2009

High-Performance Hardware and Software Implementations of the Cyclic Redundancy Check Computation

Christopher E. Kennedy

High-Performance Hardware and Software Implementations of the Cyclic Redundancy Check Computation

(Spine Title: Hardware and Software Implementations of the CRC Computation)

(Thesis Format: Monograph)

by

Christopher E. Kennedy

Faculty of Engineering
Department Electrical and Computer Engineering

Submitted in partial fulfillment
of the requirements for the degree of
Master of Engineering Science

School of Graduate and Postdoctoral Studies
The University of Western Ontario
London, Ontario, Canada
April, 2009

Abstract

The Cyclic Redundancy Check (CRC) is an error detection code used in many digital transmission and storage systems. The two major research areas surrounding CRCs concern developing computation approaches and studying error detection properties. This thesis aims to explore the various aspects of the CRC computation, with the primary objective being to propose novel computation approaches which outperform the existing ones. The work begins with a thorough examination of the formulations found throughout the literature. Then, their subsequent realizations as hardware architectures and software algorithms are investigated. During this investigation, some improvements are presented including optimizations of the state-space transformed and primitive architectures. Afterward, novel formulations are derived and the most significant contribution consists of a matrix decomposition that gives rise to a high-performance software algorithm. Simulation and implementation results are gathered for both hardware and software deployments of the investigated computation approaches. The theoretical results obtained by simulations are validated with implementation experiments. The proposed algorithm is shown to outperform the existing comparable low-memory algorithm in terms of time complexity.

Keywords: Cyclic Redundancy Check (CRC), computer arithmetic, hardware architecture, software algorithm, field-programmable gate array (FPGA), application-specific integrated circuit (ASIC).

Dedication

To my parents, Peter and Maureen, for their love, guidance, and everlasting support.

Acknowledgements

This project would not have been possible without the thoughtful and creative insight of my supervisor, Dr. Reyhani-Masoleh. His experience and encouragement kept me motivated and focused throughout this thesis project. I wish to acknowledge our research group for their positive feedback and constructive criticisms of the work. The assistance which they provided with the various tools that were required to complete the project was invaluable. The comments received from the anonymous external reviewers of our journal manuscript and conference papers were quite helpful in identifying weak areas of our work and giving praise to others. Finally, I cannot fully express my gratitude towards my parents for all the assistance and guidance that they have provided throughout my studies.

Contents

Certificate of Examination ii

Abstract iii

Dedication iv

Acknowledgements v

Contents vi

List of Tables x

List of Figures xi

List of Algorithms xiv

Nomenclature xv

Preface xvii

1 Introduction 1

1.1 Preview 1

1.1.1 Organization 2

1.2 Cyclic Redundancy Check 2

1.3 Motivation 3

1.4 Approach 3

1.5 Objectives 4

2 Preliminaries 5

2.1 Preview 5

2.1.1 Organization 5

2.2 Binary Polynomial Arithmetic 6
2.2.1 Binary Fields 6
2.2.2 Binary Polynomials 7
2.3 CRC Basics 9
2.3.1 Mathematics 9
2.3.2 Generator Polynomials 11
2.3.3 Sample Computation 13
2.3.4 Serial Implementations 14
2.3.5 Error Detection 15
2.4 Summary 18

3 Literature Analysis **19**
3.1 Preview 19
3.1.1 Organization 20
3.2 Parallel Formulation Fundamentals 20
3.2.1 Binary Polynomial Approach 20
3.2.2 State-Space Approach 25
3.3 Hardware Architectures 28
3.3.1 Primitive Architectures 29
3.3.2 Two-Step Architecture 31
3.3.3 Cascade Architecture 35
3.3.4 Look-Ahead Architecture 37
3.3.5 State-Space Transformed Architecture 42
3.3.6 Retimed Architectures 45
3.4 Software Algorithms 45
3.4.1 Assumptions 46
3.4.2 Bit-wise Algorithm 48
3.4.3 Table Look-up Algorithm 49
3.4.4 Reduced Table Look-up Algorithm 51
3.4.5 On-the-Fly Algorithm 52
3.4.6 Tea-Leaf Reader Algorithm 54
3.4.7 Joshi-Dubey-Kaplan Algorithm 54
3.4.8 Slicing Algorithms 55
3.4.9 Distributed Table Look-up Algorithm 56
3.4.10 Look-up Table Generation 57
3.5 Summary 59

4 Novel Computation Approaches **61**
4.1 Preview . . . 61
4.1.1 Organization . . . 61
4.2 Binary Polynomial to Matrix Approach . . . 62
4.2.1 Formulation . . . 62
4.2.2 Realization . . . 64
4.3 Lambda Gamma Approach . . . 68
4.3.1 Formulation . . . 68
4.3.2 Matrix Decomposition . . . 70
4.3.3 Algorithm Realization . . . 75
4.3.4 Architecture Realization . . . 77
4.4 Extended Binary Polynomial Architecture . . . 78
4.4.1 Formulation . . . 78
4.4.2 Realization . . . 83
4.5 Message Splitting Architecture . . . 86
4.5.1 Formulation . . . 86
4.5.2 Realization . . . 88
4.6 Summary . . . 90

5 Simulations and Implementations **91**
5.1 Preview . . . 91
5.1.1 Organization . . . 92
5.2 Hardware Experiments . . . 92
5.2.1 Simulations . . . 93
5.2.2 Implementations . . . 102
5.3 Software Experiments . . . 104
5.3.1 Simulations . . . 105
5.3.2 Implementations . . . 107
5.4 Summary . . . 109

6 Contributions and Future Work **110**
6.1 Preview . . . 110
6.1.1 Organization . . . 111
6.2 Contributions . . . 111
6.2.1 Chapter 3 . . . 111
6.2.2 Chapter 4 . . . 112
6.2.3 Chapter 5 . . . 112

6.3 Future Work 113
6.3.1 Hardware Architectures 113
6.3.2 Software Algorithms 114

A CRC-32 Hardware Architecture Equations **115**
A.1 Parallel LFSR Architectures 116
A.2 Lambda Gamma Architecture 118
A.3 State-Space Transformed Architecture 120
A.4 Two-Step Architecture 123

B CRC-32 Software Algorithm Data **128**
B.1 Table Look-up Algorithm 129
B.2 Reduced Table Look-up Algorithm 131
B.3 Slicing-by-4 Algorithm 132
B.4 Lambda Gamma Algorithm 138
B.5 On-the-Fly Algorithm 138

C z-Transform Approach **139**

D Literature Errata **145**

Bibliography **150**

Vita **157**

List of Tables

2.1 CRC binary polynomials. 10
2.2 Generator polynomials. 12
2.3 Gamma sets 12

3.1 Two-Step Architecture multiple polynomials. 32
3.2 Two-Step Architecture multiplicand polynomials. 34
3.3 Look-Ahead Architecture polynomial relationships. 38
3.4 Optimum state-space transform vectors. 44
3.5 Optimum state-space transform comparison. 44
3.6 Hexadecimal representations of generator polynomials. 47

4.1 Lambda sets. 77

5.1 Non-pipelined hardware architecture comparison. 93
5.2 Optimum degrees of parallelism and hardware complexity. 95
5.3 Computation time for optimum degrees of parallelism. 96
5.4 Optimum ps for the CRC LFSRp Architecture. 99
5.5 LFSR hardware architecture delay comparison. 99
5.6 State-Space Transformed Architecture coupling matrix logic hardware comparison. 101
5.7 State-Space Transformed Architecture hardware comparison. 101
5.8 Two-Step Architecture hardware comparison. 102
5.9 State-Space Transformed Architecture ASIC implementation results. . 103
5.10 Non-pipelined architecture FPGA implementation results. 104
5.11 Software algorithm comparison. 106

List of Figures

1.1 Different stages of the approach. 4

2.1 Modulo-2 truth tables. 7
2.2 Example CCITT-4 binary polynomial long division computation. . . . 13
2.3 Generalized serial LFSR Architectures. 15
2.4 Example CCITT-4 serial hardware computation. 16

3.1 Illustration of the message polynomials. 22
3.2 General state-space model. 25
3.3 Delay diagram of the serial CRC LFSR2 Architecture. 26
3.4 Delay diagram of the serial CRC LFSR1 Architecture. 27
3.5 Generalized parallel LFSR2 Architecture. 29
3.6 Generalized parallel LFSR1 Architecture. 31
3.7 Generalized Two-Step Architecture. 34
3.8 Generalized Cascade Architecture. 36
3.9 Generalized cascade combinational logic blocks. 37
3.10 Non-optimized parallel CRC computation architecture. 39
3.11 Generalized look-ahead combinational logic block. 41
3.12 Generalized Look-Ahead Architecture. 41
3.13 Design of the Flexible Look-Ahead Architecture. 42
3.14 Generalized State-Space Transformed Architecture. 45
3.15 Illustration of the message array. 47
3.16 Illustration of CRCB Endianness. 49

4.1 Matrix $\mathbf{G}_{32\times32}$ for CRC-32. 64
4.2 Example DARC-8 XOR tree architecture. 66
4.3 Generalized optimized parallel LFSR2 Architecture. 67
4.4 Illustration of the Lambda Gamma Algorithm. 76
4.5 Generalized Lambda Gamma Architecture. 78

4.6 Illustrations of the LFSRp overlapping polynomial situations. 80
4.7 Illustrations of the LFSRp non-overlapping polynomial situations. . . 81
4.8 Generalized parallel LFSRp Architecture. 83
4.9 Example DARC-6 dot notation for the parallel LFSRp Architecture. . 85
4.10 Generalized parallel Message Splitting Architecture. 88
4.11 Illustrations of the Message Splitting Architecture overlapping polynomial situations. 89

5.1 Different quantitative comparison metrics. 92
5.2 LFSR2 Architecture plot of the delay and hardware complexity. . . . 95
5.3 LFSR2 Architecture plot of the delay and computation time. 96
5.4 LFSR2 Architecture plot of the delay and time-area product. 97
5.5 LFSRp Architecture plot of the delay versus latency. 97
5.6 LFSR Architecture plot of the delay and latency. 98
5.7 State-Space Transformed Architecture pipelining blocks. 100
5.8 Algorithm timing plots. 108

A.1 Parallel LFSR2 Architecture $\mathbf{G}_{32\times32}$ equations. 116
A.2 Parallel LFSR1 Architecture $\mathbf{G}_{32\times32}$ equations. 117
A.3 Lambda Gamma Architecture $\mathbf{\Lambda}_{32\times32}$ equations. 118
A.4 Lambda Gamma Architecture $\mathbf{\Gamma}_{32\times32}$ equations. 119
A.5 State-Space Transformed Architecture $\mathbf{A}'_{32\times32}$ equations. 120
A.6 State-Space Transformed Architecture $\mathbf{B}'_{32\times32}$ equations. 121
A.7 State-Space Transformed Architecture $\mathbf{C}'_{32\times32}$ equations. 122
A.8 Two-Step Architecture first step equations. 123
A.9 Two-Step Architecture second step equations (1 of 4). 124
A.10 Two-Step Architecture second step equations (2 of 4). 125
A.11 Two-Step Architecture second step equations (3 of 4). 126
A.12 Two-Step Architecture second step equations (4 of 4). 127

B.1 CRCT(8) LUT entries (1 of 2). 129
B.2 CRCT(8) LUT entries (2 of 2). 130
B.3 CRCR(32) LUT entries. 131
B.4 CRCS4(32) LUT_56 entries (1 of 2). 132
B.5 CRCS4(32) LUT_56 entries (2 of 2). 133
B.6 CRCS4(32) LUT_48 entries (1 of 2). 134
B.7 CRCS4(32) LUT_48 entries (2 of 2). 135

B.8 CRCS4(32) LUT_40 entries (1 of 2). 136
B.9 CRCS4(32) LUT_40 entries (2 of 2). 137
B.10 CRCΛΓ(32) LUT entries: (a) Λ LUT, (b) Γ LUT. 138
B.11 CRCF(8) equations. 138

C.1 Discrete-time system illustration of the serial LFSR2 Architecture. . . 139

D.1 Cascade literature error. 148
D.2 State-Space Transformation literature error. 149

List of Algorithms

3.1 Bit-wise Algorithm. 49
3.2 Table Look-up Algorithm. 51
3.3 Reduced Table Look-up Algorithm. 53
3.4 Slicing-by-4 Algorithm. 56
3.5 Table Look-up LUT Generation Algorithm. 57
3.6 Reduced Table Look-up LUT Generation Algorithm. 58
3.7 Slicing LUT Generation Algorithm. 59
4.1 Lambda Gamma Algorithm. 76
4.2 Lambda LUT Generation Algorithm. 77

List of Acronyms

Δ	critical path delay of an architecture
Φ	computation time of an architecture
Θ	hardware cost of an architecture
C_F	hardware cost of a flip-flop
C_X	hardware cost of a two-input XOR gate
T_X	hardware delay of a two-input XOR gate
AND	logical and
ARQ	automatic repeat request
ASIC	application-specific integrated circuit
ATM	Asynchronous Transfer Mode
CMOS	complementary metal-oxide semiconductor
CPD	critical path delay
CRC	Cyclic Redundancy Check
CRC$\Lambda\Gamma$	Lambda Gamma Algorithm
CRCB	Bit-wise Algorithm
CRCD	Distrubuted Table Look-up Algorithm
CRCF	On-the-Fly Algorithm
CRCJDK	Joshi-Dubey-Kaplan Algorithm

CRCR	Reduced Table Look-up Algorithm
CRCS4	Slicing-by-4 Algorithm
CRCS8	Slicing-by-8 Algorithm
CRCT	Table Look-up Algorithm
EDC	error detection code
FCS	frame check sequence
FEC	forward error correction
FF	flip-flop
FPGA	field-programmable gate array
HDL	hardware description language
IEEE	Institute of Electrical and Electronics Engineers
IP	intellectual property
LFSR	linear feedback shift register
LSB	least significant bit
LTI	linear time-invariant
LUT	look-up table
MSB	most significant bit
PC	personal computer
PCLKS	processor clock ticks
PS	pipeline stage
VLSI	very-large-scale integration
XOR	logical exclusive-or

Preface

THIS thesis aims to provide the reader with an understanding of the principles of the Cyclic Redundancy Check (CRC) computation. We assume that the reader has minimal knowledge of CRCs and have made our best effort to present the material accordingly. Many different conventions for describing the CRC computation exist in the literature, and we have selected what we feel are the best set of notations. In our presentation, we separate the formulations from the deployments, either in hardware or software for this case. By taking this approach, we have found that the ideas are more clearly conveyed to the non-expert reader.

Our experience has taught us that examples are an excellent tool for expressing the various CRC computation approaches. However, providing completely worked in-text examples would distract the reader from the further reaching concepts of a given approach. To combat this problem, we have included two appendices which contain the implementation details for some useful hardware architectures and software algorithms.

The preliminaries and literature analysis contained in this thesis provide the reader with a solid foundation in CRCs, which allows them to understand our contributions and the open research questions in this area. From the contributions contained in this thesis, we have had two refereed conference papers accepted and are proceeding with our second revised submission of a full journal manuscript. The comments received from the reviewers of our first journal submission have helped us improve our experimental methodology and gave us thoughtful advice on ways to clarify the formulation.

Finally, we draw inspiration from Évariste Galois; the late French mathematician who laid the foundations for Galois theory, which is the branch of mathematics that CRCs are based on. Without his contributions, this work would not have been possible.

Organization

The organization of the content contained in this thesis is as follows. In Chapter 1, we provide the introduction. In Chapter 2, the preliminaries required to understand the CRC computation are reviewed. In Chapter 3, our analysis of the literature is presented. In Chapter 4, the novel CRC formulations and their resultant realizations as hardware architectures and software algorithms are proposed. In Chapter 5, the simulation and implementation comparison of the studied architectures and algorithms is presented. In Chapter 6, the conclusions and future work are discussed. In Appendix A, hardware equations for some CRC-32 computation architectures are listed. In Appendix B, software look-up table entries for some CRC-32 computation algorithms are listed. In Appendix C, the z-Transform approach to obtain parallel hardware equations is reviewed. In Appendix D, the identified technical errors in the literature are corrected.

A portrait of Évariste Galois (1811 - 1832).

Chapter 1

Introduction

1.1 Preview

AN increasing number of designers are utilizing wireless communication technology in their systems. However, wireless networks are more susceptible to transmission errors; some causes are random channel noise, fading of signals, and atmospheric conditions. Due to the greater probability of transmission errors occurring in these systems, it is necessary to verify the integrity of a received message using an error detection code (EDC). Research for fast and flexible computation methods for dependable EDCs is ongoing.

One of the more popular EDCs is the Cyclic Redundancy Check (CRC) [1]. From a certain perspective, the CRC can be considered as an insecure hash function. In other words, the CRC function maps a large variable length message to a small fixed sized checksum. This checksum is appended to its message to form a codeword. The redundancy in the codeword is typically used to verify the integrity of a message after it has been transmitted or stored.

The two major research areas surrounding CRCs concern its error detection properties and computation approaches. This thesis is primarily focused on the study of the mathematics and performance of the various approaches that perform the CRC computation. The existing formulations and their resulting realizations as hardware architectures and software algorithms are extensively reviewed, and some novel computation approaches are proposed.

In this chapter, we briefly introduce CRCs to set the stage of our study. Afterward, the motivation, approach, and objectives of this thesis are outlined.

1.1.1 Organization

The remainder of this chapter is organized as follows. In Section 1.2, the basic concepts of the CRC are introduced. In Section 1.3, the motivation for this thesis is presented. In Section 1.4, the approach taken in this thesis is explained. In Section 1.5, the objectives of our work are stated.

1.2 Cyclic Redundancy Check

In information theory, there are two major error control strategies: forward error correction (FEC) and automatic repeat request (ARQ) [2]. FEC must be used when the transmission channel is unidirectional and detected errors must be corrected by the receiver. ARQ may be used in bidirectional communication systems when it is more convenient to simply detect an error and request a retransmission. ARQ codes are typically more light-weight in terms of the number of redundant bits and computation times compared to FEC [2].

The CRC was proposed in 1961 by Peterson and Brown [1] as a separable EDC, that is now used in many digital transmission and storage systems. Before transmission, a message has its CRC appended as a frame check sequence (FCS) to form a codeword. On arrival, the FCS of the received message is computed and compared with the sent FCS; if they differ, an error is detected, else the transmission is assumed to be error free. When errors are detected, the transmission protocol dictates what action should be taken, i.e., discard the corrupted data and/or send a retransmission request.

Some examples of digital communication standards where the CRC is currently employed are, the Asynchronous Transfer Mode (ATM) [3], and the Institute of Electrical and Electronics Engineers (IEEE) communication standards, such as, IEEE 802.3 (Wired Ethernet) [4], IEEE 802.11 (WiFi) [5], and IEEE 802.16 (WiMAX) [6].

As aforementioned, the main research activities concerning CRCs consist of error detection properties and computation approaches. In [1], the authors propose using a linear feedback shift register (LFSR) to perform the CRC computation. This simple architecture operates serially, processing one message bit per clock cycle. After [1], a large amount research effort has been invested in developing parallel hardware architectures and software algorithms that perform the computation more quickly or efficiently, and this thesis continues along this path.

1.3 Motivation

In this section, we present the motivation for our study. With the current wireless telecommunications boom that is world is undergoing, ensuring data integrity will become more of an issue in the future. The CRC is an attractive option for use in communication systems, because it is easily described mathematically and the error detection properties are well understood. Moreover, there are many different approaches to perform the CRC computation, and any contributions that advance or improve upon these ideas could end up being deployed in real-world industrial systems.

In terms of the amount of research attention it receives, the CRC can be considered a hot topic. At the time of this writing, recent IEEE Transactions journal papers with contributions pertaining to the theory of CRCs include [7], [8], [9], [10], [11], and [12]. This demonstrates the large amount of current interest in this area. Furthermore, the most recent and only survey [13], was published in 1988, and many developments have happened since then. For these reasons, we feel that the time is right for a fresh investigation and discussion of this topic.

1.4 Approach

In this section, we present the approach of our study. Since this thesis is in the field of computer arithmetic, generally, all of the concepts discussed stem from a mathematical formulation. After one performs some manipulations and obtains a desired formulation, the next step is realizing that formulation as a hardware architecture or software algorithm. After realization, one proceeds to implement the architecture or algorithm, using a hardware description or programming language, respectively. Finally, the implementation is then deployed on a platform and its correctness can be verified. Figure 1.1 illustrates how we go through these steps in this study.

In this thesis, all of the formulations begin from the CRC equation, that is introduced in [1]. As shown later, many computation schemes have been deduced from the application of different techniques to manipulate that equation. Our approach begins by generalizing those existing methods found in the literature. By accomplishing this, we have more flexibility to describe and compare the different schemes which have been previously published. Afterward, we present some novel formulations and their resultant realizations as CRC computation approaches that yield new architectures and algorithms.

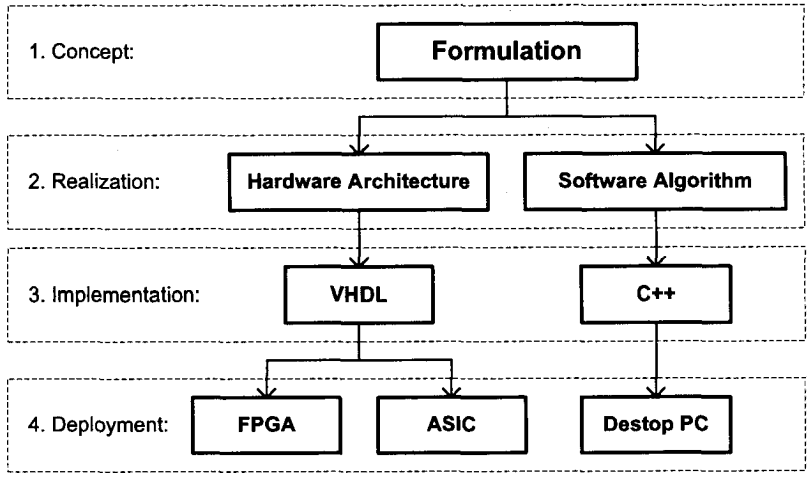


Figure 1.1: Different stages of the approach.

In terms of the experimental methodology, after an architecture or algorithm has been proposed, one can perform simulations to obtain the theoretical area/memory and time complexities. The majority of the simulation data in this thesis is obtained through custom C++ software, that has been written by the author over the course of this study. These theoretical evaluations serve to validate our experimental results gathered through deployment of our implementations on their respective platforms.

1.5 Objectives

The primary objective of this thesis is to propose novel CRC computation approaches which outperform the existing ones. In this study, the performance of an approach is measured in terms of both area/memory and time complexity. In order to achieve this primary objective, we first strive to provide the reader with a complete understanding of the fundamentals of the CRC computation. This involves using consistent notations to derive and generalize the previous computation approaches. We then perform simulations and experiments to demonstrate the high performance of our novel approaches, and we thoroughly explore the area/memory versus time complexity trade-off. Also, we aim to identify and propose some improvements to the existing computation approaches. Finally, we suggest open research questions that could be the focus of future work.

Chapter 2

Preliminaries

2.1 Preview

THE Cyclic Redundancy Check (CRC) is a term that most of us have come across at one time or another reading computer literature in the Internet age. In this chapter, we present the preliminaries required to discuss the various aspects surrounding the CRC. This includes a review of the branch of mathematics that CRCs operate in, the first hardware architectures proposed to implement the computation, and the general error detection properties.

Like most topics in information technology, the presentation style and notations used to describe the mathematics of the CRC computation have evolved considerably since being first introduced by Peterson and Brown in 1961 [1]. Emphasis has shifted from discussing the basic serial computation case to more complex parallel computation cases, and exhaustive explorations of error detection properties have been undertaken. In light of these advancements, the goal of this chapter is to review the fundamental material using more modern notations and conventions, and lay the foundation for discussing the material in later chapters.

2.1.1 Organization

The remainder of this chapter is organized as follows. In Section 2.2, the topic of binary polynomial arithmetic is reviewed. This includes a discussion of binary fields and binary polynomials, i.e., polynomials over $GF(2)$. In Section 2.3, the CRC basics are introduced using modern conventions. This includes mathematics, generator polynomials, serial hardware architectures, and basic error detection properties. This chapter is concluded with a summary in Section 2.4. We note that, most of the material contained in this chapter can be found in [1], [2], and [13].

2.2 Binary Polynomial Arithmetic

This section quickly reviews the fundamentals of the branch of mathematics that is required for an understanding of CRC computation. This involves discussing the binary Galois field $GF(2)$ and binary polynomials. This material forms the basis for the later binary polynomial formulation of the CRC computation. For a more formal and in-depth discussion of these topics, we refer the reader to [2] and [14].

2.2.1 Binary Fields

To begin, we roughly define a field to be a set of elements for which one can perform addition, subtraction, multiplication, and division without leaving the set. Furthermore, the commutative, associative, and distributive laws must be satisfied by the addition $(+)$ and multiplication $(\cdot)$ operations [2], [14].

Next, an adaptation of the formal definition of a field contained in [2] is presented. Let F be a set of elements on which the addition and multiplication operations are defined. The set F together with the addition and multiplication operations, is a field if the following three conditions are satisfied:

1. The set F is a commutative group under addition. The identity element with respect to addition is called the zero element and denoted by 0.

2. The set of nonzero elements in F is a commutative group under multiplication. The identity element with respect to multiplication is called the unit element and denoted by 1.

3. Multiplication is distributive over addition; that is, for any three elements f_0, f_1, and f_2 in F,
$$f_0 \cdot (f_1 + f_2) = f_0 \cdot f_1 + f_0 \cdot f_2.$$

It follows from the above definition that a field must contain at least two elements, namely 0 and 1. In fact, it is the field that contains only these two elements, called the binary Galois field that we are most interested in. Before discussing this field, some basic properties are listed without proof, which can be easily derived from the previous definition of a field [2].

+	0	1
0	0	1
1	1	0

(a)

·	0	1
0	0	0
1	0	1

(b)

Figure 2.1: Modulo-2 truth tables: (a) addition, (b) multiplication.

1. For every element f_0 in a field, $f_0 \cdot 0 = 0 \cdot f_0 = 0$.
2. For any two nonzero elements f_0 and f_1 in a field, $f_0 \cdot f_1 \neq 0$.
3. For any two elements f_0 and f_1 in a field, for $f_0 \cdot f_1 = 0$ and $f_0 \neq 0$, imply that $f_1 = 0$.
4. For any three elements f_0, f_1, and f_2 in a field, for $f_0 \neq 0$, $f_0 \cdot f_1 = f_0 \cdot f_2$, implies that $f_1 = f_2$.

Now consider the binary Galois field denoted as $GF(2) = \{0, 1\}$ that has two elements, with modulo-2 addition and multiplication operations defined in Figures 2.1a and 2.1b, respectively. It is clear that the commutative, associative, and distributive laws hold for the addition and multiplications operations defined on the set $GF(2)$. Thus, $\{0, 1\}$ is a field with two elements under modulo-2 addition and modulo-2 multiplication [2].

Note that binary addition and multiplication between two elements in $GF(2) = \{0, 1\}$, can be implemented in hardware using logical exclusive-or (XOR) and logical and (AND) gates, respectively. Throughout this thesis, unless otherwise noted, the addition sign, i.e., "+" is used to denote the XOR operation, and the dot sign, i.e., "·" denotes the AND operation. Finally, note that for the set $GF(2)$, addition and subtraction are defined to be the same operation, i.e., $1 \pm 1 = 0$ and $1 \pm 0 = 0 \pm 1 = 1$.

2.2.2 Binary Polynomials

From the definition of a binary field presented in the previous subsection, we are now ready to discuss binary polynomials. Consider a polynomial whose coefficients are from the binary field $GF(2)$, with the variable x, i.e.,

$$F(x) = f_0 + f_1 x + \cdots + f_n x^n,$$

where $f_i \in \{0, 1\}$ for $0 \leq i \leq n$. Polynomials of this form will be referred to as *polynomials over* $GF(2)$ [2]. In this thesis, upper case letters are used to denote polynomials over $GF(2)$ and the coefficients are lowercase letters.

The degree of a polynomial is defined as the largest power of x with a non-zero coefficient. Polynomials over $GF(2)$ can be added (or subtracted), multiplied, and divided in the usual way. As an illustration of these operations let

$$G(x) = g_0 + g_1 x + \cdots + g_m x^m,$$

be another polynomial over $GF(2)$. Assuming $m \leq n$, then addition or subtraction is computed as,

$$\begin{aligned} F(x) \pm G(x) &= (f_0 + g_0) + (f_1 + g_1)\, x + \cdots + (f_m + g_m)\, x^m + \\ &\quad f_{m+1} x^{m+1} + \cdots + f_n x^n. \end{aligned}$$

For multiplication one has the product

$$F(x) \cdot G(x) = C(x) = c_0 + c_1 x + \cdots + c_{n+m} x^{n+m},$$

where

$$\begin{aligned} c_0 &= f_0 g_0 \\ &= f_0 g_1 + f_1 g_0 \\ &= f_0 g_2 + f_1 g_1 + f_2 g_0 \\ &\vdots \\ c_i &= f_0 g_i + f_1 g_{i-1} + f_2 g_{i-2} + \cdots + f_i g_0 \\ &\vdots \\ c_{n+m} &= f_n g_m. \end{aligned}$$

Note that for CRCs, binary polynomial division is the most important operation. From the Euclidean division algorithm, one knows that when $F(x)$ is divided by $G(x)$, a unique pair of polynomials over $GF(2)$ is obtained: $Q(x)$ called the quotient and $R(x)$ called the remainder. Thus the relationship

$$F(x) = Q(x) \cdot G(x) + R(x) \tag{2.1}$$

is obtained, where the degree of $R(x)$ is less than the degree of $G(x)$. From (2.1) one can adopt the following notation for expressing the calculation of $R(x)$ from $F(x)$ and $G(x)$ as

$$R(x) = F(x) \bmod G(x).$$

The traditional elementary school long-division technique is often used when computing the division operation of two binary polynomials, and in the following section an example is provided in Figure 2.2.

Data Representation

It is convenient to represent binary strings as polynomials over $GF(2)$ [2]. However, one source of confusion often arises with the Endianness associated to mapping bit positions of the binary strings to the coefficients of the polynomial [15]. In the CRC literature, two methods are readily used, the first being mapping the most significant bit (MSB) to the coefficient of the term with the highest power of x down to the least significant bit (LSB) being mapped to coefficient of the x^0 term, i.e., 1011 1001 $\rightarrow$ $x^7 + x^5 + x^4 + x^3 + x^0$. The second method is the reverse of the first, with the MSB being mapped to the coefficient of the x^0 term up to the LSB being mapped to the coefficient of the largest power, i.e., 1011 1001 $\rightarrow$ $x^0 + x^2 + x^3 + x^4 + x^7$. These conventions will be referred to as the normal and reverse notation, for the first approach and second approach, respectively.

In this thesis, we have chosen to adopt the reverse notation, because software CRC computation algorithms are more efficiently implemented with this approach, and this is consistent the convention used in the latest software based CRC paper [7], as well as [13] and [16]. Also, in hardware it is more popular to illustrate the serial computation architecture using the reverse notation (e.g.: [1], [10], [17], [18], [19], [20], [21], [22], [23], [24], [25], [26]), rather than using the normal notation.

2.3 CRC Basics

After reviewing the related basics of binary polynomial arithmetic in the previous section, we are now prepared to discuss the fundamental aspects surrounding CRCs. In this section, the mathematics of the computation, generator polynomials, serial architectures, and error detection properties are all reviewed.

2.3.1 Mathematics

Let us begin by introducing the following binary polynomials listed in Table 2.1, which are similar to the conventions found in [13]. The m-bit CRC polynomial, also called the *syndrome* and denoted by $S(x)$ of a k-bit message, is defined as the

Table 2.1: CRC binary polynomials.

Polynomial	Name	Max Degree
$A(x)$	Quotient	$k-1$
$E(x)$	Error	$k+m-1$
$G(x)$	Generator	m
$R(x)$	Received	$k+m-1$
$S(x)$	Syndrome	$m-1$
$U(x)$	Message	$k-1$
$V(x)$	Codeword	$k+m-1$

remainder of the division between the message polynomial $U(x)$ multiplied by x^m, and the $(m+1)$-bit generator polynomial $G(x)$, i.e.,

$$S(x) = (x^m \cdot U(x)) \bmod G(x). \tag{2.2}$$

The n-bit codeword polynomial, where $n = k + m$ is defined as

$$V(x) = S(x) + x^m \cdot U(x), \tag{2.3}$$

and consists of the syndrome polynomial concatenated with the message polynomial. The codeword polynomial corresponds to the bits transmitted by the sender. Define the n-bit received polynomial $R(x)$ to consist of

$$R(x) = V(x) + E(x), \tag{2.4}$$

where $E(x)$ is an n-bit error polynomial. From (2.4) it is clear that if $E(x) = 0$, then $R(x) = V(x)$. Define $A(x)$ to the quotient of the division between $x^m \cdot U(x)$ and $G(x)$, then from the property of Euclidean division one can obtain

$$x^m \cdot U(x) = A(x) \cdot G(x) + S(x). \tag{2.5}$$

Next, in order to discuss the error detection properties one needs to show that the codeword polynomial is a multiple of the generator polynomial. Substituting (2.5) into (2.3) and one obtains,

$$\begin{aligned} V(x) &= S(x) + x^m \cdot U(x) \\ &= S(x) + (A(x) \cdot G(x) + S(x)) \\ &= A(x) \cdot G(x). \end{aligned} \tag{2.6}$$

Thus, it is concluded that codeword is a multiple of the generator polynomial. At the receiver, the integrity of a frame is verified by performing one of the two following checks:

1. Compute and test to see if $R(x) \bmod G(x) = 0$.

2. Separate $R(x)$ into the received message $x^m \cdot U'(x)$ and received syndrome $S'(x)$, and then compute and test $x^m \cdot U'(x) \bmod G(x) = S'(x)$.

