdot P_B + (1 - P_A) \cdot (1 - P_B). \tag{2.6}$$

If we define a, b, and c as the corresponding values encoded by the streams A, B, and C using the bipolar coding format, respectively (i.e., $a = 2P_A - 1$, $b = 2P_B - 1$, and $c = 2P_C - 1$), we can rewrite (2.6) as

$$\frac{c+1}{2} = \frac{a+1}{2} \cdot \frac{b+1}{2} + (1 - \frac{a+1}{2}) \cdot (1 - \frac{b+1}{2}). \tag{2.7}$$

Simplifying equation (2.7), we have $c = a \cdot b$.

2.3.4 Synthesis Method Using Combinational Logic

In 2011, Qian et al. [3] proposed a reconfigurable architecture for performing polynomial computation on stochastic bit streams. This architecture, as shown in Fig. 2.5, is composed of three parts: the *Randomizer*, the *ReSC Unit*, and the *De-Randomizer*. C_X and C_{Z_i} ($0 \leq i \leq n$, where n is the highest degree of the polynomial this architecture can compute) are the inputs. C_Y is the output. These values are represented using binary radix. The architecture is reconfigurable in the sense that it can be used to compute different functions $C_Y = f(C_X)$ by setting appropriate values for the coefficients C_{Z_i} ($0 \leq i \leq n$) [3].

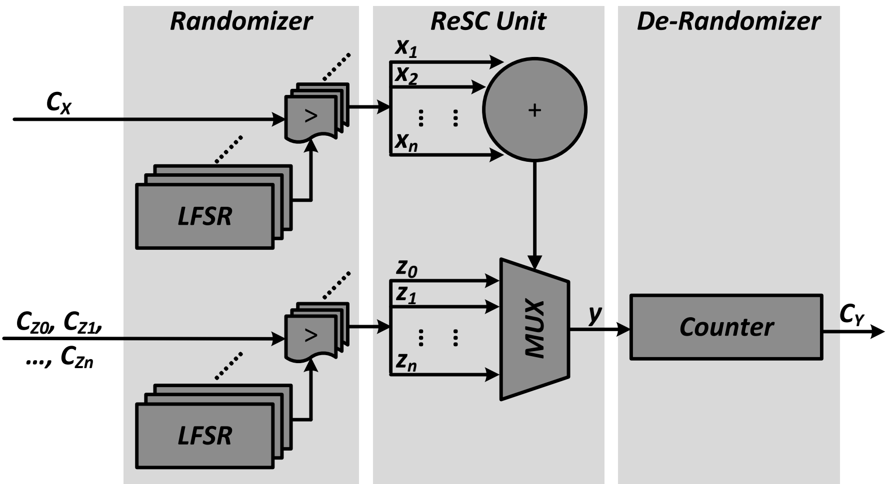


Figure 2.5: A reconfigurable stochastic computing architecture based on combinational logic (i.e., the *ReSC Unit* is implemented using an adder and a multiplexer) [3].

Their *Randomizer* uses the circuit shown in Fig. 2.1 to convert the numerical values C_X and C_{Z_i} to stochastic bit streams X_k ($1 \leq k \leq n$) and Z_i ($0 \leq i \leq n$). Their *De-Randomizer* is implemented using a counter, which converts the resulting bit streams to binary radix encoded values. The *ReSC Unit*, which processes the stochastic bit streams, is the kernel of the architecture. It is a generalized multiplexing circuit which implements Bernstein polynomials [25] with coefficients in the unit interval. This circuit can be used to approximate arbitrary continuous functions. For example, The polynomial

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

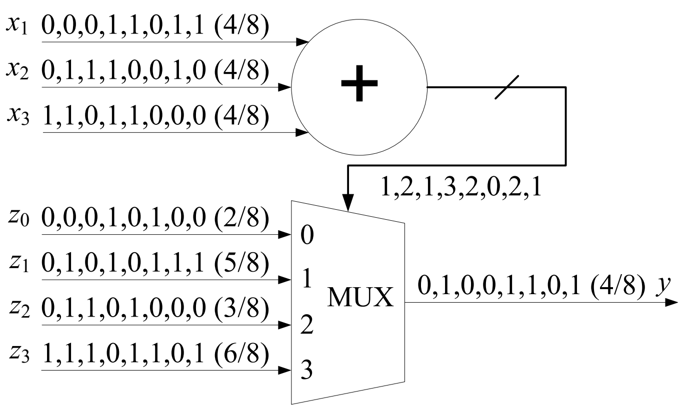


Figure 2.6: *ReSC Unit* implementing the Bernstein polynomial $f(x) = \frac{2}{8}B_{0,3}(x) + \frac{5}{8}B_{1,3}(x) + \frac{3}{8}B_{2,3}(x) + \frac{6}{8}B_{3,3}(x)$ at $x = 0.5$. Stochastic bit streams x_1, x_2 and x_3 encode the value $x = 0.5$. Stochastic bit streams z_0, z_1, z_2 and z_3 encode the corresponding Bernstein coefficients [3].

can be converted into a Bernstein polynomial of degree 3:

$$f(x) = \frac{2}{8}B_{0,3}(x) + \frac{5}{8}B_{1,3}(x) + \frac{3}{8}B_{2,3}(x) + \frac{6}{8}B_{3,3}(x), \tag{2.8}$$

where each $B_{i,3}(x)$ $(i = 0, 1, \ldots, 3)$ is a Bernstein basis polynomial of the form $B_{i,3}(x) = \binom{3}{i}x^i(1-x)^{3-i}$. A Bernstein polynomial, $B(x) = \sum_{i=0}^{n} b_i B_{i,n}(x)$, with all coefficients b_i in the unit interval, can be implemented stochastically by the *ReSC Unit* shown in Fig. 2.5. An illustration of how equation (2.8) is implemented by the *ReSC Unit* is shown in Fig. 2.6.

The *ReSC Unit* consists of an adder block and a multiplexer block. The inputs to the adder are an input set $\{x_1, \ldots, x_n\}$. The data inputs to the multiplexer are $z_0, \ldots, z_n$. The outputs of the adder are the selecting inputs to the multiplexer block. At every clock cycle, if the number of ones in the input set $\{x_1, \ldots, x_n\}$ equals i $(0 \leq i \leq n)$, then the binary number computed by the adder is i and the output of the multiplexer y is set to z_i. The inputs $x_1, \ldots, x_n$ are fed with independent stochastic bit streams $X_1, \ldots, X_n$ representing the probabilities $P(X_i = 1) = x \in [0, 1]$, for $1 \leq i \leq n$. The inputs $z_0, \ldots, z_n$ are fed with independent stochastic bit streams $Z_0, \ldots, Z_n$ representing the probabilities $P(Z_i = 1) = b_i \in [0, 1]$, for $0 \leq i \leq n$, where the b_i's are the Bernstein coefficients. The output of the circuit is a stochastic bit stream Y in which

the probability of a bit being one equals the Bernstein polynomial $B(t) = \sum_{i=0}^{n} b_i B_{i,n}(t)$ evaluated at $t = x$.

2.4 FSM-based Stochastic Computing Elements

Combinational logic can only implement polynomial functions of a specific form – namely those that map the unit interval to the unit interval [14]. Non-polynomial functions can be approximated by combinational logic, for instance with MacLaurin expansions [1]. However, highly non-linear functions such as exponentiation and tanh cannot be approximated effectively with this approach. As discussed in Section 2.3.1, the limitation stems from the fact that combinational logic can only implement scaled addition in the stochastic paradigm. The implementation of polynomials with coefficients not in the unit interval is sometimes not possible and is generally not straightforward [3].

Gaines [5] described the use of an ADaptive DIgital Element (ADDIE) for generating of arbitrary functions. The ADDIE is based on a saturating counter, that is, a counter which will not increment beyond its maximum state value or decrement below its minimum state value. In the ADDIE, the state of the counter is controlled in a closed loop fashion. The problem is that ADDIE requires the output of the counter to be converted into a stochastic bit stream in order to implement the closed loop feedback [5]. This is inefficient and hardware intensive.

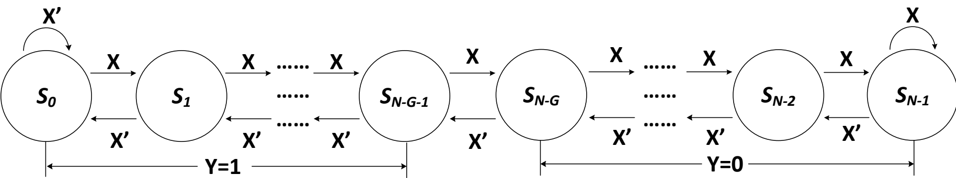


Figure 2.7: State transition diagram of the FSM-based stochastic exponentiation function.

In 2001, Brown and Card [15] presented three FSM-based constructs. The first one is called the stochastic exponentiation function, with the state transition diagram shown in Fig. 2.7. This configuration approximates an exponentiation function stochastically

as follows,

$$y \approx \begin{cases} e^{-2Gx}, & 0 \le x \le 1, \\ 1, & -1 \le x < 0, \end{cases} \tag{2.9}$$

where x is the bipolar encoding of the input bit stream X and y is the unipolar encoding of the output bit stream Y. The simulation result based on $N = 16, G = 2$ is shown in Fig. 2.8.

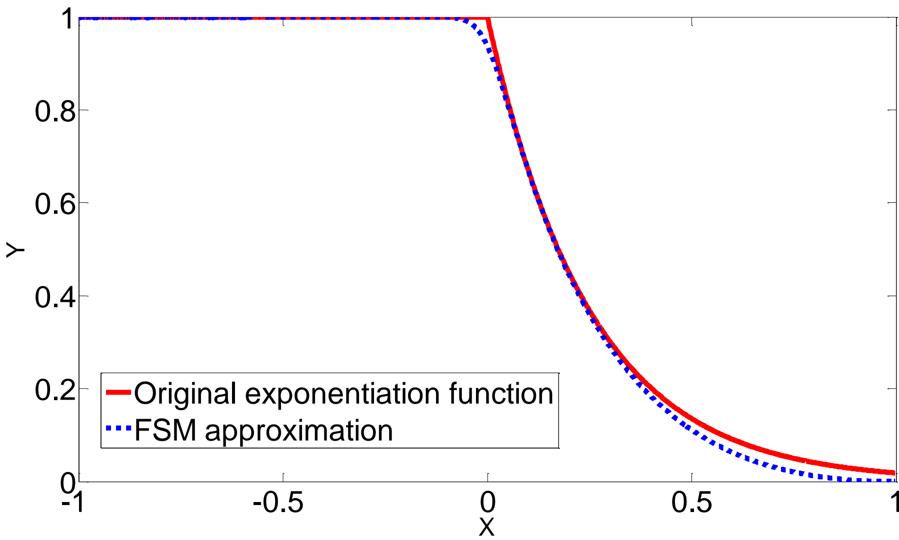


Figure 2.8: Simulation result of the FSM-based stochastic exponentiation function.

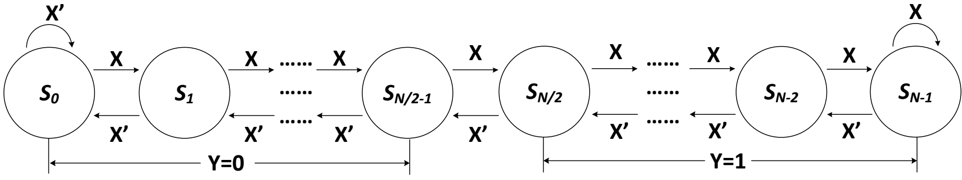


Figure 2.9: State transition diagram of the FSM-based stochastic tanh function.

The second one is called the stochastic tanh function. We show the state transition diagram in Fig. 2.9. This configuration approximates a tanh function stochastically as

follows,

$$y \approx \frac{e^{\frac{N}{2}x} - e^{-\frac{N}{2}x}}{e^{\frac{N}{2}x} + e^{-\frac{N}{2}x}}, \tag{2.10}$$

where x is the bipolar encoding of the input bit stream X and y is also the bipolar encoding of the output bit stream Y. The simulation result based on $N = 8$ is shown in Fig. 2.10.

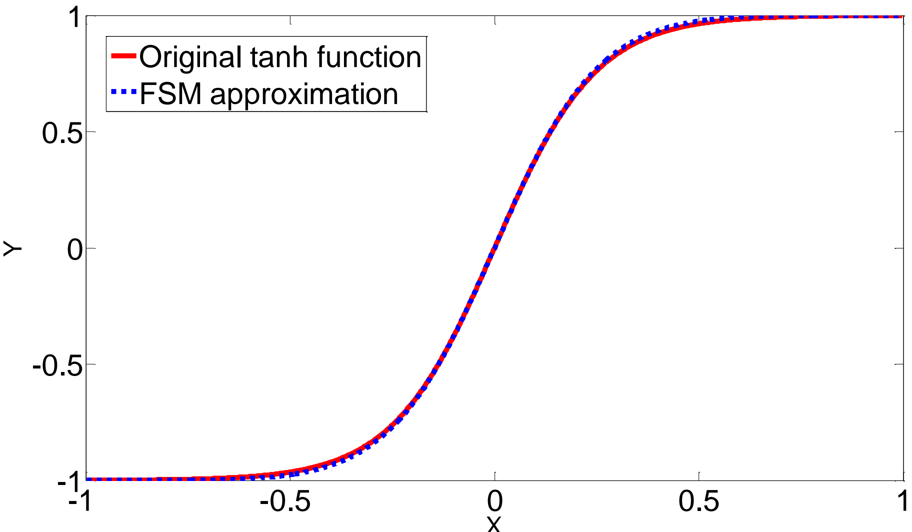


Figure 2.10: Simulation result of the FSM-based stochastic tanh function.

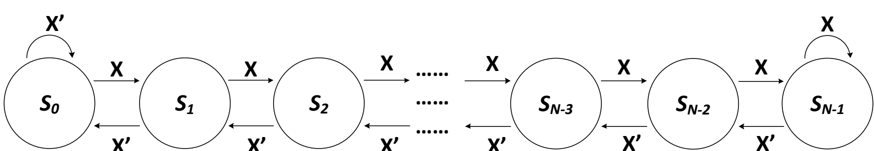


Figure 2.11: A generic linear state transition diagram.

Note that both of these FSM-based constructs use the linear state transition pattern shown in Fig. 2.11. This linear FSM is similar to Gaines' ADDIE. The difference is that this linear FSM does not use a closed loop [5, 15]; accordingly this construct is much more efficient. In the next section, we will introduce three fundamental properties of linear FSM-based stochastic computing elements.

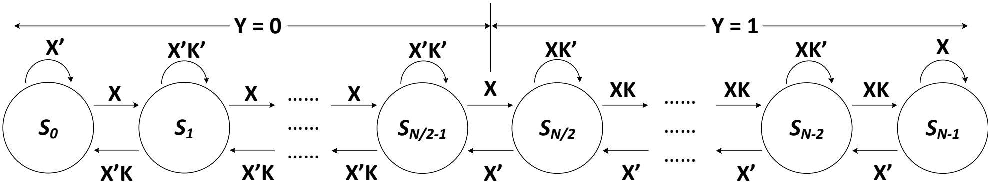


Figure 2.12: State transition diagram of the FSM implementing the stochastic linear gain function.

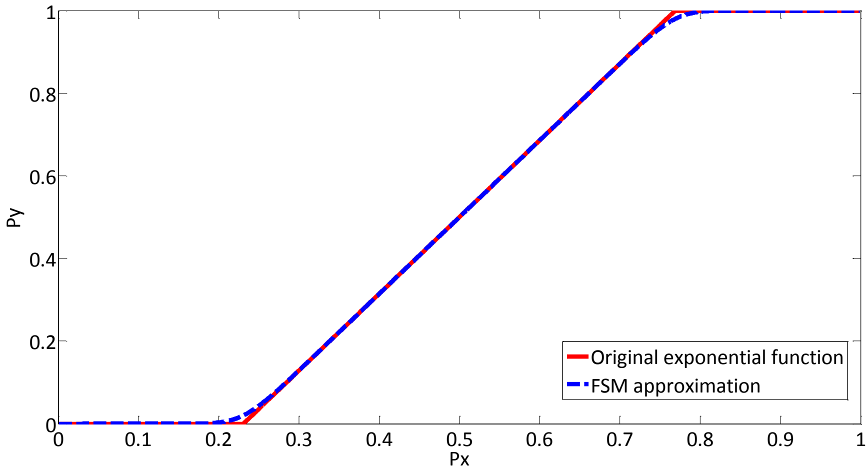


Figure 2.13: Simulation result of the FSM-based stochastic linear gain function.

The third one is called the stochastic linear function. We show the state transition diagram in Fig. 2.12. Note that this FSM has two inputs, X and K. The input K, which is also a stochastic bit stream, is used to control the linear gain. Brown and Card only empirically demonstrated that the configuration in Fig. 2.12 performs the linear gain function stochastically. However, if we define P_K as the probability that each bit in the stream K is one, the relationship between P_K and the value of the linear gain was not provided in their previous work. The configuration in Fig. 2.12 performs the following function,

$$P_Y = \begin{cases} 0, & 0 \le P_X \le \frac{P_K}{1+P_K}, \\ \\ \frac{1+P_K}{1-P_K} \cdot P_X - \frac{P_K}{1-P_K}, & \frac{P_K}{1+P_K} \le P_X \le \frac{1}{1+P_K}, \\ \\ 1, & \frac{1}{1+P_K} \le P_X \le 1. \end{cases} \tag{2.11}$$

The simulation result based on $N = 16$ and $P_K = 0.3$ is shown in Fig. 2.13.

2.5 Other Types of Stochastic Computing

Stochastic methods are normally applied to characterize uncertainty in circuit and system design [3]. These methods are quite popular and have lots of different applications. For example, statistical timing analysis is used to obtain tighter performance bounds [26, 3] and also applied in transistor sizing to maximize yield [27, 3]; Nepal et al. [28] presents a design methodology based on Markov random fields geared toward nanotechnology; Palem [29] presents a methodology based on probabilistic CMOS, with a focus on energy efficiency. Recently, researchers also proposed the design of stochastic processors [30]. The basic idea is to underdesign the digital logic circuit, such as using lower operational voltage or faster clock frequency, to allow errors in the computing. They implement fault-tolerance through software mechanisms to push it to the application layer as much as possible [3, 30]. This approach permits aggressive power reduction in the hardware design. It is particularly suitable for high-performance computing applications, such as Monte Carlo simulations, that naturally tolerate errors. However, all

these stochastic methods are still based on the deterministic binary radix computing, which is in essence different from the stochastic computing discussed in this dissertation.

Chapter 3

FSM-based Stochastic Computing Elements

This chapter presents three fundamental properties of the single input linear FSMs in stochastic computing, and analyzes five FSM-based constructs: the stochastic tanh function, the stochastic exponentiation function, the stochastic absolute value function, the stochastic exponentiation function based on an absolute value, and the stochastic linear gain function. Prior work had an empirical focus and complex constructs were validated by simulation only. The rigorous mathematical treatment of the FSM-based stochastic computing elements presented in this chapter can help us better understand the behavior of the FSM in stochastic computing and give us intuitions to implement more functions stochastically using the FSM [17, 2].

3.1 Properties of the Single Input Linear FSM

The state machine shown in Fig. 3.1 contains a set of states, $S_0, S_1, \cdots, S_{N-1}$, arranged in a linear form (i.e., a saturating counter) [15]. It has a total of N states, where N is a positive integer. We usually set $N = 2^M$, where M is also a positive integer, since the maximum number of states that can be implemented by M D-Flip-Flops (DFFs) is 2^M. X is the input of this state machine. The output Y (not shown in Fig. 3.1) of this state machine is only determined by the current state. We assume that the input X is a Bernoulli sequence (i.e., a stochastic bit stream).

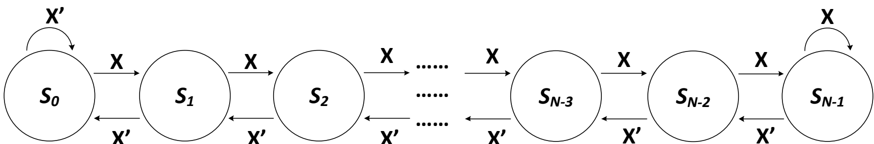


Figure 3.1: A generic linear state transition diagram.

The system can be modeled as a time-homogeneous irreducible and aperiodic Markov chain and will have one single stable hyperstate [5]. We define the probability that each bit in the input stream X is one to be P_X, the probability that each bit in the corresponding output stream Y is one to be P_Y, and in the steady state the probability that the current state is S_i $(0 \leq i \leq N-1)$ under the input probability P_X to be P_i. The individual state probability P_i in the hyperstate must sum to unity, and the probability of transitioning from state S_{i-1} to state S_i must equal the probability of transitioning from state S_i to state S_{i-1}. Thus, we have

$$P_i \cdot (1 - P_X) = P_{i-1} \cdot P_X, \tag{3.1}$$

$$\sum_{i=0}^{N-1} P_i = 1, \tag{3.2}$$

$$P_Y = \sum_{i=0}^{N-1} s_i \cdot P_i, \tag{3.3}$$

where s_i in (3.3) only has two choices of values, 0 or 1, and specifies the output Y of the system when the current state is S_i, i.e., if the current state is S_i, then the output $Y = s_i$. Based on (3.1) and (3.2), P_i can be computed as follows,

$$P_i = \frac{(\frac{P_X}{1-P_X})^i}{\sum\limits_{j=0}^{N-1} (\frac{P_X}{1-P_X})^j}. \tag{3.4}$$

Furthermore, based on different intervals of P_X, P_i can also be computed as follows,

$$P_i = \begin{cases} \frac{(\frac{P_X}{1-P_X})^i \cdot (\frac{1-2P_X}{1-P_X})}{1-(\frac{P_X}{1-P_X})^N}, & 0 \le P_X < 0.5, \\ \frac{1}{N}, & P_X = 0.5, \\ \frac{(\frac{1-P_X}{P_X})^{N-1-i} \cdot (\frac{2P_X-1}{P_X})}{1-(\frac{1-P_X}{P_X})^N}, & 0.5 < P_X \le 1. \end{cases} \tag{3.5}$$

If we define

$$t = \frac{0.5 - |P_X - 0.5|}{0.5 + |P_X - 0.5|}, \tag{3.6}$$

equation (3.5) can be rewritten as follows,

$$P_i = \begin{cases} \frac{t^i \cdot (1-t)}{1-t^N}, & 0 \le P_X < 0.5, \\ \frac{1}{N}, & P_X = 0.5, \\ \frac{t^{N-1-i} \cdot (1-t)}{1-t^N}, & 0.5 < P_X \le 1. \end{cases} \tag{3.7}$$

In this section, we introduce three fundamental properties of the linear FSMs used in stochastic computing. These properties can be proved using the above equation.

3.1.1 Property 1

P_i and P_{N-1-i} are symmetric about $P_X = 0.5$.

Proof: (1). When $P_X = 0.5$, we have

$$P_{N-1-i} = \frac{1}{N},$$

and

$$P_i = \frac{1}{N}.$$

Thus, $P_i = P_{N-1-i}$.

(2). When $0 \leq P_X < 0.5$, we have

$$P_{N-1-i} = \frac{t^{N-1-(N-1-i)} \cdot (1-t)}{1-t^N}. \tag{3.8}$$

In addition,

$$t = \frac{0.5 - |1 - P_X - 0.5|}{0.5 + |1 - P_X - 0.5|}. \tag{3.9}$$

Thus, we can rewrite (3.8) as follows,

$$P_{N-1-i} = \frac{t^{N-1-(N-1-i)} \cdot (1-t)}{1-t^N} = \frac{t^i \cdot (1-t)}{1-t^N} = P_i.$$

(3). When $0.5 < P_X \leq 1$, we have

$$P_{N-1-i} = \frac{t^{N-1-i} \cdot (1-t)}{1-t^N}. \tag{3.10}$$

Thus,

$$P_{N-1-i} = P_i.$$

As a result, we have P_i and P_{N-1-i} are symmetric about $P_X = 0.5$. ■

3.1.2 Property 2

As $N \to \infty$ (i.e., N is "large enough"),

- P_Y will be mainly determined by the configuration of the states from S_0 to $S_{N/2-1}$ when $P_X \in [0, 0.5)$;
- P_Y will be mainly determined by the configuration of the states from $S_{N/2}$ to S_{N-1} when $P_X \in (0.5, 1]$.