If either equality does not hold, then it is known that $R(x) \neq V(x)$, and a transmission error is detected. Finally, from the above definitions, it can be shown that if

$$E(x) = F(x) \cdot G(x), \tag{2.7}$$

then an error will go undetected. Beginning from (2.4) and substituting (2.6) and (2.7), one can obtain,

$$\begin{aligned} R(x) &= V(x) + E(x) \\ &= (A(x) \cdot G(x)) + (F(x) \cdot G(x)) \\ &= (A(x) + F(x)) \cdot G(x). \end{aligned}$$

It is clear that in this case, that $R(x) \bmod G(x) = 0$, and it is concluded that error patterns of this type are not detectable.

2.3.2 Generator Polynomials

Next, we discuss CRC generator polynomials. Generator polynomials are always of the form,

$$G(x) = 1 + \sum_{i=1}^{m-1} g_i x^i + x^m, \; g_i \in \{0, 1\} \tag{2.8}$$

i.e., all generator polynomials have nonzero x^0 and x^m terms [1]. Table 2.2 lists frequently referenced generator polynomials [10]. In this thesis, we denote the second greatest nonzero power of $G(x)$ as τ, i.e.,

$$G(x) = 1 + \sum_{i=1}^{\tau-1} g_i x^i + x^\tau + x^m.$$

Table 2.2: Frequently referenced generator polynomials.

Name	Polynomial
CRC-12	$1+x+x^2+x^3+x^{11}+x^{12}$
CRC-16	$1+x^2+x^{15}+x^{16}$
CCITT-16	$1+x^5+x^{12}+x^{16}$
CRC-16†	$1+x+x^{14}+x^{16}$
CCITT-16†	$1+x^4+x^{11}+x^{16}$
CRC-32	$1+x+x^2+x^4+x^5+x^7+x^8+x^{10}$ $x^{11}+x^{12}+x^{16}+x^{22}+x^{23}+x^{26}+x^{32}$

† denotes reversed polynomial coefficients.

From (2.8), we define the set Γ, as

$$\Gamma = \left\{\gamma_0, \gamma_1, \cdots, \gamma_{|\Gamma|-1}\right\} = \left\{i | g_i = 1, i \in [0, m-1]\right\}, \tag{2.9}$$

and define the cardinality of the set Gamma to be the number of elements in Γ, denoted as $|\Gamma|$. It is noted that $\gamma_0 = 0$, since generator polynomials always have a nonzero x^0 term, i.e., $g_0 = 1$. Earlier we defined τ to be the second greatest nonzero power in $G(x)$, thus $\gamma_{|\Gamma|-1} = \tau$. In later chapters, it will be convenient to show generator polynomials as $\sum_{i\in\Gamma} x^i + x^m$, with $\Gamma = \left\{\gamma_0 = 0, \gamma_1, \cdots, \gamma_{|\Gamma|-2}, \gamma_{|\Gamma|-1} = \tau\right\}$. Table 2.3 lists the Gamma sets of the commonly used generator polynomials.

Observing (2.8), and note that CRC generator polynomials are never divisible by x. If one were to use a generator polynomial that had x as a factor, then the resultant syndrome would always have its zero-order coefficient equal to zero [1]. To illustrate this fact, the following derivation is provided. Consider computing a syndrome using a generator polynomial that has x as a factor, i.e., $G(x) = x \cdot G'(x)$, then $G'(x) = x^{m-1} + \sum_{i=1}^{m-2} g'_i x^i + 1$. Using $G'(x)$ in (2.5), one obtains,

$$x^{m-1} \cdot U(x) = A(x) \cdot G'(x) + S'(x),$$

Table 2.3: Gamma sets of frequently referenced generator polynomials.

Name	Γ	$\lvert\Gamma\rvert$
CRC-12	$\{0, 1, 2, 3, 11\}$	5
CRC-16	$\{0, 2, 15\}$	3
CCITT-16	$\{0, 5, 12\}$	3
CRC-16†	$\{0, 1, 14\}$	3
CCITT-16†	$\{0, 4, 11\}$	3
CRC-32	$\{0, 1, 2, 4, 5, 7, 8, 10, 11, 12, 16, 22, 23, 26\}$	14

where $S'(x)$ is at most degree x^{m-2}. Multiplying both sides by x and one obtains,

$$x^m \cdot U(x) = A(x) \cdot G(x) + x \cdot S'(x).$$

Thus, it is shown that if $G(x) = x \cdot G'(x)$, then $S(x) = x \cdot S'(x)$ and always has a zero as its zero-order coefficient.

2.3.3 Sample Computation

In this subsection, we present a sample CRC computation using the long-division technique. Consider the generator polynomial CCITT-4 ($G(x) = 1 + x + x^4$ [27]), and the 7-bit message 101 0011. Using the reverse Endianness convention, the 7-bit binary sequence is mapped to the message polynomial as $U(x) = 1 + x^2 + x^5 + x^6$. Substituting this message polynomial into (2.2) and one obtains,

$$\begin{aligned} S(x) &= (x^m \cdot U(x)) \bmod G(x) \\ &= \left(x^4 \cdot \left(1 + x^2 + x^5 + x^6\right)\right) \bmod \left(1 + x + x^4\right) \\ &= \left(x^4 + x^6 + x^9 + x^{10}\right) \bmod \left(1 + x + x^4\right). \end{aligned} \tag{2.10}$$

It is known that when using this generator polynomial the syndrome is 4-bits long, and corresponds to a binary polynomial of at most degree x^3, i.e., $S(x) = \sum_{i=0}^{3} s_i x^i$. One can proceed to perform the reduction in (2.10) and $S(x) = x$ will be obtained. The long-division steps are shown in Figure 2.2, and we remind the reader that in this computation binary polynomials are being used, consequently, addition and

$$\begin{array}{r|l}
 & x^6 + x^5 + x^3 + x^2 + x \\
\hline
x^4 + x + 1 & x^{10} + x^9 + x^6 + x^4 \\
 & x^{10} \quad\quad + x^7 + x^6 \\
\hline
 & x^9 + x^7 + x^4 \\
 & x^9 \quad\quad + x^6 + x^5 \\
\hline
 & x^7 + x^6 + x^5 + x^4 \\
 & x^7 \quad\quad + x^4 + x^3 \\
\hline
 & x^6 + x^5 + x^3 \\
 & x^6 \quad\quad + x^3 + x^2 \\
\hline
 & x^5 + x^2 \\
 & x^5 \quad\quad + x^2 + x \\
\hline
 & x
\end{array}$$

Figure 2.2: Example CCITT-4 binary polynomial long division computation.

subtraction are the same operation. The correctness of the result can be verified by recalling (2.5), and checking that,

$$x^4 + x^6 + x^9 + x^{10} = \left(x + x^2 + x^3 + x^5 + x^6\right) \cdot \left(1 + x + x^4\right) + x.$$

The codeword polynomial $V(x)$ is formed by the concatenation of the syndrome polynomial and message polynomial (2.3), and for this example one obtains,

$$\begin{aligned} V(x) &= S(x) + x^m \cdot U(x) \\ &= x + x^4 + x^6 + x^9 + x^{10}. \end{aligned}$$

Using the long-division technique, one can verify that $V(x) \bmod G(x) = 0$. Finally, the 11-bit codeword is transmitted or stored as 0100 1010 011.

2.3.4 Serial Implementations

The historical serial implementation of the CRC computation (2.2) in hardware consists of a LFSR, which is constructed for a given generator polynomial [1]. The serial LFSR Architecture has a hardware cost

$$\Theta = m \cdot C_F + |\Gamma| \cdot C_X,$$

where C_F and C_X denote the cost of a flip-flop (FF) and a two-input XOR gate, respectively.

There are two different LFSR architectures presented in [1], named LFSR1 and LFSR2 in [26]. In terms of computation time, LFSR1 requires $k + m$ clock cycles while LFSR2 requires k clock cycles, and the general form of each architecture is shown in Figure 2.3. In practice, all the AND gates are replaced by open or short circuits depending on the coefficients of the generator polynomial, and the XOR gates without present feedback connections are removed. Notice that the critical path delay (CPD), denoted by Δ, of LFSR1 is $1 \cdot T_X$ while LFSR2 has a CPD of $2 \cdot T_X$[1], where T_X denotes the delay of a two-input XOR gate.

If the LFSR2 architecture is used, then the syndrome of a message is computed by feeding the k message bits in, beginning from the coefficient of the highest order term u_{k-1} down to u_0, afterward the syndrome of the message is stored in the FFs.

[1] If $G(x) = 1 + x^m$, then the delay of the LFSR2 architecture is $1 \cdot T_X$. However, for the common generator polynomials in Table 2.2 this is never the case.

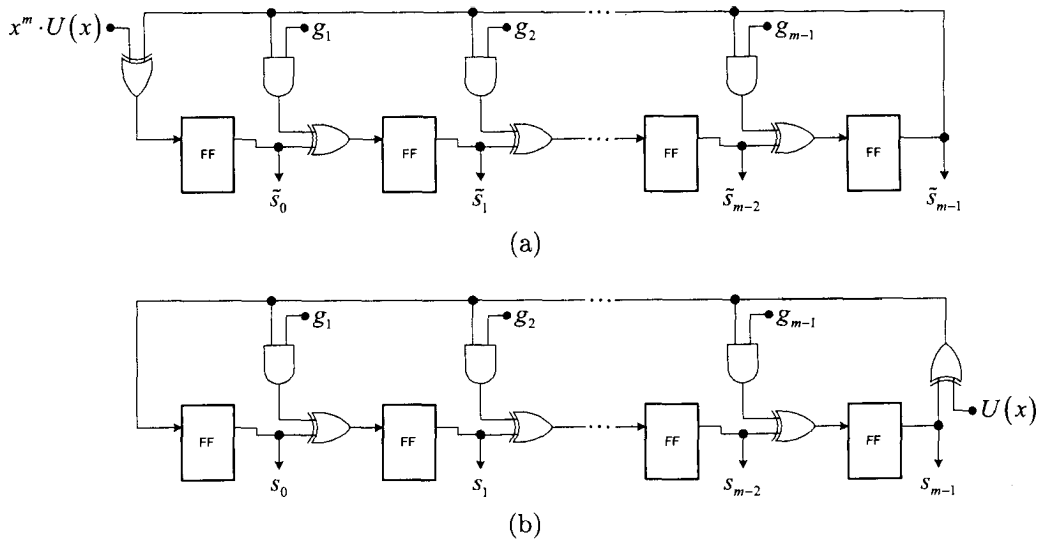


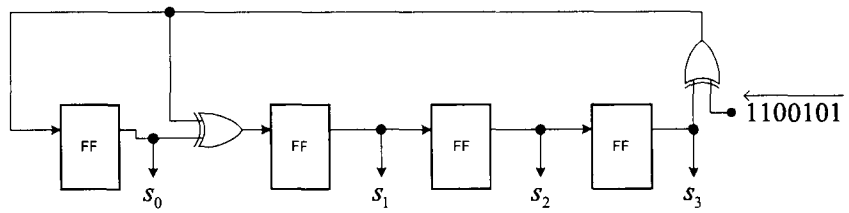
Figure 2.3: Generalized serial LFSR Architectures for $G(x) = 1 + \sum_{i=1}^{m-1} g_i x^i + x^m$: (a) LFSR1, (b) LFSR2.

Alternatively, if the LFSR1 structure is used, then m 0s must be fed in after u_0 to obtain the syndrome [1]. More formally, the LFSR2 architecture performs the CRC computation as $(x^m \cdot U(x)) \bmod G(x)$, whereas the LFSR1 architecture treats the computation as $\tilde{U}(x) \bmod G(x)$, where $\tilde{U}(x) = x^m \cdot U(x)$. These two serial implementations serve as the starting points for all the parallel hardware architectures and software algorithms that are discussed in later chapters.

We close this subsection by tracing the previous example computation through the LFSR2 architecture. In Figure 2.4a we provide an illustration of the LFSR2 architecture for the generator polynomial CCITT-4 [27], and Figure 2.4b shows the contents of the FFs after each clock cycle when processing the message $U(x) = 1 + x^2 + x^5 + x^6$. Clock cycle -1 denotes the initial all zero state of the register, and as expected, the final result (marked in boldface) is 0100, which corresponds to $S(x) = 0 \cdot x^0 + 1 \cdot x + 0 \cdot x^2 + 0 \cdot x^3 = x$.

2.3.5 Error Detection

A great amount of research effort has been invested in the study of the issues surrounding the error detection performance of CRCs. A complete discussion of this area is beyond the scope of this thesis, but for the sake of completeness, we review the basic concepts, terminology, and results. We refer the reader to the following sets of references for discussions concerning: general error detection properties [1], [13], [28], generator polynomials [8], [27], [29], [30], [31], [32], [33], [34], [35], and different CRC schemes [28], [36], [37], [38].



(a)

Cycle	Input	LFSR Contents			
		s_0	s_1	s_2	s_3
-1	–	0	0	0	0
0	1	1	1	0	0
1	1	1	0	1	0
2	0	0	1	0	1

Cycle	Input	LFSR Contents			
		s_0	s_1	s_2	s_3
3	0	1	1	1	0
4	1	1	0	1	1
5	0	1	0	0	1
6	1	**0**	**1**	**0**	**0**

(b)

Figure 2.4: Example CCITT-4 serial hardware computation: (a) LFSR2 Architecture, (b) trace.

Generally, one can say that the error detection performance of a typical CRC scheme depends on three factors:

- the degree of the generator polynomial;
- the generator polynomial coefficients; and
- the length of the message (the number of message bits k).

We say that a CRC scheme is typical, if a codeword is formed by computing and appending the m-bit syndrome to a k-bit message (2.3). Other situations are possible, such as using product generator polynomials, two-fold, and cascading; for a detailed discussion of the effectiveness of these approaches see [38]. For the remainder of this subsection, our discussion is restricted to the typical scheme.

Recall the earlier definition of the error polynomial in (2.4), and it was shown that it must be a multiple of the generator polynomial for the error pattern to go undetected. The length of a burst error is defined to distance between and including the furthest two corrupted bits in a received codeword, e.g.: the burst error described by the error polynomial $E(x) = x^i + x^j$ has a length of $j - i + 1$, assuming $0 \leq i < j < n$. It is noted that every nonzero coefficient in $E(x)$ corresponds to an inversion in the codeword at that bit position. From these concepts, we are ready to discuss some results for simple error patterns.

Consider the situations when the i-th codeword bit is inverted, then $E(x) = x^i$ for $0 \leq i < n$. Since $G(x)$ always has present x^0 and x^m terms (2.8), x^i cannot be expressed as a multiple of $G(x)$ and this type of error will always be detected.

For the cases when there are only two codeword bit inversions, they are described by the error polynomial $E(x) = x^i + x^j$ with $0 \leq i < j < n$. This error polynomial can be factored as $E(x) = x^i \cdot (1 + x^{i-j})$, and it is concluded that two bit errors will go undetected if and only if $G(x)$ evenly divides $1 + x^{i-j}$.

For all burst errors of length less than m, the error polynomial can be written as $E(x) = x^i \cdot P(x)$, where the degree of $P(x)$ is less than the degree of the generator polynomial. Then, one has $P(x) \bmod G(x) = P(x)$ and this error is always detected.

Here, some useful error detection theorems for CRCs are given without proof, and they can be found in [28].

- All single-bit errors will be detected because the generator polynomial always has more than one term. The simplest generator polynomial is $G(x) = 1 + x$ (2.8).

- All cases where an odd number of bit errors have occurred will be detected if a generator polynomial has $x^a + 1$ for $a > 0$, as a factor.

- All single- and double-bit errors will detected if the degree of the codeword polynomial is no greater than the period[2] of the polynomial.

- All single-, double-, and triple-bit errors will be detected if the generator polynomial has $x^a + 1$ for $a > 0$ as a factor, and the degree of the codeword polynomial is no greater than the period of the generator polynomial.

- All burst errors of length less than or equal to m are detected.

- The misdetection probability P_{md}, is defined to be the ratio of the total number of error patterns to the number of possible error patterns that go undetected,

$$P_{md} \approx \frac{2^k - 1}{2^n - 1} < 2^{-m},$$

  and it can be estimated by the degree of the generator polynomial.

For proofs of these theorems and further discussion, the reader is encouraged to consult [1] and [13].

[2]Period refers to the smallest degree polynomial that $G(x)$ evenly divides.

Error Correction

The authors of [1] also note the potential for the CRC to be able to correct errors. Similar to the error detection case, the remainder of the received codeword divided by the generator polynomial is computed, i.e., $R(x) \bmod G(x)$. Now, if the remainder is zero, then correct transmission is assumed. Otherwise, the remainder is compared to the n stored values of $x^i \bmod G(x)$ for $0 \leq i < n$, and if a match is found, the i-th bit of the codeword is flipped and the message is assumed correct. Alternatively, if the non-zero remainder is not found in the store values, a multi-bit error has occurred and detected. An implementation of single-bit error correction using the generator polynomial CCITT-16 with $k = 16$ can be found in [39].

2.4 Summary

In this chapter, we reviewed the concepts of binary polynomial arithmetic and discussed some of the fundamentals of CRCs. Binary polynomials or polynomials over $GF(2)$ have coefficients from the set $GF(2) = \{0, 1\}$, and they can be added (subtracted), multiplied, or divided in the usual way. Addition and multiplication of binary coefficients can be accomplished by using logical XOR and AND operations, respectively. The CRC computation involves finding the remainder of the division of the augmented message polynomial by the generator polynomial. CRC generator polynomials are of degree m and always have a present non-zero x^0 term. The classical hardware implementation of the CRC computation consists of a LFSR and it performs the computation serially. The error detection properties are well understood and depend on the generator polynomial, message length, and deployment scheme. For typical schemes, an error pattern must be a multiple of the generator polynomial to go undetected. Error correction is possible using CRCs. However, the entire codeword must be buffered and a look-up table is typically used to perform the correction.

Chapter 3

Literature Analysis

3.1 Preview

OVER the years, various parallel formulations have been proposed with the aim of deriving new hardware architectures and software algorithms to perform the CRC computation. In this chapter, we analyze the developments most related to our contributions. This includes an investigation and discussion of the various formulations that have been published and their suggested realizations as hardware architectures and/or software algorithms. The aim of this chapter is to develop generalized formulations for all of the previous works. This helps us to compare our schemes with the previous approaches, and demonstrate that our schemes presented in the thesis are indeed novel.

Excluding the bit-wise software algorithm, all of the architectures and algorithms discussed in this chapter perform the CRC computation by processing multiple message bits in an iteration. For this reason, we have included our generalized primitive parallel CRC formulation as a starting point for all the different formulations. By taking this approach, we feel the reader has a better chance of understanding the contributions made by the other authors, and it also provides a different angle for describing the published formulations. Any minor extensions that we have proposed to the surveyed works are noted, however we leave the discussion of the performance of our implementations of the architectures and algorithms to Chapter 5.

3.1.1 Organization

The remainder of this chapter is organized as follows. In Section 3.2, we present the parallel formulation fundamentals. This includes a discussion of how a message is partitioned into blocks, and the binary polynomial and state-space approaches to obtain parallel primitive CRC formulations. In Section 3.3, we discuss the existing hardware architectures. In Section 3.4, we review the existing software algorithms and LUT generation algorithms. This chapter is concluded with a summary in Section 3.5.

3.2 Parallel Formulation Fundamentals

All the parallel CRC computation architectures discussed in this thesis rely on the message being partitioned into blocks. Formulations are typically presented by binary polynomial [13], [40] or state-space (matrix) representations [20], [24], [26], and are based on either the serial CRC LFSR1 or LFSR2 Architectures.

In this section, we first review the existing primitive parallel CRC formulations, for the serial CRC LFSR1 and LFSR2 Architectures. This includes binary polynomial and state-space derivations. We begin with the binary polynomial approach since it relies on a rigorous derivation for partitioning the message polynomial into blocks, and then the state-space approach is presented. We note that the same parallel CRC expressions can be obtained from using either of these two approaches, and later formulations typically use one of these approaches as a starting point for their derivation.

In addition to the binary polynomial and state-space approaches, a third approach based on z-transforms was proposed in [21]. Since none of the architectures and algorithms in this thesis are derived from this approach, we have presented the material in Appendix C.

3.2.1 Binary Polynomial Approach

The binary polynomial approach is derived directly from the CRC equation given in (2.2). Papers that adopted this approach include [13] and [40] for primitive LFSR2 and LFSR1 formulations, respectively. To begin our discussion, we must first set up the required mathematics to rigorously define the iterative CRC computation. This involves formally partitioning the message decomposing it into blocks of equal length. The material contained in this subsection is adapted from our conference paper [41] with some small notational changes, and extended to describe the LFSR1 formulation.

Message Partitioning

Parallel implementations of the CRC computation process the message block-wise iteratively. Let $q = \lceil \frac{k}{l} \rceil$ and $\tilde{q} = \lceil \frac{k+m}{l} \rceil$ denote the number of iterations required to process a k-bit message, where at each iteration l bits are processed using parallel architectures based on the serial CRC LFSR2 and LFSR1 Architectures, respectively. In this thesis, we refer to l as the degree of parallelism and to distinguish between LFSR1 and LFSR2 formulations, we mark the LFSR1 variables with tildes. Now, we describe how the message polynomial is formally partitioned for the LFSR2 and LFSR1 binary formulations.

LFSR2: Consider an LFSR2 based formulation, one can partition $U(x)$ into $q = \lceil \frac{k}{l} \rceil$ message blocks, i.e.,

$$U(x) = \sum_{i=0}^{q-1} x^{l\cdot(q-1-i)} \cdot B^{[i]}(x), \tag{3.1}$$

where $B^{[i]}(x)$ represents a binary polynomial of at most degree $l-1$ corresponding to the l-bit message block being processed at the i-th iteration. If $k \bmod l \neq 0$, then there are basically two solutions, and in this thesis we consider the simpler solution where one has the ability to prepend $l-(k \bmod l)$ 0s to $U(x)$ increasing the message length to a multiple of l. Alternatively, $l-(k \bmod l)$ 0s can be appended to $U(x)$ and the CRC value is modified after all the blocks have been processed [11], [42].

Let $U^{[i]}(x)$ be the portion of $U(x)$ that contains all the blocks $B^{[j]}(x)$ for $0 \leq j \leq i$, and let $S^{[i]}(x)$ be the syndrome of $U^{[i]}(x)$, i.e., a binary polynomial of at most degree $m-1$. Also, define $U^{[-1]}(x) = 0$ and $S^{[-1]}(x) = S_{\text{init}}(x)$, where $S_{\text{init}}(x)$ denotes the initial content of the CRC register. Then these definitions can be written as

$$\begin{aligned} U^{[i]}(x) &= x^l \cdot U^{[i-1]}(x) + B^{[i]}(x), \\ S^{[i]}(x) &= \left(x^m \cdot U^{[i]}(x)\right) \bmod G(x), \end{aligned}$$

for $0 \leq i \leq q-1$. It is noted that $U(x) = U^{[q-1]}(x)$ and $S(x) = S^{[q-1]}(x)$. For clarification, a pictorial illustration of the relationships between these recently introduced polynomials is shown in Figure 3.1.

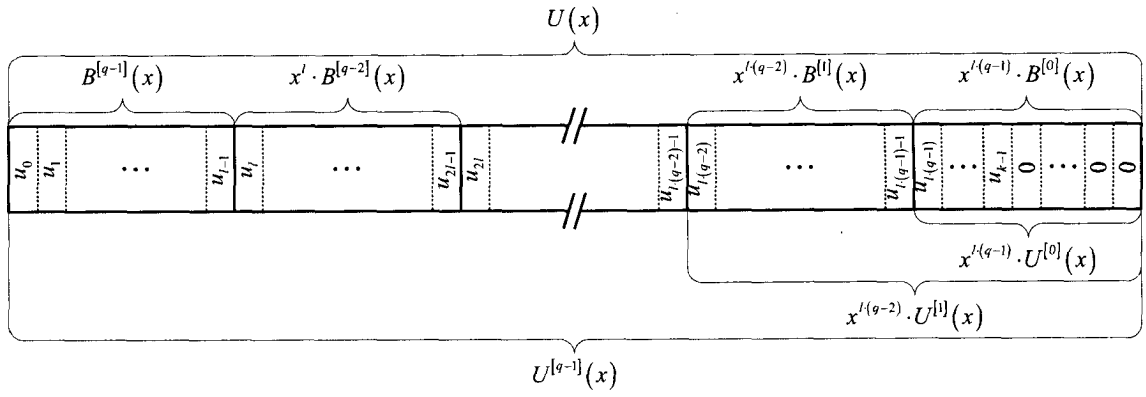


Figure 3.1: Illustration of the message polynomials.

LFSR1: A similar approach is taken for LFSR1 based formulations, where $\tilde{U}(x) = x^m \cdot U(x)$, and $\tilde{U}(x)$ is partitioned into $\tilde{q} = \lceil \frac{k+m}{l} \rceil$ message blocks, i.e.,

$$\tilde{U}(x) = \sum_{i=0}^{\tilde{q}-1} x^{l\cdot(\tilde{q}-1-i)} \cdot \tilde{B}^{[i]}(x), \tag{3.2}$$

where $\tilde{B}^{[i]}(x)$ represents a binary polynomial of at most degree $l-1$ corresponding to the l-bit message block being processed at the i-th iteration. If $(k+m) \bmod l \neq 0$, then we similarly assume one can prepend $l - ((k+m) \bmod l)$ 0s to $\tilde{U}(x)$ to increase the length of $\tilde{U}(x)$ to be a multiple of l.

Let $\tilde{U}^{[i]}(x)$ be the portion of $\tilde{U}(x)$ that contains all the blocks $\tilde{B}^{[j]}(x)$ for $0 \leq j \leq i$, and let $\tilde{S}^{[i]}(x)$ be the syndrome of $\tilde{U}^{[i]}(x)$, i.e., a binary polynomial of at most degree $m-1$. Also, define $\tilde{U}^{[-1]}(x) = 0$ and $\tilde{S}^{[-1]}(x) = S_{\text{init}}(x)$, then these definitions can be written as

$$\begin{aligned} \tilde{U}^{[i]}(x) &= x^l \cdot \tilde{U}^{[i-1]}(x) + \tilde{B}^{[i]}(x), \\ \tilde{S}^{[i]}(x) &= \tilde{U}^{[i]}(x) \bmod G(x), \end{aligned}$$

for $0 \leq i \leq \tilde{q}-1$. It is noted that $\tilde{U}(x) = \tilde{U}^{[\tilde{q}-1]}(x)$ and $S(x) = \tilde{S}^{[\tilde{q}-1]}(x)$.

LFSR2 Formulation

From the previous definitions, one can derive a recursive expression for $S^{[i]}(x)$ in terms of $S^{[i-1]}(x)$ and $B^{[i]}(x)$, i.e.,

$$\begin{aligned} S^{[i]}(x) &= \left(x^m \cdot U^{[i]}(x)\right) \bmod G(x) \\ &= \left(x^m \cdot \left(x^l \cdot U^{[i-1]}(x) + B^{[i]}(x)\right)\right) \bmod G(x) \\ &= \left(x^l \cdot x^m \cdot U^{[i-1]}(x) + x^m \cdot B^{[i]}(x)\right) \bmod G(x) \\ &= \left(x^l \cdot S^{[i-1]}(x) + x^m \cdot B^{[i]}(x)\right) \bmod G(x) \\ &= T^{[i]}(x) \bmod G(x), \end{aligned} \tag{3.3}$$

for $0 \leq i \leq q-1$, where

$$T^{[i]}(x) = x^l \cdot S^{[i-1]}(x) + x^m \cdot B^{[i]}(x). \tag{3.4}$$

Observing (3.3), there are three cases to consider: $l = m$, $l < m$, and $l > m$, called Cases I, II, and III, respectively. Each case yields a slightly different formulation with regards which present terms in the polynomial $T^{[i]}(x)$ require reduction [43]. These cases are discussed separately.

Case I: When the degree of parallelism is equal to the generator polynomial degree, i.e., $l = m$, (3.4) becomes

$$\begin{aligned} T^{[i]}(x) &= x^m \cdot \sum_{j=0}^{m-1} s_j^{[i-1]} x^j + x^m \cdot \sum_{j=0}^{m-1} b_j^{[i]} x^j \\ &= x^m \cdot \sum_{j=0}^{m-1} \left(s_j^{[i-1]} + b_j^{[i]}\right) x^j. \end{aligned} \tag{3.5}$$

In (3.5), the terms in the polynomials $S^{[i-1]}(x)$ and $B^{[i]}(x)$ completely overlap, and one can factor the x^m term out to show that all the terms in $T^{[i]}(x)$ require reduction.

Case II $l < m$: When the degree of parallelism is less than the generator polynomial degree, i.e., $l < m$, (3.4) becomes

$$\begin{aligned} T^{[i]}(x) &= x^l \cdot \sum_{j=0}^{m-1} s_j^{[i-1]} x^j + x^m \cdot \sum_{j=0}^{l-1} b_j^{[i]} x^j \\ &= x^l \cdot \sum_{j=0}^{m-l-1} s_j^{[i-1]} x^j + x^m \cdot \sum_{j=0}^{l-1} \left(s_{j+m-l}^{[i-1]} + b_j^{[i]}\right) x^j. \end{aligned} \tag{3.6}$$

In (3.6), the polynomial $T^{[i]}(x)$ has l terms that overlap and require reduction between $S^{[i-1]}(x)$ and $B^{[i]}(x)$, while $m - l$ terms of $S^{[i-1]}(x)$ do not require reduction.

Case III $l > m$**:** When the degree of parallelism is greater than the generator polynomial degree, i.e., $l > m$, (3.4) becomes

$$\begin{aligned} T^{[i]}(x) &= x^m \cdot \left(x^{l-m} \cdot \sum_{j=0}^{m-1} s_j^{[i-1]} x^j + \sum_{j=0}^{l-1} b_j^{[i]} x^j \right) \\ &= x^m \cdot \left(\sum_{j=0}^{l-m-1} b_j^{[i]} x^j + \sum_{j=0}^{m-1} \left(s_j^{[i-1]} + b_{j+l-m}^{[i]} \right) x^{j+l-m} \right). \end{aligned} \tag{3.7}$$

In (3.7), the all of the terms in the $S^{[i-1]}(x)$ overlap with the m greatest power terms of $B^{[i]}(x)$. Again, one can factor the x^m term out to show that all of the terms in $T^{[i]}(x)$ require reduction.

LFSR1 Formulation

In [40], the author proposed the first parallel LFSR1 formulation, using Galois field multiplications. Here, we take a slightly different approach and modify our LFSR2 derivation to obtain a parallel LFSR1 formulation. By combining the x^m term into the message polynomial $U(x)$ to form $\tilde{U}(x)$ in (2.2), one obtains

$$\begin{aligned} S(x) &= (x^m \cdot U(x)) \bmod G(x) \\ &= \tilde{U}(x) \bmod G(x). \end{aligned} \tag{3.8}$$

Again, from the previous definitions, one can derive a recursive expression for $\tilde{S}^{[i]}(x)$ in terms of $\tilde{S}^{[i-1]}(x)$ and $\tilde{B}^{[i]}(x)$, i.e.,

$$\begin{aligned} \tilde{S}^{[i]}(x) &= \tilde{U}^{[i]}(x) \bmod G(x) \\ &= \left(x^l \cdot \tilde{U}^{[i-1]}(x) + \tilde{B}^{[i]}(x) \right) \bmod G(x) \\ &= \left(x^l \cdot \tilde{S}^{[i-1]}(x) + \tilde{B}^{[i]}(x) \right) \bmod G(x) \\ &= \tilde{T}^{[i]}(x) \bmod G(x), \end{aligned} \tag{3.9}$$

for $0 \leq i \leq \tilde{q} - 1$, where

$$\tilde{T}^{[i]}(x) = x^l \cdot \tilde{S}^{[i-1]}(x) + \tilde{B}^{[i]}(x). \tag{3.10}$$

From (3.9), one observes that for all l and m cases, l terms in $\tilde{T}^{[i]}(x)$ require reduction and no terms between $x^l \cdot \tilde{S}^{[i-1]}(x)$ and $\tilde{B}^{[i]}(x)$ overlap. These facts are clearly illustrated by examining an expanded form of (3.10),

$$\tilde{T}^{[i]}(x) = \sum_{j=0}^{l-1} \tilde{b}_j^{[i]} x^j + x^l \cdot \sum_{j=0}^{m-1} \tilde{s}_j^{[i-1]} x^j.$$

3.2.2 State-Space Approach

The second approach that we will discuss for obtaining the primitive parallel CRC equations consists of developing a state-space model for the serial LFSR architectures, and subsequently extending them to parallel input cases.

Fundamentals

In this subsection, notations similar to those used in linear systems theory are used to describe the typical state-space model. The general linear, step-invariant, discrete-time state space is commonly represented as,

$$\begin{aligned} \mathbf{x}_{n\times 1}[k+1] &= \mathbf{A}_{n\times n} \cdot \mathbf{x}_{n\times 1}[k] + \mathbf{B}_{n\times r} \cdot \mathbf{u}_{r\times 1}[k] \\ \mathbf{y}_{m\times 1}[k] &= \mathbf{C}_{m\times n} \cdot \mathbf{x}_{n\times 1}[k] + \mathbf{D}_{m\times r} \cdot \mathbf{u}_{r\times 1}[k], \end{aligned} \tag{3.11}$$

where the vectors $\mathbf{x}_{n\times 1}$, $\mathbf{u}_{r\times 1}$, and $\mathbf{y}_{m\times 1}$ denote the state, input, and output vectors, respectively, and the matrices $\mathbf{A}_{n\times n}$, $\mathbf{B}_{n\times r}$, $\mathbf{C}_{m\times n}$, and $\mathbf{D}_{m\times r}$ denote the state, input, output, and input-to-output coupling matrices, respectively. A common illustration in block diagram form of the general state-space model is shown in Figure 3.2 [44].

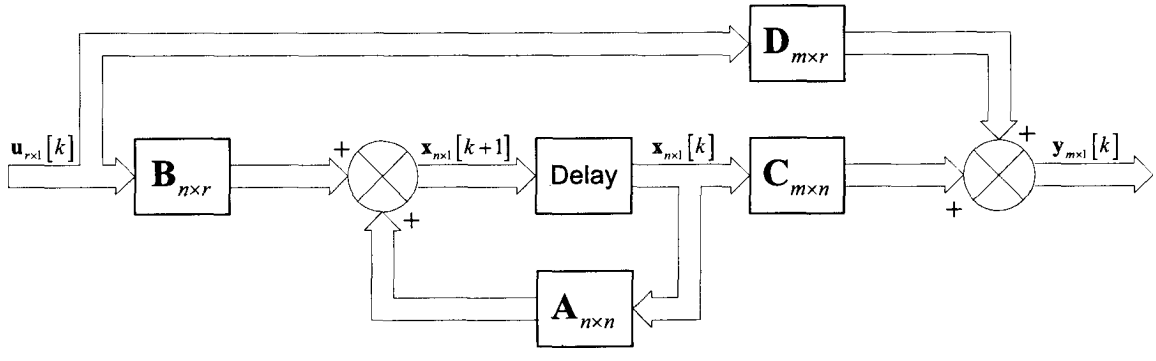


Figure 3.2: General state-space model [44].

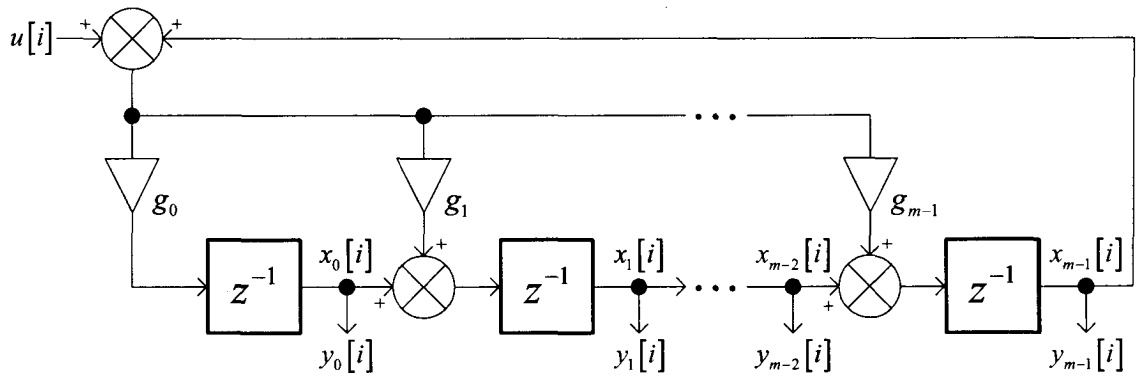


Figure 3.3: Delay diagram of the serial CRC LFSR2 Architecture.

LFSR2 State-Space Formulation

Published CRC LFSR2 state-space models that use nearly identical notations to the ones used in this thesis are presented in [20], [24], and [45], and two other examples of models with slightly different notations can be found in [17] and [22]. When comparing the notations used here to the ones from digital system theory, the same letters for the coupling matrices will be maintained, however to mark their dimensions, the CRC variables m and l will be used, therefore $n = m$ and $r = l$, also the output vector size is equal m. Finally, we substitute k with i for the iteration index. Observing the architecture in Figure 3.3, and one can derive the following state-space model for the serial CRC LFSR2 Architecture,

$$\begin{aligned} \mathbf{x}_{m\times 1}[i+1] &= \mathbf{A}_{m\times m}\cdot\mathbf{x}_{m\times 1} + \mathbf{b}_{m\times 1}\cdot u[i] \\ \mathbf{y}_{m\times 1}[i] &= \mathbf{C}_{m\times m}\cdot\mathbf{x}_{m\times 1} + \mathbf{d}_{m\times 1}\cdot u[i], \end{aligned} \tag{3.12}$$

for $0 \le i \le k-1$, where $\mathbf{A}_{m\times m} = \begin{bmatrix} 0 & 0 & 0 & & 0 & g_0 \\ 1 & 0 & 0 & & 0 & g_1 \\ 0 & 1 & 0 & & 0 & g_2 \\ \vdots & & & & & \vdots \\ 0 & 0 & 0 & \cdots & 1 & g_{m-1} \end{bmatrix}$, $\mathbf{b}_{m\times 1} = \begin{bmatrix} g_0 \\ g_1 \\ g_2 \\ \vdots \\ g_{m-1} \end{bmatrix}$, $\mathbf{C}_{m\times m} = \mathbf{I}_{m\times m}$, $\mathbf{d}_{m\times 1} = \mathbf{0}_{m\times 1}$, and $u[i] = u_{k-1-i}$ from (2.2).

The serial formulation in (3.12) can easily be extended to a parallel formulation that processes l message bits in an iteration,

$$\begin{aligned} \bar{\mathbf{x}}_{m\times 1}[i+1] &= \bar{\mathbf{A}}_{m\times m}\cdot\bar{\mathbf{x}}_{m\times 1}[i] + \bar{\mathbf{B}}_{m\times l}\cdot\bar{\mathbf{u}}_{l\times 1}[i] \\ \bar{\mathbf{y}}_{m\times 1}[i] &= \bar{\mathbf{C}}_{m\times m}\cdot\bar{\mathbf{x}}_{m\times 1}[i] + \bar{\mathbf{D}}_{m\times l}\cdot\bar{\mathbf{u}}_{l\times 1}[i], \end{aligned} \tag{3.13}$$

for $0 \leq i \leq \left\lceil \frac{k-1}{l} \right\rceil$, where $\bar{\mathbf{A}}_{m\times m} = (\mathbf{A}_{m\times m})^l$,

$$\bar{\mathbf{B}}_{m\times l} = \left[\begin{array}{cccc} (\mathbf{A}_{m\times m})^0 \cdot \mathbf{b}_{l\times 1} & (\mathbf{A}_{m\times m})^1 \cdot \mathbf{A}_{m\times m} \cdot \mathbf{b}_{l\times 1} & \cdots & (\mathbf{A}_{m\times m})^{l-1} \cdot \mathbf{b}_{l\times 1} \end{array} \right],$$

$\bar{\mathbf{C}}_{m\times m} = \mathbf{C}_{m\times m} = \mathbf{I}_{m\times m}$, $\bar{\mathbf{D}}_{m\times l} = \mathbf{0}_{m\times l}$, and

$$\bar{\mathbf{u}}_{l\times 1}[i] = \left[\begin{array}{cccc} u[i\cdot l + (l-1)] & u[i\cdot l + (l-2)] & \cdots & u[i\cdot l] \end{array} \right]^T.$$

Here we have assumed that the message has been properly partitioned, and to avoid a notation clash with the serial state-space model (3.12), we have marked the parallel state-space matrices and vectors in (3.13) with bars.

LFSR1 State-Space Formulation

In [26], a state-space model was presented for the serial CRC LFSR1 Architecture that was extended to the parallel case. We have modified the notations used in that paper to make them more consistent with the notations used in this thesis to describe the serial LFSR2 Architecture. Furthermore, the approach presented in [26] cannot be extended for situations when the degree of parallelism is greater than the generator polynomial degree, and the formulation we present next does not impose those restrictions.

Observing the architecture in Figure 3.4, and one can derive the following state space,

$$\begin{aligned} \tilde{\mathbf{x}}_{m\times 1}[i+1] &= \tilde{\mathbf{A}}_{m\times m} \cdot \tilde{\mathbf{x}}_{m\times 1}[i] + \tilde{\mathbf{b}}_{m\times 1} \cdot \tilde{u}[i] \\ \tilde{\mathbf{y}}_{m\times 1}[i] &= \tilde{\mathbf{C}}_{m\times m} \cdot \tilde{\mathbf{x}}_{m\times 1}[i] + \tilde{\mathbf{d}}_{m\times 1} \cdot \tilde{u}[i], \end{aligned} \tag{3.14}$$

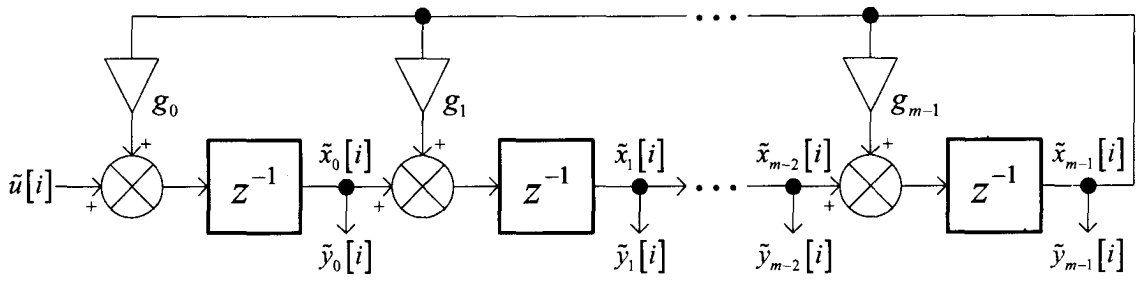


Figure 3.4: Delay diagram of the serial CRC LFSR1 Architecture.

for $0 \leq i \leq k+m-1$, where $\tilde{\mathbf{A}}_{m\times m} = \begin{bmatrix} 0 & 0 & 0 & & 0 & g_0 \\ 1 & 0 & 0 & & 0 & g_1 \\ 0 & 1 & 0 & & 0 & g_2 \\ \vdots & & & & & \vdots \\ 0 & 0 & 0 & \cdots & 1 & g_{m-1} \end{bmatrix}$, $\tilde{\mathbf{b}}_{m\times 1} = \begin{bmatrix} 1 \\ 0 \\ 0 \\ \vdots \\ 0 \end{bmatrix}$, $\tilde{\mathbf{C}}_{m\times m} = \mathbf{I}_{m\times m}$, and $\tilde{\mathbf{d}}_{m\times 1} = \mathbf{0}_{m\times 1}$. It is noted that $\tilde{\mathbf{A}}_{m\times m} = \mathbf{A}_{m\times m}$, $\tilde{\mathbf{C}}_{m\times m} = \mathbf{C}_{m\times m}$, $\tilde{\mathbf{d}}_{m\times 1} = \mathbf{d}_{m\times 1}$, and $\tilde{u}[i] = \tilde{u}_{k+m-1-i}$ from (3.8). Again, to avoid a notation clash with the LFSR2 state-space model, we mark the LFSR1 matrices and vectors in (3.14) with tildes.

For the cases when the degree of parallelism is less than or equal to the generator polynomial degree $l \leq m$, this formulation can easily be extended to

$$\begin{aligned} \bar{\tilde{\mathbf{x}}}_{m\times 1}[i+1] &= \bar{\tilde{\mathbf{A}}}_{m\times m} \cdot \bar{\tilde{\mathbf{x}}}_{m\times 1}[i] + \bar{\tilde{\mathbf{B}}}_{m\times l} \cdot \bar{\tilde{\mathbf{u}}}_{l\times 1}[i] \\ \bar{\tilde{\mathbf{y}}}_{m\times 1}[i] &= \bar{\tilde{\mathbf{C}}}_{m\times m} \cdot \bar{\tilde{\mathbf{x}}}_{m\times 1}[i] + \bar{\tilde{\mathbf{D}}}_{m\times l} \cdot \bar{\tilde{\mathbf{u}}}_{l\times 1}[i], \end{aligned} \tag{3.15}$$

for $0 \leq i \leq \left\lceil \frac{k+m-1}{l} \right\rceil$, where $\bar{\tilde{\mathbf{A}}}_{m\times m} = \left(\tilde{\mathbf{A}}_{m\times m}\right)^l$, $\bar{\tilde{\mathbf{B}}}_{m\times l} = \begin{bmatrix} \mathbf{I}_{l\times l} \\ \mathbf{0}_{(m-l)\times l} \end{bmatrix}$, $\bar{\tilde{\mathbf{C}}}_{m\times m} = \mathbf{I}_{m\times m}$, $\bar{\tilde{\mathbf{D}}}_{m\times l} = \mathbf{0}_{m\times l}$, and

$$\bar{\tilde{\mathbf{u}}}_{l\times 1}[i] = \begin{bmatrix} \tilde{u}[i\cdot l + (l-1)] & \tilde{u}[i\cdot l + (l-2)] & \cdots & \tilde{u}[i\cdot l] \end{bmatrix}^T.$$

It is noted that $\bar{\tilde{\mathbf{A}}}_{m\times m} = \bar{\mathbf{A}}_{m\times m}$, $\bar{\tilde{\mathbf{C}}}_{m\times m} = \bar{\mathbf{C}}_{m\times m}$, and $\bar{\tilde{\mathbf{D}}}_{m\times l} = \bar{\mathbf{D}}_{m\times l}$.

For the cases when $l > m$, the input coupling matrix becomes

$$\bar{\tilde{\mathbf{B}}}_{m\times l} = \begin{bmatrix} \mathbf{I}_{m\times m} & \left(\tilde{\mathbf{A}}_{m\times m}\right)^0 \cdot \mathbf{b}_{m\times 1} & \left(\tilde{\mathbf{A}}_{m\times m}\right)^1 \cdot \mathbf{b}_{m\times 1} & \cdots & \left(\tilde{\mathbf{A}}_{m\times m}\right)^{l-m-1} \cdot \mathbf{b}_{m\times 1} \end{bmatrix}$$

where $\mathbf{b}_{m\times 1}$ is the input coupling matrix of the serial LFSR2 state-space model (3.12), and all the other coupling matrices in (3.15) are unchanged.

3.3 Hardware Architectures

In this section, we review most of the published hardware CRC computation architectures found in the literature [10], [17], [20], [21], [22], [23], [24], [25], [26], [40], [46], [47], [48], and [42]. The two parallel LFSR Architectures derived from the primitive formulations developed in the previous section are examined first. Afterward, the Two-Step [46], Cascade [23], Look-Ahead [25], State-Space Transformed [24], and

Retimed Architectures [10], [42], [47] , and [48] are discussed. For most of the approaches, we present a generalized formulation that extends from a primitive one, and its subsequent hardware realization.

3.3.1 Primitive Architectures

The parallel LFSR Architectures that perform CRC computation are hardware realizations of the primitive parallel formulations that were derived in the previous section. Since the expressions are in the primitive form, they are the fastest non-pipelined hardware architectures that perform the CRC computation. In other words, there is no cancellation or sharing of terms between the parallel equations.

We have chosen to present these architectures using binary polynomial notations, and we briefly explain how the state-space or z-transform approach can also be used to obtain the equivalent parallel architectures. It is important to remember that a designer can use whichever method they are most comfortable with to obtain their desired implementation.

Before beginning this discussion, we note that the publications concerning parallel LFSR Architectures have not considered the case when $l > m$. The figures drawn in this chapter are rather general and in Chapter 4 we provide detailed design that is specific for the parallel LFSR2 Architecture when $l > m$. This can be found in our conference paper [41].

Parallel LFSR2 Architecture

The parallel LFSR2 Architecture hardware realization that is illustrated in Figure 3.5, can be obtained by directly mapping (3.4) to a hardware architecture. One observes that there is a level of XOR gates required to combine the overlapping terms between the $x^l \cdot S^{[i-1]}(x)$ and $x^m \cdot B^{[i]}(x)$ polynomials, and then a block of XOR tress to

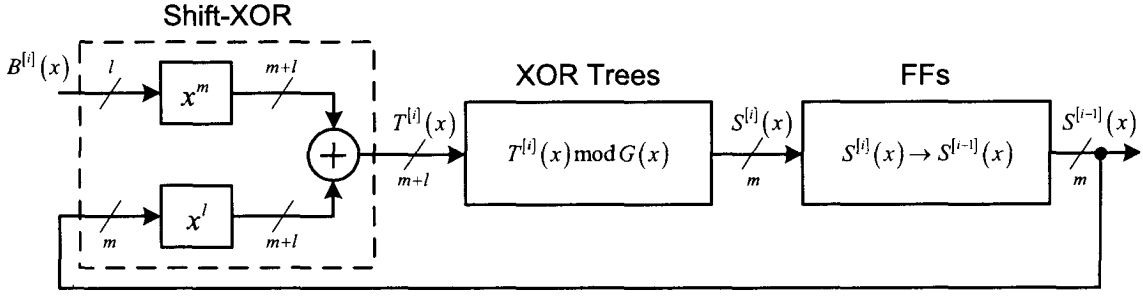


Figure 3.5: Generalized parallel LFSR2 Architecture.

perform the reduction of $T^{[i]}(x)$ mod $G(x)$. Finally there is a block that contains an array of FFs to store the $S^{[i]}(x)$ results. The latency of this approach is $\lceil \frac{k}{l} \rceil$ clock cycles, and the reader can consult [17], [20], [21], [22], and [45] for further information.

Now, to obtain an equivalent architecture using the state-space approach, we note that j-th columns of $\bar{\mathbf{A}}_{m\times m}$ and $\bar{\mathbf{B}}_{m\times l}$ correspond to the coefficients of x^{l+j} mod $G(x)$ and x^{m+j} mod $G(x)$, respectively. Depending on l and m, there will be columns that are identical between $\bar{\mathbf{A}}_{m\times m}$ and $\bar{\mathbf{B}}_{m\times l}$, and these matrices can be seen as alternative representations of $\left(x^l \cdot S^{[i-1]}(x)\right)$ mod $G(x)$ and $\left(x^m \cdot B^{[i]}(x)\right)$ mod $G(x)$. Finally, output coupling matrix $\bar{\mathbf{C}}_{m\times m}$ is an identity matrix, therefore, by combining the common columns in the input and state coupling matrices with a level of XOR gates, a representation equivalent to the binary polynomial approach is obtained.

The z-transform approach [21] is quite similar to the binary polynomial approach, only the calculation of the parallel expressions is different. With the binary polynomial approach, one selects a method to compute $S^{[i]}(x) = T^{[i]}(x)$ mod $G(x)$ and develop the m parallel equations. Alternatively, the convolution operation is used to construct the parallel equations and the realization of the result can be achieved with an architecture similar to Figure 3.5.

Finally, we note that is in this thesis we have assumed that is customary to first fix the generator polynomial and degree of parallelism before designing and implementing an architecture. Recently, a programmable CRC architecture based on the parallel LFSR2 Architecture [22] was proposed. However, we consider this topic to be outside the scope of this thesis and the interested reader can consult [49] for further information.

Parallel LFSR1 Architecture

Similarly, the parallel LFSR1 Architecture hardware realization that is illustrated in Figure 3.6 can be obtained by directly mapping (3.10) to a hardware architecture. From the figure, one observes that the high-level structure is nearly identical to that the parallel LFSR2 Architecture in Figure 3.5, but in this case the input polynomial is not augmented before it is added to the previous syndrome. Later, it is shown that in some situations the CPD of this architecture can one less T_X than the corresponding LFSR2 realization. The latency of this approach is $\lceil \frac{k+m}{l} \rceil$ clock cycles, and the reader can consult [26] and [40] for further information.

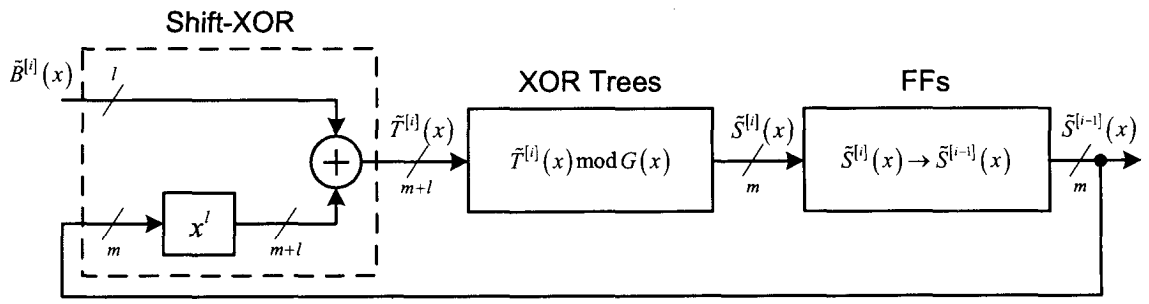


Figure 3.6: Generalized parallel LFSR1 Architecture.

3.3.2 Two-Step Architecture

In [46], the Two-Step Architecture for performing the CRC computation was purposed. Basically, the author showed that it is possible to first use a polynomial that is a multiple of the generator polynomial to perform the reduction of the message and obtain a fixed-length intermediate result. Then, the intermediate is subsequently reduced by the generator polynomial to obtain the final syndrome.

The key to this approach is to find the multiple polynomial $M(x)$ that results in a parallel LFSR1 implementation with CPD T_X. To achieve this, the present non-zero coefficients of $M(x)$ must be spaced at least l terms apart [46]. Since this method is based on an LFSR1 formulation [26], [40], an additional m 0s must be appended to the message to obtain the correct result, and this fact is not mentioned in [46]. The author notes that the second reduction only needs to be performed once, and since the intermediate is of a fixed length, the implementation of the second block does not require feedback connections.

Formulation

The formulation for the Two-Step Architecture can be expressed by first defining the multiple polynomial

$$M(x) = P(x) \cdot G(x),$$

where the multiplicand polynomial $P(x) = 1 + \sum_{i=1}^{m-1} p_i x^i + x^p$, and

$$M(x) = 1 + \sum_{i=1}^{m+p-1} m_i x^i + x^{m+p} \tag{3.16}$$

such that when $G(x)$ is multiplied by $P(x)$, $M(x)$ has its present non-zero coefficients spaced apart by at least l terms. Then, the first step of the reduction is performed as

$$\begin{aligned} S_M^{[i]}(x) &= \left(x^l \cdot \tilde{S}^{[i-1]} + \tilde{B}^{[i]}(x)\right) \bmod M(x) \\ &= \tilde{T}^{[i]}(x) \bmod M(x). \end{aligned}$$

for $0 \leq i \leq \tilde{q} - 1$, where $l < m + p$. After the intermediate $S_M(x)$ is obtained, it is subsequently reduced by the generator polynomial $G(x)$ and the final syndrome $S(x)$ is obtained as

$$S(x) = S_M^{[\tilde{q}-1]}(x) \bmod G(x).$$

Multiple Polynomial Search

As one may imagine, it is difficult to find $P(x)$ polynomials which produce suitable $M(x)$ polynomials that result in implementations with first stage CPD of T_X for large degrees of parallelism. Little insight is provided into the method used in [46] to find $P(x)$ for CRC-32 with $l = 8$ other than the author mentioning that an exhaustive search was performed. Out of curiosity, we wrote C++ code that performed a search for $P(x)$ polynomials for the other frequently referenced generator polynomials with $l = 8$ and the found $M(x)$ are listed in Table 3.1.

The search strategy that we devised can be summarized as an improved exhaustive search technique. After completing an exhaustive search to find a suitable $M(x)$ for the CRC-12, which consisted of testing each $P(x)$ of the form $P(x) = 1 + \sum_{i=1}^{p-1} p_i x^i + x^p$ up to degree $p = 32$, i.e., 2^{31} distinct polynomials, we proceeded to cover the same search space for CRC-16 without any luck. We then reconsidered the problem and noticed that it is possible to solve for the coefficients of the terms with the l smallest and greatest l powers of $P(x)$. By knowing that $M(x)$ is of the form (3.16), but

Table 3.1: Two-Step Architecture multiple polynomials for frequently referenced generator polynomials when $l = 8$.

$G(x)$	$M(x)$
CRC-12	$1 + x^{12} + x^{30} + x^{44}$
CRC-16	$1 + x^{9} + x^{22} + x^{34} + x^{51} + x^{60}$
CCITT-16	$1 + x^{10} + x^{24} + x^{32}$
CRC-16†	$1 + x^{9} + x^{26} + x^{38} + x^{51} + x^{60}$
CCITT-16†	$1 + x^{8} + x^{22} + x^{32}$
CRC-32*	$1 + x^{23} + x^{46} + x^{64} + x^{84} + x^{92} + x^{111} + x^{123}$

* reported in [46].

more specifically,

$$M(x) = 1 + \sum_{i=l}^{m+p-l} m_i x^i + x^{m+p},$$

i.e., $m_i = 0$ for $1 \leq i \leq l-1$ and $m+p-l \leq i \leq m+p-1$. In other words, after the x^0 term the next $l-1$ terms must not be present, and before the x^{m+p} term the previous $l-1$ terms are not present. Consider extracting the l smallest and greatest degree terms from $P(x)$ and $G(x)$, and then call those polynomials $P_S(x)$ and $P_G(x)$, and $G_S(x)$ and $G_G(x)$, respectively. We know that for the smallest l terms

$$P_S(x) \cdot G_S(x) = 1 + \sum_{i=1}^{l-1} 0x^i + x^{\tau_S} + \cdots,$$

where $\tau_S \geq l$, and there is only one $P_S(x)$ that satisfies that relation. A similar argument can be made for the product of the greatest l terms of $P(x)$ and $G(x)$ resulting in

$$P_G(x) \cdot G_G(x) = \cdots + x^{\tau_E} + \sum_{i=m+p-l+1}^{m+p-1} 0x^i + x^{m+p},$$

where $\tau_E \leq m+p-l$, and there is only one $P_G(x)$ that satisfies that relation. In Tables 3.2a and 3.2b the $P_S(x)$ and $P_G(x)$ polynomials for common generator polynomials with degree of parallelism $l = 8$ are listed, respectively.

To put it in perspective, the naive exhaustive search for the CRC-12 multiple polynomials lasted approximately one week (with the simulation running constantly). With the improved method, we are able to complete the same search in about one minute. However, we note that this approach cannot be easily extended to solve for more coefficients in $M(x)$. Consequently, it would not be feasible to find the $M(x)$ polynomial for CRC-32 reported in [46]. An alternative approach could test $M(x) \bmod G(x) = 0$ using one of the software CRC computation algorithms, where the coefficients in $M(x)$ are at least l terms apart.

Realization

The approach to realize the Two-Step Architecture suggested by the author consists of two blocks, and an illustration of the architecture is given in Figure 3.7. The first block which is denoted by $\tilde{T}^{[i]}(x) \bmod G(x)$, is used to perform the initial reduction by $M(x)$ and can be executed at a very high clock rate. The second block, denoted by $\tilde{S}_M(x) \bmod G(x)$, performs the final reduction by $G(x)$ once, at a much slower rate. However, since the second block does not have a feedback connection, it can be

Table 3.2: Two-Step Architecture multiplicand polynomials for frequently referenced generator polynomials when $l = 8$: (a) smallest order terms, (b) greatest order terms.

(a)

$G(x)$	$P_S(x)$
CRC-12	$1 + x + x^4 + x^5$
CRC-16	$1 + x^2 + x^4 + x^5$
CCITT-16	$1 + x^5$
CRC-16†	$1 + x + x^2 + x^3 + x^4 + x^5 + x^6 + x^7$
CCITT-16†	$1 + x^4$
CRC-32	$1 + x + x^3 + x^5 + x^7$

(b)

$G(x)$	$P_G(x)$
CRC-12	$x^{p-7} + x^{p-6} + x^{p-5} + x^{p-4} + x^{p-3} + x^{p-2} + x^{p-1} + x^p$
CRC-16	$x^{p-7} + x^{p-6} + x^{p-5} + x^{p-4} + x^{p-3} + x^{p-2} + x^{p-1} + x^p$
CCITT-16	$x^{p-4} + x^p$
CRC-16†	$x^{p-6} + x^{p-4} + x^{p-2} + x^p$
CCITT-16†	$x^{p-5} + x^p$
CRC-32	$x^{p-6} + x^p$

easily pipelined and this extension is not stated in [46]. Thus it is possible to derive a pipelined parallel implementation with CPD T_X. Furthermore, as stated earlier, the author neglects to mention the fact that an additional m 0s must be appended to the message to make the scheme functional. So, if the second block is pipelined, then the latency of this approach is $\lceil \frac{k+m}{l} \rceil + \delta_p$ clock cycles, where δ_p denotes the number of pipeline stages in the second block.

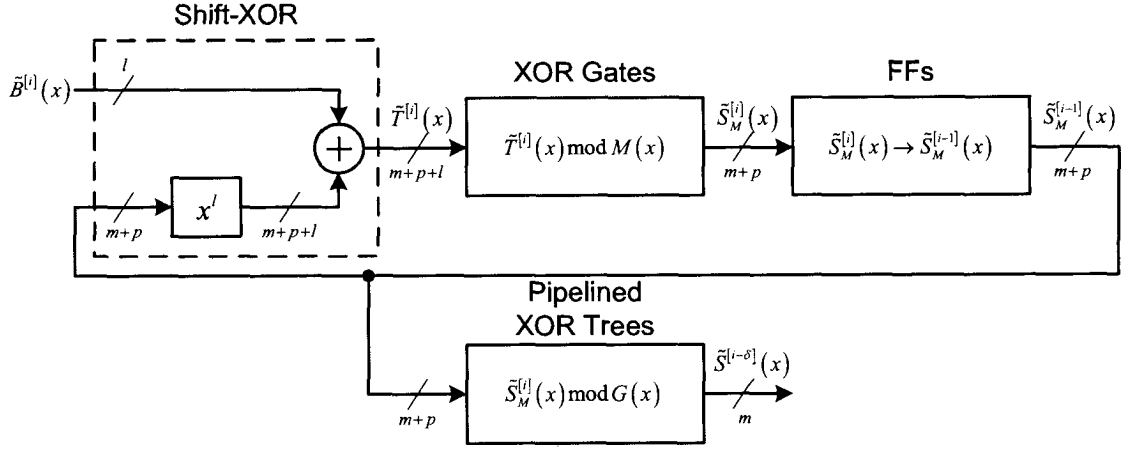


Figure 3.7: Generalized Two-Step Architecture.

3.3.3 Cascade Architecture

In [23], the idea of cascading the serial LFSR2 combination logic to obtain the Cascade Architecture was presented. In this approach, the author notes that if one separates the combinational logic from the register in the serial LFSR2 Architecture, and cascades copies of it, then a parallel architecture is obtained. The strength of this approach is its implementation simplicity. A designer with limited knowledge of CRCs can easily derive and implement a parallel architecture from the serial architecture without restriction on the desired degree of parallelism. Other papers that use different forms of this approach include [50], [51], and [52].

Formulation

The formulation of this approach is quite simple, beginning with the recursive equation in (3.3), and letting the degree of parallelism $l = 1$ one can obtain,

$$\begin{aligned} S^{[i]}(x) &= \left(x^{(1)} \cdot S^{[i-1]}(x) + x^m \cdot B^{[i]}(x)\right) \bmod G(x) \\ &= \left(x \cdot S^{[i-1]}(x) + x^m \cdot b_0^{[i]}\right) \bmod G(x) \\ &= \mathrm{crc}_1\left(S^{[i-1]}(x), b_0^{[i]}\right), \end{aligned} \tag{3.17}$$

where $\mathrm{crc}_1\left(S^{[i-1]}(x), b_0^{[i]}\right) = \left(x \cdot S^{[i-1]}(x) + x^m \cdot b_0^{[i]}(x)\right) \bmod G(x)$. Now, writing an equation for $S^{[i+1]}(x)$ from (3.17), one can obtain

$$\begin{aligned} S^{[i+1]}(x) &= \mathrm{crc}_1\left(S^{[i]}(x), b_0^{[i+1]}\right) \\ &= \mathrm{crc}_1\left(\mathrm{crc}_1\left(S^{[i-1]}(x), b_0^{[i]}\right), b_0^{[i+1]}\right), \end{aligned}$$

which can be generalized for $S^{[i+l]}(x)$ as,

$$\begin{aligned} S^{[i+l]}(x) &= \mathrm{crc}_1\left(S^{[i+l-1]}(x), b_0^{[i+l]}\right) \\ &= \mathrm{crc}_1\left(\mathrm{crc}_1\left(S^{[i+l-2]}(x), b_0^{(i+l-1)}\right), b_0^{[i+l]}\right) \\ &\vdots \\ &= \mathrm{crc}_1\left(\mathrm{crc}_1\left(\cdots \mathrm{crc}_1\left(S^{[i-1]}(x), b_0^{[i]}\right), \cdots, b_0^{[i+l-1]}\right), b_0^{[i+l]}\right). \end{aligned} \tag{3.18}$$

Now, if one considers a change of variables in (3.18) to describe a system that processes l message bits at a time, by defining $B^{[i]}(x) = \sum_{j=0}^{l-1} b_j^{[i]} x^j$ where $b_j^{[i]} = b_0^{[i+l-(j+1)]}$, and replacing $S^{[i+l]}(x)$ with $S^{[i]}(x)$, one can obtain,

$$S^{[i]}(x) = \mathrm{crc}_1\left(\mathrm{crc}_1\left(\cdots \mathrm{crc}_1\left(S^{[i-1]}(x), b_{l-1}^{[i]}\right), \cdots, b_1^{[i]}\right), b_0^{[i]}\right). \tag{3.19}$$

The formulation in (3.19) shows that the CRC operation as defined in (3.17) can be applied repeatedly during a single clock cycle to obtain a parallel architecture.

Though it was not mentioned in [23], it is clear that this approach can be generalized by defining the CRC function to process more than one message bit at a time, i.e.,

$$\mathrm{crc}_l\left(S^{[i-1]}(x), B^{[i]}(x)\right) = \left(x^l \cdot S^{[i-1]}(x) + x^m \cdot B^{[i]}(x)\right) \bmod G(x),$$

where $B^{[i]}(x) = \sum_{j=0}^{l-1} b_j^{[i]} x^j$. In fact, this concept is extended in the following section and employed to develop architectures that are flexible in the sense that the degree of parallelism can altered without modification to the architecture. However, the main advantage of the cascading approach is to allow a designer with limited knowledge to implement it, thus it is best suited for the serial cascade case.