In other words, we can rewrite (3.3) as follows,

$$P_Y \begin{cases} \approx \sum_{i=0}^{N/2-1} s_i \cdot P_i, & 0 \leq P_X < 0.5, \\ = \sum_{i=0}^{N-1} \frac{s_i}{N}, & P_X = 0.5, \\ \approx \sum_{i=N/2}^{N-1} s_i \cdot P_i, & 0.5 < P_X \leq 1. \end{cases}$$

Proof: (1). When $0 \leq P_X < 0.5$, we have

$$\sum_{i=N/2}^{N-1} P_i = \frac{\sum_{i=N/2}^{N-1} (\frac{P_X}{1-P_X})^i \cdot (\frac{1-2P_X}{1-P_X})}{1-(\frac{P_X}{1-P_X})^N} = \frac{(\frac{P_X}{1-P_X})^{\frac{N}{2}} \cdot (1-(\frac{P_X}{1-P_X})^{\frac{N}{2}})}{1-(\frac{P_X}{1-P_X})^N}.$$

Because $P_X \in [0, 0.5)$ and we assume $N \to \infty$, $(\frac{P_X}{1-P_X})^{\frac{N}{2}} \approx 0$. Thus,

$$\sum_{i=N/2}^{N-1} P_i \approx 0. \tag{3.11}$$

In addition,

$$P_Y = \sum_{i=0}^{N-1} s_i \cdot P_i = \sum_{i=0}^{N/2-1} s_i \cdot P_i + \sum_{i=N/2}^{N-1} s_i \cdot P_i.$$

Thus

$$\sum_{i=0}^{N/2-1} s_i \cdot P_i \leq P_Y \leq \sum_{i=0}^{N/2-1} s_i \cdot P_i + \sum_{i=N/2}^{N-1} P_i.$$

Because $\sum_{i=N/2}^{N-1} P_i \approx 0$ (see (3.11)), when $P_X \in [0, 0.5)$, we have

$$P_Y \approx \sum_{i=0}^{N/2-1} s_i \cdot P_i.$$

(2). When $0.5 < P_X \leq 1$, we have

$$\sum_{i=0}^{N/2-1} P_i = \frac{\sum_{i=0}^{N/2-1} (\frac{1-P_X}{P_X})^{N-1-i} \cdot (\frac{2P_X-1}{P_X})}{1-(\frac{1-P_X}{P_X})^N} = \frac{(\frac{1-P_X}{P_X})^{\frac{N}{2}} \cdot (1-(\frac{1-P_X}{P_X})^{\frac{N}{2}})}{1-(\frac{1-P_X}{P_X})^N}.$$

Because $P_X \in (0.5, 1]$ and we assume $N \to \infty$, $(\frac{1-P_X}{P_X})^{\frac{N}{2}} \approx 0$. Thus, we have

$$\sum_{i=0}^{N/2-1} P_i \approx 0. \tag{3.12}$$

In addition,

$$P_Y = \sum_{i=0}^{N-1} s_i \cdot P_i = \sum_{i=0}^{N/2-1} s_i \cdot P_i + \sum_{i=N/2}^{N-1} s_i \cdot P_i.$$

Thus,

$$\sum_{i=N/2}^{N-1} s_i \cdot P_i \le P_Y \le \sum_{i=0}^{N/2-1} P_i + \sum_{i=N/2}^{N-1} s_i \cdot P_i.$$

Because $\sum_{i=0}^{N/2-1} P_i \approx 0$ (see (3.12)), when $P_X \in (0.5, 1]$, we have

$$P_Y \approx \sum_{i=N/2}^{N-1} s_i \cdot P_i.$$

(3). When $P_X = 0.5$, based on (14), $P_i = \frac{1}{N}$. Thus, we have

$$P_Y = \sum_{i=0}^{N-1} s_i \cdot P_i = \sum_{i=0}^{N-1} \frac{s_i}{N}. \quad \blacksquare$$

3.1.3 Property 3

For the configuration

$$P_Y = \sum_{i=0}^{N-1} s_i \cdot P_i,$$

- if we set $s_i = s_{N-1-i}$, for $i = 0, 1, \cdots, \frac{N}{2} - 1$, P_Y will be symmetric about the line $P_X = 0.5$;
- if we set $s_i = 1 - s_{N-1-i}$, for $i = 0, 1, \cdots, \frac{N}{2} - 1$, P_Y will be symmetric about the point $(P_X, P_Y) = (0.5, 0.5)$.

In other words,

- if $s_i = s_{N-1-i}$, for $i = 0, 1, \cdots, \frac{N}{2} - 1$, $P_Y(P_X) = P_Y(1 - P_X)$;
- if $s_i = 1 - s_{N-1-i}$, for $i = 0, 1, \cdots, \frac{N}{2} - 1$, $P_Y(P_X) = 1 - P_Y(1 - P_X)$.

Proof: (1). When $s_i = s_{N-1-i}$, we have

$$P_Y(1 - P_X) = \sum_{i=0}^{N-1} s_i \cdot P_i.$$

Based on **Property 1**, we have,

$$\sum_{i=0}^{N-1} s_i \cdot P_i = \sum_{i=0}^{N-1} s_i \cdot P_{N-1-i}.$$

We define $j = N - 1 - i$, thus

$$\sum_{i=0}^{N-1} s_i \cdot P_{N-1-i} = \sum_{j=0}^{N-1} s_{N-1-j} \cdot P_j.$$

Because $s_j = s_{N-1-j}$, we have

$$\sum_{j=0}^{N-1} s_{N-1-j} \cdot P_j = \sum_{j=0}^{N-1} s_j \cdot P_j = P_Y(P_X).$$

(2). When $s_i = 1 - s_{N-1-i}$, we have

$$P_Y(1 - P_X) = \sum_{i=0}^{N-1} s_i \cdot P_i.$$

Based on **Property 1**, we have,

$$\sum_{i=0}^{N-1} s_i \cdot P_i = \sum_{i=0}^{N-1} s_i \cdot P_{N-1-i}.$$

We define $j = N - 1 - i$, thus,

$$\sum_{i=0}^{N-1} s_i \cdot P_{N-1-i} = \sum_{j=0}^{N-1} s_{N-1-j} \cdot P_j.$$

Because $s_j = 1 - s_{N-1-j}$, we have

$$\sum_{j=0}^{N-1} s_{N-1-j} \cdot P_j = \sum_{j=0}^{N-1} (1 - s_j) \cdot P_j = \sum_{j=0}^{N-1} P_j - \sum_{j=0}^{N-1} s_j \cdot P_j$$

$$= 1 - \sum_{j=0}^{N-1} s_j \cdot P_j = 1 - P_Y(P_X). \quad \blacksquare$$

Based on these properties, we present the FSM-based constructs for computing complex functions in the next section.

3.2 The FSM-based Stochastic Computing Elements

In this section, we present and analyze the FSM-based stochastic computing elements for the tanh function, the exponentiation function, the exponentiation function based on an absolute value, the absolute value function, and the linear gain function. The constructs for the stochastic tanh function, the stochastic exponentiation function, and the stochastic linear gain function were presented by Brown and Card [15]; they provided no proof of correctness, only empirical validation. The constructs for the stochastic exponentiation function based on an absolute value and the stochastic absolute value function are new. We prove the correctness of all five constructs.

3.2.1 Stochastic Tanh Function

Brown and Card [15] proposed the stochastic tanh function to implement sigmoid non-linear mapping with stochastic computing. They set s_i in (3.3) as follows,

$$s_i = \begin{cases} 0, & 0 \leq i \leq \frac{N}{2} - 1, \\ 1, & \frac{N}{2} \leq i \leq N - 1. \end{cases} \tag{3.13}$$

$$y \approx \frac{e^{\frac{N}{2}x} - e^{-\frac{N}{2}x}}{e^{\frac{N}{2}x} + e^{-\frac{N}{2}x}}, \tag{3.14}$$

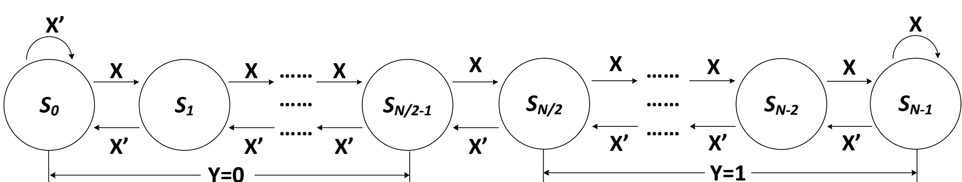


Figure 3.2: State transition diagram of the FSM-based stochastic tanh function.

This configuration is shown in Fig. 3.2, and equation (3.14) is the corresponding approximate transfer function. We will prove (3.14) based on the configuration in (3.13) as follows.

Proof: Based on the configuration in (3.13), we have

$$P_Y = \sum_{i=N/2}^{N-1} P_i. \tag{3.15}$$

If we substitute P_i in (3.15) with (3.4), we have

$$\begin{aligned} P_Y &= \frac{\sum_{i=N/2}^{N-1} (\frac{P_X}{1-P_X})^i}{\sum_{k=0}^{N-1} (\frac{P_X}{1-P_X})^k} = \frac{(\frac{P_X}{1-P_X})^{\frac{N}{2}} - (\frac{P_X}{1-P_X})^N}{1 - (\frac{P_X}{1-P_X})^N} \\ &= \frac{(\frac{P_X}{1-P_X})^{\frac{N}{2}} \cdot (1 - (\frac{P_X}{1-P_X})^{\frac{N}{2}})}{(1 + (\frac{P_X}{1-P_X})^{\frac{N}{2}}) \cdot (1 - (\frac{P_X}{1-P_X})^{\frac{N}{2}})} = \frac{(\frac{P_X}{1-P_X})^{\frac{N}{2}}}{1 + (\frac{P_X}{1-P_X})^{\frac{N}{2}}}. \end{aligned} \tag{3.16}$$

If we substitute P_X and P_Y in (3.16) with their corresponding bipolar coding format x and y, we have

$$\frac{y+1}{2} = \frac{(\frac{1+x}{1-x})^{\frac{N}{2}}}{1 + (\frac{1+x}{1-x})^{\frac{N}{2}}}. \tag{3.17}$$

Simplifying (3.17), we have,

$$y = \frac{(\frac{1+x}{1-x})^{\frac{N}{2}} - 1}{(\frac{1+x}{1-x})^{\frac{N}{2}} + 1}. \tag{3.18}$$

By using Taylor's expansion, we have

$$1 + x \approx e^x,$$

$$1 - x \approx e^{-x}.$$

Thus, we can rewrite (3.18) as follows,

$$y = \frac{(e^{2x})^{\frac{N}{2}} - 1}{(e^{2x})^{\frac{N}{2}} + 1} = \frac{e^{\frac{N}{2}x} - e^{-\frac{N}{2}x}}{e^{\frac{N}{2}x} + e^{-\frac{N}{2}x}}. \quad \blacksquare$$

Note that as $N \to \infty$, the stochastic tanh function will approximate the following threshold function,

$$y \approx \begin{cases} -1, & -1 \le x < 0, \\ \\ 1, & 0 < x \le 1, \end{cases}$$

or

$$P_Y \approx \begin{cases} 0, & 0 \le P_X < 0.5, \\ \\ 1, & 0.5 < P_X \le 1. \end{cases}$$

Proof: (1). When $P_X < 0.5$, if we substitute P_i in (3.15) with (3.5), we have

$$P_Y = \sum_{i=N/2}^{N-1} P_i = (\frac{P_X}{1-P_X})^{\frac{N}{2}}(1-(\frac{P_X}{1-P_X})^{\frac{N}{2}}). \tag{3.19}$$

Because $P_X < 0.5$, $0 < \frac{P_X}{1-P_X} < 1$. Equation (3.19) will approximate to 0 as $N \to \infty$.

(2). When $P_X > 0.5$, if we substitute P_i in (3.15) with (3.5), we have

$$P_Y = \sum_{i=N/2}^{N-1} P_i = \frac{\frac{2P_X-1}{P_X}(1-(\frac{1-P_X}{P_X})^{\frac{N}{2}})}{1-\frac{1-P_X}{P_X}}. \tag{3.20}$$

Because $P_X > 0.5$, $0 < \frac{1-P_X}{P_X} < 1$. As $N \to \infty$, (3.20) will approximate to

$$P_Y \approx \frac{\frac{2P_X-1}{P_X}}{1-\frac{1-P_X}{P_X}} = 1.$$

(3). When $P_X = 0.5$, if we substitute P_i in (3.15) with (3.5), we have

$$P_Y = \sum_{i=N/2}^{N-1} \frac{1}{N} = \frac{1}{2}. \quad \blacksquare$$

By combining the stochastic tanh function with a scaled subtraction, we can build the stochastic comparator as shown in Fig. 3.3, where A, B, S, X, and Y stand for stochastic bit streams. We define P_A, P_B, P_S, P_X, and P_Y as their corresponding probabilities. If we set $P_S = 0.5$, the function of the circuit shown in Fig. 3.3 is: *if* $(P_A < P_B)$ *then* $P_Y \approx 0$; *else* $P_Y \approx 1$.

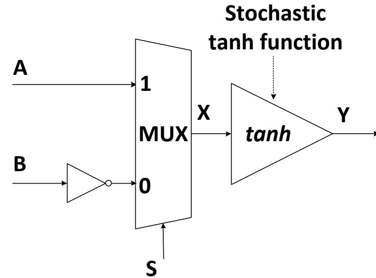


Figure 3.3: The stochastic comparator.

3.2.2 Stochastic Exponentiation Function

The stochastic exponentiation function is configured by setting the parameters s_i in (3.3) as follows,

$$s_i = \begin{cases} 1, & 0 \leq i \leq N - G - 1, \\ 0, & N - G \leq i \leq N - 1, \end{cases} \tag{3.21}$$

where G is a positive integer and $G \ll N$.

$$y \approx \begin{cases} e^{-2Gx}, & 0 \leq x \leq 1, \\ 1, & -1 \leq x < 0, \end{cases} \tag{3.22}$$

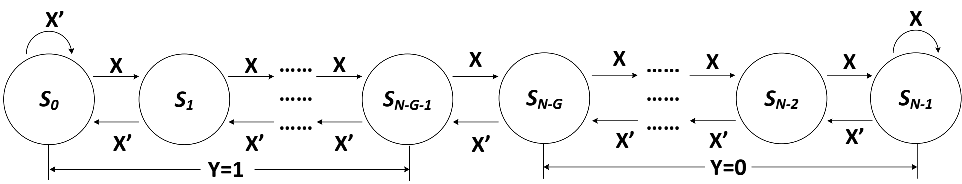


Figure 3.4: State transition diagram of the FSM-based stochastic exponentiation function.

This configuration is shown in Fig. 3.4, and equation (3.22) is the corresponding approximate transfer function. We will prove (3.22) as follows.

Proof: Based on the configuration in (3.21), we have

$$P_Y = \sum_{i=0}^{N-G-1} P_i. \tag{3.23}$$

If we substitute P_i in (3.23) with (3.5), we have

$$P_Y = \begin{cases} \sum_{i=0}^{N-G-1} \frac{t^i \cdot (1-t)}{1-t^N}, & 0 \leq P_X < 0.5, \\ \frac{N-G}{N}, & P_X = 0.5, \\ \sum_{i=0}^{N-G-1} \frac{t^{N-1-i} \cdot (1-t)}{1-t^N}, & 0.5 < P_X \leq 1. \end{cases} \tag{3.24}$$

Based on Property 2, and because $G \ll N$, we can rewrite (3.24) as follows,

$$P_Y \approx \begin{cases} \sum_{i=0}^{N/2-1} \frac{t^i \cdot (1-t)}{1-t^N}, & 0 < P_X \leq 0.5, \\ 1, & P_X = 0.5, \\ \sum_{i=N/2}^{N-G-1} \frac{t^{N-1-i} \cdot (1-t)}{1-t^N}, & 0.5 < P_X \leq 1. \end{cases}$$

(1). When $P_X < 0.5$, $t = \frac{P_X}{1-P_X} < 1$. When N is large enough, we have

$$\sum_{i=0}^{N/2-1} \frac{t^i \cdot (1-t)}{1-t^N} = \frac{1-t^{N/2}}{1-t^N} \approx 1.$$

(2). When $P_X > 0.5$, $t = \frac{1-P_X}{P_X}$. When N is large enough, we have

$$\sum_{i=N/2}^{N-G-1} \frac{t^{N-1-i} \cdot (1-t)}{1-t^N} = \frac{t^G - t^{N/2}}{1-t^N} \approx t^G.$$

Because $x = 2P_X - 1$ and $P_X > 0.5$, we can rewrite t in (3.6) as follows,

$$t = \frac{1-|x|}{1+|x|} = \frac{1-x}{1+x}.$$

By using Taylor's expansion, we have

$$1+x \approx e^{x}, \quad 1-x \approx e^{-x},$$

Thus,

$$t = \frac{1-x}{1+x} \approx \frac{e^{-x}}{e^{x}} = e^{-2x},$$

and

$$t^{G} \approx e^{-2Gx}. \quad \blacksquare$$

3.2.3 Stochastic Exponentiation Function base on an Absolute Value

This stochastic computing element can be implemented by setting s_i as follows,

$$s_i = \begin{cases} 1, & G \leq i \leq N-G-1, \\ 0, & i < G \text{ or } i > N-G-1. \end{cases} \tag{3.25}$$

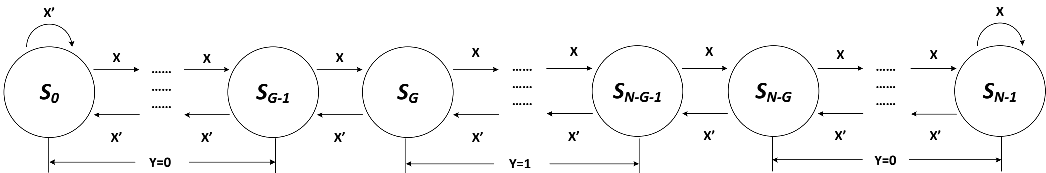


Figure 3.5: State transition diagram of the stochastic exponentiation function base on an absolute value.

We also show this configuration in Fig. 3.5. The approximate transfer function in this configuration is

$$P_Y = e^{-2G|x|}, \tag{3.26}$$

where x is the bipolar coding of P_X, i.e., $x = 2P_X - 1$.

Proof: (1). When $P_X > 0.5$ (i.e., $x > 0$), the configuration from $s_{N/2}$ to s_{N-1} is the same as in (3.21). Based on Property 2, the configuration in (3.25) has the same approximate function as in (3.21) when $P_X > 0.5$. Thus, we have

$$P_Y = e^{-2Gx}. \tag{3.27}$$

(2). When $P_X < 0.5$ (i.e., $x < 0$), because $s_i = s_{N-1-i}$, based on Property 3, P_Y is symmetric about $P_X = 0.5$, i.e., P_Y is symmetric about $x = 0$. Thus, we have

$$P_Y = e^{2Gx}. \tag{3.28}$$

As a result, based on (3.27) and (3.28), we obtain (3.26). Note that, since $G \ll N$, when $P_X = 0.5$ (i.e., $x = 0$), based on (3.7), $P_Y = (N - 2G + 1)/N \approx 1$. ■

3.2.4 Stochastic Absolute Value Function

This stochastic computing element can be implemented by setting s_i as follows,

- When $0 \leq i \leq N/2 - 1$,

$$s_i = \begin{cases} 1, & i \text{ is even}, \\ 0, & i \text{ is odd}; \end{cases} \tag{3.29}$$

- When $N/2 \leq i \leq N - 1$,

$$s_i = \begin{cases} 1, & i \text{ is odd}, \\ 0, & i \text{ is even}. \end{cases} \tag{3.30}$$

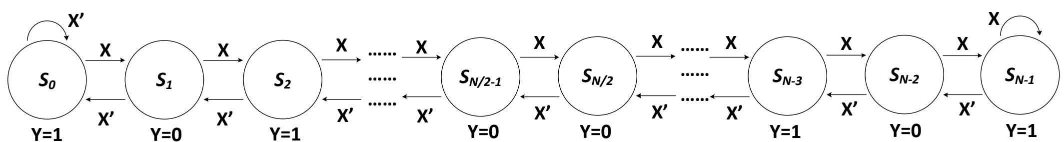


Figure 3.6: State transition diagram of the stochastic absolute value function.

We also show the configuration in Fig. 3.6. If we define x as the bipolar coding of P_X and y as the bipolar coding of P_Y. The approximate transfer function in this configuration is

$$y = |x|. \tag{3.31}$$

Proof: Based on the configuration in (3.29) and (3.30), we have

$$P_Y = \sum_{i=0}^{N/4-1} P_{2i} + \sum_{i=N/4}^{N/2-1} P_{2i+1}. \tag{3.32}$$

(1). When $P_X < 0.5$, based on Property 2, we have

$$P_Y \approx \sum_{i=0}^{N/4-1} P_{2i}. \tag{3.33}$$

If we substitute P_{2i} in (3.33) with (3.7), we have

$$\sum_{i=0}^{N/4-1} P_{2i} = \sum_{i=0}^{N/4-1} \frac{t^{2i} \cdot (1-t)}{1-t^N} = \frac{1-t^{N/2}}{1-t^N} \cdot \frac{1-t}{1-t^2} \approx \frac{1-t}{1-t^2} = \frac{1}{1+t}. \tag{3.34}$$

If we substitute t in (3.34) with (3.6), we have

$$\frac{1}{1+t} = 1 - P_X.$$

i.e., $P_Y \approx 1 - P_X$. Thus,

$$y = 2P_Y - 1 = 2(1 - P_X) - 1 = -x.$$

(2). When $P_X > 0.5$, because $s_i = s_{N-1-i}$, based on Property 3, P_Y is symmetric about $P_X = 0.5$, i.e., $P_Y \approx 1 - (1 - P_X) = P_X$. Thus,

$$y = 2P_Y - 1 = 2P_X - 1 = x.$$

(3). When $P_X = 0.5$, based on (3.5), $P_{i(P_X)} = \frac{1}{N}$. If we substitute $P_{i(P_X)}$ in (3.32) with $\frac{1}{N}$, we have

$$P_Y = \sum_{i=0}^{N/4-1} \frac{1}{N} + \sum_{i=N/4}^{N/2-1} \frac{1}{N} = 0.5,$$

i.e., $y = 0$.

As a result, we have

$$y = \begin{cases} -x, & -1 \le x < 0, \\ 0, & x = 0, \\ x, & 0 < x \le 1, \end{cases}$$

i.e., $y = |x|$. ■

3.2.5 Stochastic Linear Gain Function

In this subsection, we prove the stochastic linear gain function proposed by Brown and Card [15]. Based on the state transition diagram shown in Fig. 3.7, if the inputs X and K are stochastic bit streams with fixed probabilities, then the random state transition shown in Fig. 3.7 will eventually reach an equilibrium state, where the probability of transitioning from state S_i to its adjacent state S_{i+1} equals the probability of transitioning from state S_{i+1} to state S_i. Thus, we have

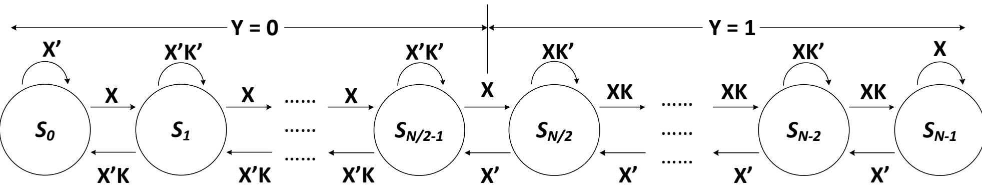


Figure 3.7: State transition diagram of the FSM implementing the stochastic linear gain function.

$$\begin{cases} P_i \cdot P_X = P_{i+1} \cdot (1 - P_X) \cdot P_K, \quad 0 \leq i \leq \frac{N}{2} - 2, \\ \\ P_{\frac{N}{2}-1} \cdot P_X = P_{\frac{N}{2}} \cdot (1 - P_X), \\ \\ P_i \cdot P_X \cdot P_K = P_{i+1} \cdot (1 - P_X), \quad \frac{N}{2} \leq i \leq N - 2. \end{cases} \tag{3.35}$$

where P_i is the probability that the current state is S_i in the equilibrium state, P_X is the probability of ones in the input bit stream X, and P_K is the probability of ones in the input bit stream K. Note that $0 < P_X < 1$ and $0 < P_K < 1$. When P_X (or P_K) = 0 or 1, we cannot use equation (3.35) to analyze the state transition diagram shown in Fig. 3.7. However, in these cases, it is easy to prove the results in equation (3.36)

based on the state transition diagram shown in Fig. 3.7.

$$P_Y = \begin{cases} 0, & 0 \le P_X \le \frac{P_K}{1+P_K}, \\ \\ \frac{1+P_K}{1-P_K} \cdot P_X - \frac{P_K}{1-P_K}, & \frac{P_K}{1+P_K} \le P_X \le \frac{1}{1+P_K}, \\ \\ 1, & \frac{1}{1+P_K} \le P_X \le 1. \end{cases} \tag{3.36}$$

Because the individual state probabilities, P_i, must sum to unity over all S_i, giving

$$\sum_{i=0}^{N-1} P_i = 1. \tag{3.37}$$

We define A and B as follows,

$$A = \sum_{i=\frac{N}{2}}^{N-1} t^i \cdot P_K^{i+1-N}, \quad B = \sum_{i=0}^{\frac{N}{2}-1} t^i \cdot P_K^{-i}. \tag{3.38}$$

Based on (3.35) and (3.37), we have

$$P_i = \begin{cases} \frac{t^i \cdot P_K^{-i}}{A+B}, & 0 \le i \le \frac{N}{2} - 1, \\ \\ \frac{t^i \cdot P_K^{i+1-N}}{A+B}, & \frac{N}{2} \le i \le N - 1. \end{cases} \tag{3.39}$$

Based on the state transition diagram shown in Fig. 3.7, we can represent P_Y, the probability of ones in the output bit stream Y, in terms of P_i as follows,

$$P_Y = \sum_{i=\frac{N}{2}}^{N-1} P_i. \tag{3.40}$$

If we substitute P_i from (3.39), we can rewrite (3.40) as $P_Y = \frac{A}{A+B}$. Note that based on equation 3.38, we can compute A and B using the formula of the sum of geometric

sequence as follows,

$$A = \begin{cases} \frac{t^{\frac{N}{2}} \cdot P_K^{1-\frac{N}{2}} \cdot (1 - t^{\frac{N}{2}} \cdot P_K^{\frac{N}{2}})}{1 - t \cdot P_K}, & t \cdot P_K > 1, \\ \frac{N}{2} \cdot P_K^{1-N}, & t \cdot P_K = 1, \\ \frac{t^{\frac{N}{2}} \cdot P_K^{1-\frac{N}{2}}}{1 - t \cdot P_K}, & t \cdot P_K < 1, \end{cases} \tag{3.41}$$

$$B = \begin{cases} \frac{1 - t^{\frac{N}{2}} \cdot P_K^{-\frac{N}{2}}}{1 - t \cdot P_K^{-1}}, & \frac{t}{P_K} > 1, \\ \frac{N}{2}, & \frac{t}{P_K} = 1, \\ \frac{1}{1 - t \cdot P_K^{-1}}, & \frac{t}{P_K} < 1. \end{cases} \tag{3.42}$$

Next we prove equation (3.36) based on different intervals of P_X.

1. $0 < P_X \leq \frac{P_K}{1+P_K}$

   Because

   $$t = \frac{P_X}{1 - P_X},$$

   we have

   $$P_X = \frac{t}{1 + t}.$$

   Since $P_X \leq \frac{P_K}{1+P_K}$, we have

   $$\frac{t}{1 + t} \leq \frac{P_K}{1 + P_K},$$

   i.e., $\frac{t}{P_K} \leq 1$, and $t \cdot P_K = \frac{t}{P_K} \cdot P_K^2 \leq P_K^2 < 1$.

   Thus, based on equation (3.41) and (3.42),

$$A = \frac{t^{\frac{N}{2}} \cdot P_K^{1-\frac{N}{2}}}{1 - t \cdot P_K},$$

$$B = \begin{cases} \frac{N}{2}, & \frac{t}{P_K} = 1, \\ \\ \frac{1}{1-t \cdot P_K^{-1}}, & \frac{t}{P_K} < 1. \end{cases}$$

If $\frac{t}{P_K} < 1$, $\lim_{N\to\infty} \frac{A}{B} = \frac{P_K - t}{1 - t \cdot P_K} \cdot \lim_{N\to\infty} \left(\frac{t}{P_K}\right)^{\frac{N}{2}} = 0.$

If $\frac{t}{P_K} = 1$, $\lim_{N\to\infty} \frac{A}{B} = \frac{2P_K}{(1 - t \cdot P_K)N} = 0.$

Thus, $\lim_{N\to\infty} P_Y = \lim_{N\to\infty} \frac{\frac{A}{B}}{\frac{A}{B} + 1} = 0.$

2. $\frac{P_K}{1+P_K} < P_X < \frac{1}{1+P_K}$

   Because $\frac{P_K}{1+P_K} < P_X$, we have $\frac{t}{P_K} > 1$.

   Because $P_X < \frac{1}{1+P_K}$, we have $t \cdot P_K < 1$.

   Thus,

$$A = \frac{t^{\frac{N}{2}} \cdot P_K^{1-\frac{N}{2}}}{1 - t \cdot P_K}, \quad B = \frac{1 - t^{\frac{N}{2}} \cdot P_K^{-\frac{N}{2}}}{1 - t \cdot P_K^{-1}}.$$

$$\lim_{N\to\infty} \frac{B}{A} = \frac{\lim_{N\to\infty} (\frac{t}{P_K})^{\frac{N}{2}} - 1}{\lim_{N\to\infty} (\frac{t}{P_K})^{\frac{N}{2}}} \cdot \frac{1 - t \cdot P_K}{t - P_K} = \frac{1 - t \cdot P_K}{t - P_K}.$$

   Thus,

$$\lim_{N\to\infty} P_Y = \lim_{N\to\infty} \frac{1}{1 + \frac{B}{A}} = \frac{1}{1 + \frac{1-t \cdot P_K}{t-P_K}} = \frac{1 + P_K}{1 - P_K} \cdot P_X - \frac{P_K}{1 - P_K}.$$

3. $\frac{1}{1+P_K} \le P_X < 1$

   In this case, we have $t \cdot P_K \ge 1$, and $\frac{t}{P_K} \ge P_K^{-2} > 1$. Thus,

$$A = \begin{cases} \frac{t^{\frac{N}{2}} \cdot P_K^{1-\frac{N}{2}} \cdot (1 - t^{\frac{N}{2}} \cdot P_K^{\frac{N}{2}})}{1 - t \cdot P_K}, & t \cdot P_K > 1, \\ \\ \frac{N}{2} \cdot P_K^{1-N}, & t \cdot P_K = 1. \end{cases}$$

$$B = \frac{1 - t^{\frac{N}{2}} \cdot P_K^{-\frac{N}{2}}}{1 - t \cdot P_K^{-1}}.$$

If $t \cdot P_K > 1$,

$$\lim_{N\to\infty} \frac{B}{A} = \frac{1 - t \cdot P_K}{P_K - t} \cdot \lim_{N\to\infty} \frac{1 - (\frac{t}{P_K})^{\frac{N}{2}}}{(1 - (t \cdot P_K)^{\frac{N}{2}}) \cdot (\frac{t}{P_K})^{\frac{N}{2}}} = \frac{1 - t \cdot P_K}{P_K - t} \cdot 0 = 0.$$

If $t \cdot P_K = 1$,

$$\lim_{N\to\infty} \frac{B}{A} = \lim_{N\to\infty} \frac{P_K^N - (t \cdot P_K)^{\frac{N}{2}}}{(P_K - t) \cdot \frac{N}{2}} = \lim_{N\to\infty} \frac{P_K^N - 1}{(P_K - t) \cdot \frac{N}{2}} = 0.$$

Thus,

$$\lim_{N\to\infty} P_Y = \lim_{N\to\infty} \frac{1}{1 + \frac{B}{A}} = 1.$$

Based on the above discussion, equation (3.36) has been proved. The simulation result can be found in [15].

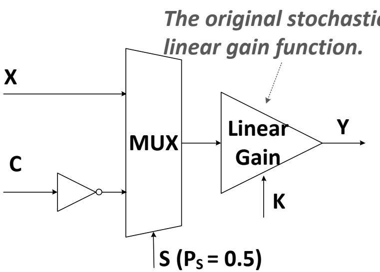


Figure 3.8: A two-parameter stochastic linear gain function.

In addition, we can build a two-parameter linear gain function, which can control both the slope and the position of the linear gain. The circuit is shown in Fig. 3.8. If we define

$$g_l = max\{0, \quad P_C - \frac{1-P_K}{1+P_K}\},$$
$$g_r = min\{1, \quad P_C + \frac{1-P_K}{1+P_K}\},$$

where P_C and P_K are the probabilities of ones in the stochastic bit streams C and K, the circuit in Fig. 3.8 performs the following function,

$$P_Y = \begin{cases} 0, & 0 \le P_X < g_l, \\ \frac{1}{2} \cdot \frac{1+P_K}{1-P_K} \cdot (P_X - P_C) + \frac{1}{2}, & g_l \le P_X \le g_r, \\ 1, & g_r < P_X \le 1, \end{cases}$$

where P_X and P_Y are the probabilities of ones in the stochastic bit streams X and Y. In Fig. 3.8, note that the input of the original stochastic linear gain function is the output of the scaled subtraction. Thus, the above equation can be proved based on this relationship. It can be seen that the center of this two-parameter stochastic linear gain function is moved to P_C. However, the new gain changes to one half of the original gain: in the original stochastic linear gain function, the gain is $\frac{1+P_K}{1-P_K}$; in the new one, the gain is $\frac{1}{2} \cdot \frac{1+P_K}{1-P_K}$.

3.3 Error Analysis

By its nature, the paradigm of computing on stochastic bit streams introduces errors and uncertainty. As discussed in [3], there are three primary sources of errors:

1. the error due to the approximation of the desired function (e_a),
2. the quantization error (e_q), and
3. the error due to random fluctuations (e_r).

We analyze each of these three errors for the FSM-based stochastic computing elements in this section.

3.3.1 Error Due to the Function Approximation

Based on the analysis from Section 3.2, it can be seen that the approximation error of the FSM-based stochastic computing elements compared to the desired functions depends on the number of states. The more states we use to implement the FSM, the less approximation error we have. Brown and Card [15] showed simulation results with 8, 16, and 32-state in the FSMs for the stochastic tanh function, the stochastic exponentiation function, and the stochastic linear gain function. Here we show the simulation results using different numbers of states for the two new FSM-based stochastic computing elements, namely stochastic exponentiation function based on an absolute value and the stochastic absolute value function.

The approximation results of the stochastic exponentiation function based on an absolute value using 8, 16, 32, and 64-state FSMs are shown in Fig. 3.9. It can be seen that the main distortion is located at $x = 0$ (i.e., $P_X = 0.5$). By using more states, this distortion can be reduced.

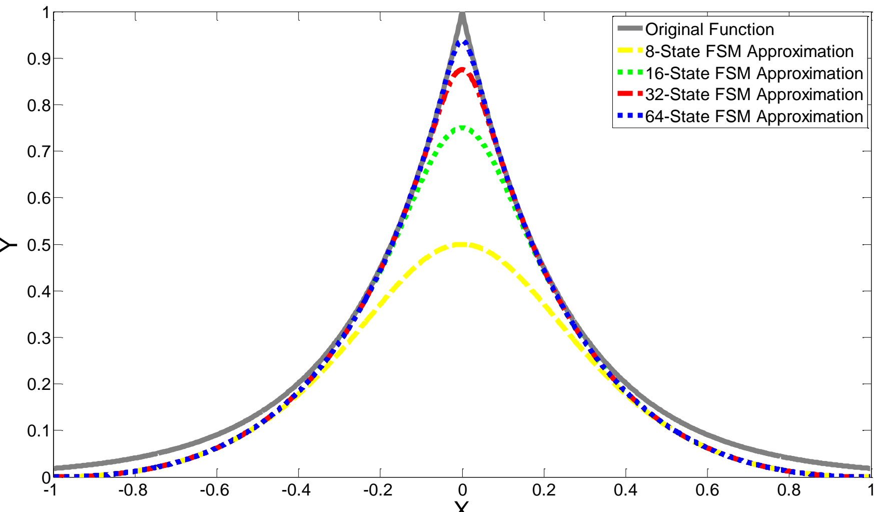


Figure 3.9: Simulation results of the stochastic exponentiation function based on an absolute value ($G = 2$ in (3.25)).

The approximation results of the stochastic absolute value function using 8, 16, 32, and 64-state linear FSMs are shown in Fig. 3.10. It can be seen that the main distortion

is again located at $x = 0$ (i.e., $P_X = 0.5$). By using more states, this distortion again can be reduced.

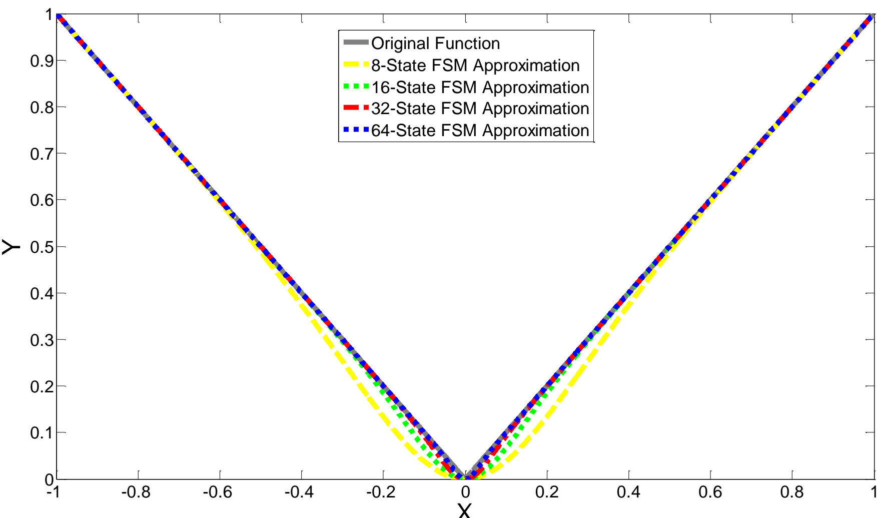


Figure 3.10: Simulation results of the stochastic absolute value function.

We list the approximation errors for each of the FSM-based stochastic computing elements versus different numbers of states in Table 3.1. The approximation error is computed using a standard least squares (2-norm) distance:

$$e_a = \int_0^1 (T(P_X) - \sum_{i=0}^{N-1} s_i \cdot P_i)^2 \cdot dP_X,$$

where $T(P_X)$ is the original function, and $\sum_{i=0}^{N-1} s_i \cdot P_i$ is the corresponding FSM-based approximation.

For many fault-tolerant applications, 10^{-3} would be considered an acceptable error rate [3]. As a result, in our subsequent experiments we choose the 8-state FSM to approximate the stochastic tanh function, the 16-state FSM to approximate the stochastic exponentiation function, the 32-state FSM to approximate the stochastic exponentiation function based on an absolute value, the 8-state FSM to approximate the stochastic absolute value function, and the 32-state FSM to approximate the stochastic linear gain function.

Table 3.1: Approximation errors (e_a) of the four FSM-based stochastic computing elements versus different numbers of states.

Functions	Number of States 8	16	32	64
tanh	0.00%	0.00%	0.00%	0.00%
exp	0.38%	0.07%	0.03%	0.02%
exp (absolute value)	1.48%	0.22%	0.07%	0.05%
absolute value	0.03%	0.00%	0.00%	0.00%
linear gain	0.23%	0.15%	0.03%	0.00%

3.3.2 Quantization Error

To analyze the quantization error, we follow the approach of Qian et al. [3]. In stochastic computing, we round an arbitrary value p ($0 \leq p \leq 1$) to the closest number p' in the set of discrete probabilities we can generate $S = \{0, \frac{1}{M}, \cdots, 1\}$, where M is the maximal integer value that can be generated by a random number generator shown in Fig. 2.1. Let L be the length of the stochastic bit stream. In order to supply good randomness, we need to choose $M \geq L$ [3]. Thus, the quantization error, e_q, for p is

$$e_q = |p - p'| \leq \frac{1}{2M} \leq \frac{1}{2L}.$$

Due to the effect of quantization, we actually compute

$$\sum_{i=0}^{N-1} s_i \cdot P_i(P_X^{'}),$$

instead of the FSM-based approximation

$$\sum_{i=0}^{N-1} s_i \cdot P_i(P_X),$$

where $P_X^{'}$ is the closest number to P_X in the set S. Thus, the quantization error is

$$e_q = |\sum_{i=0}^{N-1} s_i \cdot P_i(P_X^{'}) - \sum_{i=0}^{N-1} s_i \cdot P_i(P_X)|.$$

We define $\Delta P_X = P_X^{'} - P_X$, and using a first order approximation, the error due to

quantization is

$$e_q \approx |\sum_{i=0}^{N-1} s_i \cdot \frac{dP_i(P_X)}{dP_X} \Delta P_X|. \tag{3.43}$$

Computing (3.43) is complex, and the result depends on s_i, i.e., the configuration of the FSM. Because $\sum_{i=0}^{N-1} s_i \cdot P_i(P_X)$ approximates the desired function $T(P_X)$ closely, we can approximate e_q by using $T(P_X)$:

$$e_q \approx |\frac{dT(P_X)}{dP_X} \Delta P_X|.$$

By applying the upper bound for both $|\frac{dT(P_X)}{dP_X}|$ and ΔP_X, we can derive the maximum quantization error for each of the four FSM-based stochastic computing elements, which is listed in Table 3.2.

Table 3.2: The maximum quantization errors (e_q) of the FSM-based stochastic computing elements.

Functions	**Quantization error** e_q
tanh	$N/2L$
exp	$4G/L$
exp (absolute value)	$4G/L$
absolute value	$1/L$
linear gain	P_K/L

From Table 3.2, it can be seen that increasing L will reduce the quantization error, which is consistent with our intuition.

3.3.3 Error Due to Random Fluctuations

The output bit stream $Y(\gamma)$ ($\gamma = 1, 2, \cdots, L$, where L is the length of the stochastic bit streams) of the FSM-based stochastic computing elements has probability

$$p' = \sum_{i=0}^{N-1} s_i \cdot P_i(P'_X),$$

that each bit is one. By using a counter, we can translate this stochastic bit stream $Y(\gamma)$ to a deterministic value y, where

$$y = \frac{1}{L}\sum_{\gamma=1}^{L} Y(\gamma).$$

It is easily seen that the expected value of y is $E[y] = p'$. However, the realization of y is not, in general, exactly equal to p' due to random fluctuations in the bit streams. The error can be measured by the variance as

$$Var[y] = Var\left[\frac{1}{L}\sum_{\gamma=1}^{L} Y(\gamma)\right] = \frac{1}{L^2}\sum_{\gamma=1}^{L} Var[Y(\gamma)] = \frac{p'(1-p')}{L}.$$

Since $Var[y] = E[(y - E[y])^2] = E[(y - p')^2]$, the error due to random fluctuation is

$$e_r = |y - p'| \approx \sqrt{\frac{p'(1-p')}{L}}.$$

Thus, the error due to random fluctuations is inversely proportional to $\sqrt{L}$, and increasing the length of the stochastic bit stream will reduce this error.

3.3.4 Summary of Error Analysis

The overall error, e, is bounded by the sum of the aforementioned three error components, i.e.,

$$e = e_a + e_q + e_r.$$

We evaluate each of the five FSM-based stochastic computing elements for different lengths of stochastic bit streams on 16 points: 1/16, 2/16, 3/16, $\cdots$, 15/16, 1. For each length, we repeat the simulation 1000 times and average the errors over all simulations. The final result is shown in Table 3.3. Note that for each FSM-based stochastic computing element, we use the minimum number of states required to make the approximation error e_a less than 10^{-3}. Compared to the results from Table 3.1, we conclude that the overall error is mainly due to quantization and random fluctuations.

Table 3.3: The average overall errors (e) of the five FSM-based stochastic computing elements versus different lengths of stochastic bit streams.

	Length of Stochastic Bit Streams (L)			
Functions	**256**	**512**	**1024**	**2048**
tanh	3.71%	2.54%	1.73%	1.26%
exp	3.97%	2.39%	1.95%	1.88%
exp (absolute value)	11.10%	6.44%	4.02%	3.22%
absolute value	2.65%	2.00%	1.63%	1.42%
linear gain	3.32%	2.68%	1.76%	1.57%

3.4 Experimental Evaluation

In our experiments, we compare the stochastic implementations to deterministic implementations of the five functions in terms of hardware area and fault tolerance.

3.4.1 Comparison with Binary Radix

Hardware Area Comparison

For a comparison with a conventional binary radix implementation, we need to evaluate the cost of conventional adders and multipliers. Of course, there are a wide variety of conventional implementation types, tailored for many different purposes. Some of them require very little area but have high latency. Others require huge area, but offer better performance [31]. We evaluate the hardware cost using the area-delay product [3]. Using this metric, the performance of nearly all multiplier and adder circuits are similar. Sophisticated adders and multipliers, such as Kogge-Stone adders and Wallace Tree multipliers, were developed for additions and multiplications on wide datapaths. Such architectures are not well suited for computations on narrow datapaths. For accuracies of $M \leq 10$, where M is the number of bits used to represent a numerical value in binary radix, the overhead of such sophisticated structures outweighs their benefits.

For our studies we chose an M-bit multiplier based on the logic design of the ISCAS'85 circuit C6288, given in the benchmark as 16 bits [32]. We use this circuit because its structure is regular. The C6288 is built with *carry-save adders*. It consists of 240 full- and half-adder cells arranged in a 15×16 matrix. Each full adder is realized by 9 NOR gates. Incorporating the M-bit multiplier and optimizing it, the circuit requires $10M^2 - 4M - 9$ gates; these are inverters, fanin-2 AND gates, fanin-2 OR gates,

and fanin-2 NOR gates. The critical path of the circuit passes through $12M - 11$ logic gates [1].

We use the Maclaurin polynomial [1] approximation for the deterministic implementations of the five functions. We need a polynomial of degree 6 to approximate the tanh function, and a polynomial of degree 5 to approximate both the exponentiation function and the one with absolute value to achieve the same level of approximation errors as those of the corresponding stochastic implementations shown in Table 3.1. The corresponding Maclaurin polynomials are computed using adders and multipliers. We show the corresponding area-delay products of the deterministic implementations of the five functions in Table 3.4. To evaluate the area-delay product of the circuit which computes absolute value, we assume it has the same area-delay product of a full adder (the area is $9M$ gates, and the delay is M gates).

Table 3.4: The area and delay of the stochastic implementations and the deterministic implementations of the five functions. Delay values of the stochastic implementations stand for the overall delay. Critical path delay of the stochastic implementations can be obtained by dividing 2^M.

	Stochastic Implementation	Deterministic Implementation
Functions	Area-Delay Product	Area-Delay Product
tanh	$35 \times 3 \times 2^M$	$(10M^2 - 4M - 9) \cdot (12M - 11) \cdot 6$
exp	$75 \times 4 \times 2^M$	$(10M^2 - 4M - 9) \cdot (12M - 11) \cdot 5$
exp (absolute value)	$126 \times 4 \times 2^M$	$(10M^2 - 4M - 9) \cdot (12M - 11) \cdot 5 + 9M^2$
absolute value	$40 \times 3 \times 2^M$	$9M^2$
linear gain	$110 \times 4 \times 2^M$	$(10M^2 - 4M - 9) \cdot (12M - 11)$

As we mentioned in the previous section, we implement the five FSM-based SCEs with the minimum number of states to make the approximation error less than 10^{-3}, i.e., we implement the stochastic tanh function with an 8-state FSM, the stochastic exponentiation function with a 16-state FSM, the stochastic exponentiation function based on an absolute value with a 32-state FSM, the the stochastic absolute value function with an 8-state FSM, and the stochastic linear gain function with a 32-state FSM. The corresponding configurations have been introduced in Section 3.2. Table 3.4 also shows the area-delay products of the stochastic implementations of the five functions. Each circuit of the stochastic implementations is composed of the seven basic types of logic gates: inverters, fanin-2 AND gates, fanin-2 NAND gates, fanin-2 OR gates, fanin-2

NOR gates, fanin-2 XOR gates, and fanin-2 XNOR gates. The DFF is implemented with 6 fanin-2 NAND gates. When characterizing the area and delay, we assume that the operation of each fanin-2 logic gate requires unit area and unit delay. Note that if we assume M is the number of bits used to represent a numerical value in a binary radix, in order to get the same resolution for computation on stochastic bit streams, we need a 2^M-bit stream to represent the same value.