An extension that can be more useful is cascading the LFSR1 combinational logic instead of the LFSR2 combinational logic. The CPD of the LFSR1 logic is T_X, while the LFSR2 logic has a CPD of $2 \cdot T_X$. Thus, for some generator polynomials, one can reduce the CPD by using the LFSR1 logic. However, if the LFSR1 logic is used, then an additional m 0s must be appended to the message to obtain the correct result.

Realization

The generalized hardware realization of (3.19) is illustrated in Figure 3.8, and Figures 3.9a and 3.9b illustrate the generalized LFSR1 and LFSR2 combinational logic,

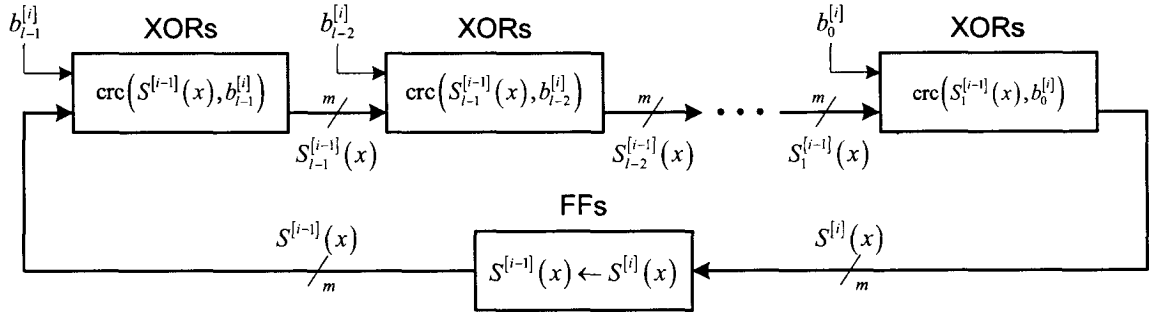


Figure 3.8: Generalized Cascade Architecture.

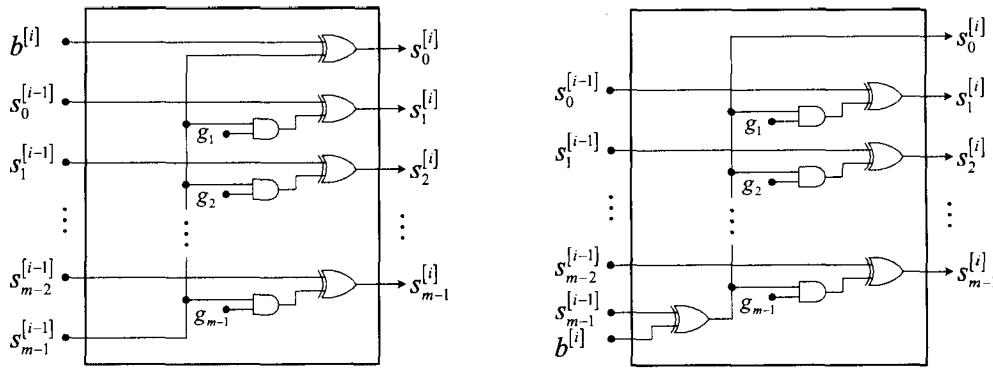


Figure 3.9: Generalized cascade combinational logic blocks for $G(x) = 1 + \sum_{i=1}^{m-1} g_i x^i + x^m$: (a) LFSR1, (b) LFSR2.

respectively. From the figures it is clear that both architectures require $|\Gamma| \times l$ XOR gates, and the computational latency is $\lceil \frac{k}{l} \rceil$ and $\lceil \frac{k+m}{l} \rceil$ clock cycles when using the LFSR2 and LFSR1 combinational logic, respectively. Finally, in [23] a sample implementation for $G(x) = 1 + x^2 + x^3$ with $l = 4$ is illustrated[1].

3.3.4 Look-Ahead Architecture

In [25], a look-ahead approach was applied to the serial LFSR2 architecture to derive the Look-Ahead Architecture. The technique described in this paper is applied to both internal and external LFSR architectures. Since the CRC is defined only for internal LFSRs, we do not discuss the applications to external LFSR architectures.

Formulation

The formulation presented in [25] is unique in the sense that it begins from the message being partitioned in l blocks, where l is the degree of parallelism. Here, we reproduce the derivation, using notations consistent with the ones used in this thesis. Begin by assuming that $k \bmod l = 0$ and partitioning the message into l blocks, i.e.,

$$U(x) = U_0(x) + U_1(x) + \cdots + U_{l-1}(x), \tag{3.20}$$

where $U_i(x) = u_i x^i + u_{l+i} x^{l+i} + \cdots + u_{k-l+i} x^{k-l+i} = \sum_{j=0}^{\frac{k}{l}-1} u_{j \cdot l+i} x^{j \cdot l+i}$. The next step is to manipulate the $U_i(x)$ polynomials; consider the effect of multiplying the message

[1] There is a typo in Figure 3a on page 110 and it is corrected in Appendix C.

polynomial by x^m, and from (3.20) one can obtain,

$$\begin{aligned} x^m \cdot U(x) &= x^m \cdot (U_0(x) + U_1(x) + \cdots + U_{l-1}(x)) \\ &= \sum_{i=0}^{l-1} (x^m \cdot U_i(x)) \\ &= \sum_{i=0}^{l-1} \left(x^m \cdot \sum_{j=0}^{\frac{k}{l}-1} u_{j \cdot l+i} x^{j \cdot l+i} \right) \\ &= \sum_{i=0}^{l-1} \left(x^{m-(l-1)+i} \cdot \sum_{j=0}^{\frac{k}{l}-1} u_{j \cdot l+i} x^{(j+1) \cdot l-1} \right) \\ &= \sum_{i=0}^{l-1} \left(x^{m-(l-1)+i} \cdot \breve{U}_i(x) \right), \end{aligned} \tag{3.21}$$

where $\breve{U}_i(x) = u_i x^{l-1} + u_{l+i} x^{2l-1} + \cdots + u_{k-l+i} x^{k-1} = \sum_{j=0}^{\frac{k}{l}-1} u_{j \cdot l+i} x^{(j+1) \cdot l-1}$. Table 3.3 is a reproduction of Table 1 in [25], showing the relationship between the $U(x)$, $U_i(x)$, and $\breve{U}_i(x)$ polynomials, but using the reverse Endianness notation.

Now substituting (3.21) into the CRC definition given in (2.2), and one obtains,

$$\begin{aligned} S(x) &= \left(\sum_{i=0}^{l-1} x^{m-(l-1)+i} \cdot \breve{U}_i(x) \right) \bmod G(x) \\ &= \left(x^{m-l+1} \cdot \breve{U}_0(x) + x^{m-l+2} \cdot \breve{U}_1(x) + \cdots + x^m \cdot \breve{U}_{l-1}(x) \right) \bmod G(x). \end{aligned} \tag{3.22}$$

The authors of [25] then make the remark that the multiplication of x^{m-l+i} with $\breve{U}_i(x)$ can be accomplished in hardware by shifting the $\breve{U}_i(x)$ into the $(m-l+i)$-th

Table 3.3: Look-Ahead Architecture polynomial relationships for $k = 9$ and $l = 3$.

deg	x^0	x^1	x^2	x^3	x^4	x^5	x^6	x^7	x^8
$U(x)$	u_0	u_1	u_2	u_3	u_4	u_5	u_6	u_7	u_8
$U_0(x)$	u_0	0	0	u_3	0	0	u_6	0	0
$U_1(x)$	0	u_1	0	0	u_4	0	0	u_7	0
$U_2(x)$	0	0	u_2	0	0	u_5	0	0	u_8
$\breve{U}_0(x)$	0	0	u_0	0	0	u_4	0	0	u_7
$\breve{U}_1(x)$	0	0	u_1	0	0	u_5	0	0	u_8
$\breve{U}_2(x)$	0	0	u_2	0	0	u_6	0	0	u_9

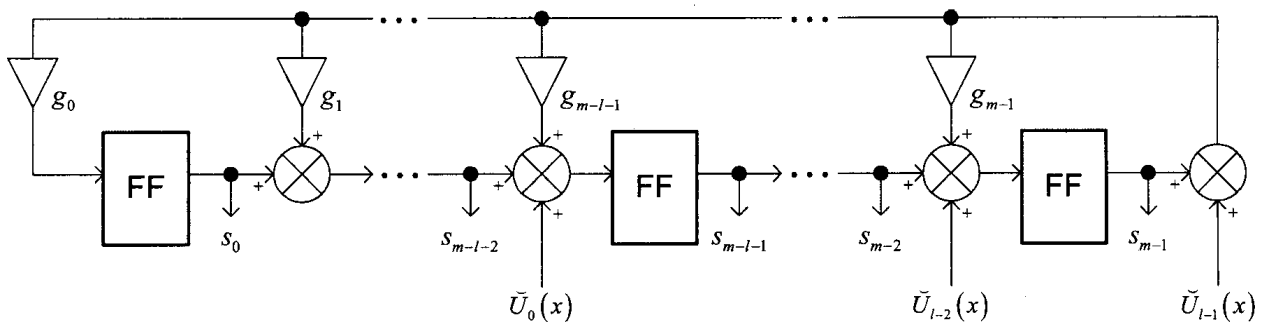


Figure 3.10: Non-optimized parallel CRC computation architecture where $l \leq m$.

right-most position of the LFSR2 architecture [2]. An illustration of a realization of (3.22) is shown in Figure 3.10, where similar to the serial LFSR architectures, the coefficients of the greatest order terms are fed in first, i.e., $\breve{u}_{i,k-1}$, $\breve{u}_{i,k-2}$, $\cdots$, $\breve{u}_{i,0}$ for $\breve{U}_i(x) = \sum_{j=0}^{k-1} \breve{u}_{i,j} x^j$. The authors note that this architecture does not have any speed advantage over the traditional serial LFSR2 architecture, requiring k clock cycles to process a k-bit message, and we note that this approach cannot be easily extended for $l > m$. However, by applying look-ahead techniques it is possible to take advantage of the $l-1$ 0s between message bits in the inputs $\breve{U}_i(x)$.

Next, we examine a simplified state-space model of the architecture in Figure 3.10 where zeros are applied to the inputs. Noting that $l \leq m$ and beginning from the system,

$$\begin{aligned} \mathbf{x}_{m\times 1}[i+1] &= \mathbf{A}_{m\times m} \cdot \mathbf{x}_{m\times 1}[i] + \mathbf{B}_{m\times m} \cdot \breve{\mathbf{u}}_{m\times 1}[i] \\ \mathbf{y}_{m\times 1}[i] &= \mathbf{C}_{m\times m} \cdot \mathbf{x}_{m\times 1}[i] + \mathbf{D}_{m\times m} \cdot \breve{\mathbf{u}}_{m\times 1}[i], \end{aligned} \tag{3.23}$$

for $0 \leq i \leq k-1$, where $\mathbf{A}_{m\times m} = \mathbf{B}_{m\times m} = \begin{bmatrix} 0 & 0 & 0 & & 0 & g_0 \\ 1 & 0 & 0 & & 0 & g_1 \\ 0 & 1 & 0 & & 0 & g_2 \\ \vdots & & & & & \vdots \\ 0 & 0 & 0 & \cdots & 1 & g_{m-1} \end{bmatrix}$, $\mathbf{C}_{m\times m} = \mathbf{I}_{m\times m}$, $\mathbf{D}_{m\times m} = \mathbf{0}_{m\times m}$, and

$$\breve{\mathbf{u}}_{m\times 1} = \begin{bmatrix} \mathbf{0}_{1\times m-l} & \breve{u}_{0,k-1-i} & \breve{u}_{1,k-1-i} & \cdots & \breve{u}_{l-1,k-1-i} \end{bmatrix}^T,$$

which describes the architecture in Figure 3.10. Consider applying zeros to the input $\breve{\mathbf{u}}_{m\times 1}[i]$, then (3.23) reduces to

$$\begin{aligned} \mathbf{x}_{m\times 1}[i+1] &= \mathbf{A}_{m\times m}\cdot\mathbf{x}_{m\times 1}[i] \\ \mathbf{y}_{m\times 1}[i] &= \mathbf{C}_{m\times m}\cdot\mathbf{x}_{m\times 1}[i], \end{aligned}$$

and after $l-1$ clock cycles the content of the state register can be expressed as

$$\mathbf{x}_{m\times 1}[i+l] = (\mathbf{A}_{m\times m})^{l-1}\cdot\mathbf{x}_{m\times 1}[i]. \tag{3.24}$$

Since the $\breve{U}_i(x)$ input polynomials were defined to have one message bit followed by $l-1$ 0s, one can consider the input bit and then apply the look-ahead transformation in (3.24) to account for the $l-1$ 0s during a single iteration without inputting them. Thus the formulation becomes,

$$\begin{aligned} \bar{\mathbf{x}}_{m\times 1}[i+1] &= (\mathbf{A}_{m\times m})^{l-1}\cdot\left(\mathbf{A}_{m\times m}\cdot\bar{\mathbf{x}}_{m\times 1}[i]+\mathbf{B}_{m\times m}\cdot\bar{\breve{\mathbf{u}}}_{m\times 1}[i]\right) \\ \mathbf{y}_{m\times 1}[i] &= \mathbf{C}_{m\times m}\cdot\bar{\mathbf{x}}_{m\times 1}[i]+\mathbf{D}_{m\times m}\cdot\bar{\breve{\mathbf{u}}}_{m\times 1}[i], \end{aligned} \tag{3.25}$$

for $0\le i\le\left\lceil\frac{k-1}{l}\right\rceil$, where the coupling matrices are as defined in (3.23) and the input vector $\bar{\breve{\mathbf{u}}}_{m\times 1}[i]$ is formed from the primitive LFSR2 state space in (3.13) with some extending zeros, i.e.,

$$\bar{\breve{\mathbf{u}}}_{m\times 1}[i] = \begin{bmatrix} \mathbf{0}_{m-l\times 1} \\ \bar{\mathbf{u}}_{l\times 1}[i] \end{bmatrix}.$$

Now, since $\mathbf{A}_{m\times m}=\mathbf{B}_{m\times m}$, one can modify (3.25) and obtain,

$$\begin{aligned} \bar{\mathbf{x}}_{m\times 1}[i+1] &= (\mathbf{A}_{m\times m})^{l}\cdot\mathbf{t}_{m\times 1}[i] \\ \mathbf{y}_{m\times 1}[i] &= \mathbf{C}_{m\times m}\cdot\bar{\mathbf{x}}_{m\times 1}[i]+\mathbf{D}_{m\times m}\cdot\bar{\breve{\mathbf{u}}}_{m\times 1}[i], \end{aligned} \tag{3.26}$$

for $0\le i\le\left\lceil\frac{k-1}{l}\right\rceil$, where $\mathbf{t}_{m\times 1}[i]=\bar{\mathbf{x}}_{m\times 1}[i]+\bar{\breve{\mathbf{u}}}_{m\times 1}[i]$.

Realization

The approach to realize the Look-Ahead Architecture consists of computing the addition $\bar{\mathbf{x}}_{m\times 1}[i]+\bar{\breve{\mathbf{u}}}_{m\times 1}[i]$ and then cascading l copies of the $\mathbf{A}_{m\times m}$ combinational logic depicted in Figure 3.11 to perform the look-ahead operation. Figure 3.12 shows the generalized look-ahead architecture. This architecture is non-primitive, cannot be easily developed for cases when $l>m$, and requires $|\Gamma|\times l$ XOR gates to implement. Finally, we note that a detailed design and trace is provided for $k=8$, $l=2$, and $G(x)=1+x+x^3+x^4$ in Section 4.1 of [25].

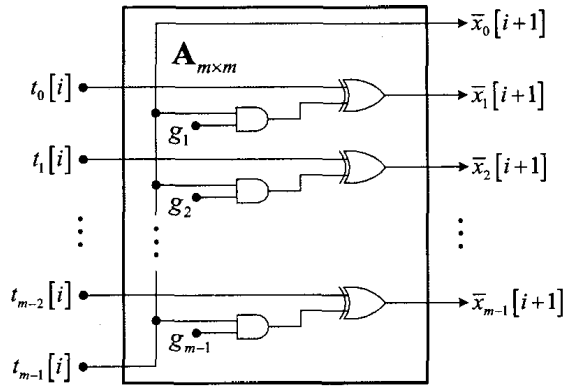


Figure 3.11: Generalized look-ahead combinational logic block for $G(x) = 1 + \sum_{i=1}^{m-1} g_i x^i + x^m$.

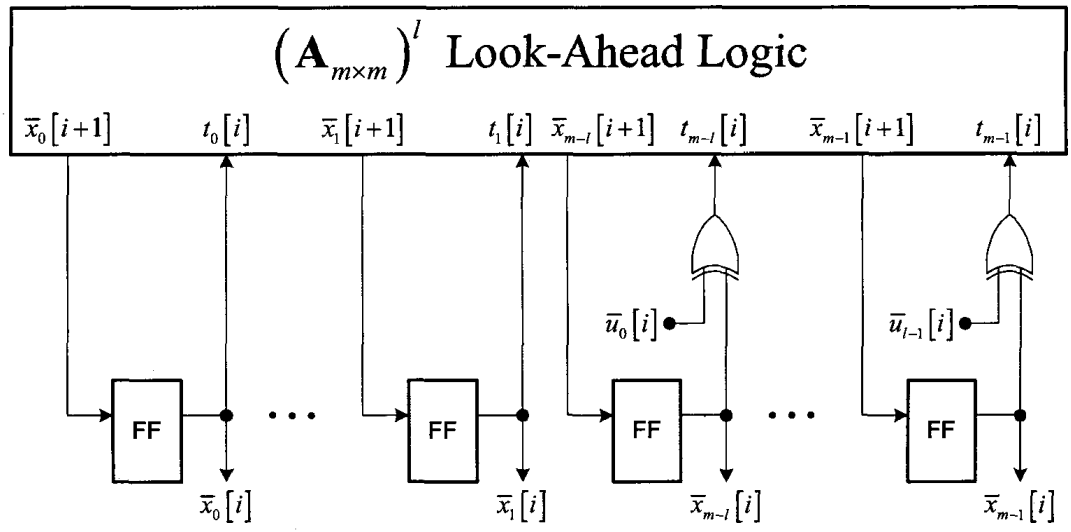


Figure 3.12: Generalized Look-Ahead Architecture.

The characteristic noted by the authors that we feel is most unique to this approach is the ability to develop an architecture that is capable of varying the degree of parallelism. By adopting an architecture shown Figure 3.13, one can realize a flexible parallel CRC computation architecture that can be adapted by simply by-passing look-ahead logic blocks. This design could be suitable for deployment in intellectual property (IP) cores or part of an adaptive system. One point not raised by the authors of [25] is the ability to adjust the generator polynomial by simply supplying control signals to the AND gates in the look-ahead blocks as depicted in Figure 3.11. We note that similar modifications could be made to the cascading approach to obtain an even more flexible design, that does not place the restriction of having the degree of parallelism less than the generator polynomial degree.

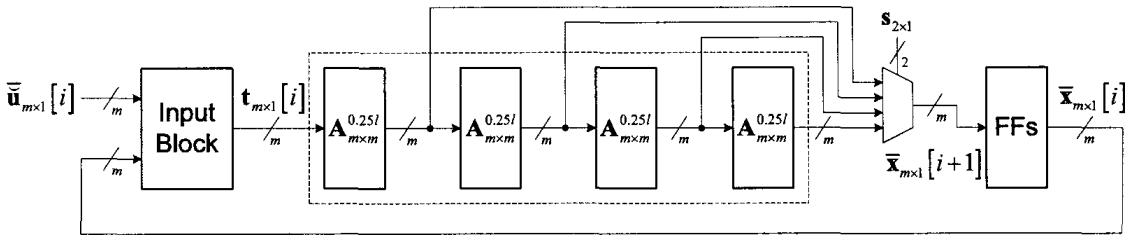


Figure 3.13: Design of the Flexible Look-Ahead Architecture.

3.3.5 State-Space Transformed Architecture

In [24], a state-space similarity transformation method was proposed to obtain the State-Space Transformed Architecture that is easily pipelined. Excluding the Two-Step Architecture, all of the previous approaches discussed thus far cannot be easily pipelined dude to the feedback loop complexity [24]. The state-space transformation proposed in [24] creates a system with a feedback loop complexity equal to that of the serial LFSR2 architecture, and the input and output logic blocks can be pipelined resulting in a parallel architecture with CPD $2 \cdot T_X$.

Formulation

Beginning with the parallel LFSR2 state-space equations (3.13), consider a similarity transformation $\bar{\mathbf{x}}_{m\times 1}[i] = \mathbf{T}_{m\times m} \cdot \bar{\mathbf{x}}'_{m\times 1}[i]$, which takes the state coupling matrix to companion form. Then, the transformed state-space equations can be written as,

$$\begin{aligned} \bar{\mathbf{x}}'_{m\times 1}[i+1] &= \bar{\mathbf{A}}'_{m\times m} \cdot \bar{\mathbf{x}}'_{m\times 1}[i] + \bar{\mathbf{B}}'_{m\times l} \cdot \bar{\mathbf{u}}_{l\times 1}[i] \\ \bar{\mathbf{y}}_{m\times 1}[i] &= \bar{\mathbf{C}}'_{m\times m} \cdot \bar{\mathbf{x}}'_{m\times 1}[i] + \bar{\mathbf{D}}_{m\times l} \cdot \bar{\mathbf{u}}_{l\times 1}[i], \end{aligned} \tag{3.27}$$

for $0 \le i \le \lceil \frac{k-1}{l} \rceil$, where $\bar{\mathbf{A}}'_{m\times m} = \mathbf{T}^{-1}_{m\times m} \cdot \bar{\mathbf{A}}_{m\times m} \cdot \mathbf{T}_{m\times m}$, $\bar{\mathbf{B}}'_{m\times l} = \mathbf{T}^{-1}_{m\times m} \cdot \bar{\mathbf{B}}_{m\times l}$, and $\bar{\mathbf{C}}'_{m\times m} = \bar{\mathbf{C}}_{m\times m} \cdot \mathbf{T}_{m\times m} = \mathbf{T}_{m\times m}$.

In [24], it is shown that the matrix $\mathbf{T}_{m\times m}$ that takes $\bar{\mathbf{A}}_{m\times m}$ to companion form $\bar{\mathbf{A}}'_{m\times m}$, can be found as

$$\mathbf{T}_{m\times m} = \left[\left((\mathbf{A}_{m\times m})^l\right)^0 \cdot \mathbf{b}^*_{m\times 1} \quad \left((\mathbf{A}_{m\times m})^l\right)^1 \cdot \mathbf{b}^*_{m\times 1} \quad \cdots \quad \left((\mathbf{A}_{m\times m})^l\right)^{m-1} \cdot \mathbf{b}^*_{m\times 1} \right] \tag{3.28}$$

and the vector[2] $\mathbf{b}^*_{m\times 1}$ can be selected freely subject to the constraint that the columns of $\mathbf{T}_{m\times m}$ are linearly independent, i.e., $\mathbf{T}_{m\times m}$ is invertible. Furthermore, if the generator polynomial is irreducible, then any $\mathbf{b}^*_{m\times 1}$ can be selected.

[2] In [24], $\mathbf{b}_1$ denotes the $m \times 1$ vector that is used to create the matrix $\mathbf{T}_{m\times m}$.

In [24] an example is presented for CRC-32 with $l = 32$, and for simplicity the author selects $\mathbf{b}^*_{32\times 1} = \begin{bmatrix} 1 & 0 & 0 & \cdots & 0 \end{bmatrix}^T$ to construct the transformation matrix $\mathbf{T}_{32\times 32}$. The author makes the comment in the discussion proposing an open research question that "it may also be possible ... using a different choice for $[\mathbf{b}^*_{m\times 1}]$, to find cases [with lesser hardware requirements]" [24]. The hardware requirements in this case consist of the number of XOR gates, FFs, and pipeline stages (PSs). We chose to investigate this open research question in our conference paper [53] and our search approach is explained next.

We make one final note concerning the formulation of this approach. In [54], the authors extend the formulation in [24] to handle the case when the message length is not a multiple of the degree of parallelism. We consider this to be outside the scope of this thesis, and we will omit it from our discussion.

Vector $\mathbf{b}^*_{m\times 1}$ Search

We perform a brute-force search testing all possible $\mathbf{b}^*_{m\times 1}$ vectors for their result implementation complexity. Since designers take different approaches to implement an architecture (possibly depending on what different hardware elements are available), to consider a $\mathbf{b}^*_{m\times 1}$ to be optimum, we feel that it is best to count the number of 1s in the coupling matrices for the frequently referenced generator polynomials. This may always result in the globally optimum value for all the different hardware libraries, but the end result will likely have a lower hardware complexity than selecting the simple vector $\mathbf{b}^*_{m\times 1} = \begin{bmatrix} 1 & 0 & 0 & \cdots & 0 \end{bmatrix}^T$ vector.

To perform our analysis of the selected generator polynomials with $l = m$, we wrote C++ code that precomputed the state-space matrices $\bar{\mathbf{A}}_{m\times m}$, $\bar{\mathbf{B}}_{m\times l}$ in (3.13) for each generator polynomial. After the precomputation stage, we loop over all the $2^m - 2$ possibilities for the different $\mathbf{b}^*_{m\times 1}$ vectors. For each $\mathbf{b}^*_{m\times 1}$ candidate, the matrix $\mathbf{T}_{m\times m}$ is computed from (3.28) and inverted using Gauss-Jordan elimination, i.e.,

$$\begin{bmatrix} \mathbf{T}_{m\times m} & \mathbf{I}_{m\times m} \end{bmatrix} \sim \begin{bmatrix} \mathbf{I}_{m\times m} & \mathbf{T}^{-1}_{m\times m} \end{bmatrix},$$

to obtain $\mathbf{T}^{-1}_{m\times m}$. Next, we verify that $\mathbf{T}_{m\times m} \cdot \mathbf{T}^{-1}_{m\times m} = \mathbf{I}_{m\times m}$, and afterward the transformed coupling matrices $\bar{\mathbf{A}}'_{m\times m}$, $\bar{\mathbf{B}}'_{m\times l}$, and $\bar{\mathbf{C}}'_{m\times m}$ in (3.27) are obtained. We proceed to count the number of 1s in $\bar{\mathbf{A}}'_{m\times m}$, $\bar{\mathbf{B}}'_{m\times l}$, and $\bar{\mathbf{C}}'_{m\times m}$, and the set of $\mathbf{b}^*_{m\times 1}$ vectors that produced minimum values is retained. One optimization for the computation exploits the fact that if $l = m$, then $\bar{\mathbf{A}}_{m\times m} = \bar{\mathbf{B}}_{m\times(m)}$ [24], thus we are able to first compute $\bar{\mathbf{B}}'_{m\times m} = \mathbf{T}^{-1}_{m\times m} \cdot \bar{\mathbf{B}}_{m\times m}$ and then compute $\bar{\mathbf{A}}'_{m\times m} = \bar{\mathbf{B}}'_{m\times m} \cdot \mathbf{T}_{m\times m}$.

Table 3.4: Optimum $\hat{\mathbf{b}}^*_{m\times 1}$ for frequently referenced generator polynomials when $l = m$.

$G(x)$	$\hat{\mathbf{b}}^*_{m\times 1}$
CRC-12	$[0x814]^T$
CRC-16	$[0xC00D]^T$
CCITT-16	$[0x648B]^T$, * $[0x908C]^T$, $[0xC916]^T$, * $[0xF664]^T$
CRC-16†	$[0x00E0]^T$, * $[0x7401]^T$
CCITT-16†	$[0x390D]^T$, $[0x721A]^T$, * $[0xAC1F]^T$
CRC-32	$[0x0AA4\ 1D98]^T$, * $[0x3C9C\ 8222]^T$

Table 3.5: Comparison of the total number of 1s in the transformed coupling matrices.

$G(x)$	$\mathbf{b}^*_{m\times 1}$	$\hat{\mathbf{b}}^*_{m\times 1}$	% Savings
CRC-12	136	120	11.8
CRC-16	218	188	13.8
CCITT-16	238	226	5.0
CRC-16†	250	190	24.0
CCITT-16†	248	226	8.9
CRC-32	1031	929	9.9

By taking this approach, we save one matrix multiplication operation in our main simulation $\mathbf{b}^*_{m\times 1}$ loop.

The results of our search are summarized using a compact hexadecimal notation in Table 3.4, and the comparison with the simple $\mathbf{b}^*_{m\times 1} = \begin{bmatrix} 1 & \mathbf{0}_{1\times m-1} \end{bmatrix}^T$ vectors is shown in Table 3.5. We note that at this writing, we have completed over 50% of the CRC-32 search, and the final result will appear in our conference paper [53]. As an example of the hexadecimal notation, the optimum vector for CRC-12 with $l = 12$ is found to be

$$\hat{\mathbf{b}}^*_{12\times 1} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \end{bmatrix}^T,$$

we denote it as $\hat{\mathbf{b}}^*_{12\times 1} = [0x814]^T$, and hats are used to mark the optimum vectors.

Realization

A block diagram of the pipelined architecture is shown in Figure 3.14. In our realization, we assume only two-input XOR gates are available and pipelined the input and output blocks such that their output wires were clocked. Any required retiming FFs are placed at the roots of the pipelined XOR trees. Finally, we note a typo in the matrix $\mathbf{T}^{-1}$ listed at the end of [24] and we have corrected it in Appendix C.

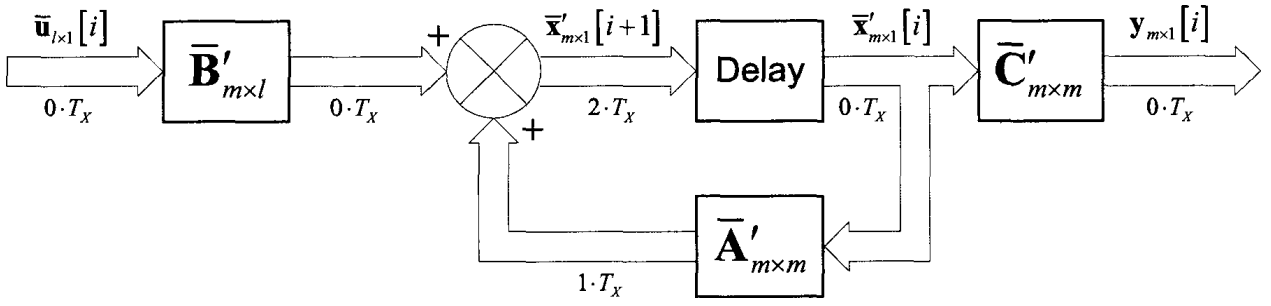


Figure 3.14: Generalized State-Space Transformed Architecture.

3.3.6 Retimed Architectures

In [10], the authors manipulated the serial LFSR2 architecture by unfolding, pipelining, and retiming it. Since no closed form formulation for this approach is presented, we neglected to study it in detail. However, the results achieved by this approach are quite good for CRC-32, but it is difficult to gage its performance with other degrees of parallelism. We note that a possible future study could consist deriving the parallel expressions of this approach based on mathematical manipulations.

Other proposed retimed architectures include [42], [47], and [48]. In [47], a parallel architecture is proposed based on the LFSR1 formulation. The input reduction logic is unfolded and retimed, however the critical path in the feedback loop is unchanged limiting the operational frequency. The other two retimed architectures, i.e., [42] and [48], rely on assumptions that may limit their deployment. For instance, in [42] the message length must be known beforehand and [48] assumes that there are buffered packets to achieve good throughput. For these reasons, we consider all of these retimed architectures to be outside the scope of this thesis and they are omitted from our comparisons.

3.4 Software Algorithms

In this section, the published software algorithms that perform the CRC computation are surveyed. CRC software algorithms are generally slower than their hardware architecture counterparts, however they offer the designer greater flexibility and are useful in systems that employ micro-controllers. The Bit-wise Algorithm, Table Look-up Algorithm, Reduced Table Look-up Algorithm, On-the-Fly Algorithm, Slicing Algorithms, and Distributed Table Look-up Algorithm, as well as the LUT Generation Algorithms are discussed in detail.

In our discussion, we take an approach similar to what used to discuss the hardware architectures in the previous section. Each algorithm is summarized by presenting a generalized formulation and a possible software realization using C++ pseudo code. All of the algorithms are based on the primitive LFSR2 formulation, and are presented using binary polynomial derivations. We begin our discussion by outlining our assumptions in the following subsection. Published papers concerning CRC software algorithms include [7], [13], [15], [16], [55], [56], [57], [58], [59], and [60].

3.4.1 Assumptions

To simplify our discussion of the CRC software algorithms, in this thesis we assume the following:

- w-bit bus width;
- datapath with standard instruction set, e.g.: desktop personal computer (PC);
- all typical bit-wise operations are assumed to take equal run time;
- the leading x^m term of the generator polynomial is considered implicit;
- generator polynomial degree is less than or equal to the bus width, i.e., $m \leq w$;
- degree of parallelism is less than or equal to the bus width, i.e., $l \leq w$;
- word-sized chunks of the message are fetched at a time;
- reverse Endianness convention is employed; and
- $W = \lceil \frac{k}{w} \rceil$ words are required to store a k-bit message in memory.

Since the target deployment for our software experiments consisted of a desktop PC with $w = 32$, most of these assumptions are rather trivial. Before moving on to the discussion of the algorithms, we provide some justification for the assumptions that may not be obvious to the reader.

First, we feel that the most important issue for CRC software algorithms is the Endianness convention of the computation. We have chosen to maintain the reverse convention in all our implementations, because this notation simplifies the implementations of most CRC algorithms, including our later proposed algorithm. Also, it is used in the most recent software CRC publication [7] and this convention is illustrated in Figure 3.15.