From Table 3.4, we can see that, except the stochastic absolute value function and the stochastic linear gain function, the area-delay product of the stochastic implementation is less than that of the deterministic implementation when $M \leq 10$. Although the stochastic implementation of the stochastic absolute value function and the stochastic linear gain function has larger area-delay product than the corresponding deterministic implementation, these two stochastic computing elements are necessary components for some applications, such as the stochastic implementation of an edge detection algorithm in digital image processing, and the overall hardware cost is still less than that of the deterministic implementation [6].

In fact, both Qian et al. [3] and Brown et al. [15] had shown that, when $M \leq 10$, computation on stochastic bit streams has better performance than those based on binary radix using adders and multipliers in terms of area-delay product. Furthermore, since the energy consumed by computation is generally proportional to the product of the circuit area and the computation delay, the stochastic implementation also has an advantage in energy consumption. Note that stochastic computation is usually applied to those applications that do not require a high resolution, typically with an M in the range from 8 to 10. For example, in image processing applications [3, 6], M is normally set to 8; in artificial neural network applications [15], M is normally set to 10. Thus, for most applications of stochastic computation, the stochastic implementation has an advantage both in area-delay product and energy consumption [3, 15]. In addition, computing on stochastic bit streams offers tunable precision—as the length of the stochastic bit stream increases, the precision of the value represented by it also increases. Thus, without hardware redesign, we have the flexibility to trade-off precision and computation time.

Fault-Tolerance Comparison

To make comparisons on fault-tolerance, we compare the performance of the deterministic implementations versus the stochastic implementations of the five functions when the input data is corrupted with noise. Suppose that the input data of a deterministic implementation is represented using $M = 10$ bits. In order to achieve the same resolution, the bit stream of a stochastic implementation contains $2^M = 1024$ bits. We choose the error ratio ϵ of the input data to be 0, 0.001, 0.002, 0.005, 0.01, 0.02, 0.05, and 0.1, as measured by the fraction of random bit flips that occur.

We evaluated each of the five functions on 16 points: 1/16, 2/16, 3/16, $\cdots$, 15/16, 1. For each error ratio ϵ, each function, and each evaluation point, we simulated both the stochastic and the deterministic implementations 1000 times. We averaged the relative errors over all simulations. Finally, for each error ratio ϵ, we averaged the relative errors over all evaluation points. Table 3.5 shows the average relative error of the stochastic implementations and the deterministic implementations for the five functions versus different error ratios ϵ. We average the relative errors of the five functions, and plot the results in Fig. 3.11 to give a clear comparison.

Table 3.5: Relative errors of the stochastic implementations and the deterministic implementations of the five functions versus different error ratios ϵ in the input data.

	tanh Rel. Error of		**exp** Rel. Error of		**exp (abs.)** Rel. Error of		**absolute value** Rel. Error of		**linear gain** Rel. Error of	
Error Ratio ϵ	Stoch. Impl. (%)	Deter. Impl. (%)	Stoch. Impl. (%)	Deter. Impl. (%)	Stoch. Impl. (%)	Deter. Impl. (%)	Stoch. Impl. (%)	Deter. Impl. (%)	Stoch. Impl. (%)	Deter. Impl. (%)
0	1.73	0.00	1.95	0.00	4.02	0.00	1.63	0.00	2.13	0.00
0.001	1.93	0.52	2.17	0.35	4.21	0.67	1.65	0.43	2.33	0.36
0.002	2.21	1.23	2.25	0.79	4.27	1.58	1.69	0.62	2.53	1.67
0.005	2.23	2.42	2.33	2.11	4.32	3.19	1.73	1.62	3.21	2.89
0.01	2.26	5.36	2.37	3.47	4.39	4.89	1.92	4.10	3.76	5.94
0.02	2.35	9.36	2.42	7.43	4.46	8.32	2.36	6.01	3.98	9.33
0.05	2.69	19.68	2.46	14.50	5.06	15.78	3.91	15.44	4.13	15.76
0.1	3.83	34.95	2.97	25.36	7.21	26.83	6.55	29.56	4.67	25.98

When $\epsilon = 0$, meaning that no noise is injected into the input data, the deterministic implementation computes without any error. However, due to the inherent variance, the stochastic implementation produces a small relative error [3]. When $\epsilon > 0$, the relative

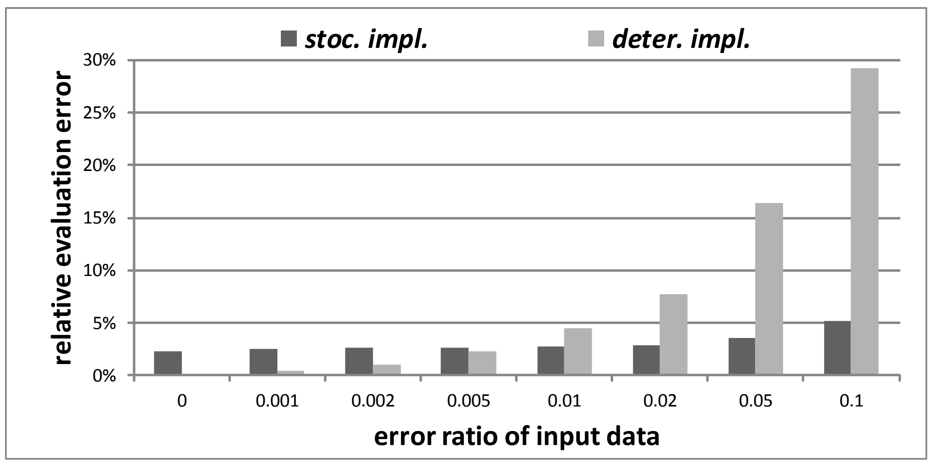


Figure 3.11: The average of the relative errors of the four functions shown in Table 3.5.

error of the deterministic implementation blows up dramatically as ϵ increases. Even for small values, the stochastic implementation performs much better.

It is not surprising that the deterministic implementation is so sensitive to errors, given that the representation used is binary radix. In a noisy environment, bit flips affect all the bits with equal probability. In the worst case, the most significant bit gets flipped, resulting in an error of $2^{M-1}/2^M = 1/2$ on the input value. In contrast, in a stochastic implementation, the data is represented as the fractional weight on a bit stream of length 2^M. Thus, a single bit flip only changes the input value by $1/2^M$, which is minuscule in comparison [3].

Chapter 4

Digital Image Processing Case Studies

This chapter presents the application of the FSM-based stochastic computing elements for five specific digital image processing algorithms as case studies. These algorithms are: image edge detection, median filter-based noise reduction, image contrast stretching, frame difference-based image segmentation, and KDE-based image segmentation. We analyze the error tolerate of the stochastic implementations of these algorithms and compare it to the corresponding conventional implementations using the binary encoding. We also analyze and compare the hardware cost, latency and energy consumption [4, 6, 18, 19].

4.1 Stochastic Implementations of Image Processing Algorithms

In this section, we demonstrate the stochastic implementations of the five digital image processing algorithms. In our implementations, we use a 16-state FSM to compute the absolute value, and 32-state FSMs to compute the tanh function, the exponentiation function, and the linear gain function in all the five algorithms.

4.1.1 Edge Detection

Classical methods of edge detection involve convolving the image with an operator (a 2-D filter), which is constructed to be sensitive to large gradients in the image while returning values of zero in uniform regions [33]. There are a large number of edge detection operators available, each designed to be sensitive to certain types of edges. Most of these operators can be efficiently implemented by the SCEs introduced in this paper. Here we consider only Robert's cross operator as shown in Fig. 4.1 as an example [33].

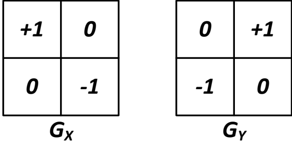


Figure 4.1: Robert's cross operator for edge detection.

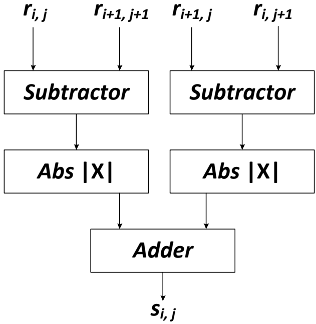


Figure 4.2: The conventional implementation of the Robert's cross operator based edge detection.

This operator consists of a pair of 2×2 convolution kernels. One kernel is simply the other rotated by 90°. An approximate magnitude is computed using: $G = |G_X| + |G_Y|$, i.e.,

$$s_{i,j} = \frac{1}{2}(|r_{i,j} - r_{i+1,j+1}| + |r_{i,j+1} - r_{i+1,j}|),$$

where $r_{i,j}$ is the pixel value at location (i, j) of the original image and $s_{i,j}$ is the pixel value at location (i, j) of the processed image. Note that the coefficient $\frac{1}{2}$ is used to

scale $s_{i,j}$ to [0, 255], which is the range of the grayscale pixel value. The conventional implementation of this algorithm is shown in Fig. 4.2.

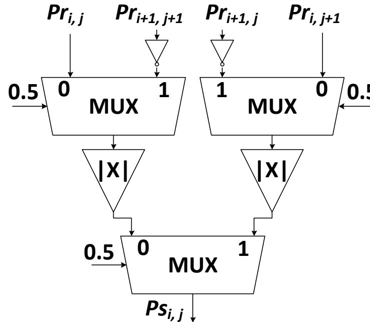


Figure 4.3: The stochastic implementation of the Robert's cross operator based edge detection.

The stochastic implementation of this algorithm is shown in Fig. 4.3, in which $P_{r_{i,j}}$ is the probability of ones in the stochastic bit stream which is converted from $r_{i,j}$, i.e., $P_{r_{i,j}} = \frac{r_{i,j}}{256}$. So are $P_{r_{i+1,j}}$, $P_{r_{i,j+1}}$, and $P_{r_{i+1,j+1}}$. Suppose that under the bipolar encoding, the values represented by the stochastic bit streams $P_{r_{i,j}}$, $P_{r_{i+1,j}}$, $P_{r_{i,j+1}}$, $P_{r_{i+1,j+1}}$, and $P_{s_{i,j}}$ are $a_{r_{i,j}}$, $a_{r_{i+1,j}}$, $a_{r_{i,j+1}}$, $a_{r_{i+1,j+1}}$, and $a_{s_{i,j}}$, respectively. Then, based on the circuit, we have

$$a_{s_{i,j}} = \frac{1}{4}(|a_{r_{i,j}} - a_{r_{i+1,j+1}}| + |a_{r_{i,j+1}} - a_{r_{i+1,j}}|).$$

Because $a_{s_{i,j}} = 2P_{s_{i,j}} - 1$ and $a_{r_{i,j}} = 2P_{r_{i,j}} - 1$ ($a_{r_{i+1,j}}$, $a_{r_{i,j+1}}$, $a_{r_{i+1,j+1}}$ are defined in the same way), we have

$$P_{s_{i,j}} = \frac{1}{4}\left(\left|P_{r_{i,j}} - P_{r_{i+1,j+1}}\right| + \left|P_{r_{i,j+1}} - P_{r_{i+1,j}}\right|\right) + \frac{1}{2} = \frac{s_{i,j}}{512} + \frac{1}{2}.$$

Thus, by counting the number of ones in the output bit stream, we can convert it back to $s_{i,j}$.

4.1.2 Noise Reduction Based on The Median Filter

The median filter replaces each pixel with the median of neighboring pixels. It is quite popular because, for certain types of random noise (such as salt-and-pepper noise), it

provides excellent noise-reduction capabilities, with considerably less blurring than the linear smoothing filters of a similar size [33]. A hardware implementation of a 3 × 3 median filter based on a sorting network is shown in Fig. 4.4. Its basic unit and the corresponding conventional implementation are shown in Fig. 4.5 and Fig. 4.6. This unit is used to sort two inputs in ascending order.

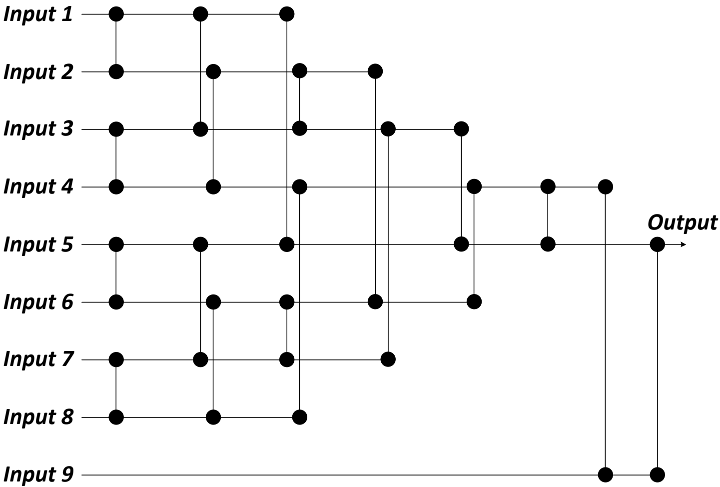


Figure 4.4: Hardware implementation of the 3 × 3 median filter based on a sorting network.

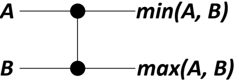


Figure 4.5: The basic sorting unit.

The stochastic version of this basic unit is shown in Fig. 4.7, which is implemented by the stochastic comparator introduced in the previous chapter with a few modifications.

In Fig. 4.7, if $P_A > P_B$, $P_S \approx 1$, the probability of ones in the output of "$MUX1$" is P_B, which is the minimum of (P_A, P_B), and the probability of ones in the output of "$MUX2$" is P_A, which is the maximum of (P_A, P_B).

If $P_A < P_B$, $P_S \approx 0$, the probability of ones in the output of "$MUX1$" is P_A, which is the minimum of (P_A, P_B), and the probability of ones in the output of "$MUX2$" is P_B, which is the maximum of (P_A, P_B).

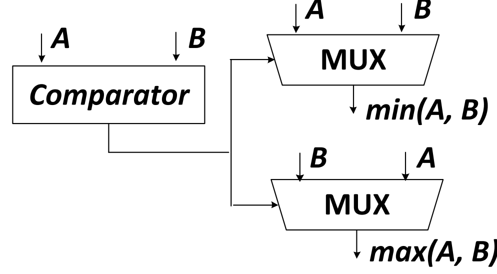


Figure 4.6: The conventional implementation of the basic sorting unit. It requires 379 logic gates in total.

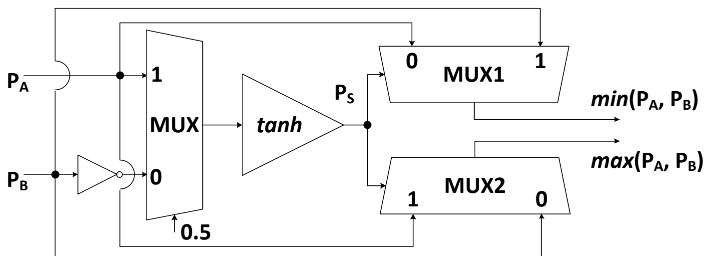


Figure 4.7: The stochastic implementation of the basic sorting unit, which can be implemented using only 66 logic gates.

If $P_A = P_B$, $P_S \approx 0.5$, both the probabilities of ones in the outputs of $MUX1$ and $MUX2$ should be very close to $\frac{P_A+P_B}{2} = P_A = P_B$. Based on this circuit, we can implement the sorting network shown in Fig. 4.4 stochastically.

4.1.3 Contrast Stretching

Image contrast stretching, or normalization, is used to increase the dynamic range of the gray levels in an image. One of the typical transformations used for contrast stretching is shown in Fig. 4.8.

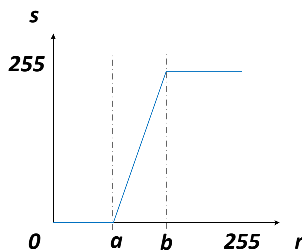


Figure 4.8: A piecewise linear gain function used in image contrast stretching.

It can be seen that a closed-form expression of the piecewise linear gain function shown in Fig. 4.8 is,

$$s = \begin{cases} 0, & 0 \le r \le a, \\ \frac{255}{b-a} \cdot (r - 0.5 \cdot (a+b)) + 128, & a < r < b, \\ 255, & b \le r \le 255. \end{cases} \tag{4.1}$$

The conventional implementation of the above function is shown in Fig. 4.9, which includes some complex arithmetic circuits such as subtractor and multiplier. However, this function can be efficiently implemented stochastically using the two-parameter stochastic linear gain function introduced in Fig.3.8 of the previous chapter. In that circuit, we set

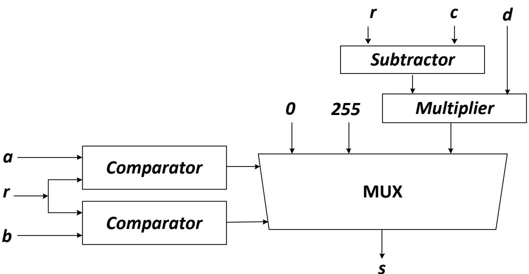


Figure 4.9: The conventional implementation of the piecewise linear gain function used in image contrast stretching, where $d = \frac{255}{b-a}$ and $c = \frac{255a}{b-a}$ based on equation (4.1).

$$P_X = \frac{r}{255}, P_K = \frac{510 + a - b}{510 - a + b}, P_C = \frac{a+b}{510},$$

then,

$$P_Y = \begin{cases} 0, & 0 \leq P_X \leq \frac{a}{255}, \\ \frac{r - 0.5 \cdot (a+b)}{b-a} + \frac{1}{2}, & \frac{a}{255} < P_X < \frac{b}{255}, \\ 1, & \frac{b}{255} \leq P_X \leq 1. \end{cases}$$

It can be seen that $P_Y = \frac{s}{255}$. Thus, by counting the number of ones in the output bit stream, we can convert it back to s.

4.1.4 Frame Difference-Based Image Segmentation

If we define the value of a pixel at the current frame as X_t, and the value of a pixel at the same location of the previous frame as X_{t-1}, the frame difference-based image segmentation uses the difference between X_t and X_{t-1} to see if it is greater than a predefined threshold Th. If yes, the pixel at the current frame is set to the foreground; otherwise, it is set to the background. The conventional implementation of this algorithm are shown in Fig. 4.10.

The stochastic implementation of this algorithm is shown in Fig. 4.11, in which we set

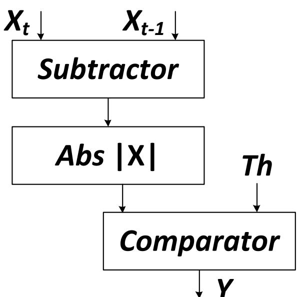


Figure 4.10: The conventional implementation of the frame difference-based image segmentation. The total number of logic gates of this implementation is 486.

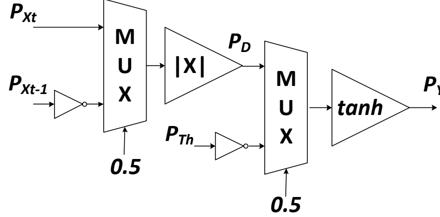


Figure 4.11: The stochastic implementation of the frame difference-based image segmentation. The total number of logic gates of this implementation is 107.

$$P_{X_t} = \frac{X_t}{255}, P_{X_{t-1}} = \frac{X_{t-1}}{255}, P_{Th} = \frac{Th}{510} + \frac{1}{2},$$

then the probability (P_D) of the ones in the output bit stream of the stochastic absolute value function is,

$$P_D = \frac{|X_t - X_{t-1}|}{510} + \frac{1}{2}.$$

Notice that we convert the bipolar encodings into the probabilities of ones in the bit stream. The bit stream P_Y is the output of the stochastic comparator with inputs P_D and P_{Th}. Based on the function of the stochastic comparator, if $|X_t - X_{t-1}| < Th$, $P_Y = 0$, which means the current pixel belongs to the background; else $P_Y = 1$, which means the current pixel belongs to the foreground.

4.1.5 KDE-Based Image Segmentation

KDE is another image segmentation algorithm which is normally used for object recognition, surveillance, and tracking. The basic approach of this algorithm is to build a background model to capture the very recent information about a sequence of images while continuously updating this information to capture fast changes in the scene. This can be used to extract changes in a video stream in real-time, for instance. Since the intensity distribution of a pixel can change quickly, the density function of this distribution must be estimated at any moment of time given only the very recent history information. Let (X_t, X_{t-1}, X_{t-2}, ..., X_{t-n}) be a recent sample of intensity values of a pixel. Using this sample of values, the probability density function (PDF) describing the distribution of intensity values that this pixel will have at time t can be non-parametrically estimated using the kernel estimator K,

$$PDF(X_t) = \frac{1}{n} \sum_{i=1}^{n} K(X_t - X_{t-i}).$$

The kernel K should be a symmetric function. For example, if we choose our kernel estimator K to be $e^{-4|x|}$, then the PDF can be estimated as:

$$PDF(X_t) = \frac{1}{n} \sum_{i=1}^{n} e^{-4|X_t - X_{t-i}|}. \tag{4.2}$$

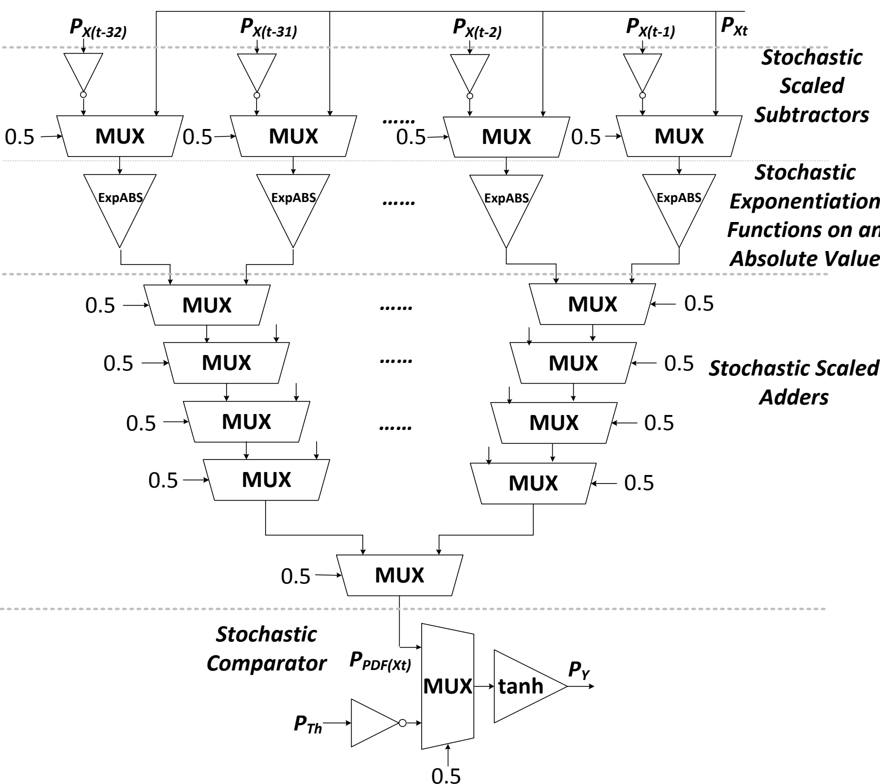


Figure 4.12: The stochastic implementation of the KDE-based image segmentation algorithm.

Using this probability estimator, a pixel is considered a background pixel if $PDF(X_t)$ is less than a predefined threshold Th. Both the conventional implementation and the stochastic implementation of this algorithm are mapped from equation (4.2). For example, Fig. 4.12 shows the stochastic implementation of this algorithm based on 32 frame parameters (i.e., $n = 32$ in equation (4.2)). Similar to the previous four algorithms, we can get the conventional implementation of this algorithm by mapping the stochastic computing elements to the corresponding conventional computing elements. The block diagram of the conventional implementation is shown in Fig. 4.13. Note that in the conventional implementation, the multipliers will be used multiple times to compute the exponentiation function based on an absolute value.

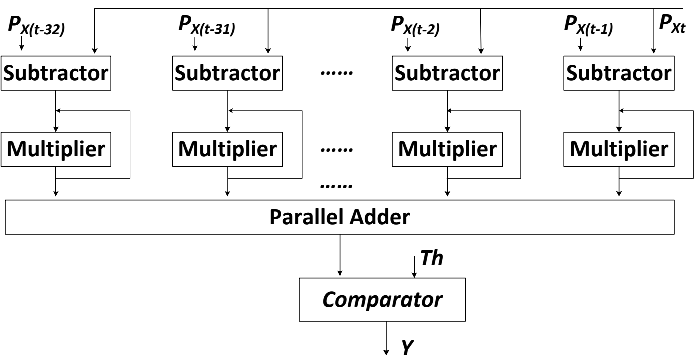


Figure 4.13: The conventional implementation of the KDE-based image segmentation algorithm.

In the stochastic implementation shown in Fig.4.12, the first part is 32 *"Stochastic Scaled Subtracters,"* which are used to compute $0.5 \cdot (P_{Xt} - P_{X(t-i)})$; the second part is 32 *"Stochastic Exponentiation Functions on an Absolute Value,"* which is used to implement the kernel estimator and has been introduced in the previous chapter; the third part of this circuit is 31 *"Stochastic Scaled Adders,"* which computes $P_{PDF}(Xt)$; and the last part of this circuit is a *"Stochastic Comparator,"* which is used to produce the final segmentation output. Based on the circuit shown in Fig. 4.12, if $PDF(X_t) < Th$, $P_Y = 0$, which means the current pixel belongs to the background; else $P_Y = 1$, which means the current pixel belongs to the foreground.

4.2 Experimental Results

In this section, we present the experimental results of the stochastic and conventional implementations of the five digital image processing algorithms discussed in Section 4.1. The experiments include comparisons in terms of fault-tolerance and hardware resources. We use the Xilinx System Generator [34] to estimate hardware costs since the systems built by this tool are very close to the real hardware, and can be easily verified using an FPGA.

4.2.1 Simulation Results

We use 1024 bits to represent a pixel value in stochastic computing. The simulation results are shown in Fig. 4.14. It can be seen that the simulation results generated by the stochastic implementations are almost the same as the ones generated by the conventional implementations based on binary radix. In fact, computation based on stochastic bit streams does have some errors compared to computation based on a binary radix. For applications such as image/audio processing and artificial neural networks, however, these errors can be ignored because humans can hardly see such small differences.

4.2.2 Hardware Resources Comparison

In Section 4.1, we introduced circuit structures of both the stochastic and conventional implementations of the five digital image processing algorithms. The hardware cost of these implementations in terms of the equivalent two-input NAND gates is shown in Table 4.1. It can be seen that the stochastic implementation uses substantially fewer hardware resources than the conventional implementation especially when the algorithm needs many computational elements. This is mainly because SCEs take much less hardware than the ones based on the binary radix encoding used in the conventional implementation. For example, multiplication can be implemented using a single AND gate stochastically, and the exponentiation function can be implemented stochastically using an FSM.

Note that in our hardware evaluation, we did not consider the hardware cost of the interface circuitry, i.e., the hardware cost of encoding values by random bit streams

	Original Image	Conventional Implementation	Stochastic Implementation
Edge Detection			
Noise Reduction			
Contrast Stretching			
Frame Difference			
KDE			

Figure 4.14: Simulation results of the conventional and the stochastic implementations of the image processing algorithms.

and decoding random bit streams into other representations. In our current implementations, the interface circuitry is designed based on the linear feedback shift register (LFSR) because it is easy to implement and simulate. However, LFSR-based interface circuitry is expensive. In future work, we are going to design the interface circuitry based on a sigma-delta analog to digital conversion technique. The hardware cost for this technique is much less than the LFSR-based technique, and the random bit streams can be generated almost for free.

Table 4.1: Hardware resources comparison in terms of the equivalent two-input NAND gates.

	Conventional	Stochastic	Ratio (conv. vs. stoc.)
Edge Detection	776	110	7.05
Frame Difference	486	107	4.54
Noise Reduction	7.2K	1.25K	5.76
Contrast Stretching	896	54	16.6
KDE	400K	1.5K	267

4.2.3 Energy Consumption Comparison

Although the stochastic implementations of digital image processing algorithms have lower hardware cost and can tolerate more errors, they might consume more energy. It depends on how complex the algorithms are and how many bits are used to represent a pixel value stochastically. If we use L-bit to represent a pixel value stochastically, the processing time of the stochastic implementation will be L times slower than the corresponding conventional implementation. We evaluate the energy consumption using the product of the hardware cost (shown in Table 4.1) and the corresponding processing time. Specifically, for each algorithm,

$$\frac{E_{conv}}{E_{stoch}} = \frac{Area_{conv}}{Area_{stoch}} \cdot \frac{Time_{conv}}{Timestoch} = \frac{Area\ Ratio}{L},$$

where *Area Ratio* is shown in the fourth column of Table III. We show the comparison results in Table 4.2. It can be seen that for a simple algorithm, such as the image edge detection, the stochastic implementation consumes more energy even if $L > 7$. For a complex algorithm, though, such as the KDE-based image segmentation, the stochastic implementation consumes less energy than the conventional implementation if $L \leq 256$.

Table 4.2: Energy consumption comparison.

	Conventional vs. Stochastic
Edge Detection	7.05 : L
Frame Difference	4.54 : L
Noise Reduction	5.76 : L
Contrast Stretching	16.6 : L
KDE	267 : L

We also compare the area-delay product for different implementations, and the results are shown in Table 4.3. The area has been given in Table 4.1. The delay is evaluated using the number of gates in the critical path of different implementations. Note that the delay of the stochastic implementations are multiplied by 2^10, because we represent the pixel value using 1024 bits in the current experiments. The data in Table 4.3 conclude the same results as the ones in Table 4.2: the area-delay products of different implementations depend on the the algorithm complexity. For a simple algorithm, such as the image edge detection, the area-delay product of the convention implementation is smaller. For a complex algorithm, though, such as the KDE-based image segmentation, the area-delay product of the stochastic implementation is smaller.

Table 4.3: Area-delay product comparison.

	delay		**area-delay product**		**ratio**
	conv.	**stoc.**	**conv.**	**stoc.**	
Edge Detection	48	$4 \cdot 2^{10}$	37248	450560	0.08
Frame Difference	37	$5 \cdot 2^{10}$	17982	547840	0.03
Noise Reduction	85	$5 \cdot 2^{10}$	612000	6400000	0.10
Contrast Stretching	48	$5 \cdot 2^{10}$	43008	276480	0.16
KDE	85	$5 \cdot 2^{10}$	34000000	7680000	4.43

4.2.4 Fault-Tolerance Comparison

We test the fault-tolerance of both implementations by randomly injecting soft errors into the internal circuits, and measuring the corresponding average output error for each implementation. To inject soft errors into a computational element, such as a MUX shown in Fig. 4.15, we insert XOR gates into all of its inputs and output, which is shown in Fig. 4.16. For each XOR gate, one of its inputs is connected to the original signal of the MUX and the other is connected to a global random soft error source, which

is implemented using a linear feedback shift register (LFSR) and a comparator [3]. Note that if the output of the computational element is connected to the input of another computational element, we do not inject the error twice on the intermediate line.

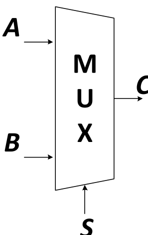


Figure 4.15: Test circuit for the error injection.

If the error signal (*Error A*, *Error B*, *Error S*, or *Error C* in Fig. 4.16(b)) equals one, its corresponding original signal (A, B, S, or C in Fig. 4.16(b)) will be flipped due to the XOR gate. If the error signal equals zero, it has no influence on its corresponding original signal. The *Register* shown in Fig. 4.16(b) is used to control the soft error injection rate. If we use a k-bit LFSR and want to generate $p\%$ soft errors, we can set the register's value to $2^k \cdot p\%$. For example, in Fig. 4.16(b), assuming that we use a 10-bit LFSR and want to generate an error rate of 30%, we should set the register's value to $2^{10} \times 30\% = 307$. Note that the LFSR has a parameter called the initial value. By setting different initial values for each of the LFSRs, we can guarantee that the random soft errors generated by them are independent of each other. In addition, at each clock cycle, we only enable one LFSR to generate the soft error.

We apply this approach to each basic computational element in a circuit, such as comparator, adder, subtractor, and multiplier in the conventional implementations; and NOT gate, AND gate, XNOR gate, MUX, stochastic absolute value function, stochastic comparator, and stochastic linear gain function in the stochastic implementations. Fig. 4.17 uses the KDE-based image segmentation as an example to visualize the effect. We summarize the average output error of the two implementations under different soft error injection rates for all of the five algorithms in Table 4.4. We define the height of the image to be H and the width to be W, and calculate the average output error E in

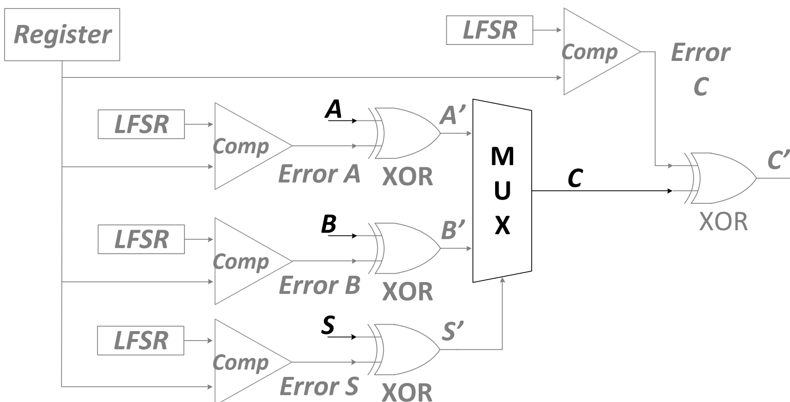


Figure 4.16: The circuit for simulating injected soft errors. A, B, C, and S are the original signals. $Error\ A$, $Error\ B$, $Error\ C$, and $Error\ S$ are the soft error signals generated by the random soft error source. A', B', C', and S' are the signals corrupted by the soft errors.

Table 4.4 as follows,

$$E = \frac{\sum_{i=1}^{H} \sum_{j=1}^{W} |T_{i,j} - S_{i,j}|}{255 \cdot H \cdot W}, \tag{4.3}$$

where S is the output image of the conventional implementation of an algorithm without any injected soft errors, and T is the output image of the implementation of the same algorithm with injected soft errors. We define the soft error injection rate to be R, and use a linear regression model [35],

$$E = a + b \cdot R, \tag{4.4}$$

to analyze the relationship between R and E based on the data in Table 4.4. For the stochastic implementations, we get $a < 5\%$ and $b \approx 0$ for all the five algorithms. This analysis means that the stochastic implementations are not sensitive to the error injection rate R because the slope of the fitted line is $b \approx 0$. However, it does have small errors even though we do not inject any soft errors in the circuit. These errors are due to approximation, quantization, and random fluctuations [3], and can be quantified by the parameter a in (4.4). For the conventional implementations (except the noise reduction algorithm), we get $a \approx 0$ and $b > 50\%$. This means that the conventional

implementations are very sensitive to the soft error injection rate R because the slope of the fitted line $b > 50\%$.

Table 4.4: Fault tolerance test results.

	The average output error E of the stochastic implementations						
Noise	0%	1%	2%	5%	10%	15%	30%
Edge Detection	2.05%	1.97%	2.10%	2.10%	2.09%	2.10%	2.54%
Frame Difference	0.31%	0.40%	1.17%	0.38%	0.34%	0.49%	0.74%
Noise Reduction	1.02%	1.04%	1.05%	1.10%	1.22%	1.34%	1.73%
Contrast Stretching	2.55%	2.43%	2.33%	2.27%	2.70%	3.56%	6.88%
KDE	0.89%	0.91%	0.91%	0.93%	0.96%	0.99%	1.11%
	The average output error E of the conventional implementations						
Noise	0%	1%	2%	5%	10%	15%	30%
Edge Detection	0.00%	1.37%	1.73%	3.22%	6.42%	9.92%	20.50%
Frame Difference	0.00%	0.82%	2.24%	6.67%	13.71%	20.56%	38.36%
Noise Reduction	0.00%	0.07%	0.20%	0.57%	1.13%	1.65%	3.06%
Contrast Stretching	0.00%	0.66%	1.93%	5.56%	10.89%	15.49%	25.49%
KDE	0.00%	0.54%	1.20%	3.20%	6.60%	9.71%	19.02%

4.2.5 Fault-Tolerance Analysis

In this section, we explain why the stochastic computing technique can tolerate more errors than the conventional computing technique based on a binary radix. Assume that the binary radix number has M bits. Consider a binary radix number

$$p = x_1 2^{-1} + x_2 2^{-2} + \cdots + x_M 2^{-M}.$$

Assume that each bit x_i $(1 \leq i \leq M)$ has probability ϵ being flipped. Since bit flips happen independently, we can use M independent random Boolean variable $R_1, \ldots, R_M$ to indicate the bit flips. Specifically, if $R_i = 1$, then bit x_i is flipped; otherwise, bit x_i is not flipped. We have $P(R_i = 1) = \epsilon$.

Now we can model the overall error with random variables $R_1, \ldots, R_M$. If the i-th bit is flipped, then the bit value becomes $1 - x_i$; otherwise, it is still x_i. Therefore, with bit flips randomly occurring, the value of the i-th bit is also a random variable, which can be represented as

$$X_i = (1 - x_i)R_i + x_i(1 - R_i)$$

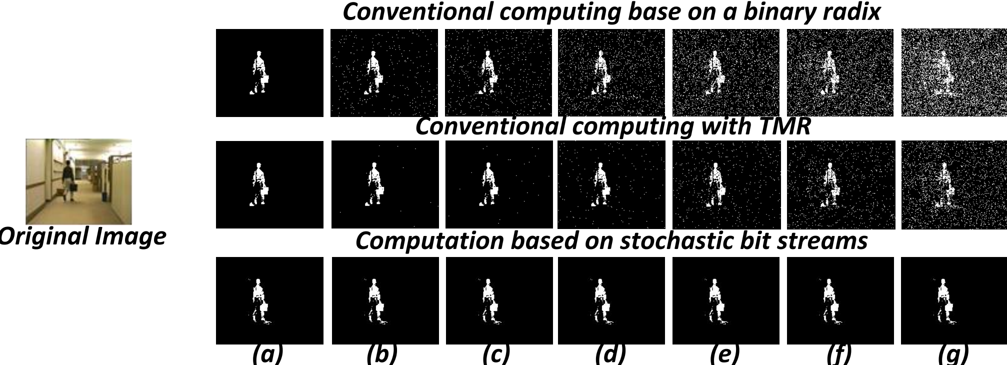


Figure 4.17: A comparison of the fault tolerance capabilities of different hardware implementations for the KDE-based image segmentation algorithm. The images in the top row are generated by a conventional deterministic implementation. The images in the second row are generated by the conventional implementation with a TMR-based approach. The images in the bottom row are generated using a stochastic implementation [4]. Soft errors are injected at a rate of (a) 0%; (b) 1%; (c) 2%; (d) 5%; (e) 10%; (f) 15%; (g) 30%.

It can be easily seen that

$$P(X_i = 1 - x_i) = \epsilon, \quad P(X_i = x_i) = 1 - \epsilon.$$

Now with independent random bit flips occurring at each bit, the binary radix number is

$$q = X_1 2^{-1} + X_2 2^{-2} + \cdots + X_M 2^{-M} = \sum_{i=1}^{M} [(1 - x_i) R_i + x_i (1 - R_i)] 2^{-i}.$$

The error is

$$e = q - p = \sum_{i=1}^{M} [(1 - x_i) R_i + x_i (1 - R_i)] 2^{-i} - \sum_{i=1}^{M} x_i 2^{-i} = \sum_{i=1}^{M} (1 - 2x_i) R_i 2^{-i}, \quad (4.5)$$

which is also a random variable.

Now we consider the mean of the error e and the variance of the error e. Since

$R_1, \ldots, R_M$ are independent, from equation (4.5), we have

$$E[e] = \sum_{i=1}^{M} (1 - 2x_i) 2^{-i} E[R_i]$$
$$Var[e] = \sum_{i=1}^{M} (1 - 2x_i)^2 2^{-2i} Var[R_i]$$

For each R_i, it can be easily shown that

$$E[R_i] = \epsilon, \quad Var[R_i] = \epsilon(1 - \epsilon).$$

Therefore

$$E[e] = \sum_{i=1}^{M} (1 - 2x_i) 2^{-i} \epsilon = \epsilon(\sum_{i=1}^{M} 2^{-i} - 2\sum_{i=1}^{M} x_i 2^{-i}) \approx (1 - 2p)\epsilon$$

Notice that x_i is either 0 or 1. Thus $(1 - 2x_i)^2 = 1$. Therefore,

$$Var[e] = \sum_{i=1}^{M} (1 - 2x_i)^2 2^{-2i} \epsilon(1 - \epsilon) = \sum_{i=1}^{M} 2^{-2i} \epsilon(1 - \epsilon) \approx \frac{1}{3}\epsilon(1 - \epsilon).$$

Now we consider the stochastic encoding of the same value p as in the binary radix case. We need a bit stream of length $N = 2^M$. Suppose that the bit stream is $y_1 y_2 \ldots y_N$. We have

$$p = \frac{1}{N} \sum_{i=1}^{N} y_i.$$

Similarly, we use the random Boolean variable S_i to indicate whether bit y_i is flipped or not. If $S_i = 1$, bit y_i is flipped to $1 - y_i$; otherwise, it stays the same. Assume that the bit flip rate for the stochastic encoding is the same as that for the binary radix encoding, then we have $P(S_i = 1) = \epsilon$.

With bit flips randomly occurring, the value of the i-th bit is also a random variable, which can be represented as

$$Y_i = (1 - y_i)S_i + y_i(1 - S_i).$$

Now with independent random bit flips occurring at each bit, the actual value represented by the stream is

$$q = \frac{1}{N} \sum_{i=1}^{N} Y_i = \frac{1}{N} \sum_{i=1}^{N} [(1 - y_i)S_i + y_i(1 - S_i)].$$

The error is

$$e = q - p = \frac{1}{N}\sum_{i=1}^{N}[(1-y_i)S_i + y_i(1-S_i)] - \frac{1}{N}\sum_{i=1}^{N} y_i = \frac{1}{N}\sum_{i=1}^{N}(1-2y_i)S_i$$

Now we evaluate $E[e]$ and $Var[e]$. We apply the independence and obtain

$$E[e] = \frac{1}{N}\sum_{i=1}^{N}(1-2y_i)E[S_i] = \frac{1}{N}\sum_{i=1}^{N}(1-2y_i)\epsilon = (1-2p)\epsilon.$$

and

$$Var[e] = \sum_{i=1}^{N}\frac{(1-2y_i)^2}{N^2}Var[S_i] = \frac{1}{N}\epsilon(1-\epsilon).$$

It can be seen that bit flips cause errors in both the binary radix encoding and the stochastic encoding. With the same bit flip rate, both the binary radix and the stochastic encoding have the same average error. However, in terms of the variation of the error, they are different. The variation of the error for the binary radix encoding is a constant independent of the number of bits. The variation of the error for the stochastic encoding is inversely proportional to the length of the bit stream. Increasing the length reduces the variation of error.

Now consider error distributions. The error distribution of the binary radix encoding and that of the stochastic encoding essentially have the same center, since they have the same mean value. However, the error distribution of the binary radix encoding is much wider than that of the stochastic encoding, since the former's variance is larger than the latter's. Thus, we have a large chance to obtain a large error by sampling the error distribution of the binary radix encoding. For example, in the KDE-based image segmentation experiment shown in Fig. 4.17, we can consider it a sample from the error distribution. Thus, in the experimental results, we observe large errors for the binary radix encoding.

Note that the introduction of the finite-state machine computing elements adds interdependence between bits in stochastic computing. The analysis presented above does not consider this, and instead assuming independence among the bit streams. Compared to the error in the combinational logic-based stochastic computing elements, the error due to the effect of soft errors affecting multiple bits in the finite-state machine computing elements is affected by two additional factors: the number of states of the

finite-state machine and the relative location of the multiple bit flips. More states cause fewer errors, and adjacent multiple bit flips cause more errors. Based on our experiments, if the finite-state machine has more than 8 states, the adjacent multiple bit flips will not contribute too many errors (less than 1%) in the final results. We plan to make a detailed analysis of the errors in the finite-state machine-based stochastic computing elements in our future work.

Chapter 5

FSM-based Synthesis Methods

As we discussed in the previous chapters, FSMs can be used to implement complex functions stochastically. However, the proposed FSM-based stochastic computing elements are developed based on the specific configurations. A general method about how to synthesize a given target function stochastically using the FSM is missing. This chapter introduces such a synthesis method. In addition, it introduces three different FSM topologies for synthesizing the target function, and discusses the corresponding trade-offs [20, 21, 22, 23].

5.1 Single Input FSM-Based Synthesis Approach

We noticed that the stochastic exponentiation and tanh functions proposed by Brown and Card [15] use a very similar state transition diagram shown in Fig. 2.11. The only difference of the state transition diagrams of the two functions is the output configuration. For example, in Fig. 2.11 if we set the output to be the one shown in Fig. 2.7, the output will be a stochastic exponentiation function of the input. If we set the output to be the one shown in Fig. 2.9, the output will be a stochastic tanh function of the input. This means that, based on different output configurations, the FSM shown in Fig. 2.11 can also compute other functions stochastically. To synthesis a target function, the problem becomes how to configure the output of the FSM.

To solve this problem, we propose a circuit structure shown in Fig. 5.1, where the state transition diagram of the FSM is shown in Fig. 5.2. It can be seen that the output

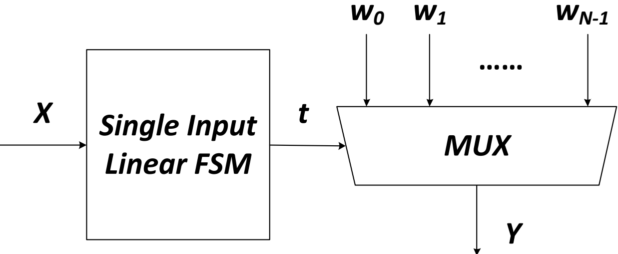


Figure 5.1: The circuit for synthesizing target functions based on the single input linear FSM.

of the FSM is the current state number. In Fig. 5.1, the output of the FSM is connected to the selection inputs of the multiplexer "*MUX*", which has N data inputs (w_0, w_1, $\cdots$, w_{N-1}). Note that if the current state of the FSM is S_t ($0 \leq t \leq N-1$), then the channel that connects w_t to Y will be selected in the "*MUX*." Using this circuit, we can implement the FSM which can compute the exponentiation/tanh function stochastically proposed by Brown and Card [15]. For example, if we set $w_t = 0$ if $0 \leq t \leq N/2-1$ and $w_t = 1$ if $N/2 \leq t \leq N-1$, it will be the same as the FSM shown in Fig. 2.9. In this section, we first analyze the FSM shown in Fig. 5.2. Then we introduce a synthesis approach based on the circuit shown in Fig. 5.1.

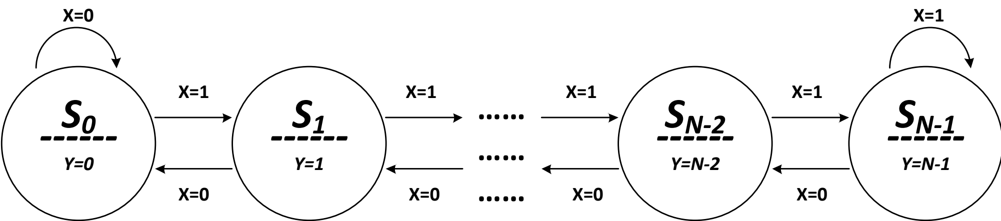


Figure 5.2: The FSM has a single input X. The numbers on each arrow represent the transition condition. The FSM has $log_2\lceil N \rceil$ outputs, encoding a value in binary radix. In the figure, the number below each state S_t ($0 \leq t \leq N-1$) represents the value encoded by the output of the FSM when the current state is S_t.