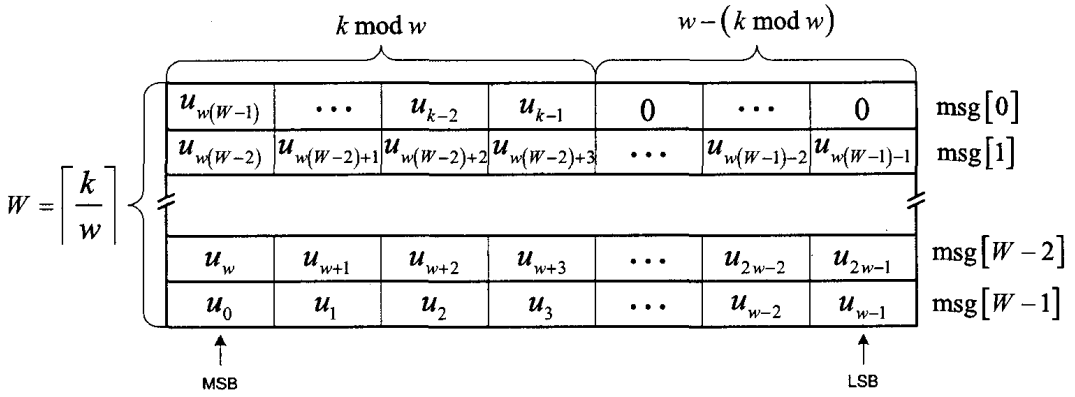


Figure 3.15: Illustration of the message array.

Next, when representing generator polynomials in memory, there are traditionally two approaches. Since generator polynomials always have non-zero x^0 and x^m terms [1], one of those terms can considered implicit and we do not need to store it in memory. The overwhelming majority of papers adopt the approach to make the coefficient of the x^m term implicit, e.g.: [7], [13], [56], and [58], and we also follow that convention. Moreover, the Endianness of that representation is also important, and in Table 3.6, we provide both representations for the common generator polynomials with the leading x^m term implicit.

Again, like a hardware architecture, an iteration of a software algorithm is defined as the operations to process l message bits. We note that since $l \leq w$ was assumed, then it is possible for an implementation of an algorithm to require multiple iterations to process one w-bit message chunk. Finally, since we have assumed $m \leq w$, the final syndrome and all the LUT entries can fit inside a single word. For discussion of implementations when $m > w$, the reader can consult [15].

Table 3.6: Standard hexadecimal representations of frequently referenced generator polynomials.

$G(x)$	Normal*	Reverse*
CRC-12	0x80F	0xF01
CRC-16	0x8005	0xA001
CCITT-16	0x1021	0x8408
CRC-16†	0x4003	0xC002
CCITT-16†	0x0811	0x8810
CRC-32	0x04C1 1DB7	0xEDB8 8320

* x^m term is implicit.

3.4.2 Bit-wise Algorithm

The Bit-wise Algorithm (CRCB) [56] is an emulation of the serial LFSR2 Architecture (Figure 2.3b) in software. The CRCB does not capitalize on the advantage that datapaths can manipulate multiple bits (a word) at a time, and consequently it is quite slow.

Formulation

The formulation for this approach is quite simple, recalling the equation that was used to develop (3.17),

$$S^{[i]}(x) = \left(x \cdot S^{[i-1]}(x) + x^m \cdot B^{[i]}(x)\right) \bmod G(x). \tag{3.29}$$

If one expands the $S^{[i-1]}(x)$ polynomial and replaces $B^{[i]}(x)$ with $b_0^{[i]}$ in (3.29), then

$$\begin{aligned} S^{[i]}(x) &= \left(s_0^{[i-1]}x + s_1^{[i-1]}x^2 + \cdots + s_{m-1}^{[i-1]}x^m + b_0^{[i]}x^m\right) \bmod G(x) \\ &= \left(s_0^{[i-1]}x + s_1^{[i-1]}x^2 + \cdots + s_{m-2}^{[i-1]}x^{m-1} + \right. \\ &\quad \left.\left(s_{m-1}^{[i-1]} + b_0^{[i]}\right)x^m\right) \bmod G(x) \\ &= s_0^{[i-1]}x + s_1^{[i-1]}x^2 + \cdots + s_{m-2}^{[i-1]}x^{m-1} + \\ &\quad \left(s_{m-1}^{[i-1]} + b_0^{[i]}\right)x^m \bmod G(x) \end{aligned} \tag{3.30}$$

is obtained. Since generator polynomials can be written as

$$x^m = g_0 + g_1x + g_2x^2 + \cdots + g_{m-1}x^{m-1},$$

equation (3.30) becomes

$$\begin{aligned} S^{[i]}(x) &= \left(s_0^{[i-1]}x + s_1^{[i-1]}x^2 + \cdots + s_{m-2}^{[i-1]}x^{m-1}\right) + \\ &\quad \left(s_{m-1}^{[i-1]} + b_0^{[i]}\right) \cdot \left(g_0 + g_1x + g_2x^2 + \cdots + g_{m-1}x^{m-1}\right). \end{aligned} \tag{3.31}$$

Realization

The realization of (3.31) is rather straightforward, and in Algorithm 3.1 we show a code snippet for a possible C++ implementation of CRCB(1) assuming $m = w = 32$. Observe that the parentheses are used to denote the number of message bits processed

Algorithm 3.1 Bit-wise Algorithm.

```
crc = INIT_VALUE;
while (p_buf < p_end) {
     msg = *(uint32_t *)p_buf;

     for (i=0; i<32; i++) {
          if ((msg ^ crc) & 0x1) == 0x1)
               crc = (crc >> 1) ^ polynomial;
          else
               crc >>= 1;

          msg >>= 1;
     }

     p_buf += 4;
}
return crc ^ FINAL_VALUE;
```

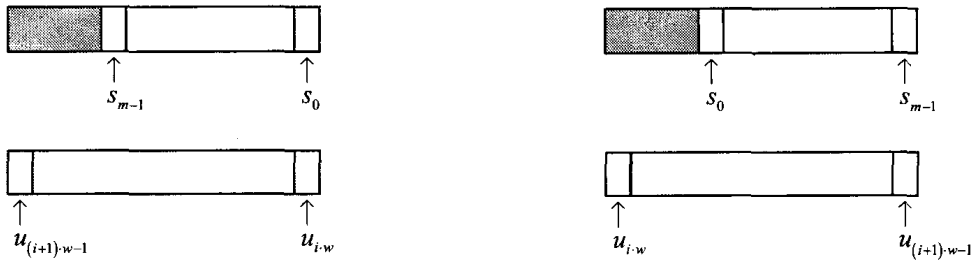


Figure 3.16: Illustration of CRCB Endianness for $m < w$: (a) normal, (b) reverse.

in an iteration for a realization of an algorithm. For further discussion concerning implementations of CRCB, the author may consult [13], [15], and [56]. Finally we note that, if the normal Endianness convention is used to implement this algorithm and $m < w$, then an extra shift is required in the condition in Algorithm 3.1 to form $s_{m-1}^{[i-1]} + b_0^{[i]}$ (as well as all the other shifts being reversed), see Figure 3.16. In fact, all the CRC software algorithms in this thesis suffer this problem for normal Endianness convention when $m < w$.

3.4.3 Table Look-up Algorithm

The Table Look-up Algorithm (CRCT) first suggested in [56], but often credited to [58], is rather easily implemented and useful for software deployments of high-speed CRC computations where memory is readily available. Consequently, this algorithm is popular for desktop PC deployments.

Formulation

The formulation of this approach consists of combining l iterations of CRCB into a single iteration. Beginning with (3.3), assuming $l \leq m$, and expanding the $S^{[i-1]}(x)$ and $B^{[i]}(x)$ polynomials, one can obtain,

$$\begin{aligned}
S^{[i]}(x) &= \left(x^l \cdot S^{[i-1]}(x) + x^m \cdot B^{[i]}(x)\right) \bmod G(x) \\
&= \left(s_0^{[i-1]}x^l + s_1^{[i-1]}x^{l+1} + \cdots + s_{m-1}^{[i-1]}x^{l+m-1} + \right. \\
&\quad \left. b_0^{[i]}x^m + b_1^{[i]}x^{m+1} + \cdots + b_{l-1}^{[i]}x^{m+l-1}\right) \bmod G(x) \\
&= s_0^{[i-1]}x^l + s_1^{[i-1]}x^{l+1} + \cdots + s_{m-l-1}^{[i-1]}x^{m-1} + \left(\left(s_{m-l}^{[i-1]} + b_0^{[i]}\right)x^m + \right. \\
&\quad \left. \left(s_{m-l+1}^{[i-1]} + b_1^{[i]}\right)x^{m+1} + \cdots + \left(s_{m-1}^{[i-1]} + b_{l-1}^{[i]}\right)x^{m+l-1}\right) \bmod G(x) \\
&= s_0^{[i-1]}x^l + s_1^{[i-1]}x^{l+1} + \cdots + s_{m-l-1}^{[i-1]}x^{m-1} + \\
&\quad \left(x^m \cdot H^{[i]}(x)\right) \bmod G(x), \qquad (3.32)
\end{aligned}$$

where $H^{[i]}(x) = \sum_{j=0}^{l-1}\left(s_{m-l+j}^{[i-1]} + b_{l-1}^{[i]}\right)x^j$, and $H^{[i]}(x)$ is used to represent the terms that require reduction in $x^l \cdot S^{[i-1]}(x) + x^m \cdot B^{[i]}(x)$. If one performs the reduction of $x^m \cdot H^{[i]}(x)$ in (3.32) during a single iteration, then the performance of this approach will be better than CRCB.

Realization

An LUT is created with 2^l m-bit entries for the syndromes of each of the possible l-bit patterns of $H^{[i]}(x)$, where

$$\text{lut}\left(H^{[i]}(x)\right) = \left(x^m \cdot H^{[i]}(x)\right) \bmod G(x).$$

The most common degree of parallelism for implementations of this algorithm is $l = 8$. Selecting l such that $w \bmod l \neq 0$ makes little sense, because the number of bits processed in the iterations of a word fetch will not be equal. The next logical value would be $l = 16$, but that results in a LUT with 2^{16} entries, and the designer may not have that amount of memory available. Furthermore, even when there is enough memory available to implement $l = 16$, the access times are likely to be poor mainly due to caching [12], [61].

Algorithm 3.2 Table Look-up Algorithm.

```
crc = INIT_VALUE;
while (p_buf < p_end) {
     msg = *(uint32_t *)p_buf;

     for (i=0; i<4; i++) {
          term = (crc ^ msg) & 0xFF;
          crc = lut[term] ^ (crc >> 8);

          msg >>= 8;
     }

     p_buf += 4;
}
return crc ^ FINAL_VALUE;
```

In Algorithm 3.2, we show a code snippet for a possible C++ implementation of CRCT(8) assuming $m = w = 32$. For the cases when $l \geq m$, the development of the formulation in (3.32) is slightly different with

$$H^{[i]}(x) = \sum_{j=0}^{l-m-1} b_j^{[i]} x^j + \sum_{j=0}^{m-1} \left(s_j^{[i-1]} + b_{j+l-m}^{[i]} \right) x^j.$$

In our discussion of the next algorithm, we develop this formulation in detail. Finally, additional implementations and further discussion of this algorithm can be found in [7], [13], and [15].

3.4.4 Reduced Table Look-up Algorithm

The Reduced Table Look-up Algorithm (CRCR) first suggested in [13], also known as the Virtual Table Algorithm [15], is derived from the linearity property of the modulus operation. Typical implementations slightly outperform CRCB, at the cost of a small amount of additional memory.

Formulation

The approach of this algorithm is based on decomposition of the polynomial $H^{[i]}(x)$ and the distribution of the modulus operation. Beginning from (3.3), assuming $l \geq m$, and expanding the $S^{[i-1]}(x)$ and $B^{[i]}(x)$ polynomials, one can obtain,

$$
\begin{aligned}
S^{[i]}(x) &= \left(x^l \cdot S^{[i-1]}(x) + x^m \cdot B^{[i]}(x)\right) \bmod G(x) \\
&= \left(s_0^{[i-1]}x^l + s_1^{[i-1]}x^{l+1} + \cdots + s_{m-1}^{[i-1]}x^{l+m-1} + \right. \\
&\quad \left. b_0^{[i]}x^m + b_1^{[i]}x^{m+1} + \cdots + b_{l-1}^{[i]}x^{m+l-1}\right) \bmod G(x) \\
&= \left(b_0^{[i]}x^m + b_1^{[i]}x^{m+1} + \cdots + b_{l-m-1}^{[i]}x^{l-1} + \left(s_0^{[i-1]} + b_{l-m}^{[i]}\right)x^l + \right. \\
&\quad \left. \left(s_1^{[i-1]} + b_{l-m+1}^{[i]}\right)x^{l+1} + \cdots + \left(s_{m-1}^{[i-1]} + b_{l-1}^{[i]}\right)x^{m+l-1}\right) \bmod G(x) \\
&= \left(x^m \cdot H^{[i]}(x)\right) \bmod G(x), \qquad (3.33)
\end{aligned}
$$

where $H^{[i]}(x) = \sum_{j=0}^{m-l-1} s_j^{[i-1]}x^j + \sum_{j=m-l}^{m-1}\left(s_j^{[i-1]} + b_{j-m+l}^{[i]}\right)x^j$. Then, the polynomial $x^m \cdot H^{[i]}(x)$ in (3.33) is decomposed and the mod operation is distributed, i.e.,

$$
\begin{aligned}
S^{[i]}(x) &= \left(x^m \cdot H^{[i]}(x)\right) \bmod G(x) \\
&= \left(x^m \cdot \sum_{j=0}^{l-1} h_j^{[i]}x^j\right) \bmod G(x) \\
&= h_0^{[i]}x^m \bmod G(x) + h_1^{[i]}x^{m+1} \bmod G(x) + \cdots + \\
&\quad h_{l-1}^{[i]}x^{m+l-1} \bmod G(x). \qquad (3.34)
\end{aligned}
$$

Realization

To implement (3.34) in software, one can create a reduced LUT with l m-bit entries, where

$$\text{lut}\left(x^j\right) = x^{m+j} \bmod G(x).$$

Then, during each iteration, the CRCT LUT entry is constructed from the required look-ups (depending on values of $h_j^{[i]}$ for $0 \leq j \leq l-1$) to the reduced LUT table. This approach serves as a low-memory alternative to CRCT. Selecting the degree of parallelism equal to bus width is generally the best choice for the fastest implementations of this algorithm [13]. In Algorithm 3.3, we show a code snippet for a possible C++ implementation of CRCR(32) assuming $m = w = 32$.

3.4.5 On-the-Fly Algorithm

The On-the-Fly Algorithm (CRCF) [13], [56], also known as the Optimized Virtual Table Look-up Algorithm [15], relies on implementing the primitive LFSR2 equations in software with datapath operations. Unlike the previously described software CRC

Algorithm 3.3 Reduced Table Look-up Algorithm.

```
crc = INIT_VALUE;
while (p_buf < p_end) {
     term = crc ^ *(uint32_t *)p_buf;

     crc = 0x0;
     for (i=0; i<32; i++) {
          if ((term & 0x1) == 0x1)
               crc ^= lut[i];

          term >>= 1;
     }

     p_buf += 4;
}
return crc ^ FINAL_VALUE;
```

computation algorithms CRCB, CRCT, and CRCR, the performance of CRCF is dependent on the degree of parallelism, coefficients of the generator polynomial, and creativity of the implementer.

Formulation

As we have demonstrated though out this chapter, there are many different ways to obtain the primitive parallel LFSR2 equations. In typical explanations of CRCF, the equations are obtained from the entries of the CRCR LUT, and both [13] and [56] present a table outlining those equations for CRC-16 with $l = 8$. We have produced a similar table in Appendix B, that outlines the equations for CRC-32 with $l = 8$. For designers not experienced with hardware, this is probably the best method to obtain the CRCF expressions.

Realization

The implementations of CRCF(8) for CRC-16 that are described in [13] and [56] make use of special memory flags such the register parity. In many cases, these flags may not be known of or available to the designer. Referring to the expressions for CRCF(8) with CRC-32 in Appendix B, one can see that it would be difficult to derive an implementation that outperforms CRCR(32) on any datapath. Also, we feel that this algorithm is best suited for smaller datapaths such as $w = 8$ or $w = 16$. For these reasons, we omit this algorithm from our comparisons.

3.4.6 Tea-Leaf Reader Algorithm

The Tea-Leaf Reader Algorithm was one of the earliest CRC algorithms proposed by the authors of [57]. We note that the presentation style and notations contained in the paper are quite cumbersome and difficult to understand. This algorithm was subsequently shown to be less efficient in terms of the number of number of instructions and LUT sizes when compared to CRCT in [58]. Furthermore, in [62] the authors discuss the differences between internal and external LFSRs, and they state the CRC computation is implemented with an internal LFSR architecture, whereas the Tea-Leaf reader algorithm is based on an external LFSR architecture. It is noted that there exists a unique one-to-one mapping between the signatures created from an input sequence inputted to each architecture, but the result produced from the external LFSR architecture is not the CRC computation as defined in (2.2). Therefore, the Tea-Leaf Reader Algorithm does not perform the CRC computation. For these reasons, we omit this algorithm from our comparisons, and for further discussion on internal and external LFSRs and the signatures generated by them, the reader may consult [63] and [64].

3.4.7 Joshi-Dubey-Kaplan Algorithm

In [16], a parallel CRC algorithm was proposed, which we refer to as the Joshi-Dubey-Kaplan Algorithm (CRCJDK). It was designed to take advantage of the instruction set extensions present in the IBM PowerPC 128-bit architecture. It is difficult to compare implementations of CRCJDK [7], because similar to CRCF its performance relies on specific datapath instructions that are not always available to the designer. We attempted to implement it on our $w = 32$ system and derive a formulation that modeled the operation of this algorithm, but we were unsuccessful. For these reasons, we omit this algorithm from our software comparisons.

At the end of [16], a parallel CRC architecture is proposed by mapping the CRCJDK to hardware. However, the explanation of its implementation is not clear, and we failed at attempts to implement it. We have found no other papers in the literature that provide comparisons of the theoretical or implementation performance of this architecture, and for these reasons we omit it from our hardware comparisons.

3.4.8 Slicing Algorithms

The Slicing Algorithms are the fastest known software CRC computation algorithms. The derivation of these algorithms presented in [7] is quite complex involving many theorems and lemmas. In this subsection, we derive the slicing formulation though simple binary polynomial manipulations, and show this approach is really a combination of the CRCR and CRCT algorithms. Two variants of approach are presented in [7] called the CRC Slicing-by-4 Algorithm (CRCS4) and the CRC Slicing-by-8 Algorithm (CRCS8). They are suitable to process one word and two words portions of the message stream during an iteration for CRCS4 and CRCS8, respectively. Our formulation for this approach focuses on CRCS4, but it is easily extended to the CRCS8 case.

Formulation

The formulation of this approach can be considered as an extension of the CRCR algorithm, and it is typically developed for situations when $l \geq m$. For CRCS4, continuing from (3.33) and assuming that $l \bmod 4 = 0$, slicing $H^{[i]}(x)$ into four polynomials, then distributing the modulus operation, and one obtains

$$\begin{aligned} S^{[i]}(x) &= \left(x^m \cdot H^{[i]}(x)\right) \bmod G(x) \\ &= \Big(x^m \cdot \Big(H_0^{[i]}(x) + x^{\frac{l}{4}} \cdot H_1^{[i]}(x) + \\ &\quad x^{\frac{l}{2}} \cdot H_2^{[i]}(x) + x^{\frac{3l}{4}} \cdot H_3^{[i]}(x)\Big)\Big) \bmod G(x) \\ &= \left(x^m \cdot H_0^{[i]}(x)\right) \bmod G(x) + \left(x^{m+\frac{1}{4}l} \cdot H_1^{[i]}(x)\right) \bmod G(x) + \\ &\quad \left(x^{m+\frac{l}{2}} \cdot H_2^{[i]}(x)\right) \bmod G(x) + \left(x^{m+\frac{3l}{4}} \cdot H_3^{[i]}(x)\right) \bmod G(x), \end{aligned} \tag{3.35}$$

where $H_0^{[i]}(x) = \sum_{j=0}^{\frac{l}{4}-1} h_j^{[i]} x^j$, $H_1^{[i]}(x) = \sum_{j=0}^{\frac{l}{4}-1} h_{\frac{l}{4}+j}^{[i]} x^j$, $H_2^{[i]}(x) = \sum_{j=0}^{\frac{l}{4}-1} h_{\frac{l}{2}+j}^{[i]} x^j$, $H_3^{[i]}(x) = \sum_{j=0}^{\frac{l}{4}-1} h_{\frac{3l}{4}+j}^{[i]} x^j$, and assuming that $l \bmod 4 = 0$.

Observing (3.35) and one notices that a similar approach was taken with CRCR, but in this case the reduction of $H^{[i]}(x)$ has been split in such a way that four polynomials of $\frac{l}{4}$ bits require reduction. Also, note that each polynomial requires reduction at a different offset. For CRCS8, one splits $H^{[i]}(x)$ into eight polynomials assuming that $l \bmod 8 = 0$.

Algorithm 3.4 Slicing-by-4 Algorithm.

```
crc = INIT_VALUE;
while (p_buf < p_end) {
     term = crc ^ *(uint32_t *)p_buf;

     crc = lut_56[term & 0xFF] ^
           lut_48[(term >> 8) & 0xFF] ^
           lut_40[(term >> 16) & 0xFF] ^
           lut_32[term >> 24];

     p_buf += 4;
}
return crc ^ FINAL_VALUE;
```

Realization

The realization of this approach in software is rather straightforward. For CRCS4, four LUTs are created for the different offsets x^m, $x^{m+\frac{l}{4}}$, $x^{m+\frac{l}{2}}$, and $x^{m+\frac{3l}{4}}$. With $w = 32$, $l = 32$ is generally the best choice with size of the LUTs exploding for greater values of l, and awkward alignments and poor performance for smaller values of l. In Algorithm 3.4, we show a code snippet for a possible C++ implementation of CRCS4(32) assuming $m = w = 32$, and this is similar to what is presented in [7]. This approach outperforms CRCT because it avoids alignment shifts by processing an entire word in an iteration and the caching problem by requiring four LUTs, each with $2^{\frac{l}{4}}$ m-bit entries.

3.4.9 Distributed Table Look-up Algorithm

Recently, the Distributed Table Look-up Algorithm (CRCD) was proposed in [60]. Basically, this approach adapts CRCS4 to perform the CRC computation with distributed accumulators, i.e., one on each processor. The message is then partitioned, and portions are processed in parallel before the final result is obtained from combining all the partial results. Since we did not have access to a distributed environment, we opted not to implement this algorithm.

However, in the future work section of that paper, the authors suggest an investigation into a hardware realization of their approach. In the following chapter, we investigate this idea and show that it is likely that cases exist where this approach can offer improvement over the existing hardware approaches. However, simulations need to be performed to verify that these cases exist and that they are for useful situations.

3.4.10 Look-up Table Generation

In this subsection, the three LUT generation algorithms for the LUT based CRC computation algorithms discussed in this chapter are presented. We note that LUT entries are typically computed offline, so efficiency is not a primary concern. Therefore, we feel that designers should aim to implement algorithms that are more easily maintained, so their correctness that can be verified. We remind the reader of our assumption that $m \leq w$, i.e., all LUT entries can be stored in a single word. Notice that we drop the iteration superscript from the $H(x)$ polynomials, because the iteration has no impact when we are considering the creation of an LUT.

CRCT LUT Generation

As previously discussed, the m-bit LUT entries for CRCT were defined as

$$\text{lut}(H(x)) = (x^m \cdot H(x)) \bmod G(x),$$

where $H(x) = \sum_{j=0}^{l-1} h_j x^j$. To generate the entries for the LUT, the bit-wise CRC algorithm can be used. One can loop over the 2^l different bit patterns of $H(x)$, compute the CRC with CRCB, and store the result in a table at the index described by the bit pattern of $H(x)$. In Algorithm 3.5, we show a code snippet for a possible C++ implementation of the CRCT LUT generation algorithm.

Algorithm 3.5 Table Look-up LUT Generation Algorithm.

```
lut_size = pow(2, dop);

for (i=0; i<lut_size; i++) {
     msg = i;
     lfsr = 0;

     for (j=0; j<dop; j++) {
          if (((msg ^ lfsr) & 0x1) == 0x1)
               lfsr = (lfsr >> 1) ^ polynomial;
          else
               lfsr >>= 1;

          msg >>= 1;
     }

     lut[i] = lfsr;
}
```

Algorithm 3.6 Reduced Table Look-up LUT Generation Algorithm.

```
lut_size = dop;

for (i=0; i<lut_size; i++) {
     lfsr = polynomial;

     for (j=0; j<i; j++) {
          if ((lfsr & 0x1) == 0x1)
               lfsr = (lfsr >> 1) ^ polynomial;
          else
               lfsr >>= 1;
     }

     lut[i] = lfsr;
}
```

CRCR LUT Generation

Recalling, that the m-bit LUT entries for CRCR were defined as

$$\text{lut}\left(x^j\right) = \left(x^m \cdot x^j\right) \bmod G\left(x\right),$$

for $0 \leq j \leq l-1$. The cleaver reader may notice that to compute the CRC of a message of the form $U\left(x\right) = x^j$ involves shifting one 1 followed by $j-1$ 0s into the Serial LFSR2 Architecture. Again, using CRCB one can easily generate this LUT, and in Algorithm 3.6, we show a code snippet for a possible C++ implementation of the CRCR LUT Generation Algorithm. One final point worth noting, the CRCT LUT can be generated from the CRCR LUT entries by means of the CRCR Algorithm, and this is discussed by the authors of [13].

CRCS LUT Generation

Recalling, that the LUT entries for the slicing algorithms depend on the offset and the degree of parallelism, we define the generalized CRCS LUT as

$$\text{lut}_o\left(H\left(x\right)\right) = \left(x^m \cdot x^o \cdot H\left(x\right)\right) \bmod G\left(x\right),$$

which is slightly different than the notation used in [7]. This is easily implemented as combination of the two previous approaches, using CRCB to reduce $H\left(x\right)$ and shifting in an additional o 0s. In other words, combining the $x^o \cdot H\left(x\right)$ together, and

Algorithm 3.7 Slicing LUT Generation Algorithm.

```
lut_size = pow(2, dop);

for (i=0; i<lut_size; i++) {
     msg = i;
     lfsr = 0;

     for (j=0; j<(dop+offset); j++) {
          if (((msg ^ lfsr) & 0x1) == 0x1)
               lfsr = (lfsr >> 1) ^ polynomial;
          else
               lfsr >>= 1;

          msg >>= 1;
     }

     lut[i] = lfsr;
}
```

one obtains,

$$\begin{aligned} \mathrm{lut}_o\left(H\left(x\right)\right) &= \left(x^m \cdot x^o \cdot H\left(x\right)\right) \bmod G\left(x\right) \\ &= \left(x^m \cdot H'\left(x\right)\right) \dot{\bmod} G\left(x\right), \end{aligned}$$

where $H'\left(x\right) = x^o \cdot H\left(x\right)$. In Algorithm 3.7, we show a code snippet for a possible C++ implementation of the CRCS LUTs generation algorithm.

3.5 Summary

In this chapter we reviewed the CRC formulations, architectures, and algorithms most relevant to our work. We identified three methods to obtain the primitive parallel CRC equations: binary polynomial, state-space, and z-transform. Parallel hardware architectures are based on either LFSR1 or LFSR2 formulations, while software algorithms are based solely on LFSR2 formulations.

The LFSR Architectures that perform the CRC computation are direct hardware realizations of their mathematical formulations, and are the fastest known non-pipelined architectures. The Two-Step Architecture can result in parallel implementation of the CRC computation with CPD T_X, however it is difficult to find $M\left(x\right)$ polynomials for degrees of parallelism greater than $l = 8$. The approach used to derive the Cascade Architecture places no restrictions on the degree of parallelism and

generator polynomial degree, and is useful for designers with limited knowledge who wish to implement a parallel CRC architecture. The Look-Ahead Architecture is yet another systematic method to realize a parallel CRC computation in hardware. The State-Space Transformed Architecture, can be used to derive a realization that can be pipelined with CPD $2 \cdot T_X$. We performed a brute-force search to obtain the set of optimum vectors used to construct the transformation matrices for frequently referenced generator polynomials. Unfolding, pipelining, and retiming techniques were applied to the serial LFSR2 Architecture to obtain high-speed parallel CRC computation circuits.

The Bit-wise Algorithm is software emulation of the LFSR2 Architecture. The Table Look-up Algorithm uses LUTs to perform the computation at a fast rate and requires 2^l m-bit LUT entries. The Reduced Table Look-up Algorithm is slightly faster than the Bit-wise Algorithm requiring l m-bit LUT entries. The On-the-Fly Algorithm relies on the designer being able to implement the parallel LFSR2 equations using bit-wise operations. Unlike the previously mentioned algorithms, the performance depends on the coefficients of the generator polynomial. Although the Tea-Leaf Reader Algorithm is frequently cited in software CRC papers, it does not actually perform a proper CRC computation. The Joshi-Dubey-Kaplan Algorithm was designed to operate on a 128-bit Power PC architecture, and relied on instructions specific to that architecture. The Slicing Algorithms use multiple LUTs to perform the CRC computation, and they are the fastest known software algorithms. The Distributed Look-up Algorithm extends the slicing approach to operate across multiple processors. All of the LUTs for the previously mentioned software algorithms can be computed using a variant of the Bit-wise Algorithm.

Chapter 4

Novel Computation Approaches

4.1 Preview

THE previous chapter studied the existing CRC formulations. These formulations can be used to derive subsequent hardware architectures and software algorithms. We have seen that many approaches exist, all having different implementation trade-offs. Now, with the existing work sufficiently described and analyzed, we are poised to develop our novel computation approaches.

In this chapter, we present a few novel extensions to the CRC formulation. These contributions give rise to new computation approaches that are realized as architectures and algorithms. We show that some of these approaches improve upon the existing ones, while the others are explored and could be the focus of future work. In the following chapter, we present the performance comparison of our proposed approaches versus the exiting ones.

4.1.1 Organization

The remainder of this chapter is organized as follows. In Section 4.2, we present the binary polynomial to matrix derivation and our optimized parallel LFSR2 Architecture for the case when the degree of parallelism is greater than the generator polynomial degree. In Section 4.3, the derivation, algorithm, and architecture of Lambda Gamma approach are presented. In Section 4.4, we extend the existing binary polynomial approaches discussed in the previous chapter and propose a novel architecture. In Section 4.5, we derive and present the Message Splitting Architecture which is based on the Distributed Table Look-up Algorithm and show that it may outperform the existing approaches. This chapter is concluded with a summary in Section 4.6.

4.2 Binary Polynomial to Matrix Approach

In this section, we construct a matrix-based representation of the parallel CRC LFSR2 formulation. Unlike the previous matrix-based approaches that are obtained through state-space manipulations, our approach begins from the binary polynomial description of the CRC computation presented in the previous chapter. After completing the derivation, we propose an architecture specific to the case when the degree of parallelism is greater than the degree of the generator polynomial. The majority of this material appears in our conference paper [41], with some notational changes to bring the formulation in-line with the material in the previous chapter.