5.1.1 FSM Analysis

As we introduced in Chapter 3, the basic form of the FSM shown in Fig. 5.2 is a set of states $S_0 \rightarrow S_{N-1}$ arranged in a linear form (e.g. like a saturating counter) [15]. It has totally N states. X is the input of this state machine. It consists of a stochastic bit stream. We define the probability that each bit in the input stream X is one to be P_X. Because the input X is a stochastic bit stream with fixed probability, the state transition process of the FSM can be modeled as a time-homogeneous Markov chain. It can be shown that this Markov chain is irreducible and aperiodic. Then, based on the theory of Markov chain, the FSM has an equilibrium state distribution [36]. We define the probability that the current state is S_t $(0 \leq t \leq N-1)$ in the equilibrium (or the probability that the current output is t) to be P_t. Intuitively, P_t is a function of P_X. In the following, we derive a closed form expression of P_t in terms of P_X. This expression is used to synthesize a given target function $T(P_X)$ on P_X.

Based on the theory of Markov chains [36], at the equilibrium stage, the probability of transitioning from the state S_{t-1} to its next state S_t, equals the probability of transitioning from the state S_t to the state S_{t-1}. Thus, we have

$$P_t \cdot (1 - P_X) = P_{t-1} \cdot P_X. \tag{5.1}$$

Because all the individual state probabilities P_t must sum to unity, we have

$$\sum_{t=0}^{N-1} P_t = 1. \tag{5.2}$$

Based on equation (5.1) and (5.2), we obtain

$$P_t = \frac{(\frac{P_X}{1-P_X})^t}{\sum_{i=0}^{N-1}(\frac{P_X}{1-P_X})^i}. \tag{5.3}$$

5.1.2 Synthesis Approach

We use the circuit shown in Fig. 5.1 to synthesize a target function $T(P_X)$. We let X and w_t be stochastic bit streams, and define P_X to be the probability of ones in X, P_{w_t} to be the probability of ones in w_t, and P_Y to be the probability of ones in Y. Based on

the circuit shown in Fig. 5.1, it can be seen that the probability that the "*MUX*" input w_t is selected as its output is P_t, because this probability is the same as the probability that the current state of the FSM is S_t. Thus, we can obtain P_Y as

$$\begin{aligned} P_Y &= P(Y = 1) \\ &= \sum_{t=0}^{N-1} P(Y = 1 \mid w_t \textit{ is selected}) \cdot P(w_t \textit{ is selected}) \\ &= \sum_{t=0}^{N-1} P(w_t = 1) \cdot P(w_t \textit{ is selected}) \\ &= \sum_{t=0}^{N-1} P_{w_t} \cdot P_t. \end{aligned} \tag{5.4}$$

Note that P_Y is a function of P_X and P_{w_t}, because P_t is a function of P_X (refer to (5.3)). It can be seen that equation (5.4) is a closed form expression of P_Y. This expression is used to synthesize the given function $T(P_X)$. We define the approximation error ϵ as

$$\epsilon = \int_0^1 (T(P_X) - P_Y)^2 \cdot d(P_X). \tag{5.5}$$

The synthesis goal is to compute P_{w_t} to minimize ϵ. By expanding (5.5), we can rewrite ϵ as

$$\epsilon = \int_0^1 T(P_X)^2 \cdot d(P_X) - 2\int_0^1 T(P_X) \cdot P_Y \cdot d(P_X) + \int_0^1 P_Y^2 \cdot d(P_X).$$

The first term $\int_0^1 T(P_X)^2 \cdot d(P_X)$ is a constant because $T(P_X)$ is given. Thus minimizing ϵ is equivalent to minimizing the following objective function φ:

$$\varphi = \int_0^1 P_Y^2 \cdot d(P_X) - 2\int_0^1 T(P_X) \cdot P_Y \cdot d(P_X). \tag{5.6}$$

We define a vector $\mathbf{b}$, a vector $\mathbf{c}$, and a matrix $\mathbf{H}$ as follows,

$$\mathbf{b} = [P_{w_0}, P_{w_1}, \cdots, P_{w_{N-1}}]^T,$$

$$\mathbf{c} = \begin{bmatrix} -\int_0^1 T(P_X) \cdot P_0 \cdot d(P_X) \\ -\int_0^1 T(P_X) \cdot P_1 \cdot d(P_X) \\ \vdots \\ -\int_0^1 T(P_X) \cdot P_{N-1} \cdot d(P_X) \end{bmatrix},$$

$$\mathbf{H} = [\mathbf{H_0}, \mathbf{H_1}, \cdots, \mathbf{H_{N-1}}]^T,$$

where P_t $(0 \leq t \leq N-1)$ in vector $\mathbf{c}$ are defined by equation (5.3) and H_t $(0 \leq t \leq N-1)$ in matrix $\mathbf{H}$ is a row vector defined as follows,

$$\mathbf{H_t} = \begin{bmatrix} \int_0^1 P_t \cdot P_0 \cdot d(P_X) \\ \int_0^1 P_t \cdot P_1 \cdot d(P_X) \\ \vdots \\ \int_0^1 P_t \cdot P_{N-1} \cdot d(P_X) \end{bmatrix}^T.$$

Note that (refer to the expression of P_Y in (5.4)),

$$\mathbf{b}^T\mathbf{H}\mathbf{b} = \int_0^1 P_Y^2 \cdot d(P_X),$$

$$\mathbf{c}^T\mathbf{b} = -\int_0^1 T(P_X) \cdot P_Y \cdot d(P_X),$$

Thus, the objective function φ in (5.6) can be rewrite as

$$\varphi = \mathbf{b}^T\mathbf{H}\mathbf{b} + 2\mathbf{c}^T\mathbf{b}. \tag{5.7}$$

We notice that, computing P_{w_t} (i.e., the vector $\mathbf{b}$) to minimize φ in the form of equation (5.7) is a typical constrained quadratic programming problem. This is because P_t is a function of P_X (please refer to (5.3)). The integral of P_t on P_X is a constant, so are the vector $\mathbf{c}$ and the matrix $\mathbf{H}$. Based on (5.7), the solution of P_{w_t} (i.e., the vector $\mathbf{b}$) can be obtained using standard techniques [37]. Based this synthesis approach, if we set the target function to the tanh function in (2.10) or to the exponentiation function in (2.9), we will get the exactly same results proposed by Brown and Card [15].

However, one issue of this single input linear FSM is that it can synthesize only a small number of nonlinear functions. Some important functions, for instance, a Gaussian

distribution function and the functions used in the low-density parity-check (LDPC) decoding [38] cannot be synthesized by it, because the approximation error ϵ is normally greater than 10^{-3}. To solve this issue, a state transition diagram with more degrees of freedom must be developed.

5.2 Two-Input Linear FSM-Based Synthesis Approach

We notice that it is hard to increase the degrees of freedom based on the single input FSM, because the base functions of this topology has only one single variable. Thus, we need to use an FSM with more inputs. We study the FSM which has two inputs, and realized that the topology of the stochastic linear gain function proposed by Brown and Card [15] is a good candidate. However, for synthesis purpose we need to change the state transition diagram of the stochastic linear gain function to the one shown in Fig. 5.3. In this figure, we do not change the topology of the FSM. The only difference between this FSM and the one proposed by Brown and Card [15] is the output configuration. The output of the FSM proposed by Brown and Card [15] has only two values, 0 or 1. The output of the FSM shown in Fig. 5.3 has N different values, where N is the number of states in the FSM. The purpose of this configuration is to detect the current state of the FSM based on its current output.

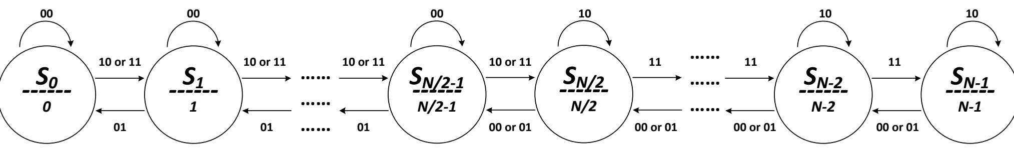


Figure 5.3: The state transition diagram of an N-state Moore style FSM. It has two inputs X and K. The numbers on each arrow represent the transition condition, with the first corresponding to the input X and the second corresponding to the input K. This FSM has $\lceil log_2 N \rceil$ outputs, encoding a value in binary radix. In the figure, the number below each state S_i ($0 \leq i \leq N-1$) represents the output of the FSM when the current state is S_i.

5.2.1 FSM Analysis

As shown in Fig. 5.3, this FSM has two inputs X and K. Its output is the current state number. For example, if the current state of the FSM is S_i $(0 \leq i \leq N-1)$, then the output of the FSM is i. Note that although the output of the FSM looks like an up/down counter, its state transition is quite different from a conventional up/down counter. Given a current state S_i, the next state of the FSM will be

- S_{i+1} if $X = 1$ and $0 \leq i \leq \frac{N}{2} - 1$;
- S_{i-1} if $(X, K) = (0, 1)$ and $1 \leq i \leq \frac{N}{2} - 1$;
- S_{i+1} if $(X, K) = (1, 1)$ and $\frac{N}{2} \leq i \leq N - 2$;
- S_{i-1} if $X = 0$ and $\frac{N}{2} \leq i \leq N - 1$;
- S_i in any other cases.

If the inputs X and K are stochastic bit streams with fixed probabilities, then the random state transition will eventually reach an equilibrium state, where the probability of transitioning from state S_i to its adjacent state S_{i+1}, will equal the probability of transitioning from state S_{i+1} to state S_i. Thus, we have

$$\begin{cases} P_i \cdot P_X = P_{i+1} \cdot (1 - P_X) \cdot P_K, \quad 0 \leq i \leq \frac{N}{2} - 2, \\ \\ P_{\frac{N}{2}-1} \cdot P_X = P_{\frac{N}{2}} \cdot (1 - P_X), \\ \\ P_i \cdot P_X \cdot P_K = P_{i+1} \cdot (1 - P_X), \quad \frac{N}{2} \leq i \leq N - 2, \end{cases} \tag{5.8}$$

where P_i is the probability that the current state is S_i in the equilibrium state (or the probability that the current output is i). P_X and P_K have been already defined in the beginning of this section. Note that the individual state probability P_i must sum to unity over all S_i, i.e.,

$$\sum_{i=0}^{N-1} P_i = 1. \tag{5.9}$$

Using (5.8) and (5.9), we can write P_i in terms of P_X and P_K as

$$P_i = \begin{cases} \frac{(\frac{P_X}{1-P_X})^i \cdot P_K^{-i}}{\alpha}, & 0 \leq i \leq \frac{N}{2} - 1, \\ \\ \frac{(\frac{P_X}{1-P_X})^i \cdot P_K^{i+1-N}}{\alpha}, & \frac{N}{2} \leq i \leq N - 1, \end{cases} \tag{5.10}$$

where α is,

$$\alpha = \sum_{i=0}^{\frac{N}{2}-1} (\frac{P_X}{1-P_X})^i \cdot P_K^{-i} + \sum_{i=\frac{N}{2}}^{N-1} (\frac{P_X}{1-P_X})^i \cdot P_K^{i+1-N}.$$

5.2.2 Synthesis Approach

To synthesize the given target function $T(P_X)$, we use a circuit shown in Fig. 5.4, which is very similar to the one shown in Fig. 5.1.

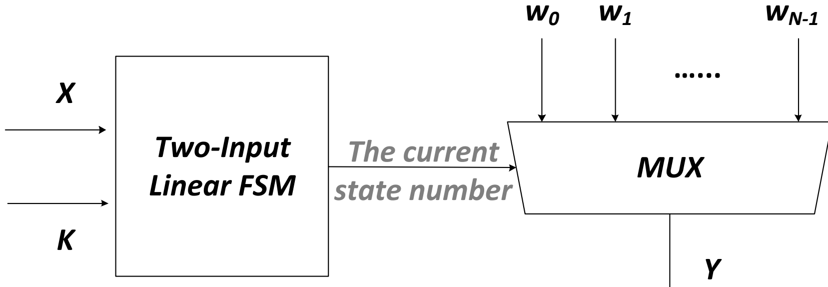


Figure 5.4: The circuit for synthesizing target functions.

In Fig. 5.4, the state transition diagram of the "*Two-Input Linear FSM*" has been shown in Fig. 5.3. Its inputs are X and K. Its output is the current state number, which is connected to the selection inputs of the multiplexer "*MUX*", which has N data inputs (w_0, w_1, $\cdots$, w_{N-1}). Note that if the current state of the FSM is S_t $(0 \leq t \leq N-1)$, then the channel that connects w_t to Y will be selected in the "*MUX*."

We let X, K, and w_t be stochastic bit streams, and define P_X to be the probability of ones in X, P_K to be the probability of ones in K, P_{w_t} to be the probability of ones in w_t, and P_Y to be the probability of ones in Y. Based on the circuit shown in Fig. 5.4, it can be seen that the probability that the "*MUX*" input w_t is selected as its output is

P_t, because this probability is the same as the probability that the current state of the FSM is S_t. Thus, we can obtain P_Y as

$$\begin{aligned} P_Y &= P(Y = 1) \\ &= \sum_{t=0}^{N-1} P(Y = 1 \mid w_t \ is \ selected) \cdot P(w_t \ is \ selected) \\ &= \sum_{t=0}^{N-1} P(w_t = 1) \cdot P(w_t \ is \ selected) \\ &= \sum_{t=0}^{N-1} P_{w_t} \cdot P_t. \end{aligned} \tag{5.11}$$

Note that P_Y is a function of P_X, P_K, and P_{w_t}, because P_t is a function of P_X and P_K (refer to (5.10)). It can be seen that equation (5.11) is a closed form expression of P_Y. This expression is used to synthesize the given function $T(P_X)$, and the synthesis goal is to compute P_{w_t} and P_K to minimize ϵ, which has been defined in (5.5). We noticed that if P_K is set to a constant, P_{w_t} can be obtained using the same method introduced in Section 5.1.2. To compute P_K, we needs to use a numerical approach. More specifically, we first set P_K to 0.001, and compute the corresponding P_{w_t} and ϵ. Next, we set P_K to 0.002, and compute the corresponding P_{w_t} and ϵ. So on and so forth. Finally, we set P_K to 1, and compute the corresponding P_{w_t} and ϵ. Among these 1000 results of ϵ, we select the minimum one, and the corresponding P_K and P_{w_t}. After we get the optimal values of P_K and P_{w_t}, the stochastic bit streams K and w_t in Fig. 5.4 will be generated to implement the target function $T(P_X)$ stochastically. The two-input linear FSM can be used to synthesize more functions such as high order polynomials and other non-polynomials.

5.2.3 Synthesis Examples

Example 1: Synthesizing Gaussian distribution function:

$$T(x) = \frac{1}{\delta\sqrt{2\pi}} e^{-\frac{(x-\mu)^2}{2\delta^2}}, \quad (-1 \le x \le 1).$$

Because x could be negative values, we need to convert x to P_X using bipolar coding,

i.e., we set $P_X = 0.5(x+1)$, and rewrite the target function as

$$T(P_X) = \frac{1}{\delta\sqrt{2\pi}} e^{-\frac{(2P_X-1-\mu)^2}{2\delta^2}}, \quad (0 \le P_X \le 1). \tag{5.12}$$

Note that in (5.12), δ could be any value as long as $\delta \ge \frac{1}{\sqrt{2\pi}}$, because in computation on stochastic bit streams, the output is a probability value that should be greater than or equal to 0 and less than or equal to 1 and when $\delta \ge \frac{1}{\sqrt{2\pi}}$, we guarantee that the maximal value of the function $T(P_X)$ is less than or equal to 1. We normally set $\mu = 0$ for simplicity, because using different value of μ will only shift the curve along on the x-axis instead of changing the shape of the curve. If we set $\delta = 2$ and $\mu = 0$, for example, we compute P_K and P_{w_t} using the synthesis approach discussed in the previous section. The results are listed in Table 5.1. The approximation error ϵ (defined in (5.5)) is 4.9×10^{-7}. Fig. 5.5 shows the simulation result.

Table 5.1: P_K and P_{w_t} $(0 \le t \le 7)$ for synthesizing the target function in (5.12) with $\delta = 2$ and $\mu = 0$.

$P_K = 0.56$			
$P_{w_0} = 0.11$	$P_{w_1} = 0.64$	$P_{w_2} = 1$	$P_{w_3} = 0.87$
$P_{w_4} = 0.87$	$P_{w_5} = 1$	$P_{w_6} = 0.64$	$P_{w_7} = 0.11$

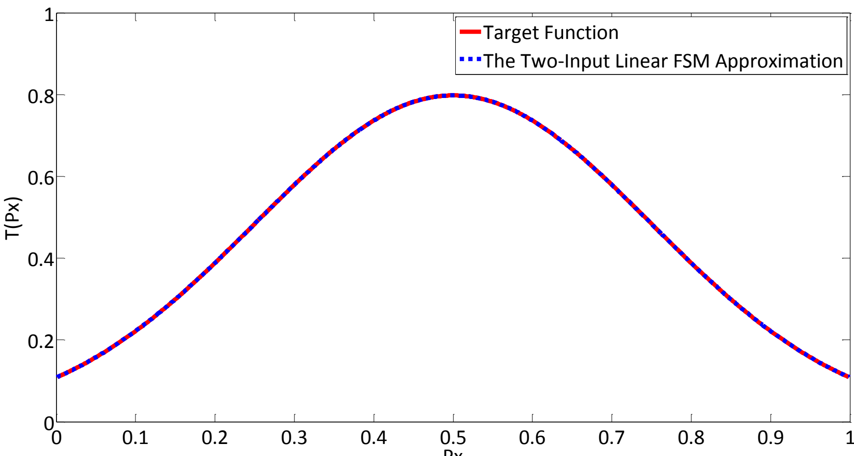


Figure 5.5: Synthesis result of the target function in (5.12) with $\delta = 2$ and $\mu = 0$.

This means that, using the circuit shown in Fig. 5.4, if the probability of ones in the input bit stream K equals 0.56 and the probabilities of ones in the input bit streams w_t equal P_{w_t} shown in Table 5.1, the probability of ones in the output bit stream Y will be

$$\frac{1}{2\sqrt{2\pi}}e^{-\frac{(2P_X-1)^2}{8}}, \quad (0 \le P_X \le 1),$$

where P_X is the probability of ones in the input X. Note that if P_{w_t} equal 0 or 1, then w_t can be set to a constant '0' or '1' correspondingly, and the hardware implementation can be further simplified. In addition, if $0 < w_t < 1$, the corresponding bit streams can be generated with extremely low cost using the technique proposed by Qian et al. [39].

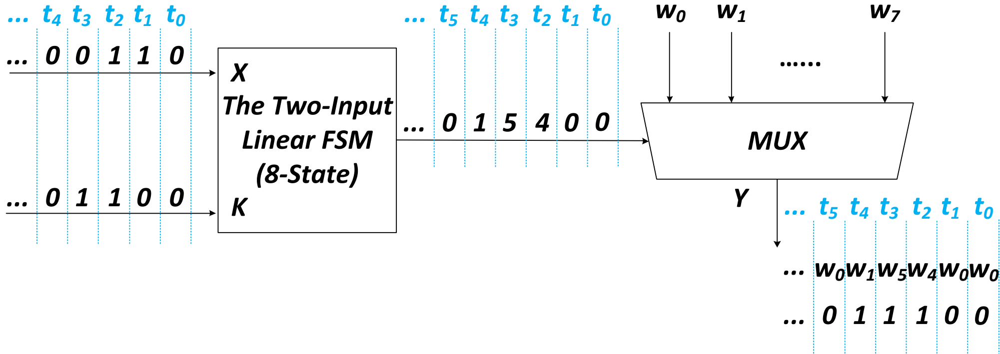


Figure 5.6: An example about how the circuit shown in Fig. 5.4 works (the FSM is implemented with the state transition diagram shown in Fig. 5.3).

We give an example in Fig. 5.6 to show how this circuit works. Assume that the circuit starts working at clock cycle t_0 and the initial state of the FSM is S_0 (please note that the initial state has no influence on the final results, it could be any one of the 8 states). The output of the FSM at t_0 is 0 (because its initial state is S_0) and the output of the multiplexer "*MUX*" at t_0 is w_0 (because its selection input equals 0 at t_0), and $w_0 = 0$ based on Table 5.1.

Because at t_0, $X = K = 0$ and the initial state is S_0, in the next clock cycle t_1, the current state of the FSM is still S_0 (and the output of the FSM is still 0) based on the state transition diagram shown in Fig. 5.3. The output of the "*MUX*" at t_1 is still w_0, which equals 0 based on Table 5.1.

In the next clock cycle t_2, based on the state transition diagram shown in Fig. 5.3, the current state becomes S_1 because $X = 1$ $K = 0$ at the previous clock cycle t_1, and the output of the FSM is 1. Thus, the output of the "MUX" at t_2 becomes the current input at w_1.

So on and so forth. Assume that we use 1024 bits to represent a value stochastically. After 1024 clock cycles, if the probability of ones in X equals P_X, and the probability of ones in K equals 0.56, then the probability of ones in Y will be $\frac{1}{2\sqrt{2\pi}}e^{-\frac{(2P_X-1)^2}{8}}$. ■

Example 2: Synthesizing the following high order polynomial $\lambda(x)$ used in low-density parity-check coding [40]:

$$\lambda(x) = 0.1575x + 0.3429x^2 + 0.0363x^5 + 0.059x^6 \\ + 0.279x^8 + 0.1253x^9, \quad (0 \le x \le 1).$$

For this example, we use unipolar coding because we do not deal with negative values and x is in the unitary range. We rewrite the target function in terms of P_X ($P_X = x$) as

$$T(P_X) = 0.1575P_X + 0.3429P_X^2 + 0.0363P_X^5 + 0.059P_X^6 \\ + 0.279P_X^8 + 0.1253P_X^9, \quad (0 \le P_X \le 1). \tag{5.13}$$

We compute P_K and P_{w_t} using the proposed synthesis approach and show the results in Table 5.2. The approximation error ϵ (defined in (5.5)) is 1.1×10^{-7}. Fig. 5.7 shows the simulation result. ■

Table 5.2: P_K and P_{w_t} for synthesizing the target function in (5.13).

$P_K = 0.1875$			
$P_{w_0} = 0$	$P_{w_1} = 0$	$P_{w_2} = 0$	$P_{w_3} = 0$
$P_{w_4} = 0.2$	$P_{w_5} = 0.8$	$P_{w_6} = 0.1$	$P_{w_7} = 1$

It can be seen that by using the two-input linear FSM, more nonlinear functions can be synthesized stochastically. However, this FSM can still not synthesize all the nonlinear functions. For example, the function $\phi(x)$ used in low-density parity-check decoding [41],

$$\phi(x) = log\frac{e^x + 1}{e^x - 1}, \quad (x > 0), \tag{5.14}$$

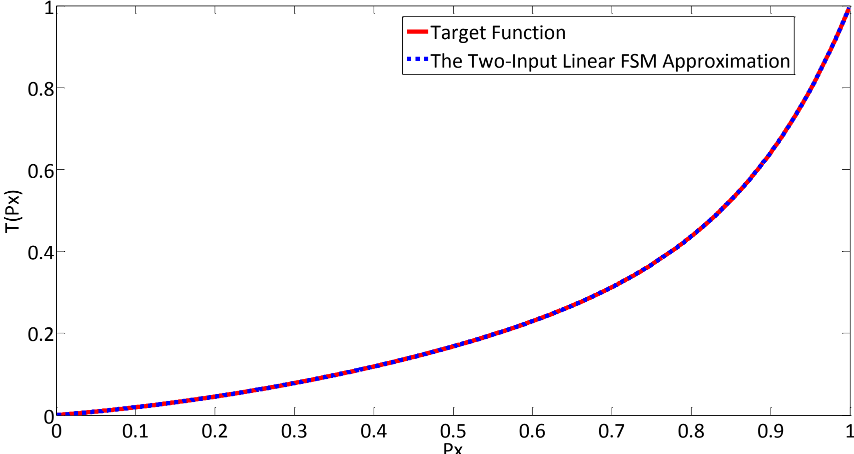


Figure 5.7: Synthesis result of the target function in (5.13).

cannot be synthesized using the two-input linear FSM, because the approximation error ϵ is normally greater than 10^{-3}. To solve this issue, another state transition diagram with more degrees of freedom must be developed.