4.2.1 Formulation

Assuming that degree of parallelism is greater than the generator polynomial degree, i.e, $l > m$, and continuing from the LFSR2 derivation presented in (3.7), one can obtain

$$\begin{aligned} T^{[i]}(x) &= x^m \cdot \left(\sum_{j=0}^{l-m-1} b_j^{[i]} x^j + \sum_{j=0}^{m-1} \left(s_j^{[i-1]} + b_{j+l-m}^{[i]} \right) x^{j+l-m} \right) \\ &= \sum_{j=0}^{l-m-1} b_j^{[i]} x^{j+m} + \sum_{j=0}^{m-1} \left(s_j^{[i-1]} + b_{j+l-m}^{[i]} \right) x^{j+l} \\ &= \sum_{j=0}^{l+m-1} t_j^{[i]} x^j \end{aligned} \tag{4.1}$$

where

$$t_j^{[i]} = \begin{cases} 0 & 0 \le j < m \\ b_{j-m}^{[i]} & m \le j < l \\ s_{j-l}^{[i]} + b_{j-m}^{[i]} & l \le j < l+m \end{cases} . \tag{4.2}$$

Now, we introduce a matrix multiplication for representing the non-zero portion of scalar $T^{[i]}(x)$ in (4.1), as the product of a row vector $\mathbf{x}_{1\times l} = \begin{bmatrix} 1 & x & \cdots & x^{l-1} \end{bmatrix}$ and the column vector $\mathbf{t}_{l\times 1}^{[i]} = \begin{bmatrix} t_m & t_{m+1} & \cdots & t_{l+m-1} \end{bmatrix}^T$, i.e.,

$$\begin{aligned} \sum_{j=m}^{l+m-1} t_j^{[i]} x^j &= x^m \cdot \sum_{j=m}^{l+m-1} t_j^{[i]} x^{j-m} \\ &= x^m \cdot \left(\mathbf{x}_{1\times l} \cdot \mathbf{t}_{l\times 1}^{[i]} \right). \end{aligned}$$

From (4.2), it is clear that $T^{[i]}(x) = \sum_{j=m}^{l+m-1} t_j x^j$, consequently,

$$T^{[i]}(x) = x^m \cdot \mathbf{x}_{1\times l} \cdot \mathbf{t}^{[i]}_{l\times 1}. \tag{4.3}$$

From the recursive definition of the CRC given in (3.3), and using (4.3), one can derive a generalized matrix-based formulation for the CRC computation as

$$\begin{aligned}
S^{[i]}(x) &= \left(x^m \cdot \mathbf{x}_{1\times l} \cdot \mathbf{t}^{[i]}_{l\times 1}\right) \bmod G(x) \\
&= \begin{bmatrix} x^m & x^{m+1} & \cdots & x^{l+m-1} \end{bmatrix} \cdot \mathbf{t}^{[i]}_{l\times 1} \bmod G(x) \\
&= \begin{bmatrix} x^m \bmod G(x) \\ x^{m+1} \bmod G(x) \\ \vdots \\ x^{m+l-1} \bmod G(x) \end{bmatrix}^T \cdot \mathbf{t}^{[i]}_{l\times 1} \\
&= \begin{bmatrix} g_{0,0} + g_{1,0}x + \cdots + g_{m-1,0}x^{m-1} \\ g_{0,1} + g_{1,1}x + \cdots + g_{m-1,1}x^{m-1} \\ \vdots \\ g_{0,l-1} + g_{1,l-1}x + \cdots + g_{m-1,l-1}x^{m-1} \end{bmatrix}^T \cdot \mathbf{t}^{[i]}_{l\times 1} \\
&= \mathbf{x}_{1\times m} \cdot \begin{bmatrix} g_{0,0} & g_{0,1} & \cdots & g_{0,l-1} \\ g_{1,0} & g_{1,1} & \cdots & g_{1,l-1} \\ \vdots & \vdots & \ddots & \vdots \\ g_{m-1,0} & g_{m-1,1} & \cdots & g_{m-1,l-1} \end{bmatrix} \cdot \mathbf{t}^{[i]}_{l\times 1} \\
&= \mathbf{x}_{1\times m} \cdot \mathbf{G}_{m\times l} \cdot \mathbf{t}^{[i]}_{l\times 1},
\end{aligned} \tag{4.4}$$

for $0 \le i \le q-1$, where $\mathbf{x}_{1\times m} = \begin{bmatrix} 1 & x & \cdots & x^{m-1} \end{bmatrix}$ and

$$\mathbf{G}_{m\times l} = \begin{bmatrix} g_{0,0} & g_{0,1} & \cdots & g_{0,l-1} \\ g_{1,0} & g_{1,1} & \cdots & g_{1,l-1} \\ \vdots & \vdots & \ddots & \vdots \\ g_{m-1,0} & g_{m-1,1} & \cdots & g_{m-1,l-1} \end{bmatrix}.$$

Similar to the method used to express $T^{[i]}(x)$ as the product of a row and column vector in (4.3), one can write the left-hand side of (4.4) as

$$S^{[i]}(x) = \mathbf{x}_{1\times m} \cdot \mathbf{s}^{[i]}_{m\times 1}. \tag{4.5}$$

$$\begin{bmatrix}
1&0&0&0&0&0&1&0&0&1&1&0&1&0&0&0&1&0&0&0&0&0&0&0&1&1&1&0&1&1&1&1\\
1&1&0&0&0&0&1&1&0&1&0&1&1&1&0&0&1&1&0&0&0&0&0&0&1&0&0&1&1&0&0&0\\
1&1&1&0&0&0&1&1&1&1&0&0&0&1&1&0&1&1&1&0&0&0&0&0&1&0&1&0&0&0&1&1\\
0&1&1&1&0&0&0&1&1&1&1&0&0&0&1&1&0&1&1&1&0&0&0&0&0&1&0&1&0&0&0&1\\
1&0&1&1&1&0&1&0&1&0&0&1&1&0&0&1&0&0&1&1&1&0&0&0&1&1&0&0&0&1&1&1\\
1&1&0&1&1&1&1&1&0&0&1&0&0&1&0&0&0&0&0&1&1&1&0&0&1&0&0&0&1&1&0&0\\
0&1&1&0&1&1&1&1&1&0&0&1&0&0&1&0&0&0&0&0&1&1&1&0&0&1&0&0&0&1&1&0\\
1&0&1&1&0&1&0&1&1&0&1&0&0&0&0&1&1&0&0&0&0&1&1&1&1&1&0&0&1&1&0&0\\
1&1&0&1&1&0&0&0&1&0&1&1&1&0&0&0&0&1&0&0&0&0&1&1&0&0&0&0&1&0&0&1\\
0&1&1&0&1&1&0&0&0&1&0&1&1&1&0&0&0&0&1&0&0&0&0&1&1&0&0&0&0&1&0&0\\
1&0&1&1&0&1&0&0&0&1&0&0&0&1&1&0&1&0&0&1&0&0&0&0&0&0&1&0&1&1&0&1\\
1&1&0&1&1&0&0&0&0&1&0&0&1&0&1&1&1&1&0&0&1&0&0&0&1&1&1&1&1&0&0&1\\
1&1&1&0&1&1&1&0&0&1&0&0&1&1&0&1&0&1&1&0&0&1&0&0&1&0&0&1&0&0&1&1\\
0&1&1&1&0&1&1&1&0&0&1&0&0&1&1&0&1&0&1&1&0&0&1&0&0&1&0&0&1&0&0&1\\
0&0&1&1&1&0&1&1&1&0&0&1&0&0&1&1&0&1&0&1&1&0&0&1&0&0&1&0&0&1&0&0\\
0&0&0&1&1&1&0&1&1&1&0&0&1&0&0&1&1&0&1&0&1&1&0&0&1&0&0&1&0&0&1&0\\
1&0&0&0&1&1&0&0&1&0&0&0&1&1&0&0&0&1&0&1&0&1&1&0&1&0&1&0&0&1&1&0\\
0&1&0&0&0&1&1&0&0&1&0&0&0&1&1&0&0&0&1&0&1&0&1&1&0&1&0&1&0&0&1&1\\
0&0&1&0&0&0&1&1&0&0&1&0&0&0&1&1&0&0&0&1&0&1&0&1&1&0&1&0&1&0&0&1\\
0&0&0&1&0&0&0&1&1&0&0&1&0&0&0&1&1&0&0&0&1&0&1&0&1&1&0&1&0&1&0&0\\
0&0&0&0&1&0&0&0&1&1&0&0&1&0&0&0&1&1&0&0&0&1&0&1&0&1&1&0&1&0&1&0\\
0&0&0&0&0&1&0&0&0&1&1&0&0&1&0&0&0&1&1&0&0&0&1&0&1&0&1&1&0&1&0&1\\
1&0&0&0&0&0&0&0&0&1&0&1&1&0&1&0&1&0&1&1&0&0&0&1&1&0&1&1&0&1&0&1\\
1&1&0&0&0&0&1&0&0&1&0&0&0&1&0&1&1&1&0&1&1&0&0&0&0&0&1&1&0&1&0&1\\
0&1&1&0&0&0&0&1&0&0&1&0&0&0&1&0&1&1&1&0&1&1&0&0&0&0&0&1&1&0&1&0\\
0&0&1&1&0&0&0&0&1&0&0&1&0&0&0&1&0&1&1&1&0&1&1&0&0&0&0&0&1&1&0&1\\
1&0&0&1&1&0&1&0&0&0&1&0&0&0&0&0&0&0&1&1&1&0&1&1&1&1&1&0&1&0&0&1\\
0&1&0&0&1&1&0&1&0&0&0&1&0&0&0&0&0&0&0&1&1&1&0&1&1&1&1&1&0&1&0&0\\
0&0&1&0&0&1&1&0&1&0&0&0&1&0&0&0&0&0&0&0&1&1&1&0&1&1&1&1&1&0&1&0\\
0&0&0&1&0&0&1&1&0&1&0&0&0&1&0&0&0&0&0&0&0&1&1&1&0&1&1&1&1&1&0&1\\
0&0&0&0&1&0&0&1&1&0&1&0&0&0&1&0&0&0&0&0&0&0&1&1&1&0&1&1&1&1&1&0\\
0&0&0&0&0&1&0&0&1&1&0&1&0&0&0&1&0&0&0&0&0&0&0&1&1&1&0&1&1&1&1&1
\end{bmatrix}$$

Figure 4.1: Matrix $\mathbf{G}_{32\times32}$ for the generator polynomial CRC-32 when $l = 32$.

Then dropping $\mathbf{x}_{1\times m}$ from (4.4) and (4.5) and one obtains

$$\mathbf{s}^{[i]}_{m\times1}(x) = \mathbf{G}_{m\times l} \cdot \mathbf{t}^{[i]}_{l\times1}. \tag{4.6}$$

We note that having the matrix $\mathbf{G}_{m\times l}$ expressed in this form (4.4) simplifies the task of developing software to generate it for different generator polynomials and degrees of parallelism. An example of the matrix $\mathbf{G}_{32\times32}$ using the generator polynomial CRC-32 is shown in Figure 4.1.

4.2.2 Realization

We begin by noting that since the iteration number has no impact on an architecture, for convenience, one can drop the superscript notation from the terms in the parallel CRC formulations. However, since these equations are recursive and contain both $s_j^{[i]}$ and $s_j^{[i-1]}$ for $0 \leq i \leq q-1$ and $0 \leq j \leq m-1$, we can mark the next CRC terms, i.e., $s_j^{[i]}$, with primes. This notation is also used in [22] and [26].

Observing (4.6), it should be clear that there are m equations (one equation per syndrome bit),

$$\begin{aligned} s'_0 &= f_0\left(t_m, t_{m+1}, \cdots, t_{m+l-1}\right) \\ s'_1 &= f_1\left(t_m, t_{m+1}, \cdots, t_{m+l-1}\right) \\ &\vdots \\ s'_{m-1} &= f_{m-1}\left(t_m, t_{m+1}, \cdots, t_{m+l-1}\right). \end{aligned}$$

A parallel CRC circuit can be constructed by direct realization of the m equations described by the multiplication $\mathbf{G}_{m\times l}\cdot\mathbf{t}_{l\times 1}$, i.e., independent realizations of the functions f_is.

It is possible to use complexity reduction techniques to share hardware between these m parallel equations, i.e., sub-expression sharing. Some of the published sub-expression sharing techniques include [65], [66], [67], [68], and [69]. Initially, we attempted to develop our own complexity reduction approach tailored for CRCs, but afterward we found [68] which outperformed our approach and subsequently we abandoned this topic.

Now, recalling from Chapter 2, where Δ and Θ are defined to be the time and area complexity of a hardware architecture, respectively, we now let δ_i and θ_i represent the delay and hardware complexity of f_i for $0 \leq i \leq m-1$, respectively. Then, the CPD of a hardware architecture Δ is determined by,

$$\Delta = \max\left(\delta_0, \delta_1, \cdots, \delta_{m-1}\right),$$

and we define the total hardware complexity of a parallel CRC architecture as

$$\Theta = m \cdot C_X + \sum_{i=0}^{m-1} \theta_i + m \cdot C_F,$$

where, C_X and C_F were previously defined to be the cost of an XOR gate and FF, respectively.

As we are interested in the fast realization of f_is, XOR trees can be utilized to implement the equations. It is known that an n-bit XOR tree has hardware cost $(n-1)\cdot C_X$ and $\lceil \log_2 n \rceil$ gate levels. Therefore, if all the terms in an equation have delay $0 \cdot T_X$, this results in a delay of $\lceil \log_2 n \rceil \cdot T_X$, where T_X is the delay of an XOR gate.

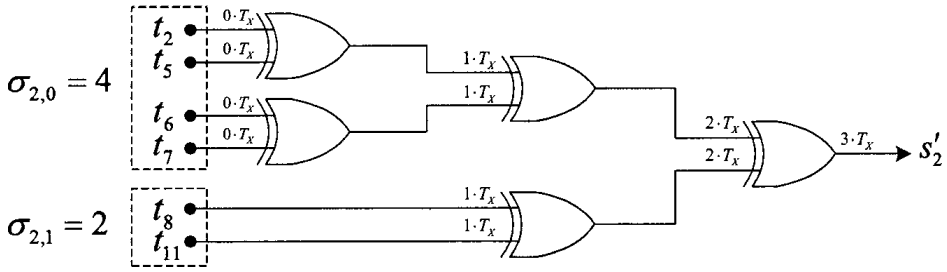


Figure 4.2: Example DARC-8 XOR tree architecture for $s'_2 = t_2 + t_5 + t_6 + t_7 + t_8 + t_{12}$ with $l = 16$.

Since $l > m$, t_is are computed with different delays (either $0 \cdot T_X$ or $1 \cdot T_X$). Thus it is possible to reduce the δ_is if proper care is taken when constructing the XOR trees. As can be seen in (4.2), terms t_m to t_{l-1} can be obtained with no delay, whereas the terms t_l to t_{m+l-1} have the delay $1 \cdot T_X$. Therefore, in this architecture, we begin to construct the XOR tree by pair-wise XORing the terms t_m to t_{l-1} that are present in each f_i, and at the same time we obtain the present t_l to t_{m+l-1} terms. Assuming that in f_i there are $\sigma_{i,0}$ terms with no delay and $\sigma_{i,1}$ terms with $1 \cdot T_X$ delay, if $\sigma_{i,1} > 0$ then the results can be summed up after $\lceil \log_2 (\sigma_{i,1} + \lceil \frac{\sigma_{i,0}}{2} \rceil) \rceil$ levels. Thus, the delay of s'_i is equal to

$$\delta_i = \left(1 + \left\lceil \log_2 \left(\sigma_{i,1} + \left\lceil \frac{\sigma_{i,0}}{2} \right\rceil\right)\right\rceil\right) \cdot T_X, \tag{4.7}$$

and the hardware complexity is

$$\theta_i = (\sigma_{i,0} + \sigma_{i,1} - 1) \cdot C_X. \tag{4.8}$$

Figure 4.2 graphically shows how one can build an XOR tree for the s'_2 equation of the generator polynomial DARC-8 ($G(x) = 1 + x^3 + x^4 + x^5 + x^8$ [27]) with $l = 16$.

Considering all the above remarks, we can now present a generalized design of the parallel LFSR2 Architecture when $l > m$ as shown in Figure 4.3, which appears in [41]. It is an extension of the parallel CRC architecture presented in [22] and illustrated with conventions similar to those of the parallel LFSR1 Architecture for $l = m$ in Figure 4 of [26], with some optimizations specific to the case $l > m$. The first optimization is trivial, like previously published primitive architectures, we share the feedback XOR gates that combine s_{j-l} and b_{j-m} for $l \leq j \leq l + m - 1$ (4.2). The second optimization comes from the previous discussion about how one can reduce the overall delay by first having a level of XOR gates that combine the present t_j for $m \leq j \leq l - 1$ terms with no delay, and then construct an XOR tree with the remaining terms.

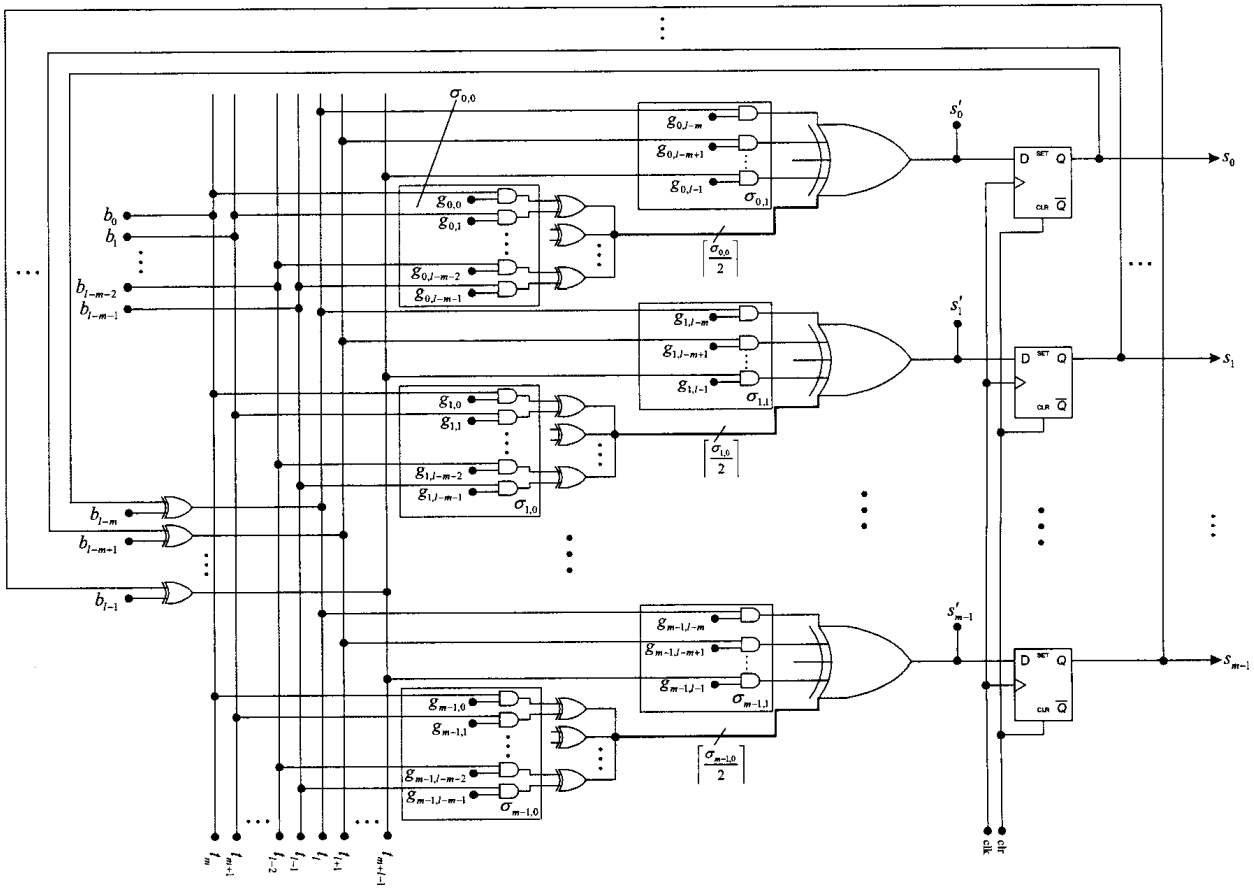


Figure 4.3: Generalized optimized parallel LFSR2 Architecture.

This architecture was novel in the sense that no one had undertaken a study of the design when $l > m$. In the following chapter, simulations are performed for this situation and the optimum degrees of parallelism are determined.

4.3 Lambda Gamma Approach

In this section, we present the Lambda Gamma approach for performing the CRC computation. We consider this to be the most significant contribution contained the thesis. This approach gives rise to both a software algorithm and hardware architecture. Compared to the existing approaches, the novel software algorithm provides high-performance and requires low-memory usage.

4.3.1 Formulation

Here, we examine the relationships between the columns of $\mathbf{G}_{m\times l}$ and discuss how the matrix is constructed. Afterward, we propose a matrix decomposition that we call the Lambda Gamma decomposition. This derivation assumes that the degree of parallelism is greater than or equal to the degree of the generator polynomial, i.e., $l \geq m$. We note that in [70], the original formulation for the modular reduction with $l = m - 2$ is introduced and it has been used to design a bit-serial multiplier. In the following, we extend this development to the CRC computation for $l \geq m$ and propose software and hardware approaches.

Beginning from the formulation in (4.6), the entries of the left-most column of $\mathbf{G}_{m\times l}$ are obtained from the coefficients of $G(x)$ as follows: $g_{i,0} = g_i$, where $g_{i,0} \in \mathbf{G}_{m\times l}$ and $g_i \in G(x)$, for $0 \leq i \leq m-1$; this is because

$$\begin{aligned} x^m \bmod G(x) &= \left(1 + \sum_{i=1}^{m-1} g_i x^i\right) \bmod G(x) \\ &= 1 + \sum_{i=1}^{m-1} g_i x^i \end{aligned}$$

Therefore, when computing the remaining $l-1$ right-most columns of $\mathbf{G}_{m\times l}$, in general, each column represents the result of $x^{m+j} \bmod G(x)$ for $0 < j < l$, and we have

$$x^{m+j} \bmod G(x) = \left(x^j + \sum_{i=1}^{\tau-1} g_i x^{i+j} + x^{\tau+j}\right) \bmod G(x).$$

Recall that x^{τ} denotes the order of the second greatest nonzero term in the generator polynomial, i.e.,

$$G(x) = 1 + \sum_{i=1}^{\tau-1} g_i x^i + x^{\tau} + x^m.$$

Now, if $j < m - \tau$ then no further reduction by $G(x)$ is required [70], and for these cases, the top-most entry of the j-th column is equal to zero, i.e., $g_{0,j} = 0$. However, when $\tau + j = m$, the term $x^{\tau+j}$, requires reduction, i.e., substitute $x^{\tau+j} = x^m = x^{\tau} + \sum_{i=1}^{\tau-1} g_i x^i + 1$; this results in the entry $g_{0,m-\tau} = 1$. For the remaining cases $\tau + j > m$, the value of $g_{0,j}$ depends on the generator polynomial. Also, it is interesting to note that $g_{0,j}$s are fixed for a given generator polynomial.

From the entries in the top-most row of $\mathbf{G}_{m\times l}$, i.e., $g_{0,j}$ for $0 \leq j \leq l-1$, we define the set Λ such that

$$\begin{aligned} \Lambda &= \{\lambda_0, \lambda_1, \cdots, \lambda_{|\Lambda|-1}\} \\ &= \{j \mid g_{0,j} = 1, j \in [0, l-1], g_{0,j} \in \mathbf{G}_{m\times l}\}. \end{aligned} \tag{4.9}$$

In other words, Λ can be seen as the set of js for $0 \leq j \leq l-1$, for which the coefficient of the least significant term, i.e., x^0, in the polynomial representation of $x^{m+j} \bmod G(x)$ is 1. Note that $\lambda_0 = 0$, since $g_0 = 1$ for all generator polynomials [1]. Moreover, the set Λ is easily computed using the serial LFSR2 Architecture (Figure 2.3b), by feeding in $1, 0, 0, \cdots, 0$, and recording the cycle numbers when $s_0 = 1$ (counting the first cycle as cycle 0).

From the earlier discussion concerning reduction, it is clear that $\lambda_0 = 0$ and $\lambda_1 = m - \tau$. With the left-most column of $\mathbf{G}_{m\times l}$, which contains the coefficients of the generator polynomial, one can obtain the columns between λ_0 and λ_1 of $\mathbf{G}_{m\times l}$, as

$$\begin{aligned} x^{m+j} \bmod G(x) &= x^j \cdot \left(1 + \sum_{i=1}^{\tau-1} g_i x^i + x^{\tau}\right) \\ &= x^j + \sum_{i=1}^{\tau-1} g_i x^{i+j} + x^{\tau+j}, \end{aligned}$$

for $\lambda_0 = 0 < j < \lambda_1 = \tau$; which is equivalent to a j-fold down-shift operation of the

left-most column of $\mathbf{G}_{m\times l}$. For the case $j = \lambda_1$, we have

$$\begin{aligned} x^{m+\lambda_1} \bmod G(x) &= x^{\lambda_1} \cdot \left(1 + \sum_{i=1}^{\tau-1} g_i x^i + x^\tau\right) \\ &= x^{\lambda_1} + \sum_{i=1}^{\tau-1} g_i x^{i+\lambda_1} + x^{\tau+\lambda_1}. \end{aligned} \tag{4.10}$$

Substituting $\lambda_1 = m - \tau$ into the right-hand side of (4.10), one obtains

$$\begin{aligned} x^{m+\lambda_1} \bmod G(x) &= x^{m-\tau} + \sum_{i=1}^{\tau-1} g_i x^{i+m-\tau} + x^{\tau+m-\tau} \\ &= x^{m-\tau} + \sum_{i=1}^{\tau-1} g_i x^{i+m-\tau} + x^m, \end{aligned}$$

which is equal to the addition of the left-most column of $\mathbf{G}_{m\times l}$ with its λ_1-fold down-shifts. Next, for $\lambda_1 < j < \lambda_2$, $x^{m+j} \bmod G(x)$ can be obtained by $(j - \lambda_1)$-fold down-shifts of column λ_1. This is the same as the addition of the j-fold and $(j - \lambda_1)$-fold down-shifts of the left-most column in $\mathbf{G}_{m\times l}$. The remaining columns can be obtained similarly.

4.3.2 Matrix Decomposition

From the set Λ, we introduce a group of $|\Lambda|$ matrices that have the dimension $m \times l$ denoted as $\breve{\mathbf{\Lambda}}^{[\lambda_k]}_{m\times l}$ for $0 \le k \le |\Lambda| - 1$. To construct $\breve{\mathbf{\Lambda}}^{[\lambda_k]}_{m\times l}$, begin with an $m \times l$ null matrix, add a south-east ($\searrow$) diagonal string of 1s, which originate from λ_k-th column in the top-most row of $\breve{\mathbf{\Lambda}}^{[\lambda_k]}_{m\times l}$, and extend across to the opposite side of the matrix; the remaining entries of $\breve{\mathbf{\Lambda}}^{[\lambda_k]}_{m\times l}$ are 0s, i.e.,

$$\breve{\lambda}^{[\lambda_k]}_{i,j} = \begin{cases} 0 & i \neq j - \lambda_k \\ 1 & i = j - \lambda_k \end{cases},$$

where

$$\breve{\mathbf{\Lambda}}^{[\lambda_k]}_{m\times l} = \begin{bmatrix} \breve{\lambda}^{[\lambda_k]}_{0,0} & \breve{\lambda}^{[\lambda_k]}_{0,1} & \cdots & \breve{\lambda}^{[\lambda_k]}_{0,l-1} \\ \breve{\lambda}^{[\lambda_k]}_{1,0} & \breve{\lambda}^{[\lambda_k]}_{1,1} & \cdots & \breve{\lambda}^{[\lambda_k]}_{1,l-1} \\ \vdots & \vdots & \ddots & \vdots \\ \breve{\lambda}^{[\lambda_k]}_{m-1,0} & \breve{\lambda}^{[\lambda_k]}_{m-1,1} & \cdots & \breve{\lambda}^{[\lambda_k]}_{m-1,l-1} \end{bmatrix}.$$

Alternatively, the matrix $\breve{\mathbf{\Lambda}}^{[\lambda_k]}_{m\times l}$ can be expressed as

$$\breve{\mathbf{\Lambda}}^{[\lambda_k]}_{m\times l} = \begin{bmatrix} \mathbf{0}_{m\times\lambda_k} & \mathbf{D}_{m\times(l-\lambda_k)} \end{bmatrix}$$

where $\mathbf{D}_{m\times(l-\lambda_k)}$ is a rectangular diagonal matrix[1]. Also, note that square brackets have been chosen to describe the $\breve{\mathbf{\Lambda}}^{[\lambda_k]}_{m\times l}$ matrices, and this should not be confused with the square brackets used to denote iteration numbers elsewhere in the thesis.

Next, define the matrix down-shift operator, denoted by $\downarrow i$, which acts on the matrix $\mathbf{A}_{p\times q}$ as

$$(\mathbf{A}_{p\times q})_{\downarrow i} = \begin{bmatrix} [\mathbf{0}_{i\times q}] \\ [\mathbf{A}_{(p-i)\times q}] \end{bmatrix},$$

where $[\mathbf{A}_{(q-i)\times q}]$ denotes the $p-i$ top-most rows of $\mathbf{A}_{p\times q}$ and $i \leq q$. We note that this matrix $\downarrow i$ operator can be expressed by the matrix multiplication

$$(\mathbf{A}_{p\times q})_{\downarrow i} = \begin{bmatrix} [\mathbf{0}_{i\times p}] \\ [\mathbf{D}_{(p-i)\times p}] \end{bmatrix} \cdot \mathbf{A}_{p\times q}.$$

With the matrix down-shift operator defined and generalizing the above construction explanations, we can decompose the matrix $\mathbf{G}_{m\times l}$ as

$$\mathbf{G}_{m\times l} = \sum_{j\in\Gamma} \left(\sum_{k\in\Lambda} \breve{\mathbf{\Lambda}}^{[k]}_{m\times l} \right)_{\downarrow j}. \tag{4.11}$$

Furthermore, we can express the equation given in (4.11) as a matrix product

$$\mathbf{G}_{m\times l} = \mathbf{\Gamma}_{m\times m} \cdot \mathbf{\Lambda}_{m\times l}, \tag{4.12}$$

where

$$\Gamma_{m\times m} = \begin{bmatrix} g_0 & 0 & \cdots & 0 \\ g_1 & g_0 & \cdots & 0 \\ \vdots & \vdots & \ddots & 0 \\ g_{m-1} & g_{m-2} & \cdots & g_0 \end{bmatrix}$$

[1] $\mathbf{D} = [d_{i,j}]$ and $d_{i,j} = 1$ if $i = j$ else $d_{i,j} = 0$.

and

$$\mathbf{\Lambda}_{m\times l} = \begin{bmatrix} g_{0,0} & g_{0,1} & \cdots & g_{0,m-1} & \cdots & g_{0,l-1} \\ 0 & g_{0,0} & \cdots & g_{0,m-2} & \cdots & g_{0,l-2} \\ \vdots & \vdots & \ddots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & g_{0,0} & \cdots & g_{0,l-m} \end{bmatrix},$$

recalling that $g_i \in G(x)$ and $g_{i,j} \in \mathbf{G}_{m\times l}$. To prove (4.12) is equivalent to (4.11), we make the following observation

$$\mathbf{\Lambda}_{m\times l} = \sum_{k\in\Lambda} \breve{\mathbf{\Lambda}}^{[k]}_{m\times l}, \tag{4.13}$$

which is obvious from our definition of the $\breve{\mathbf{\Lambda}}^{[\lambda_k]}_{m\times l}$ matrices. Then, substituting (4.13) into (4.11) and one obtains

$$\begin{aligned} \mathbf{G}_{m\times l} &= \sum_{j\in\Gamma} (\mathbf{\Lambda}_{m\times l})_{\downarrow j} \\ &= (\mathbf{\Lambda}_{m\times l})_{\downarrow\gamma_0} + (\mathbf{\Lambda}_{m\times l})_{\downarrow\gamma_1} + \cdots + (\mathbf{\Lambda}_{m\times l})_{\downarrow\gamma_{|\Gamma|-1}}. \end{aligned} \tag{4.14}$$

Replacing the down-shift operations with their equivalent rectangular diagonal matrix products, one obtains

$$\begin{aligned} \mathbf{G}_{m\times l} &= (\mathbf{\Lambda}_{m\times l})_{\downarrow\gamma_0} + (\mathbf{\Lambda}_{m\times l})_{\downarrow\gamma_1} + \cdots + (\mathbf{\Lambda}_{m\times l})_{\downarrow\gamma_{|\Gamma|-1}} \\ &= \begin{bmatrix} \mathbf{0}_{\gamma_0\times l} \\ \mathbf{D}_{(l-\gamma_0)\times l} \end{bmatrix} \cdot \mathbf{\Lambda}_{m\times l} + \begin{bmatrix} \mathbf{0}_{\gamma_1\times l} \\ \mathbf{D}_{(l-\gamma_1)\times l} \end{bmatrix} \cdot \mathbf{\Lambda}_{m\times l} + \cdots + \\ &\quad \begin{bmatrix} \mathbf{0}_{\gamma_{|\Gamma|-1}\times l} \\ \mathbf{D}_{(l-\gamma_{|\Gamma|-1})\times l} \end{bmatrix} \cdot \mathbf{\Lambda}_{m\times l} \\ &= \left(\begin{bmatrix} \mathbf{0}_{\gamma_0\times l} \\ \mathbf{D}_{(l-\gamma_0)\times l} \end{bmatrix} + \begin{bmatrix} \mathbf{0}_{\gamma_1\times l} \\ \mathbf{D}_{(l-\gamma_1)\times l} \end{bmatrix} + \cdots + \begin{bmatrix} \mathbf{0}_{\gamma_{|\Gamma|-1}\times l} \\ \mathbf{D}_{(l-\gamma_{|\Gamma|-1})\times l} \end{bmatrix} \right) \cdot \mathbf{\Lambda}_{m\times l} \end{aligned} \tag{4.15}$$

From the definition of the set Γ in (2.9), it is clear that the expanded sum in (4.15) is equal to $\mathbf{\Gamma}_{m\times m}$. Thus, we have shown both decompositions to be equivalent. Finally, we note that the sets Λ and Γ serve as compact and convenient methods to represent the $\mathbf{\Lambda}_{m\times l}$ and $\mathbf{\Gamma}_{m\times m}$ matrices, respectively.

Next, the matrix $\mathbf{G}_{m\times l}$ decomposition expressed in summation form (4.11) is used to propose a software algorithm, whereas, the product form decomposition (4.12) is used to propose a hardware architecture. Before proceeding we provide the following example to illustrate how one can decompose a matrix $\mathbf{G}_{m\times l}$.

An Example

Consider constructing and decomposing the matrix $\mathbf{G}_{4\times5}$ using the CCITT-4 generator polynomial ($G(x) = 1 + x + x^4$ [27]) and $l = 5$. From $G(x)$ one can quickly obtain $m = 4$ and $\Gamma = \{\gamma_0, \gamma_1\} = \{0, 1\}$. Next,

$$\mathbf{x}_{1\times4} \cdot \mathbf{G}_{4\times5} = \begin{bmatrix} x^4 \bmod G(x) \\ x^5 \bmod G(x) \\ \vdots \\ x^8 \bmod G(x) \end{bmatrix}^T \Rightarrow \mathbf{G}_{4\times5} = \begin{bmatrix} \mathbf{1} & \mathbf{0} & \mathbf{0} & \mathbf{1} & \mathbf{1} \\ 1 & 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 & 0 \end{bmatrix}$$

is computed. From the top-most row of $\mathbf{G}_{4\times5}$ (shown in boldface), one extracts $\Lambda = \{\lambda_0, \lambda_1, \lambda_2\} = \{0, 3, 4\}$.