5.3 Two-Input Two-Dimension FSM-Based Synthesis Approach

To increase the degrees of freedom of the state transition diagram shown in Fig. 5.3, an intuitive solution is to update the two-input linear FSM into a two-dimensional structure shown in Fig. 5.8. It has two inputs, which are X and K. The FSM has in total $M \times N$ states, arranged as an $M \times N$ two-dimensional array. We normally set $M \times N = 2^R$, where R is a positive integer, because we can implement an FSM with 2^R states by R D flip-flops (DFFs). In addition, we set $M = 2^{\lfloor \frac{R}{2} \rfloor}$, and $N = 2^{\lceil \frac{R}{2} \rceil}$. The FSM shown in Fig. 5.8 is a Moore-style machine, which has $log_2\lceil MN \rceil$ outputs encoding an integer in the range $[0, MN-1]$ using binary radix. If the current state is S_t $(0 \leq t \leq MN-1)$, the value encoded by its output is just the current state number t. In short, we say that the output of the FSM is t. It can be seen that the output configuration of the FSM looks like an up/down counter. However, the state transitions of this FSM are quite

different from a conventional up/down counter. For example, if we assume that the current state is S_{N+1} in Fig. 5.8, its next state and the corresponding output will be: S_{N+2} and $N+2$, if $(X, K) = (1, 1)$; S_N and N, if $(X, K) = (0, 0)$; S_{2N+1} and $2N+1$, if $(X, K) = (1, 0)$; S_2 and 2, if $(X, K) = (0, 1)$. An example of such an FSM with 8 states is shown in Fig. 5.9.

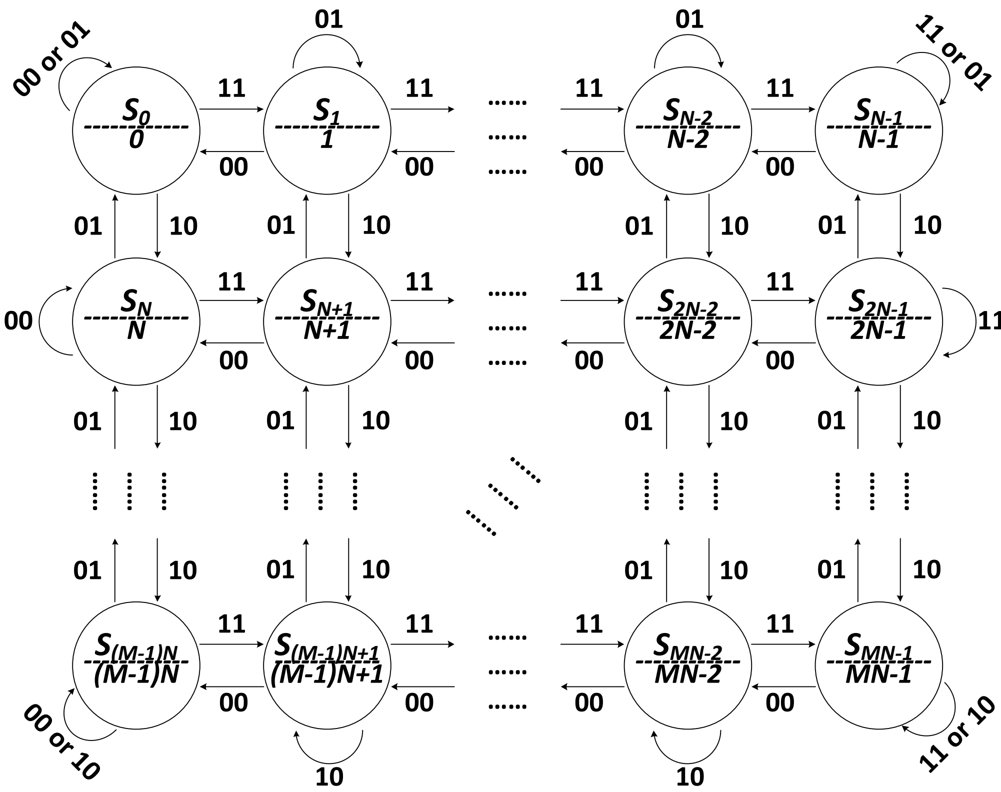


Figure 5.8: The FSM has two inputs X and K. The numbers on each arrow represent the transition condition, with the first corresponding to the input X and the second corresponding to the input K. The FSM has $log_2\lceil MN \rceil$ outputs, encoding a value in binary radix. In the figure, the number below 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.

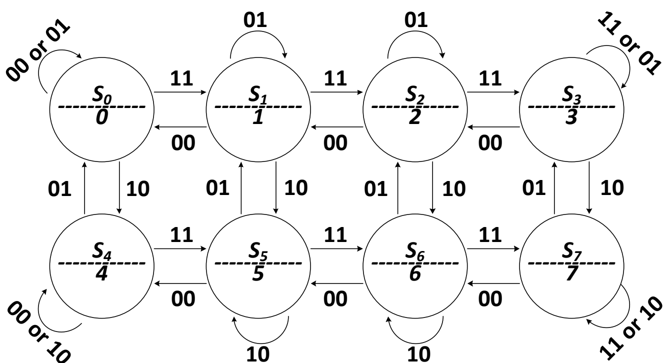


Figure 5.9: An example of the state transition digram of the proposed FSM with 8 states. Here, $M = 2$ and $N = 4$.

5.3.1 FSM Analysis

The inputs X and K of this FSM consist of stochastic bit streams. We define the probability that each bit in the input stream X is one to be P_X, and the probability that each bit in the input stream K is one to be P_K. Because both inputs X and K are stochastic bit streams with fixed probabilities, the state transition process of the FSM can be modeled as a time-homogeneous Markov chain. It can be shown that this Markov chain is irreducible and aperiodic. Then, based on the theory of Markov chain, the FSM has an equilibrium state distribution [36]. We define the probability that the current state is S_t ($0 \leq t \leq MN - 1$) in the equilibrium (or the probability that the current output is t) to be P_t. Intuitively, P_t is a function of both P_X and P_K. In the following, we derive a closed form expression of P_t in terms of P_X and P_K. This expression is used to synthesize a given target function $T(P_X)$ on P_X. The synthesis details will be discussed in the next section.

In the following discussion, we define $i = \lfloor \frac{t}{N} \rfloor$ and $j = t$ modulo N, i.e., i and j are the quotient and the remainder of t divided by N, respectively (or $t = i \times N + j$). Based on the theory of Markov chains [36], at the equilibrium stage, the probability of transitioning from the state $S_{i\times N+j}$ to its horizontal adjacent state $S_{i\times N+j-1}$, equals the probability of transitioning from the state $S_{i\times N+j-1}$ to the state $S_{i\times N+j}$. Thus, we

have

$$P_{i\times N+j}\cdot(1-P_X)\cdot(1-P_K)=P_{i\times N+j-1}\cdot P_X\cdot P_K. \tag{5.15}$$

In addition, the probability of transitioning from the state $S_{i\times N+j}$ to its vertical adjacent state, $S_{(i-1)\times N+j}$, equals the probability of transitioning from the state $S_{(i-1)\times N+j}$ to the state $S_{i\times N+j}$:

$$P_{i\times N+j}\cdot(1-P_X)\cdot P_K=P_{(i-1)\times N+j}\cdot P_X\cdot(1-P_K). \tag{5.16}$$

Because all the individual state probabilities $P_{i\times N+j}$ (or P_t) must sum to unity, we have

$$\sum_{t=0}^{MN-1}P_t=\sum_{i=0}^{M-1}\sum_{j=0}^{N-1}P_{i\times N+j}=1. \tag{5.17}$$

Based on equation (5.15), (5.16), and (5.17), we obtain

$$P_t=P_{i\times N+j}=\frac{t_x^i\cdot t_y^j}{\sum_{u=0}^{M-1}\sum_{v=0}^{N-1}t_x^u\cdot t_y^v}, \tag{5.18}$$

where t_x and t_y are,

$$t_x=\frac{P_X}{1-P_X}\cdot\frac{P_K}{1-P_K},$$
$$t_y=\frac{P_X}{1-P_X}\cdot\frac{1-P_K}{P_K}.$$

It can be seen that equation (5.18) is a closed form expression of P_t in terms of P_X and P_K.

5.3.2 Synthesis Approach

To synthesize the given target function $T(P_X)$, we use a circuit shown in Fig. 5.10, which is very similar to the ones shown in Fig. 5.1 and Fig. 5.4.

In Fig. 5.10, the state transition diagram of the "*Two-Input Mesh FSM*" has been shown in Fig. 5.8. Its inputs are X and K. Its output is the current state number, which is connected to the selection inputs of the multiplexer "*MUX*", which has $M\times N$ data inputs (w_0, w_1, $\cdots$, w_{MN-1}). Note that if the current state of the FSM is S_t

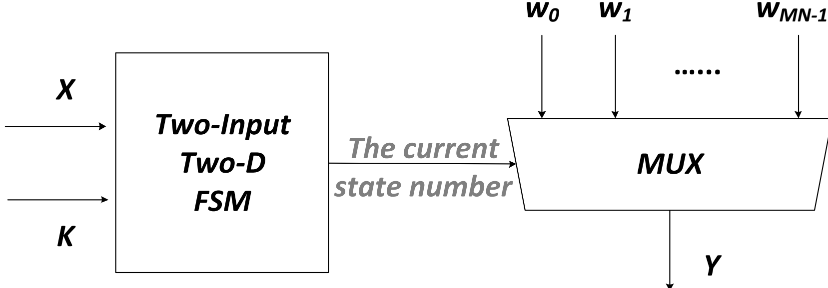


Figure 5.10: The circuit for synthesizing target functions.

$(0 \leq t \leq MN-1)$, then the channel that connects w_t to Y will be selected in the "MUX."

We let X, K, and w_t be stochastic bit streams, and define P_X to be the probability of ones in X, P_K to be the probability of ones in K, P_{w_t} to be the probability of ones in w_t, and P_Y to be the probability of ones in Y. Based on the circuit shown in Fig. 5.10, it can be seen that the probability that the "MUX" input w_t is selected as its output is P_t, because this probability is the same as the probability that the current state of the FSM is S_t. Thus, we can obtain P_Y as

$$\begin{aligned}
P_Y &= P(Y=1) \\
&= \sum_{t=0}^{MN-1} P(Y=1 \mid w_t \textit{ is selected}) \cdot P(w_t \textit{ is selected}) \\
&= \sum_{t=0}^{MN-1} P(w_t=1) \cdot P(w_t \textit{ is selected}) \\
&= \sum_{t=0}^{MN-1} P_{w_t} \cdot P_t.
\end{aligned} \tag{5.19}$$

Note that P_Y is a function of P_X, P_K, and P_{w_t}, because P_t is a function of P_X and P_K (refer to (5.18)). It can be seen that equation (5.19) is a closed form expression of P_Y. This expression is used to synthesize the given function $T(P_X)$, and the synthesis goal is to compute P_{w_t} and P_K to minimize ϵ, which has been defined in (5.5). We noticed that if P_K is set to a constant, P_{w_t} can be obtained using the same method introduced in

Section 5.1.2. To compute P_K, we needs to use a numerical approach. More specifically, we first set P_K to 0.001, and compute the corresponding P_{w_t} and ϵ. Next, we set P_K to 0.002, and compute the corresponding P_{w_t} and ϵ. So on and so forth. Finally, we set P_K to 1, and compute the corresponding P_{w_t} and ϵ. Among these 1000 results of ϵ, we select the minimum one, and the corresponding P_K and P_{w_t}. After we get the optimal values of P_K and P_{w_t}, the stochastic bit streams K and w_t in Fig. 5.10 will be generated to implement the target function $T(P_X)$ stochastically. The two-input FSM can be used to synthesize more functions such as high order polynomials and other non-polynomials.

5.3.3 Synthesis Examples

Example: Synthesizing the function $\phi(x)$ used in low-density parity-check decoding [41]:

$$\phi(x) = log\frac{e^x + 1}{e^x - 1}, \quad (x > 0).$$

For this example, we use unipolar coding because we do not deal with negative values, and set $P_X = x/\alpha$, where α is a scaling factor to map the range of x to unitary. We rewrite the target function in terms of P_X as

$$T(P_X) = log\frac{e^{\alpha P_X} + 1}{e^{\alpha P_X} - 1}, \quad (0 < P_X \leq 1). \tag{5.20}$$

Table 5.3: P_K and P_{w_t} for synthesizing the target function in (5.20) with $\alpha = 30$.

$P_K = 0.9688$			
$P_{w_0} = 1$	$P_{w_1} = 0.5$	$P_{w_2} = 0$	$P_{w_3} = 0$
$P_{w_4} = 0$	$P_{w_5} = 0$	$P_{w_6} = 0$	$P_{w_7} = 0$

If we set $\alpha = 30$, for example, we compute P_K and P_{w_t} using the proposed synthesis approach and show the results in Table 5.3. The approximation error ϵ (defined in (5.5)) is 9.7×10^{-4}. Fig. 5.11 shows the simulation result. ■

5.4 Comparison of Different Synthesis Approaches

In this section, we compare hardware cost of the single input FSM, the two-input linear FSM, and the two-input two-dimension FSM introduced in this paper. In addition, we

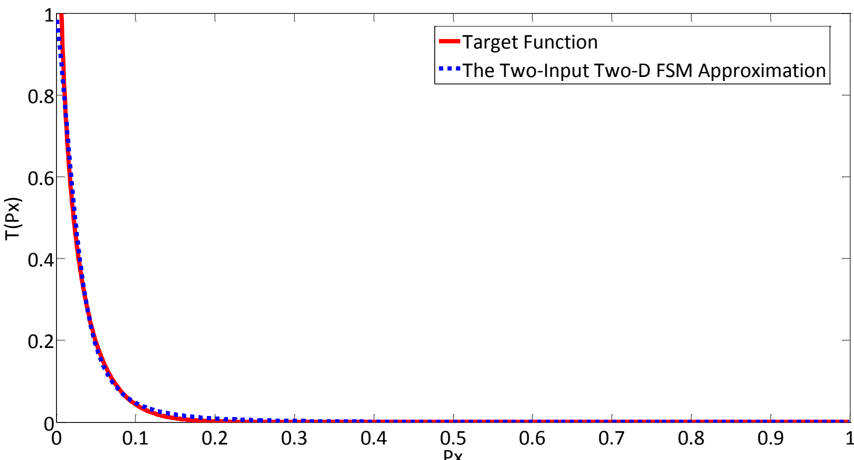


Figure 5.11: Synthesis result of the target function in (5.20) with $\alpha = 30$.

compare the proposed FSM-based synthesis approach to the Bernstein polynomial-based synthesis approach.

5.4.1 Hardware Cost of Different FSM Topologies

We use Synopsys Design Compiler to evaluate the hardware cost of the different FSM topologies in terms of silicon area based on FreePDK45 standard cell library [42]. Table 5.4 shows the evaluation results.

Table 5.4: Hardware area (um^2) of three different FSMs.

	8-state	16-state	32-state
Single Input Linear FSM	60.07	115.92	154.87
Two-Input Linear FSM	111.69	164.72	275.95
Two-Input Two-Dimension FSM	68.52	97.61	126.71

It can be seen that when we use 8-state FSMs, the single input linear FSM has the least hardware area. When we use 16-state and 32-state FSMs, the two-input two-dimension FSM has the least hardware area. However, the two-input two-dimension FSM needs an additional input compared to the single input linear FSM. Since the

interface circuit to generate the additional input is normally implemented using 10-bit linear feedback shift register, it has more area than the FSM itself. Thus, the most area-efficient topology is the single input linear FSM. But the issue of this single input linear FSM is that the functions can be synthesized stochastically using this FSM are very limited.

Based on the results shown in Table 5.4, for the two-input FSMs, the two-dimension topology has less hardware area than the linear topology. In addition, as we discussed before, it can synthesize more functions than the linear topology, such as the $\phi(x)$ function in (5.14) used in low-density parity-check decoding [41]. Thus, if the target function can be synthesized using the single input linear FSM, it has the most area-efficiency. Otherwise, one should use the two-input two-dimension FSM to synthesize the target function.

5.4.2 Comparison with the Bernstein Polynomial-based Approach

The Bernstein polynomial-based approach uses combinational logic (an adder and a multiplexer, as shown in Fig. 5.12) to perform computation on stochastic bit streams. Hardware area required by this approach depends on the degree of polynomial. Table 5.5 lists its area in terms of the number of fan-in two logic gates [1].

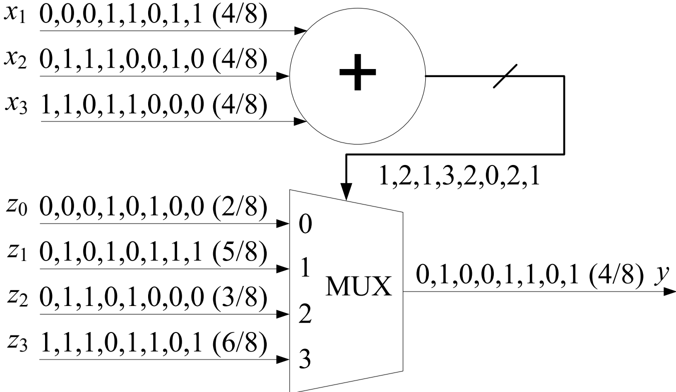


Figure 5.12: Logical computation on stochastic bit streams implementing the Bernstein polynomial $f(x) = \frac{2}{8}B_{0,3}(x) + \frac{5}{8}B_{1,3}(x) + \frac{3}{8}B_{2,3}(x) + \frac{6}{8}B_{3,3}(x)$ at $x = 0.5$. Stochastic bit streams x_1, x_2 and x_3 encode the value $x = 0.5$. Stochastic bit streams z_0, z_1, z_2 and z_3 encode the corresponding Bernstein coefficients [1].

Table 5.5: The number of the fan-in two logic gates for computing Bernstein polynomials of degree 3, 4, 5, and 6 [1].

Degree n	3	4	5	6
Number of Gates	22	40	49	58

In this section, we compare the Bernstein polynomial-based approach to the two-input two-dimension FSM-based approach. Note that by using the 8-state two-input FSM we can synthesize a polynomial of degree up to 9 (refer to **Example 2** in Section 5.2.3). In addition, for those non-polynomials, such as the target functions introduced in this chapter, the Bernstein polynomial-based approach normally takes at least degree 6 to obtain the same level of approximation error. The 8-state two-input two-dimension FSM can be implemented using 3 D-flip-flops (DFFs) as follows,

$$\begin{aligned}
D_2 &= XKQ_0 + XQ_1 + KQ_1 + Q_1Q_0, \\
D_1 &= X\bar{K} + XQ_2 + \bar{K}Q_2, \\
D_0 &= \bar{X}\bar{K}Q_1\bar{Q}_0 + \bar{X}KQ_0 + XKQ_1 + X\bar{K}Q_0 + XKQ_0,
\end{aligned}$$

where D_0, D_1, and D_2 are the inputs of the three DFFs, and Q_0, Q_1, and Q_2 are the corresponding outputs. Because it is a Moore FSM, we assign $Q_2Q_1Q_0 = 000$ for state S_0, $Q_2Q_1Q_0 = 001$ for state S_1, $\cdots$, and $Q_2Q_1Q_0 = 111$ for state S_7. Based on the report of the logic synthesis tool Synopsys Design Compiler, the entire circuit (including the multiplexer in Fig. 5.10) can be implemented using 45 fan-in two logic gates. Please note that the evaluation is based on a generalized version of the circuit shown in Fig. 5.10, if w_t is set to a constant '0' or '1', the circuit can be further simplified and the number of logic gates can be further reduced. It can be seen that, to synthesize non-polynomials and polynomials of degree greater than 4, the two-input two-dimension FSM takes less hardware.

In terms of performance, because both techniques compute on stochastic bit streams, they have equivalent processing time. In terms of energy consumption, we assume that given a CMOS technology, a digital circuit consumes a constant power dissipation per unit area. We use the product of area and processing time as a metric of the energy consumption [15]. Because these two techniques have equivalent processing time, the

FSM-based approach consumes less energy when computing non-polynomials and high order polynomials with degree larger than 4.

In terms of fault-tolerance, we compare the two techniques when the input data is corrupted with noise. We evaluate the fault-tolerant performance on circuits implementing the target functions introduced in the last section and other functions such as trigonometric functions ($sin(x)$, $cos(x)$, and $tan(x)$) and logarithmic functions ($y = log_2(x)$, $y = log_{10}(x)$, and $y = ln(x)$). The length of the stochastic bit streams which are used to represent a value is set to 1024. We define the error ratio γ as the percentage of random bit flips that occur in the computation. We choose the error ratio γ to be 0%, 0.5%, 1%, 5%, and 10%. For example, under 10% error ratio, 102 of 1024 bits will be flipped in the computation. To measure the impact of the noise, we evaluated each target function at 13 distinct input data points: 0.2, 0.25, 0.3, $\cdots$, 0.8. For each error ratio γ, each target function, and each evaluation point, we simulated both the FSM-based implementation and the Bernstein polynomial-based implementation 1000 times. We averaged the relative errors over all simulations. Finally, for each error ratio γ, we averaged the relative errors over all target functions and all evaluation points. Table 5.6 shows the average relative error of the two different implementations versus different γ values. It can be seen that these two techniques has almost equivalent fault-tolernace (the difference is less than 0.5%), because both techniques perform computation on stochastic bit streams.

Table 5.6: Relative error of the FSM-based and the Bernstein polynomial-based implementations of target function computation versus the error ratio γ in the input data.

Error Ratio γ (%)	0	0.5	1	5	10
Relative Error of the FSM (%)	2.26	2.78	3.16	6.75	11.2
Relative Error of Bernstein (%)	2.21	2.72	3.36	6.25	11.7

Chapter 6

Other Encoding Schemes

Besides the stochastic encoding scheme introduced in the previous chapters and the conventional binary radix encoding scheme, we believe there is a spectrum of encodings in computing systems. At one end of this spectrum is the binary encoding. At the other end is the stochastic encoding. Each encoding has its advantages and disadvantages. The best encoding scheme could be different for different applications. This chapter introduces another two encoding schemes. This first on is called *equally weighted non-stochastic encoding*, and the second one is called *overlapped stochastic encoding* [24].

6.1 Equally Weighted Non-stochastic Encoding

The goal of this encoding scheme is to eliminate the random fluctuations in the stochastic encoding by custom designing the positions of ones in the bit stream. For example, we can use an 8-bit stream "00011111" to represent "5/8" by putting all the ones on the right side, and all the zeros the left side. This is actually the basic idea of the proposed encoding. As shown in Fig. 6.1, we use an N-bit stream to represent a value $\frac{L}{N}$ in the range $[0, 1]$. In this bit stream, the left $N - L$ bits are set to zero, and the right L bits are set to one. Based on this encoding, some functions can be implemented using much simpler logic than the ones based on the stochastic and the binary encoding. We will introduce each of these computing elements in the following subsections.

N-L L
0, 0, …, 0, 1, 1, …, 1

Figure 6.1: Representing the value $\frac{L}{N}$ using the proposed encoding scheme.

N-L1 L1 N-max(L1,L2) |L1-L2| min(L1,L2)
A 0, 0, …, 0, 1, 1, …, 1
B 0, 0, …, 0, 1, 1, …, 1
XOR
0, 0, …, 0, 1, 1, …, 1, 0, 0, …, 0 C
N-L2 L2

Figure 6.2: Computing absolute values using a single XOR gate based on the proposed encoding.

6.1.1 Computing Absolute Values

We can use a single XOR gate to compute the absolute value of the difference between two inputs using the proposed encoding. As shown in Fig. 6.2, L_1 is the number of ones in the input A or the XOR gate, and L_2 is the number of ones in the input B. If $L_1 \geq L_2$, the output C (from left to right) will have $N - L_1$ zeros, $L_1 - L_2$ ones, and L_2 zeros. If $L_1 < L_2$, the output C (from left to right) will have $N - L_2$ zeros, $L_2 - L_1$ ones, and L_1 zeros. Thus, in either case, the number of ones in the output C is simply $|L_1 - L_2|$.

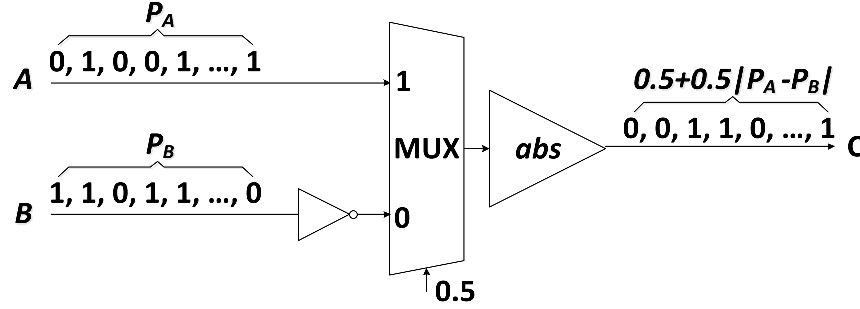


Figure 6.3: Computing absolute values using the stochastic encoding.

In the stochastic encoding, the same function is implemented using the block diagram shown in Fig. 6.3, which consists of a multiplexer (MUX), a NOT gate, and a stochastic abs function [6]. P_A is the probability of ones in the input A, and P_B is the probability of ones in the input B. The probability of ones in the output C is $0.5+0.5 \cdot |P_A - P_B|$. The stochastic abs function is implemented using an FSM with the state transition diagram shown in Fig. 3.6. This FSM has N states, which are S_0, S_1, $\cdots$, S_{N-1}. The input of the FSM is X, and the output of the FSM is Y. It can be seen that the stochastic encoding-based implementation for computing absolute values requires more hardware and has output errors due to the random fluctuations. For example, an 8-state FSM implementation of this function requires at least 21 logic gates, and has 2% - 5% output errors [6].

6.1.2 Making Comparisons

As shown in Fig. 6.4, we can compute the minimum value of n inputs using a single n-input AND gate. Based on the proposed encoding and the logic function of the AND gate, the number of ones in the output is the same as the number of ones in the input which has the least number of ones, i.e., the number of ones in the output C is $min(L_1, L_2, ..., L_n)$.

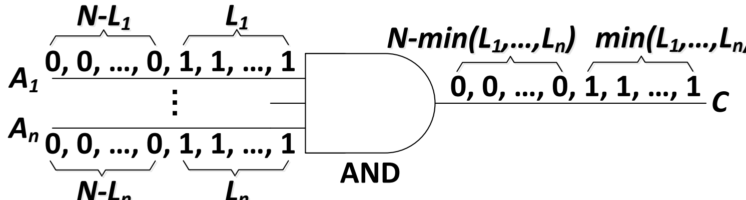


Figure 6.4: Computing the minimum value using an n-input AND gate.

As shown in Fig. 6.5, we can compute the maximum value of n inputs using a single n-input OR gate. Based on the proposed encoding and the logic function of the OR gate, the number of ones in the output is the same as the number of ones in the input which has the most ones, i.e., the number of ones in the output C is $max(L_1, L_2, ..., L_n)$.

In addition, by using the AND and OR gates, we can implement a comparator as shown in Fig. 6.6 to compute the maximum and minimum values of the n input values.

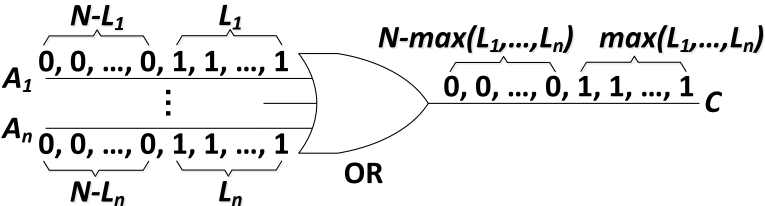


Figure 6.5: Computing the maximum value using an n-input OR gate.

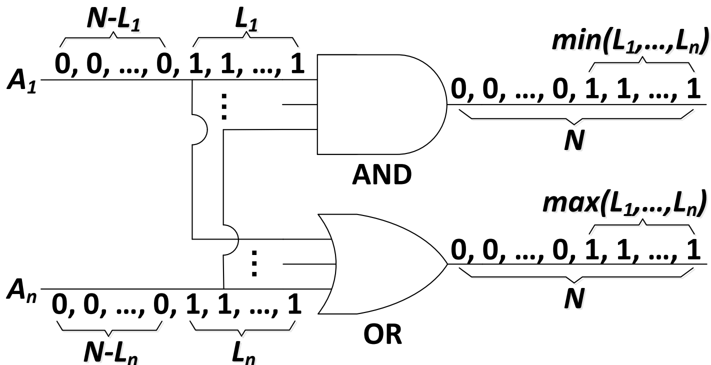


Figure 6.6: Comparator based on the proposed encoding.

In the stochastic encoding, this comparator is implemented using the block diagram shown in Fig. 3.3, which consists of three MUXs, a NOT gate, and a stochastic tanh function [6]. P_A is the probability of ones in the input A, and P_B is the probability of ones in the input B. The outputs of this comparator are C and D. The probability of ones in the output C is $min(P_A, P_B)$, and the probability of ones in the output D is $max(P_A, P_B)$. The stochastic tanh function used in this comparator is implemented using an FSM with the state transition diagram shown in Fig. 2.9. It can be seen that the stochastic encoding-based comparator requires more hardware than the one using the proposed encoding. It also has output errors due to the random fluctuations. For example, a 32-state FSM implementation of this function requires at least 39 logic gates, and has 2% - 3% output errors [6].

6.2 Overlapped Stochastic Encoding

We noticed that, although the stochastic encoding needs a long bit stream to represent a deterministic value, we can actually share parts of the bits to represent consecutive deterministic values. For example, assume that we have four sequential deterministic inputs $X_0 = 0.5$, $X_1 = 0.3$, $X_2 = 0.4$, and $X_3 = 0.6$, and we use a 10-bit stream to represent each value stochastically. In the conventional stochastic encoding, we need 40 bits to represent these 4 values. However, let us take a look at the example shown in Fig. 6.7.

$X_0 = 0.5$ $X_2 = 0.4$

0,1,1,0,1,0,1,0,1,0,0,0,1,0,0,0,1,1,0,1,1,0,1,1,0

$X_1 = 0.3$ $X_3 = 0.6$

Figure 6.7: An example of sharing 50% of bits in the consecutive bit stream to represent the deterministic value stochastically.

In Fig. 6.7, we generate 10 bits to represent the first value X_0. When the next value X_1 comes, we only generate another 5 new bits, and share the last 5 bits of the bit stream which represents X_0. When X_2 comes, we also only generate another 5 new bits, and

share the last 5 bits of the bit stream which represents X_1. And so on. Thus, by sharing bits from the previous bit streams, we can represent the 4 values using only 25 bits. In addition, each value is still represented using a 10-bit stream without any precision degradation. Based on this example, we see a 37.5% performance improvement. In the next section, we discuss this method in detail.

6.2.1 Sharing Bits based on Fixed Length

In the example shown in Fig. 6.7, we actually share 50% of the bits from the consecutive bit streams. We call this method sharing bits based on fixed length, because every bit stream shares the same number of bits. Assume that we share $p\%$ $(0 \leq p < 100)$ bits from the previous bit stream, the performance will be improved by

$$1 - \frac{L + (n-1) \times (1 - p\%) \times L}{n \times L} = \frac{n-1}{n} \times p\%,$$

where n is the number of consecutive deterministic input values in the entire computation, and L is the length of the bit stream that represents each deterministic value.

Now the question is how to generate new bits. In the following discussions, we define the current deterministic value to be X_t, the previous deterministic value to be X_{t-1}, and the deterministic value for generating new bits to be X. By sharing $p\%$ of the bits from the previous bit stream which represents X_{t-1}, we only need to generate $(1-p)\% \times L$ new bits to represent X_t. Note that we still use an L-bit stream to represent X_t. In these L bits, $p\% \times L$ bits are from the bit stream which represents X_{t-1}, and the other $(1-p)\% \times L$ bits are generated using X. The goal is that the shared bits and the new generated bits together should represent X_t as closely as possible. In other words, we have

$$\frac{X \times (1-p)\% \times L + X_{t-1} \times p\% \times L}{L} = X_t,$$

i.e.,

$$X = \frac{X_t - X_{t-1} \times p\%}{1 - p\%}. \tag{6.1}$$

Based on Eq. 6.1, we actually have a limitation, which is

$$X_t - X_{t-1} \times p\% \geq 0. \tag{6.2}$$

Otherwise, the results will be wrong. Thus, this method has a trade-off: using a larger p, we share more bits and have better performance, but this introduces more limitations on the values of X_t and X_{t-1}, i.e., X_t must not be greater than $X_{t-1} \times p\%$; using a smaller p, we have less performance improvement, but we also have less limitations on the values of X_t and X_{t-1}. For example, if $p = 0$, i.e., we use the conventional stochastic encoding, and we have no limitations on the values of X_t and X_{t-1}.

6.2.2 Variable Overlap Bits Sharing

By sharing bits based on a fixed length, all the consecutive values share the same fraction of bits without considering the actual difference between these two values. Ideally, if the two consecutive values are the same, all the bits in the bit stream which represents the previous value can be shared by the bit stream which represents the current value. However, if the previous value is 1, and the current value is 0, no bits will be shared[1] . Based on this idea, we can share bits based on a variable overlap, which can share more bits without additional output error than the sharing method based on a fixed overlap.

Generally speaking, the closer the two consecutive values are arithmetically, the more bits can be shared between them. For example, in Fig. 6.7, we represent X_0, X_1, X_2, and X_3 using 25 bits in total based on the fixed length sharing method. However, if we take a look at the similarity of the adjacent values[2] , we will find that X_0 and X_1 have 60% similarity, X_1 and X_2 have 75% similarity, and X_2 and X_3 have 66.7% similarity. This means that the bit streams representing X_1 and X_2 can share more bits than the bit streams representing X_2 and X_3, and the bit streams representing X_2 and X_3 can share more bits than the bit streams representing X_0 and X_1. We show the new sharing results in Fig. 6.8. It can be seen that the bit stream representing X_1 shares 5 bits with the bit stream representing X_0, the bit stream representing X_2 shares 7 bits with the bit stream representing X_1, and the bit stream representing X_2 shares 6 bits with the bit stream representing X_1. Thus, we use 22 bits to represent all four of the deterministic values, and each of them is still represented using a 10-bit stream. This is a better result than the sharing method based on the fixed overlap.

[1] Here we assume the range of the deterministic values is between $[0, 1]$.

[2] We compute the similarity between the two adjacent deterministic values by using the smaller one of the two divided by the bigger one of the two.

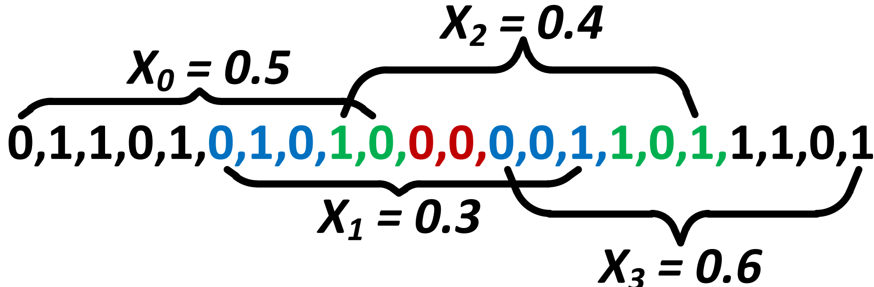


Figure 6.8: An example of sharing a variable number of bits based on the arithmetic difference of the adjacent values.

Now we discuss how to compute the sharing length in detail. We define the current deterministic value to be X_t, the previous deterministic value is X_{t-1}, the value for generating new bits is X, and the shared length is L_s. The formula for computing L_s based on the adjacent values is given in Eq. 6.3,

$$L_s = q\% \times L \times \frac{min(X_t,\ X_{t-1})}{max(X_t,\ X_{t-1})}, \tag{6.3}$$

where q ($0 \leq q \leq 100$) is used to control the randomness of the newly generated bits, and L is the length of the bit stream for representing a single deterministic value. Based on Eq. 6.3, we can see that the sharing length between the two adjacent streams depends on how close their deterministic values are. In addition, it also depends on the value of q. The smaller q is, the more randomness we have in the new bit stream. For example, if we set $q = 0$, we share no bits, and all the new bit streams have the best randomness since we generate them from scratch as in the conventional stochastic encoding method. If we set $q = 100$, the shared bits will be

$$L_s = L \times \frac{min(X_t,\ X_{t-1})}{max(X_t,\ X_{t-1})},$$

which means that the new bit streams will be either all zeros or all ones, which is not random at all.

Another question is how to compute the value X for generating new bits. The goal is that the shared bits and the newly generated bits together should represent X_t as closely as possible. In other words, we have

$$X \times (L - L_s) + min(X_t, X_{t-1}) \times L_s = X_t \times L,$$

i.e.,

$$X = \frac{X_t \times L - min(X_t, X_{t-1}) \times L_s}{L - L_s}. \tag{6.4}$$

From Eq. 6.4, we can see that $X_t \times L$ is always greater than $min(X_t, X_{t-1}) \times L_s$. Thus, this equation has no limitation on the range of the adjacent values. However, compared to Eq. 6.1, it is more complex.

Chapter 7

Conclusion and Future Work

In this dissertation, we first provides background information including the details of the stochastic encoding scheme, combinational logic for simple functions, and several FSM-based constructs proposed by Brown and Card [15].

Second, we present and analyze five FSM-based constructs: tanh, exponentiation, absolute values, exponentiation based on absolute values, and linear gain function. The sources of error in these five constructs are analyzed. Experimental results for the cost, the performance, and the error-tolerance of both stochastic and deterministic implementations of the five constructs are also demonstrated [2, 17].

Third, we introduce the FSM-based stochastic implementations of the five digital image processing algorithms. We analyze the error tolerate of these FSM-based stochastic implementations and compare them to the implementations using the binary encoding. We also analyze and compare the hardware cost, latency and energy consumption [4, 6, 18, 19].

Fourth, we introduce a general method for synthesizing the given target function stochastically using different FSMs, and discusses the trade-offs among these FSMs by comparing their hardware cost and limitations [20, 21, 22, 23].

Finally, we introduces the equally weighted non-stochastic encoding and the overlapped stochastic encoding [24].

The primary contributions of this dissertation are:

- The analysis of the three stochastic computing elements, including stochastic tanh,

exponentiation, and linear gain functions [17].

- The analysis of three fundamental properties of the linear FSMs used to construct those computing elements [2].
- The development and analysis of two new stochastic computing elements, including stochastic absolute values and exponentiation based on absolute values [4, 6].
- The stochastic implementations of five digital image processing algorithms, which include image edge detection, median filter-based noise reduction, image contrast stretching, frame difference-based image segmentation, and KDE-based image segmentation [4, 6].
- The comparison between the stochastic implementations and the deterministic implementations of these five algorithms in terms of the hardware cost, latency and energy consumption [18, 19].
- The general synthesis method using three different FSM topologies [20, 21].
- The analysis of the trade-off among these FSM topologies [22, 23].
- The development of two new encoding schemes, i.e., the equally weighted non-stochastic encoding and the overlapped stochastic encoding [24].

Lots of other interesting problems still need to be solved in this area. For example,

- What applications or algorithms can be benefit from this computing technique? Intuitively, applications in artificial neural networks and complex algorithms in digital image processing should be good candidates. Other applications, such as Boltzmann machine [43] and MIMO system [44] can also be implemented using stochastic computing technique to save both hardware cost and energy consumption compared to conventional binary radix-based implementations.
- Can we implement the FSM-based stochastic computing elements in parallel? As we know, it is not a problem to implement the combinational logic-based stochastic computing elements in parallel, because each bit in a bit stream is independent to the other bits in the same bit stream [45, 46]. However, the FSMs are sequential

logic, there are dependencies between continuous bits for the FSM-based stochastic computing elements. Thus, it is very hard to implement the FSM-based stochastic computing elements in parallel. But if we cannot implement them in parallel, the latency might be an issue for certain applications.

- The FSM-based stochastic computing elements has a correlation issue due to the dependencies between continuous bits [15]. However, we do not find a detailed analysis about this issue yet. How to analyze this issue and how to solve it are important for this research area.

References

[1] W. Qian and M.D. Riedel. The synthesis of robust polynomial arithmetic with stochastic logic. In *45th ACM/IEEE Design Automation Conference, DAC'08*, pages 648–653, 2008.

[2] Peng Li, Weikang Qian, M.D. Riedel, K. Bazargan, and D.J. Lilja. The synthesis of linear finite state machine-based stochastic computational elements. In *Design Automation Conference (ASP-DAC), 2012 17th Asia and South Pacific*, pages 757 –762, 2012.

[3] W. Qian, X. Li, M. Riedel, K. Bazargan, and D. Lilja. An architecture for fault-tolerant computation with stochastic logic. *IEEE Transactions on Computers*, 60(1):93–105, January 2010.

[4] P. Li and D. J. Lilja. A low power fault-tolerance architecture for the kernel density estimation based image segmentation algorithm. In *IEEE International Conference on Application - specific Systems, Architectures and Processors, ASAP'11*, 2011.

[5] B. R. Gaines. Stochastic computing systems. *Advances in Information System Science, Plenum*, 2(2):37–172, 1969.

[6] Peng Li and David J. Lilja. Using stochastic computing to implement digital image processing algorithms. In *29th IEEE International Conference on Computer Design, ICCD'11*, 2011.

[7] J.F. Keane and L.E. Atlas. Impulses and stochastic arithmetic for signal processing. In *Acoustics, Speech, and Signal Processing, 2001. Proceedings.(ICASSP'01). 2001 IEEE International Conference on*, volume 2, pages 1257–1260. IEEE, 2001.

[8] V.C. Gaudet and A.C. Rapley. Iterative decoding using stochastic computation. *Electronics Letters*, 39(3):299–301, 2003.

[9] W.J. Gross, V.C. Gaudet, and A. Milner. Stochastic implementation of ldpc decoders. In *Signals, Systems and Computers, 2005. Conference Record of the Thirty-Ninth Asilomar Conference on*, pages 713–717. IEEE, 2005.

[10] D.K. McNeill and H.C. Card. Refractory pulse counting processes in stochastic neural computers. *Neural Networks, IEEE Transactions on*, 16(2):505–508, 2005.

[11] H. Li, D. Zhang, and S.Y. Foo. A stochastic digital implementation of a neural network controller for small wind turbine systems. *Power Electronics, IEEE Transactions on*, 21(5):1502–1507, 2006.

[12] M. Hori and M. Ueda. Fpga implementation of a blind source separation system based on stochastic computing. In *Soft Computing in Industrial Applications, 2008. SMCia'08. IEEE Conference on*, pages 182–187. IEEE, 2008.

[13] T. Onomi, T. Kondo, and K. Nakajima. Implementation of high-speed single flux-quantum up/down counter for the neural computationusing stochastic logic. *Applied Superconductivity, IEEE Transactions on*, 19(3):626–629, 2009.

[14] W. Qian, M. D. Riedel, and I. Rosenberg. Uniform approximation and Bernstein polynomials with coefficients in the unit interval. *European Journal of Combinatorics*, 32:448–463, 2011.

[15] B. D. Brown and H. C. Card. Stochastic neural computation I: Computational elements. *IEEE Transactions on Computers*, 50(9):891–905, September 2001.

[16] B. D. Brown and H. C. Card. Stochastic neural computation II: Soft competitive learning. *IEEE Transactions on Computers*, 50(9):906–920, September 2001.

[17] P. Li, D.J. Lilja, W. Qian, M.D. Riedel, and K. Bazargan. Logical computation on stochastic bit streams with linear finite state machines. *Computers, IEEE Transactions on*, 2012.

[18] P. Li, D.J. Lilja, W. Qian, K. Bazargan, and M.D. Riedel. Computation on stochastic bit streams digital image processing case studies. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 2013.

[19] Peng Li, Weikang Qian, David J Lilja, Kia Bazargan, and Marc D Riedel. Case studies of logical computation on stochastic bit streams. In *Integrated Circuit and System Design. Power and Timing Modeling, Optimization and Simulation*, pages 235–244. Springer, 2013.

[20] P. Li, D. J. Lilja, W. Qian, and K. Bazargan. Using a two-dimensional finite-state machine for stochastic computation. In *International Workshop on Logic and Synthesis, IWLS'12*, 2012.

[21] Weikang Qian, Chen Wang, Peng Li, David J Lilja, Kia Bazargan, and Marc D Riedel. An efficient implementation of numerical integration using logical computation on stochastic bit streams. In *Computer-Aided Design (ICCAD), 2012 IEEE/ACM International Conference on*. IEEE, 2012.

[22] P. Li, D. J. Lilja, W. Qian, K. Bazargan, and M. Riedel. The synthesis of complex arithmetic computation on stochastic bit streams using sequential logic. In *Computer-Aided Design (ICCAD), 2012 IEEE/ACM International Conference on*. IEEE, 2012.

[23] P. Li, W. Qian, and D. J. Lilja. A stochastic reconfigurable architecture for fault-tolerant computation with sequential logic. In *Computer Design (ICCD), 2011 IEEE 30th International Conference on*. IEEE, 2012.

[24] P. Li and D. J. Lilja. Accelerating the performance of stochastic encoding-based computations by sharing bits in consecutive bit streams. In *IEEE International Conference on Application - specific Systems, Architectures and Processors, ASAP'13*, 2013.

[25] G. Lorentz. *Bernstein Polynomials*. University of Toronto Press, 1953.

[26] Hongliang Chang and Sachin S Sapatnekar. Statistical timing analysis under spatial correlations. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 24(9):1467–1482, 2005.

[27] Daniel K Beece, Jinjun Xiong, Chandu Visweswariah, Vladimir Zolotov, and Yifang Liu. Transistor sizing of custom high-performance digital circuits with parametric yield considerations. In *Design Automation Conference (DAC), 2010 47th ACM/IEEE*, pages 781–786. IEEE, 2010.

[28] Kundan Nepal, R Iris Bahar, Joseph Mundy, WR Patterson, and A Zaslavsky. Designing logic circuits for probabilistic computation in the presence of noise. In *Design Automation Conference, 2005. Proceedings. 42nd*, pages 485–490. IEEE, 2005.

[29] Krishna V Palem. Energy aware computing through probabilistic switching: A study of limits. *Computers, IEEE Transactions on*, 54(9):1123–1137, 2005.

[30] Sriram Narayanan, John Sartori, Rakesh Kumar, and Douglas L Jones. Scalable stochastic processors. In *Proceedings of the Conference on Design, Automation and Test in Europe*, pages 335–338. European Design and Automation Association, 2010.

[31] B. Parhami. *Computer arithmetic*. Oxford university press, 2000.

[32] ISCAS'85 C6288 16×16 multiplier. In *http://www.eecs.umich.edu/~jhayes/iscas/c6288.html*.

[33] R. C. Gonzalez and R. E. Woods. Digital image processing, 3rd edition. *Prentice Hall*, 2008.

[34] System generator for dsp user guide. *Xilinx Inc.*, version 12.3, 2010.

[35] David J. Lilja. Measure computer performance: a practitioner's guide, 1st edition. *Cambridge University Press*, 2000.

[36] A. A. Markov. Extension of the limit theorems of probability theory to a sum of variables connected in a chain. *reprinted in Appendix B of: R. Howard. Dynamic Probabilistic Systems, volume 1: Markov Chains. John Wiley and Sons*, 1971.

[37] G.H. Golub and C.F. Van Loan. *Matrix computations*, volume 3. Johns Hopkins Univ Pr, 1996.

[38] S. S. Tehrani, S. Mannor, and W. J. Gross. Fully parallel stochastic ldpc decoders. *IEEE Transactions on Signal Processing*, 56(11):5692–5703, November 2008.

[39] W. Qian, M.D. Riedel, K. Bazargan, and D.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.

[40] T.J. Richardson, M.A. Shokrollahi, and R.L. Urbanke. Design of capacity-approaching irregular low-density parity-check codes. *Information Theory, IEEE Transactions on*, 47(2):619–637, 2001.

[41] W.E. Ryan. An introduction to ldpc codess. 2003.

[42] The FreePDK45 process design kit.

[43] Geoffrey Hinton. A practical guide to training restricted boltzmann machines. *Momentum*, 9:1, 2010.

[44] Arogyaswami J Paulraj, Dhananjay A Gore, Rohit U Nabar, and Helmut Bolcskei. An overview of mimo communications-a key to gigabit wireless. *Proceedings of the IEEE*, 92(2):198–218, 2004.

[45] Weijun Xiao, Peng Li, and David J Lilja. Comparing the performance of stochastic simulation on gpus and openmp. *International Journal of Computational Science and Engineering*, 8(1):34–46, 2013.

[46] Peng Li, Weijun Xiao, and David J Lilja. A gpu-based simulation for stochastic computing. In *2nd International Workshop on GPUs and Scientific Applications (GPUScA 2011)*, page 15, 2011.