Next, using the set Λ, the three $\breve{\mathbf{\Lambda}}_{4\times5}^{[\lambda_k]}$ matrices are constructed as

$$\breve{\mathbf{\Lambda}}_{4\times5}^{[0]} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \end{bmatrix}, \quad \breve{\mathbf{\Lambda}}_{4\times5}^{[3]} = \begin{bmatrix} 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix},$$

and

$$\breve{\mathbf{\Lambda}}_{4\times5}^{[4]} = \begin{bmatrix} 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}.$$

Now, consider the inner summation of (4.11), and compute the sum

$$\begin{aligned} \sum_{k\in\Lambda} \left(\breve{\mathbf{\Lambda}}_{4\times5}^{[k]} \right) &= \breve{\mathbf{\Lambda}}_{4\times5}^{[0]} + \breve{\mathbf{\Lambda}}_{4\times5}^{[3]} + \breve{\mathbf{\Lambda}}_{4\times5}^{[4]} \\ &= \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \end{bmatrix} + \begin{bmatrix} 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix} + \begin{bmatrix} 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix} \\ &= \begin{bmatrix} 1 & 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \end{bmatrix}. \end{aligned}$$

Finally,

$$\sum_{j\in\Gamma}\left(\begin{bmatrix}1&0&0&1&1\\0&1&0&0&1\\0&0&1&0&0\\0&0&0&1&0\end{bmatrix}\right)_{\downarrow j} = \begin{bmatrix}1&0&0&1&1\\0&1&0&0&1\\0&0&1&0&0\\0&0&0&1&0\end{bmatrix}_{\downarrow 0} + \begin{bmatrix}1&0&0&1&1\\0&1&0&0&1\\0&0&1&0&0\\0&0&0&1&0\end{bmatrix}_{\downarrow 1}$$

$$= \begin{bmatrix}1&0&0&1&1\\0&1&0&0&\mathbf{1}\\0&0&1&0&0\\0&0&0&1&0\end{bmatrix} + \begin{bmatrix}0&0&0&0&0\\1&0&0&1&\mathbf{1}\\0&1&0&0&1\\0&0&1&0&0\end{bmatrix}$$

$$= \begin{bmatrix}1&0&0&1&1\\1&1&0&1&\mathbf{0}\\0&1&1&0&1\\0&0&1&1&0\end{bmatrix}$$

$$= \mathbf{G}_{4\times 5}.$$

Notice the single cancellation which occurs in the matrix addition (shown in boldface).

Alternatively, using (4.12) and starting from the sets Γ and Λ, one constructs

$$\mathbf{\Gamma}_{4\times 4} = \begin{bmatrix}1&0&0&0\\1&1&0&0\\0&1&1&0\\0&0&1&1\end{bmatrix} \text{ and } \mathbf{\Lambda}_{4\times 5} = \begin{bmatrix}1&0&0&1&1\\0&1&0&0&1\\0&0&1&0&0\\0&0&0&1&0\end{bmatrix}.$$

Then

$$\mathbf{\Gamma}_{4\times 4}\cdot\mathbf{\Lambda}_{4\times 5} = \begin{bmatrix}1&0&0&0\\1&1&0&0\\0&1&1&0\\0&0&1&1\end{bmatrix}\cdot\begin{bmatrix}1&0&0&1&1\\0&1&0&0&1\\0&0&1&0&0\\0&0&0&1&0\end{bmatrix}$$

$$= \begin{bmatrix}1&0&0&1&1\\1&1&0&1&0\\0&1&1&0&1\\0&0&1&1&0\end{bmatrix}$$

$$= \mathbf{G}_{4\times 5}.$$

As a final remark, we observe that the equality stated in (4.13) holds in this example.

4.3.3 Algorithm Realization

From the matrix decomposition presented in (4.11), we propose a new software algorithm that performs the CRC computation, we call it the Lambda Gamma Algorithm (CRC$\Lambda\Gamma$). Like the all the previously published algorithms, our proposed algorithm processes the message iteratively and we adopt the reverse Endianness convention for representing the message and syndrome polynomials in memory (see Figure 3.15).

Beginning with the matrix and vector product in (4.6) and substituting (4.11), one can obtain

$$\mathbf{s}^{[i]}_{m\times 1} = \left(\sum_{j\in\Gamma} \left(\sum_{k\in\Lambda} \breve{\mathbf{\Lambda}}^{[k]}_{m\times l} \right)_{\downarrow j} \right) \cdot \mathbf{t}^{[i]}_{l\times 1}. \tag{4.16}$$

When realizing (4.16) in software, after forming $\mathbf{t}^{[i]}_{l\times 1}$, we move it into the inner summation of (4.16) and perform the multiplications with the $\breve{\mathbf{\Lambda}}^{[\lambda_k]}_{m\times l}$ matrices. This corresponds to extracting and summing m-bit groups from $\mathbf{t}^{[i]}_{l\times 1}$ beginning at the λ_k-th left-most bit position. If λ_k points to a position in $\mathbf{t}^{[i]}_{l\times 1}$ where m consecutive bits would run off the end of $\mathbf{t}^{[i]}_{l\times 1}$, then zeros are used to fill the higher order positions (as would be the result of $\breve{\mathbf{\Lambda}}^{[\lambda_k]}_{m\times l} \cdot \mathbf{t}^{[i]}_{l\times 1}$). We denote the m-bit intermediate result of the set Λ summation by $\mathbf{l}^{[i]}_{m\times 1}$, i.e.,

$$\mathbf{l}^{[i]}_{m\times 1} = \sum_{k\in\Lambda} \left(\breve{\mathbf{\Lambda}}^{[k]}_{m\times l} \cdot \mathbf{t}^{[i]}_{l\times 1} \right). \tag{4.17}$$

Since we are discussing a software algorithm, in fact (4.17) is realized as

$$\mathbf{l}^{[i]}_{m\times 1} = \breve{\mathbf{\Lambda}}^{[\lambda_0]}_{m\times l} \cdot \mathbf{t}^{[i]}_{l\times 1} + \breve{\mathbf{\Lambda}}^{[\lambda_1]}_{m\times l} \cdot \mathbf{t}^{[i]}_{l\times 1} + \cdots + \breve{\mathbf{\Lambda}}^{\left[\lambda_{|\Lambda|-1}\right]}_{m\times l} \cdot \mathbf{t}^{[i]}_{l\times 1}.$$

Afterward, the set Γ summation is carried out on the intermediate vector $\mathbf{l}^{[i]}_{m\times 1}$ and the result is stored as the next syndrome. The set Γ summation is realized in software as

$$\mathbf{s}^{[i]}_{m\times 1} = \left(\mathbf{l}^{[i]}_{m\times 1}\right)_{\downarrow\gamma_0} + \left(\mathbf{l}^{[i]}_{m\times 1}\right)_{\downarrow\gamma_1} + \cdots + \left(\mathbf{l}^{[i]}_{m\times 1}\right)_{\downarrow\gamma_{|\Gamma|-1}}.$$

Finally, the order of operations for CRC$\Lambda\Gamma$ is denoted by parentheses as

$$\mathbf{s}^{[i]}_{m\times 1} = \sum_{j\in\Gamma} \left(\sum_{k\in\Lambda} \left(\breve{\mathbf{\Lambda}}^{[k]}_{m\times l} \cdot \left(\mathbf{t}^{[i]}_{l\times 1} \right) \right) \right)_{\downarrow j}. \tag{4.18}$$

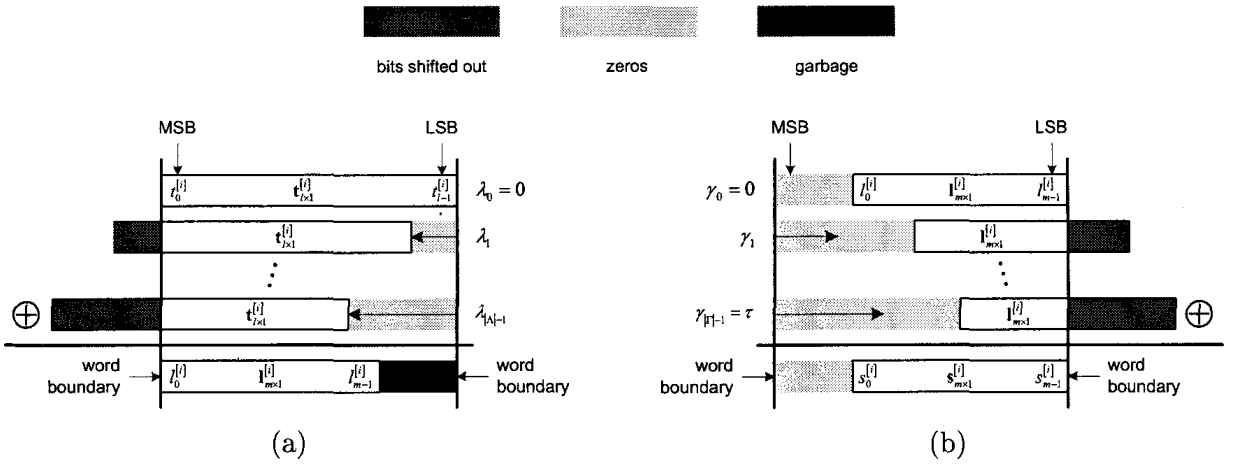


Figure 4.4: Illustration of the Lambda Gamma Algorithm when $l \geq m$ and $l = w$: (a) left-shifting by the set Λ, (b) right-shifting by the set Γ.

Figure 4.4 shows a pictorial representation of the Lambda Gamma Algorithm when $l \geq m$. The Λ and Γ summations are shown as a group of left-shifts and right-shifts, in Figure 4.4a and 4.4b, respectively. Again, we remind the reader that we are using the reverse Endianness convention and the algorithmic form of CRC$\Lambda\Gamma$(32) is shown in Algorithm 4.1 as a C++ code snippet. One observers that the implementation is straightforward and consists of two for-loops.

Similar to the CRCT, CRCR, and CRCS4 algorithms, the CRC$\Lambda\Gamma$ algorithm requires precomputation operations, consisting of the sets Λ and Γ for a given generator polynomial and degree of parallelism. In Algorithm 4.2, we show a code snippet for a possible C++ implementation of the Lambda LUT generation algorithm. The algorithm is derived from the definition of the set Λ in (4.9), and uses a modified version

Algorithm 4.1 Lambda Gamma Algorithm.

```
crc = INIT_VALUE;
while (p_buf < p_end) {
     term = crc ^ *(uint32_t *)p_buf;

     intermediate = 0x0;
     for (i=0; i<lambda_size; i++)
          intermediate ^= term << lambda[i];

     crc = 0x0;
     for (i=0; i<gamma_size; i++)
          crc ^= intermediate >> gamma[i];

     p_buf += 4;
}
return crc ^ FINAL_VALUE;
```

Algorithm 4.2 Lambda LUT Generation Algorithm.

```
lut_size = 0;
lfsr = polynomial;

for (i=0; i<dop; i++) {
     if (((lfsr >> (gpd-1)) & 0x1) == 0x1)
          lut[lut_size++] = i;

     if ((lfsr & 0x1) == 0x1)
          lfsr = (lfsr >> 1) ^ polynomial;
     else
          lfsr >>= 1;
}
```

Table 4.1: Lambda sets for the frequently referenced generator polynomials.

$G(x)$	Λ for $l = 32$	$\lvert\Lambda\rvert$
CRC-12	$\{0, 1, 2, 3, 4, 5, 6, 7, 8, 11, 12, 13, 14, 15,$ $16, 17, 22, 23, 24, 25, 26, 29, 30\}$	23
CRC-16	$\{0, 1, 2, 3, 4, 5, 6, 7, 9, 10, 11, 12, 13, 15, 16, 17,$ $18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 30, 31\}$	29
CCITT-16	$\{0, 4, 8, 11, 12, 19, 20, 22, 26, 27, 28\}$	11
CRC-16†	$\{0, 2, 4, 6, 8, 10, 12, 14, 15, 18, 19, 22, 23, 26, 27, 31\}$	16
CCITT-16†	$\{0, 5, 10, 12, 15, 16, 20, 22, 24, 25, 26, 29, 30\}$	13
CRC-32	$\{0, 6, 9, 10, 12, 16, 24, 25, 26, 28, 29, 30, 31\}$	13

of CRCB to perform the $x^{m+i} \bmod G(x)$ reductions for $0 \leq i \leq l-1$ and records the iterations when the term x^0 is present in the result. The Λ sets for the frequently referenced generator polynomials with $l = 32$ are provided in Table 4.1. The set Γ is easily obtained from the coefficients of the generator polynomial, and Γ sets for the frequently referenced generator polynomials were provided earlier in Table 2.3.

4.3.4 Architecture Realization

From the matrix and vector product presented in (4.12) and substituting (4.11), one can obtain

$$\mathbf{s}_{m\times 1}^{[i]} = (\mathbf{\Gamma}_{m\times m} \cdot \mathbf{\Lambda}_{m\times l}) \cdot \mathbf{t}_{l\times 1}^{[i]}. \tag{4.19}$$

When realizing (4.19) in hardware, $\mathbf{t}_{l\times 1}^{[i]}$ is formed by the addition of the input message block and the previous syndrome. Then, the multiplication between the matrix $\mathbf{\Lambda}_{m\times l}$ and vector $\mathbf{t}_{l\times 1}^{[i]}$ is performed, resulting in an m-bit intermediate vector, denoted by $\mathbf{l}_{m\times 1}^{[i]}$, i.e.,

$$\mathbf{l}_{m\times 1}^{[i]} = \mathbf{\Lambda}_{m\times l} \cdot \mathbf{t}_{l\times 1}^{[i]}.$$

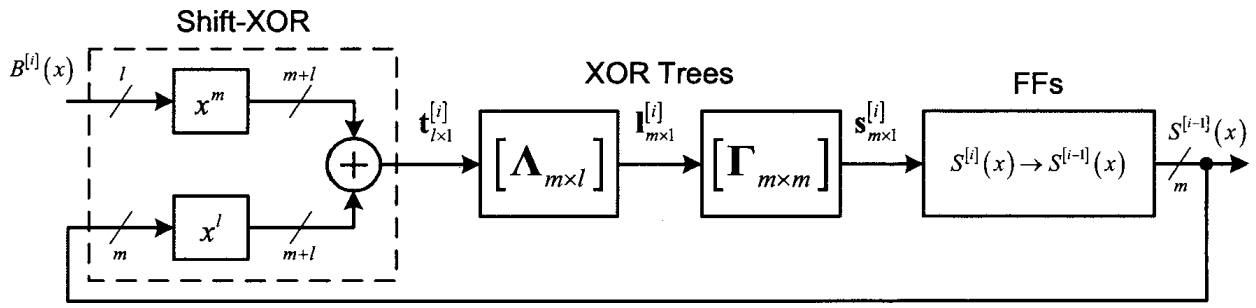


Figure 4.5: Generalized Lambda Gamma Architecture.

The vector $\mathbf{l}^{[i]}_{m\times 1}$ is subsequently multiplied by the matrix $\mathbf{\Gamma}_{m\times m}$ and the result is stored as the next syndrome in the m-bit FF array. The order of operations is denoted by parentheses as

$$\mathbf{s}^{[i]}_{m\times 1} = \left(\mathbf{\Gamma}_{m\times m} \cdot \left(\mathbf{\Lambda}_{m\times l} \cdot \left(\mathbf{t}^{[i]}_{l\times 1}\right)\right)\right). \tag{4.20}$$

A pictorial representation of (4.20) when $l \geq m$ is shown in Figure 4.5. Since the delays of the vector inputs to these matrices can be different, one should proceed by pairing terms in order of least delay first, in order to obtain the fastest overall CPD. We have taken this approach for our implementations and our results are stated in the following chapter.

4.4 Extended Binary Polynomial Architecture

In this section, we introduce a new parameter that extends the existing binary polynomial formulations of the CRC computation. This parameter, denoted as p, allows one to derive both the LFSR2 and LFSR1 formulations from a common staring point. We call this the LFSRp approach, and show it is useful for obtaining optimized primitive hardware architectures in terms of CPD and latency for a given generator polynomial and degree of parallelism.

4.4.1 Formulation

We begin by modifying the original CRC equation (2.2) as

$$S(x) = \left(x^{m-p} \cdot \hat{U}(x)\right) \bmod G(x), \tag{4.21}$$

where

$$\hat{U}(x) = x^p \cdot U(x). \tag{4.22}$$

Next, we decompose our discussion into two parts that consider non-negative and negative values of p. From this point forward, hats are used to denote variables used to describe the LFSRp formulation.

Non-negative p

Assuming that $p \geq 0$, we provide recursive polynomial-based definitions similar to the ones in the previous chapter, and derive a generalized parallel polynomial-based CRC formulation with the parameter p.

Message Partitioning The approach taken is similar to what is done for the primitive parallel LFSR2 and LFSR1 formulations presented in the previous chapter. Begin by partitioning $\hat{U}(x)$ into $\hat{q}$ message blocks, i.e.,

$$\hat{U}(x) = \sum_{i=0}^{\hat{q}-1} x^{l\cdot(\hat{q}-1-i)} \cdot \hat{B}^{[i]}(x),$$

where $\hat{B}^{[i]}(x)$ represents a binary polynomial of at most degree $l-1$ corresponding to the l-bit message block being processed at the i-th iteration. Again, if $(k+p) \bmod G(x) \neq 0$, then we assume one can prepend $l - ((k+p) \bmod G(x))$ 0s to $\hat{U}(x)$ to increase its length to a multiple of l.

Let $\hat{U}^{[i]}(x)$ be the portion of $\hat{U}(x)$ that contains all the blocks $\hat{B}^{[j]}(x)$ for $0 \leq j \leq i$, and let $\hat{S}^{[i]}(x)$ be the syndrome of $\hat{U}^{[i]}(x)$. Also, define $\hat{U}^{[-1]}(x) = 0$ and $\hat{S}^{[-1]}(x) = S_{\text{init}}(x)$, then these definitions can be written as

$$\begin{aligned} \hat{U}^{[i]}(x) &= x^l \cdot \hat{U}^{[i-1]}(x) + \hat{B}^{[i]}(x), \\ \hat{S}^{[i]}(x) &= \left(x^{m-p} \cdot \hat{U}^{[i]}(x)\right) \bmod G(x), \end{aligned}$$

for $0 \leq i \leq \hat{q}-1$. It is noted that $\hat{U}(x) = \hat{U}^{[\hat{q}-1]}(x)$ and $S(x) = \hat{S}^{[\hat{q}-1]}(x)$. For more information, we refer the reader to examine Figure 3.1, where the message polynomial relationships for the LFSR2 approach are illustrated.

Derivation From the previous definitions, one can derive a generalized recursive expression for $\hat{S}^{[i]}(x)$ in terms of $\hat{S}^{[i-1]}(x)$ and $\hat{B}^{[i]}(x)$, i.e.,

$$\begin{aligned}
\hat{S}^{[i]}(x) &= \left(x^{m-p} \cdot \hat{U}^{[i]}(x)\right) \bmod G(x) \\
&= \left(x^{m-p} \cdot \left(x^{l} \cdot \hat{U}^{[i-1]}(x) + \hat{B}^{[i]}(x)\right)\right) \bmod G(x) \\
&= \left(x^{l} \cdot x^{m} \cdot \hat{U}^{[i-1]}(x) + x^{m} \cdot \hat{B}^{[i]}(x)\right) \bmod G(x) \\
&= \left(x^{l} \cdot \hat{S}^{[i-1]}(x) + x^{m-p} \cdot \hat{B}^{[i]}(x)\right) \bmod G(x) \\
&= \hat{T}^{[i]}(x) \bmod G(x), \qquad (4.23)
\end{aligned}$$

where

$$\hat{T}^{[i]}(x) = x^{l} \cdot \hat{S}^{[i-1]}(x) + x^{m-p} \cdot \hat{B}^{[i]}(x). \qquad (4.24)$$

for $0 \leq i \leq \hat{q} - 1$.

Observing (4.23), there are a few cases to consider depending on the values of l, m, and p. The discussion of these cases is first partitioned by the ranges $0 \leq p \leq m$ and $p > m$.

Case I: $0 \leq p \leq m$. First, we note that for any l and $G(x)$, if $p = 0$ then a primitive parallel LFSR2 formulation is obtained. Conversely, if $p = m$, then we obtain a primitive parallel LFSR1 formulation. Similar to the LFSR2 approach, we first form $\hat{T}^{[i]}(x)$ by performing the addition of (3.10), then the final reduction is carried out (4.23). Depending on l and m there are different overlapping situations between the terms of $x^{l} \cdot \hat{S}^{[i-1]}(x)$ and $x^{m-p} \cdot \hat{B}^{[i]}(x)$ and some of them are illustrated in Figure 4.6.

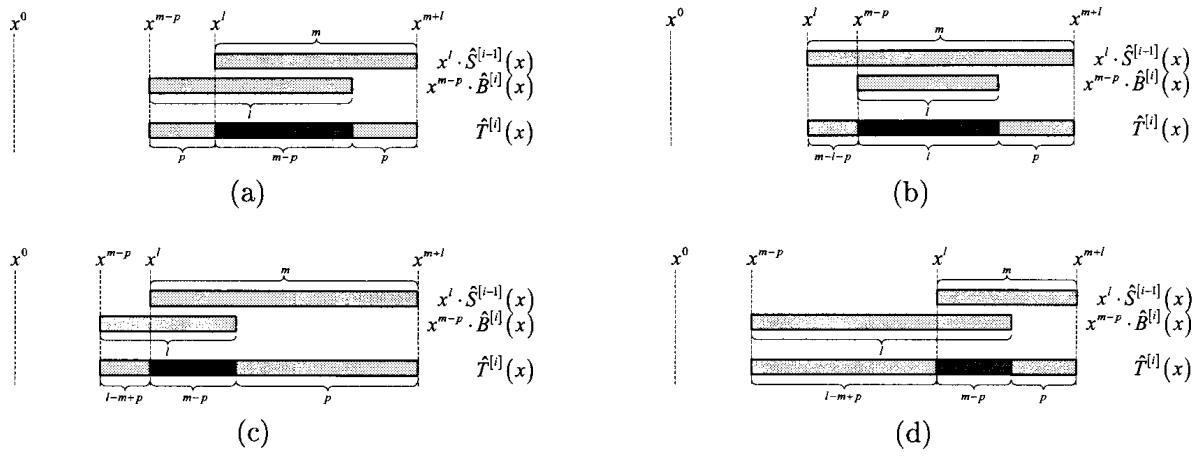


Figure 4.6: Illustrations of some of the LFSRp overlapping situations between $x^{l} \cdot \hat{S}^{[i-1]}(x)$ and $x^{m-p} \cdot \hat{B}^{[i]}(x)$ in $\hat{T}^{[i]}(x)$: (a) $l = m$ and $0 \leq p \leq m$, (b) $l < m$ and $0 \leq p \leq m - l$, (c) $l < m$ and $m - l < p < m$, (d) $l > m$ and $0 \leq p < m$.

From Figure 4.6 one can observe that only the terms in the overlap will require XOR gates to complete the addition in (4.24), and consequently we note that for $0 \leq p \leq m$, all implementations have equal hardware complexities. As proof, first it is clear that for all situations where we have complete overlap between $x^l \cdot \hat{S}^{[i-1]}(x)$ and $x^{m-p} \cdot \hat{B}^{[i]}(x)$ the hardware complexity is equal. Since in these cases, $\hat{T}^{[i]}(x)$ has the same range of powers of x present, thus the reduction (4.24) is identical and the number of overlapping terms is equal, see Figure 4.6b. Now, we note that regardless of the value of p, the degree of $\hat{T}^{[i]}(x)$ is at least x^m because we have $l \geq 1$. Since the degree of $\hat{T}^{[i]}(x)$ is greater than or equal to $G(x)$, by properties of modular reduction each coefficient of $\hat{S}^{[i]}(x)$ will be influenced by at least one coefficient of $\hat{S}^{[i-1]}(x)$. Therefore, when we shift the degrees of the present $\hat{B}^{[i]}(x)$ terms to positions which do not require reduction, i.e., degrees less than x^m, we remove one XOR gate required to form $\hat{T}^{[i]}(x)$, but one XOR gate is added to sum the newly added $x^{m-p} \cdot \hat{B}^{[i]}(x)$ term that does not require reduction with the reduced $\hat{S}^{[i-1]}(x)$ terms.

Case II: $p > m$ When $p \geq m$, as shown in Figure (4.7), there are no overlapping terms between $x^l \cdot \hat{S}^{[i-1]}(x)$ and $x^{m-p} \cdot \hat{B}^{[i]}(x)$ in (4.23). Consequently, the implementation of this computation can be carried out as

$$\begin{aligned} \hat{S}^{[i]}(x) &= \left(x^l \cdot \hat{S}^{[i-1]}(x) + x^{m-p} \cdot \hat{B}^{[i]}(x)\right) \bmod G(x) \\ &\quad \left(x^l \cdot \hat{S}^{[i-1]}(x)\right) \bmod G(x) + \left(x^{m-p} \cdot \hat{B}^{[i]}(x)\right) \bmod G(x), \end{aligned}$$

for $0 \leq i \leq \hat{q} - 1$.

Furthermore, when $p > m$ there are terms in $x^{m-p} \cdot \hat{B}^{[i]}(x)$ which have negative powers of x. Generally, the reduction of these negative power terms is as complex as the terms with powers greater than x^m. Therefore, it is not of any interest to explore this case because the latency is equal to $\lceil \frac{k+p}{l} \rceil$ cycles.

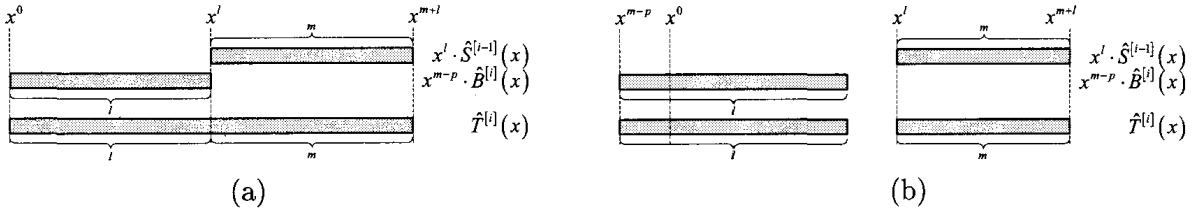


Figure 4.7: Illustrations of some of the LFSRp non-overlapping situations between $x^l \cdot \hat{B}^{[i]}(x)$ in $\hat{T}^{[i]}(x)$: (a) $l = m$ and $p = m$, (b) $l > m$ and $p > m$.

Negative p

Assuming $p < 0$, we can rewrite (4.22) as

$$\hat{U}(x) = x^p \cdot \left(u_0 + u_1 x + \cdots + u_{k-1} x^{k-1}\right),$$

or

$$\hat{U}(x) = u_0 x^p + u_1 x^{p+1} + \cdots + u_{|p|-1} x^{-1} + u_{|p|} + u_{|p|+1} x + \cdots + u_{k-1} x^{k+p-1}. \quad (4.25)$$

One can observe that there are two groups of terms in (4.25). One group includes the terms with negative powers of x and the other one includes the terms with non-negative powers of x. Therefore, one can split (4.25) into two parts as

$$\hat{U}(x) = \hat{U}_1(x) + \hat{U}_2(x), \quad (4.26)$$

where,

$$\hat{U}_1(x) = u_0 x^p + u_1 x^{p+1} + \cdots + u_{|p|-1} x^{-1}, \quad (4.27)$$

and

$$\hat{U}_2(x) = x_{|p|} + u_{|p|+1} x + \cdots + u_{k-1} x^{k+p-1}. \quad (4.28)$$

As shown above, (4.27) has $|p|$ terms with negative powers of x and (4.28) has $k - |p|$ terms with non-negative powers of x. Substituting (4.26) into (4.21) and we obtain

$$\hat{S}(x) = \left(x^{m-p} \cdot \left(\hat{U}_1(x) + \hat{U}_2(x)\right)\right) \bmod G(x),$$

which, by properties of modular reduction, can be split into two independent computations as

$$\hat{S}_1(x) = \left(x^{m-p} \cdot \hat{U}_1(x)\right) \bmod G(x), \quad (4.29)$$

and

$$\hat{S}_2(x) = \left(x^{m-p} \cdot \hat{U}_2(x)\right) \bmod G(x). \quad (4.30)$$

Now, the final CRC can be obtained using the following

$$\hat{S}(x) = \hat{S}_1(x) + \hat{S}_2(x). \quad (4.31)$$

However, this approach has two drawbacks, the message length k must be known in advance and the first half of the message buffered, before the parallel computation can be performed. We feel that these dependencies are not desirable for typical implementations of the CRC computation, and for this reason we do not consider this approach further.

4.4.2 Realization

For non-negative p values, the generalized parallel LFSRp Architecture is illustrated in Figure 4.8. As previously noted, this approach can give rise to both generalized parallel LFSR2 (Figure 3.5) and LFSR1 (Figure 3.6) Architectures by selecting $p = 0$ and $p = m$, respectively.

Discussion

Since we have shown that the hardware complexity is constant for all the implementations of LFSRp with $0 \leq p \leq m$, we are interested in finding the optimum p for $0 \leq p \leq m$, in order to minimize the overall time complexity of the CRC computation. This is accomplished by finding all the set of ps that result in a realization with minimum CPD. Then, the smallest p in the set is identified as the optimum p which reduces the computational latency.

Revisiting the serial LFSR Architectures shown in Figure 2.3, if we wish to modify those architectures and find the optimum p in $0 \leq p \leq m$ for $l = 1$, then we search for the maximum value of i where $g_i = 0$ and $g_i \in G(x)$, and then we select $p = m - i$. In other words, we are looking to place the input at the right-most LFSR position without a present feedback connection, and this results in an implementation with CPD $1 \cdot T_X$ and computation time $k + p$ cycles.

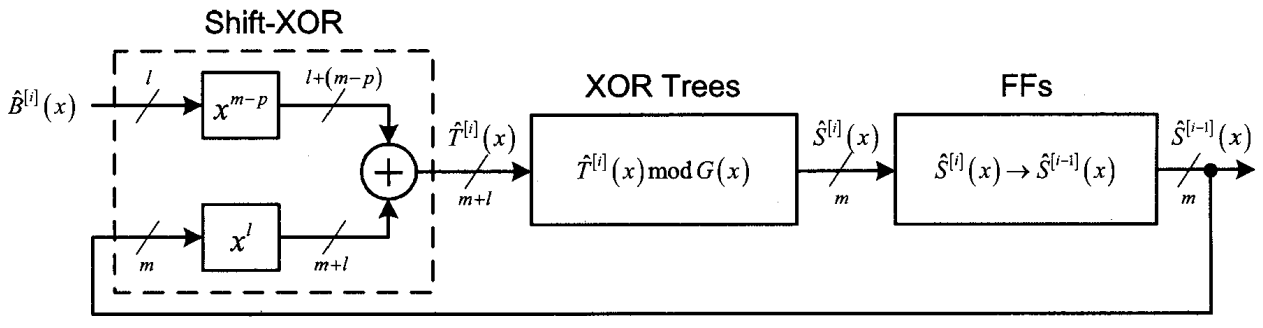


Figure 4.8: Generalized parallel LFSRp Architecture.

For parallel CRC computation architectures, determining the optimum p is not as straightforward. To this end, we provide an illustrative example with $m = l = 6$ and show the affect of three different ps.

Example

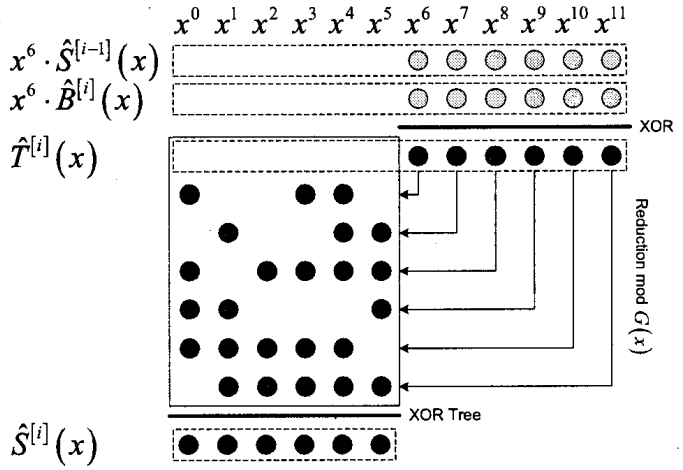
Consider the generator polynomial DARC-6 ($G(x) = 1+x^3+x^4+x^6$ [27]) with degree of parallelism $l = 6$. We provide dot notations for the LFSRp implementations with $p = 0$, 4, and 6, in Figures 4.9a, 4.9b, 4.9c, respectively. We use white and gray dots to denote terms with no delay and $1 \cdot T_X$ delay, respectively, while the syndrome bits are denoted with black dots.

In these figures, we first show how to form $\hat{T}^{[i]}(x)$ and then perform the reduction $\hat{T}^{[i]}(x)$ mod $G(x)$, observing the reduction in Figure 4.9a (marked with arrows), beginning with the first term requiring reduction, we show

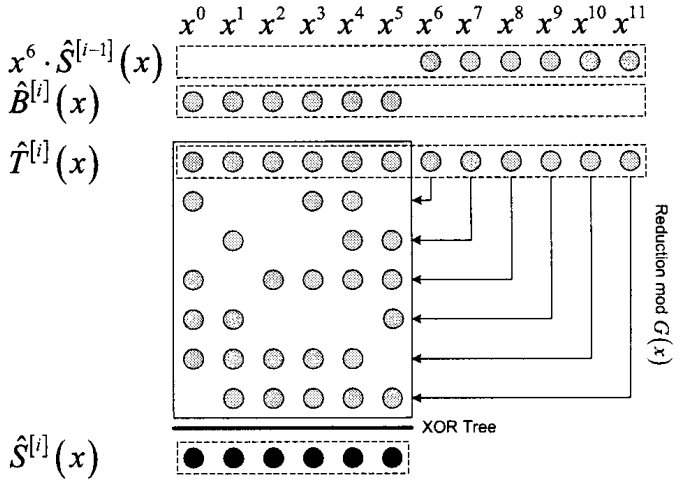
$$\begin{aligned} x^6 &= 1 + x^3 + x^4 \\ x^7 &= x + x^4 + x^5 \\ &\vdots \\ x^{11} &= x + x^2 + x^3 + x^4 + x^5. \end{aligned}$$

The other two figures have similar reduction patterns, however $\hat{T}^{[i]}(x)$ has some terms with different delays and powers that do not require reduction.

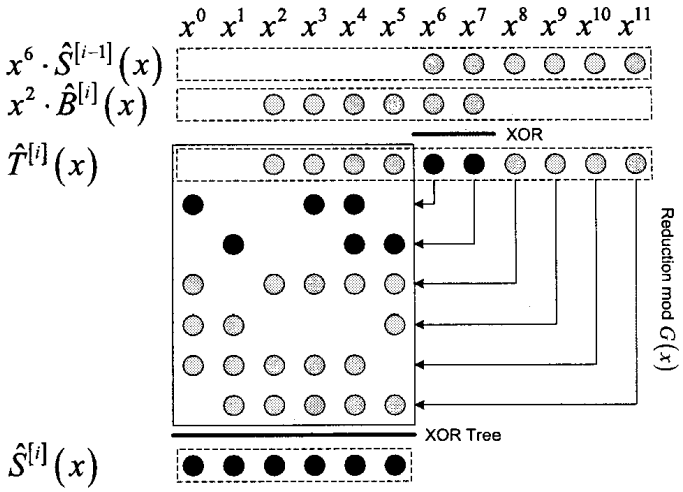
Using the XOR construction technique where we pair $0 \cdot T_X$ terms first before construction XOR trees, the CPDs are $4 \cdot T_X$, $3 \cdot T_X$, and $3 \cdot T_X$ for the $p = 0$, 4, and 6 implementations, respectively. One can observe that all three implementations require 24 XOR gates. For the other ps in $0 \leq p \leq 6$ the corresponding CRC implementations can be obtained similarly, all require 24 XOR gates, $\lceil \frac{k+p}{l} \rceil$ clock cycles, and have CPDs of $4 \cdot T_X$, $4 \cdot T_X$, $4 \cdot T_X$, $3 \cdot T_X$, $3 \cdot T_X$, and $3 \cdot T_X$, respectively. However, the CRC realization with $p = 4$ is identified as the optimum p for $0 \leq p \leq m$, because it minimizes CPD and latency.



(a)



(b)



(c)

Figure 4.9: Example DARC-6 dot notation for the parallel LFSRp Architecture with $l = 6$: (a) $p = 0$, (b) $p = 4$, (c) $p = 6$.

4.5 Message Splitting Architecture

In this section, we explore the application of idea of message splitting proposed in [60] as a software algorithm to a hardware architecture. The authors of [60] suggested that a hardware architecture based on their software algorithm could be the source of some future work.

In this approach, one computes a number of separate syndromes for different portions of the message. These syndromes are finally combined to obtain the syndrome of the entire message. The formulation of this approach begins by following the LFSR2 binary polynomial formulation discussed in the previous chapter. Unlike the generalized LFSRp formulation with $p < 0$ that also splits the message, this approach does not depend on the message length k. The derivation we provide is for situations when the message is partitioned into to two groups. Extending this approach to higher orders, such as four, is rather straightforward.

4.5.1 Formulation

By recalling how the message polynomial was partitioned into l-bit blocks in (3.1), one can perform the following expansion,

$$\begin{aligned} U(x) &= \sum_{i=0}^{q-1} x^{l\cdot(q-1-i)} \cdot B^{[i]}(x) \\ &= x^{l\cdot(q-1)} \cdot B^{[0]}(x) + x^{l\cdot(q-2)} \cdot B^{[1]}(x) + x^{l\cdot(q-3)} \cdot B^{[2]}(x) + \\ &\quad x^{l\cdot(q-4)} \cdot B^{[3]}(x) + \cdots + x^{l} \cdot B^{[q-2]}(x) + B^{[q-1]}(x). \end{aligned}$$

Now, assuming that one has the ability to prepend the required number of zeros to extend the message to a length such that there are an even number of message blocks, i.e, $q \bmod 2 = 0$ and then grouping the blocks in $U(x)$ into evens and odds, one obtains

$$\begin{aligned} U(x) &= x^{l\cdot(q-1)} \cdot B^{[0]}(x) + x^{l\cdot(q-3)} \cdot B^{[2]}(x) + \cdots + x^{l} \cdot B^{[q-2]}(x) + \\ &\quad x^{l\cdot(q-2)} \cdot B^{[1]}(x) + x^{l\cdot(q-4)} \cdot B^{[3]}(x) + \cdots + B^{[q-1]}(x) \\ &= U_e(x) + U_o(x), \end{aligned}$$

where $U_e(x) = x^{l\cdot(q-1)} \cdot B^{[0]}(x) + x^{l\cdot(q-3)} \cdot B^{[2]}(x) + \cdots + x^{l} \cdot B^{[q-2]}(x)$ and $U_o(x) = x^{l\cdot(q-2)} \cdot B^{[1]}(x) + x^{l\cdot(q-4)} \cdot B^{[3]}(x) + \cdots + B^{[q-1]}(x)$.

Let $U_e^{[i]}(x)$ and $U_o^{[i]}(x)$ be the portions of $U(x)$ that contain all the blocks $B^{[2j]}(x)$ and $B^{[2j+1]}(x)$ for $0 \leq j \leq i$, respectively, and let $S_e^{[i]}(x)$ and $S_o^{[i]}(x)$ be the syndromes of $U_e^{[i]}(x)$ and $U_o^{[i]}(x)$, respectively. For convenience, define $B_e^{[i]}(x) = B^{[2i]}(x)$ and $B_o^{[i]}(x) = B^{[2i+1]}(x)$. Also, define $U_e^{[-1]}(x) = U_o^{[-1]}(x) = 0$ and $S_e^{[-1]}(x) = S_{\text{init}}(x)$ and $S_o^{[-1]}(x) = 0$, where $S_{\text{init}}(x)$ denotes the initial content of the CRC register. Then these definitions can be written as

$$\begin{aligned} U_e^{[i]}(x) &= x^{2l} \cdot U_e^{[i-1]}(x) + B_e^{[i]}(x), \\ S_e^{[i]}(x) &= \left(x^m \cdot U_e^{[i]}(x)\right) \bmod G(x), \end{aligned}$$

and

$$\begin{aligned} U_o^{[i]}(x) &= x^{2l} \cdot U_o^{[i-1]}(x) + x^l \cdot B_o^{[i]}(x), \\ S_o^{[i]}(x) &= \left(x^m \cdot U_o^{[i]}(x)\right) \bmod G(x), \end{aligned}$$

for $0 \leq i \leq \lceil \frac{q-1}{2} \rceil$. It is noted that $U(x) = U_e^{[\lceil \frac{q-1}{2} \rceil]}(x) + U_o^{[\lceil \frac{q-1}{2} \rceil]}$ and $S(x) = S_e^{[\lceil \frac{q-1}{2} \rceil]}(x) + S_o^{[\lceil \frac{q-1}{2} \rceil]}(x)$.

Derivation

From the previous definitions, one can derive recursive expressions for $S_e^{[i]}(x)$ and $S_o^{[i]}(x)$ using a manner similar to the polynomial based derivations, i.e,

$$\begin{aligned} S_e^{[i]}(x) &= \left(x^m \cdot U_e^{[i]}(x)\right) \bmod G(x) \\ &= \left(x^m \cdot \left(x^{2l} \cdot U_e^{[i-1]}(x) + B_e^{[i]}(x)\right)\right) \bmod G(x) \\ &= \left(x^{2l} \cdot x^m \cdot U_e^{[i-1]}(x) + x^m \cdot B_e^{[i]}(x)\right) \bmod G(x) \\ &= \left(x^{2l} \cdot S_e^{[i-1]}(x) + x^m \cdot B_e^{[i]}(x)\right) \bmod G(x) \\ &= T_e^{[i]}(x) \bmod G(x), \end{aligned} \tag{4.32}$$

for $0 \leq i \leq \lceil \frac{q-1}{2} \rceil$, where

$$T_e^{[i]}(x) = x^{2l} \cdot S_e^{[i-1]}(x) + x^m \cdot B_e^{[i]}(x), \tag{4.33}$$

and

$$\begin{aligned} S_o^{[i]}(x) &= \left(x^m \cdot U_o^{[i]}(x)\right) \bmod G(x) \\ &= \left(x^m \cdot \left(x^{2l} \cdot U_o^{[i-1]}(x) + x^l \cdot B_o^{[i]}(x)\right)\right) \bmod G(x) \\ &= \left(x^{2l} \cdot x^m \cdot U_o^{[i-1]}(x) + x^{m+l} \cdot B_o^{[i]}(x)\right) \bmod G(x) \\ &= \left(x^{2l} \cdot S_o^{[i-1]}(x) + x^{m+l} \cdot B_o^{[i]}(x)\right) \bmod G(x) \\ &= T_o^{[i]}(x) \bmod G(x), \end{aligned} \tag{4.34}$$

for $0 \leq i \leq \left\lceil \frac{q-1}{2} \right\rceil$, where

$$T_o^{[i]}(x) = x^{2l} \cdot S_o^{[i-1]}(x) + x^{m+l} \cdot B_o^{[i]}(x). \tag{4.35}$$

4.5.2 Realization

The hardware realization of the split approach is shown in Figure 4.10. One observes that two accumulators are required, each with m FFs and the message block is split in half (into even and odd digits). Next, we analyze the CPD of this approach and show that it can outperform the extended binary polynomial approach for some cases. However, simulations should be undertaken to see if useful cases exist, and this could be the focus of some future work as outlined in Chapter 6.

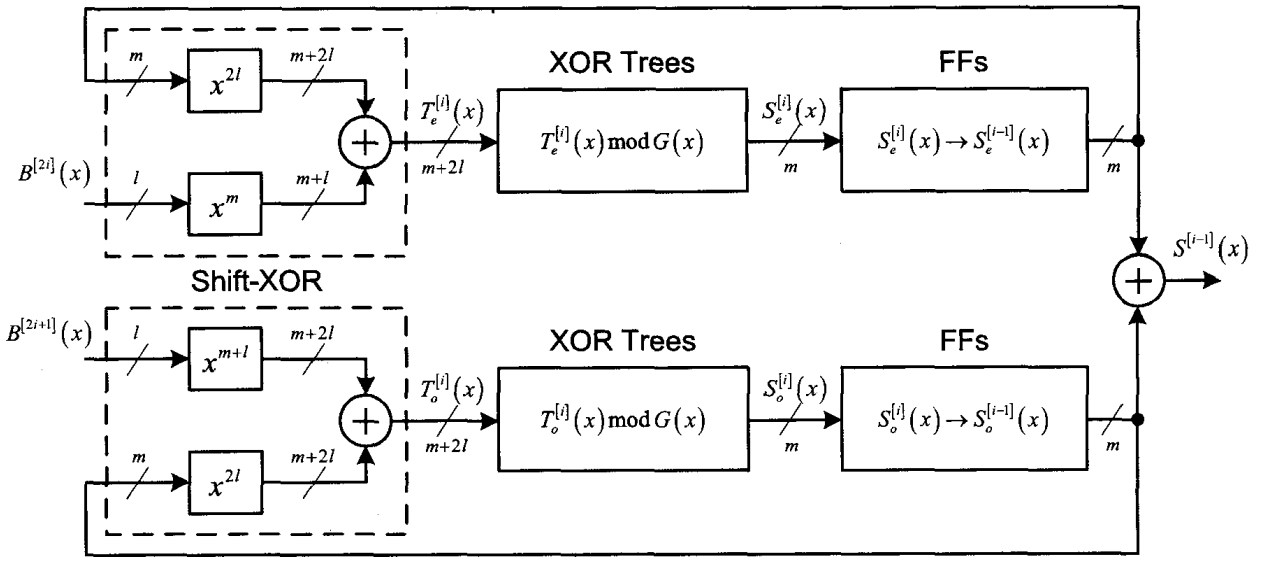


Figure 4.10: Generalized parallel Message Splitting Architecture.

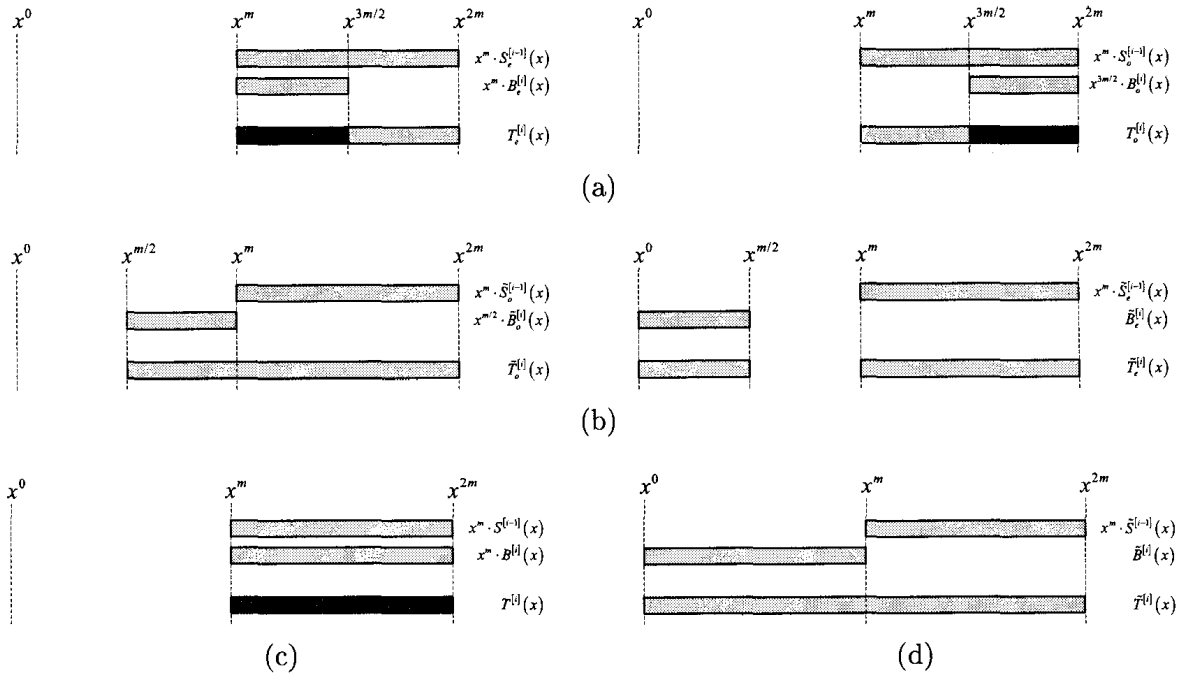


Figure 4.11: Illustrations of the Message Splitting Architecture overlapping polynomial situations when $l = m$: (a) Splitting LFSR2, (b) Splitting LFSR1, (c) LFSR2, (d) LFSR1.

Discussion

The formulation described by (4.32), (4.33), (4.34), and (4.35) is strongly based on the binary polynomial approach analyzed in the previous chapter. Here, we have shown that one is free to split the of blocks the message polynomial into groups and process these groups in parallel. It is noted that these blocks need not be of equal length, but for simplicity we have chosen to split the message into even and odd digits. As noted in [60], this approach is different than the message slicing approach [7] because it uses multiple accumulators.

Observing the polynomial manipulations in (4.32) and (4.34), it should be clear that the extended binary polynomial approach (4.21) can easily be applied to generalize the splitting formulation. Here, the final LFSR1 result is provided without derivation,

$$\begin{aligned} \tilde{T}_e^{[i]}(x) &= x^{2l} \cdot \tilde{S}_e^{[i-1]}(x) + \tilde{B}_e^{[i]}(x) \\ \tilde{S}_e^{[i]}(x) &= \tilde{T}_e^{[i]}(x) \bmod G(x), \end{aligned}$$

and

$$\begin{aligned} \tilde{T}_o^{[i]}(x) &= x^{2l} \cdot \tilde{S}_o^{[i-1]}(x) + x^{l} \cdot \tilde{B}_o^{[i]}(x) \\ \tilde{S}_o^{[i]}(x) &= \tilde{T}_o^{[i]}(x) \bmod G(x). \end{aligned}$$

Now, we show that this approach can offer speed improvement on the existing parallel LFSR Architectures at the cost of additional hardware. Consider implementing the Splitting Architecture with $2l = m$, i.e., this approach splits message blocks of $2l$ bits into two l-bit sub-blocks during each iteration. The terms $\tilde{T}_e^{[i]}(x)$ and $\tilde{T}_o^{[i]}(x)$ to be reduced are formed as illustrated in Figure 4.11b, and Figure 4.11d shows the terms $\tilde{T}^{[i]}(x)$ for parallel LFSR1. It is clear that the reduction of the polynomials $x^m \cdot \tilde{S}_e^{[i-1]}(x)$, $x^m \cdot \tilde{S}_o^{[i-1]}(x)$, and $x^m \cdot \tilde{S}^{[i-1]}(x)$ require equal time complexity, and $\tilde{B}_e^{[i]}(x) + x^{\frac{m}{2}} \cdot \tilde{B}_o^{[i]}(x) = \tilde{B}^{[i]}(x)$. Therefore, the equation with the CPD in the LFSR1 approach also exists in one of the split architecture groups. Hence, we conclude that both have equal CPDs, but the hardware complexity of the Message Splitting Architecture is nearly double.

For the LFSR2 case, we reason from the bar Figures 4.11a and 4.11c as follows. It is clear that the delay of the realization of each half of the LFSR2, split approach is less than or equal to the delay of LFSR2 since not all the positions in $T_o^{[i]}(x)$ and $T_e^{[i]}$ have delay $1 \cdot T_X$. Therefore, it is possible that the CPD split architecture can be less than LFSR2. Furthermore, we have shown that the delay of LFSR1 is less than or equal to that of LFSR2, and for all the cases when LFSR2 and LFSR1 have equal CPDs the splitting approach has a chance to outperform the primitive approaches. We conclude that these situations are possible, however it is not known if they will exist for the frequently referenced generator polynomials and/or useful degrees of parallelism. A possible future study could involve performing simulations to determine if and what situations this is possible for.

4.6 Summary

In this chapter, we presented the detailed design of the parallel LFSR2 Architecture specific to the case when the degree of parallelism is greater than the degree of the generator polynomial. Afterward, three novel formulations of the CRC computation were presented. The Lambda Gamma formulation yields a high-performance low-memory software algorithm, which is suitable for implementations when the generator polynomial degree is less than or equal to the word size. The extended binary polynomial formulation demonstrates how the classical binary polynomial formulation can be generalized to allow one to derive LFSR2 and LFSR1 based formulations from a common starting point. Finally, the future work suggested in [60] is explored and shown that their proposed Message Splitting Architecture could offer improvement over the existing primitive architectures. However, simulations are required to identify those cases and we have opted to leave them for a possible future investigation.

Chapter 5

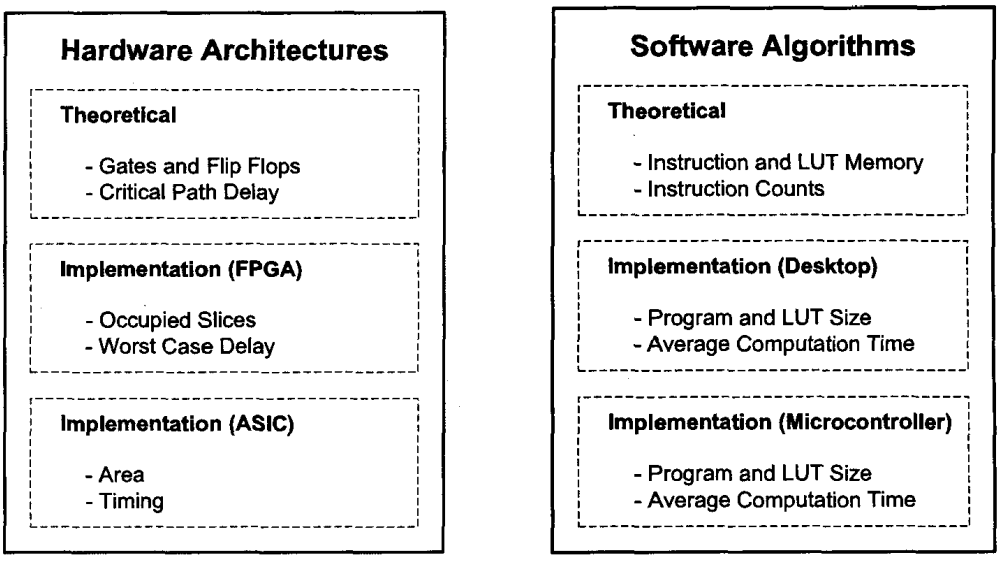
Simulations and Implementations

5.1 Preview

THE performance of a hardware architecture is evaluated in terms of its area and time complexities. Whereas, the performance of a software algorithm is measured by its memory and time complexities. Other qualitative factors such as, ease of implementation and regularity can also be considered. In the CRC computation domain for a fixed generator polynomial and degree of parallelism, hardware architectures are quantitatively compared in terms of their XOR gate and FF counts, as well as their critical path delay and latency values. Conversely, software algorithms are quantitatively compared by their memory requirements (code and data), as well as their execution times.

In this chapter, we present the data that we have gathered through our simulation and implementation experiments, which were conducted over the course of this study. We begin with the examination of the hardware architectures by performing simulations followed by comparisons and analysis. Some of these simulations are followed up with implementations for validation. Then, the performance of the software algorithms are measured with a thorough theoretical analysis, which is followed by implementations for verification. Similar to what is found the literature, our comparisons are carried out using the frequently referenced generator polynomials with useful degrees of parallelism.

In terms of the novel approaches presented in this thesis, we compare the Lambda Gamma Architecture against its non-pipelined counterparts. The parallel LFSRp Architecture is compared against the parallel LFSR1 and LFSR2 Architectures. The CRC$\Lambda\Gamma$(32) is compared against the existing software algorithms. Additionally, we present the experimental results contained in our conference papers [41] and [53].



(a) (b)

Figure 5.1: Different quantitative comparison metrics for implementations of the CRC computation: (a) hardware architectures, (b) software algorithms.

The implementations of the hardware architectures are deployed on application-specific integrated circuits (ASICs) and field-programmable gate arrays (FPGAs). The ASIC data is reported as the required area and timing, whereas, the FPGA data is given in terms of the number of occupied slices and worse case delay. These hardware metrics are summarized in Figure 5.1a. The software algorithms are compared in terms of memory, i.e., number of instructions and look-up table (LUT) sizes, and computation speed. The implementation data provides execution times for various message lengths, and these software metrics are summarized in Figure 5.1b.

5.1.1 Organization

The remainder of this chapter is organized as follows. In Section 5.2, we present the results of our hardware experiments. This includes our simulations and implementations data on FPGA and ASIC platforms. In Section 5.3, we present the results of our software experiments. This includes a theoretical analysis and some benchmarking results. This chapter is concluded with a summary in Section 5.4.

5.2 Hardware Experiments

This section presents the studies conducted on the CRC hardware architectures. First we detail the results obtained through simulations and then implementation data is provided for some selected simulation experiments.

5.2.1 Simulations

Here we provide the theoretical comparison between all the hardware architectures that perform the CRC computation studied in this thesis. In addition to the standard analysis of the algorithms in terms of the hardware and time complexity, we have undertaken two additional studies. The first study investigates and finds the optimum degrees of parallelism in terms of CPD for the parallel LFSR2 Architecture and was the focus of our conference paper [41]. The second study investigates and finds the optimum p values on the interval $0 \leq p \leq m$ for the LFSRp Architecture.

Comparison: Non-Pipelined Architectures

We begin our hardware simulations by comparing our proposed Lambda Gamma Architecture against the existing non-pipelined architectures, i.e., [22], [40], [23], and [25], when the degree of parallelism is equal to the generator polynomial degree. We remind the reader that all of these architectures require m FFs to implement and have a computation latency of $\lceil \frac{k}{l} \rceil$ clock cycles, except [40] which requires $\lceil \frac{k+m}{l} \rceil$ clock cycles to process a k-bit message. The comparison results are presented in Table 5.1.

Comparing the Lambda Gamma Architecture against the systematic approaches, i.e., [23] and [25], one observes that the proposed architecture has an equal or lower CPD, but requires more XOR gates. The only exception being CRC-32 where the Lambda Gamma Architecture requires the fewest gates of all the approaches. We remind the reader that the proposed architecture is expected to perform poorly for digit sizes less than the degree of parallelism, and its implementation is not easily described for those input sizes. However, if $l \geq m$ then we expect the area complexity to improve.

Table 5.1: Theoretical non-pipelined hardware architecture comparison for frequently referenced generator polynomials when $l = m$.

$G(x)$	CRC-12		CRC-16		CCITT-16		CRC-16†		CCITT-16†		CRC-32	
Arch	Θ	Δ	Θ	Δ	Θ	Δ	Θ	Δ	Θ	Δ	Θ	Δ
[22]	52	5	72	5	88	4	154	5	84	4	452	6
[40]	52	4	72	4	88	4	154	4	84	4	452	5
[23]	60	24	48	32	48	8	48	16	48	8	448	18
[25]	60	13	48	17	48	5	48	9	48	5	448	16
$\Lambda\Gamma$	104	7	149	6	60	5	90	6	53	5	439	8

Study: Optimum Degrees of Parallelism

This portion of the thesis outlines our results in our conference paper [41], concerning the optimum degrees of parallelism in terms of CPD for the parallel LFSR2 Architecture. We begin by discussing how we obtain the optimum degrees of parallelism. Then, with these optimum degrees of parallelism, we show how the computation time can be minimized. Finally, the time-area product efficiency metric is investigated.

For this study, we have written C++ software that computes the matrix $\mathbf{G}_{m\times l}$ for a given generator polynomial and degree of parallelism. One can compute the Δ and Θ values from the number of 1s in a row of the matrix $\mathbf{G}_{m\times l}$, and they are stated in terms of the maximum number of XOR gate levels and the total number of XOR gates, respectively. The computation time Φ of a k-bit message using the parallel LFSR2 Architecture is equal to q clock cycles multiplied by the CPD of the architecture, i.e.,

$$\Phi = \left\lceil \frac{k}{l} \right\rceil \times \Delta.$$

We restate that for a given generator polynomial, Δ is a function of l. Therefore, to obtain the fastest architectures, we are interested in finding the maximum l for a given Δ, i.e., minimize Φ.

As an illustration of how to minimize Φ, consider the DARC-8 generator polynomial ($G(x) = 1 + x^3 + x^4 + x^5 + x^8$ [27]). Through simulations, we obtain the Δ and Θ, for $1 \leq l \leq 256$, and the results are plotted in Figure 5.2. Since the CPD is computed from a ceiling function, the Δ plots resemble step functions with some spikes for values of l that are greater than m. Observing Figure 5.2, we see the first spike at $l = 16, 17$. As we are interested in finding degrees of parallelism that result in the fastest circuits, rather than selecting $l = 3$ with $\Delta = 3 \cdot T_X$ (which is the point before the first transition from $\Delta = 3 \cdot T_X$ to $\Delta = 4 \cdot T_X$), we select $l = 17$, which is the maximum l with $\Delta = 3 \cdot T_X$, and we record the hardware complexity of $\Theta = 48 \cdot C_X$.

The plots for the frequently referenced generator polynomials resemble the DARC-8 plot, however they are generally not as noisy. The results for those generator polynomials are given in Table 5.2.

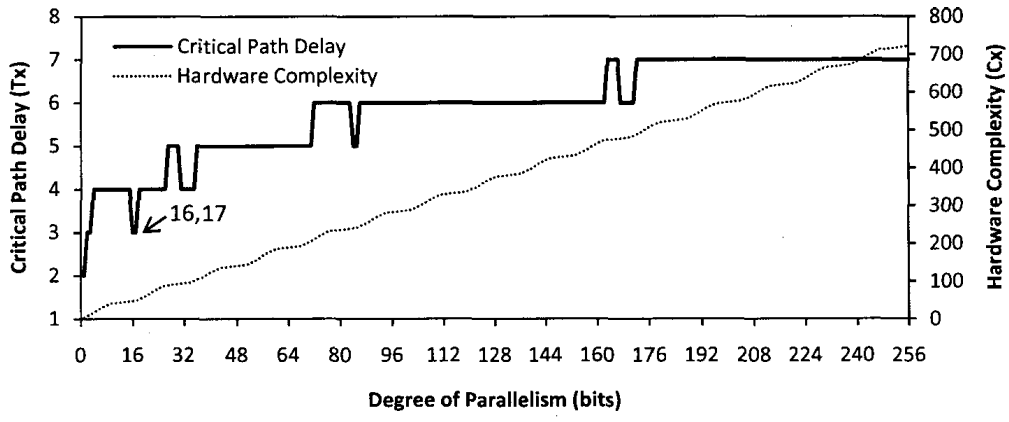


Figure 5.2: Plot of the critical path delay and hardware complexity versus the degree of parallelism, for the parallel LFSR2 Architecture using DARC-8 with $1 \leq l \leq 256$.

Table 5.2: Hardware complexity for the maximum degree of parallelism of different critical path delays using frequently referenced generator polynomials, using the parallel LFSR2 Architecture.

Δ	2		3		4		5		6		7	
$G(x)$	l	Θ	l	Θ	l	Θ	l	Θ	l	Θ	l	Θ
CRC-12	1	5	3	13	7	33	46	192	107	517	247	1345
CRC-16	1	3	3	13	7	33	18	82	61	295	146	826
CCITT-16	4	12	8	32	16	88	49	307	98	722	234	1786
CRC-16†	2	6	4	16	8	48	35	329	90	794	191	1697
CCITT-16†	5	15	10	40	19	105	55	333	106	752	236	1810
CRC-32	**1**	14	**4**	56	**13**	179	**31**	434	**80**	1169	**209**	3255

After we obtain the CPD versus the degree of parallelism data for a given generator polynomial, obtaining the computation time is rather trivial. For the generator polynomial CRC-32, we show the Δ and Φ, versus l plot for $k = 1500$ bytes (which is the MTU size for Ethernet) in Figure 5.3. For clarity, the points shown in boldface in Table 5.2 are marked on the plot. Obtaining similar plots for the other generator polynomials is not difficult. But, due to the vast number of different standards that employ these polynomials, presenting specific data for all the k-values would prove to be too cumbersome. Nevertheless, we provide timing data for the points found in Table 5.2 using a message length of $k = 1024$ bits for all the frequently referenced generator polynomials (excluding CRC-32 whose results are for $k = 1500$ bytes) and the values are shown in Table 5.3. The l values marked with asterisks are situations where the timing result of $l + 1$ is less than or equal to that of l.

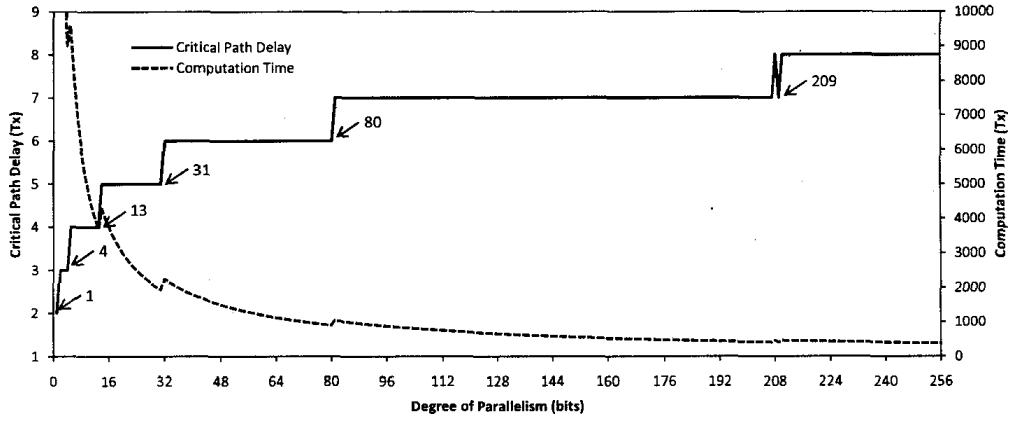


Figure 5.3: Plot of the critical path delay and computation time for $k = 1500$ bytes, versus the degree of parallelism using CRC-32, for $1 \leq l \leq 256$.

Table 5.3: Computation time for the maximum degree of parallelism of different CPDs using frequently referenced generator polynomials.

Δ	2		3		4		5		6		7	
$G(x)$	l	Φ	l	Φ	l	Φ	l	Φ	l	Φ	l	Φ
CRC-12	1*	2048	3*	1026	7	588	46	115	107	60	247	35
CRC-16	1*	2048	3*	1026	7	588	18	285	61	102	146*	56
CCITT-16	4	512	8	384	16	256	49	105	98	66	234	35
CRC-16†	2	1024	4	768	8	512	35	150	90	72	191	42
CCITT-16†	5	410	10	309	19	216	55	95	106	60	236	35
CRC-32	1*	24000	4	9000	13	3696	31	1940	80	900	209	406

The last comment we make in this experiment concerns the time-area efficiency. The hardware complexity of all the frequently referenced generator polynomials is a strictly increasing function, i.e., Θ is increased as l is increased. If we plot the time-area product $\Phi \cdot \Theta$ versus l, we can compare designs in terms of their time-area efficiency. Figure 5.4 shows the results for CRC-32 with $k = 1500$ bytes. As expected, the time-area product roughly tracks the CPD plot, and as the degree of parallelism is increased the time-area product also increases, meaning that smaller degrees of parallelism result in more efficient designs. This result shows that increasing l generally results in diminishing time-area returns.

In summary, through simulations this experiment obtained the maximum degree of parallelism for a given CPD of the parallel LFSR2 Architecture for the frequently referenced generator polynomials. Investigating the computation times for the obtained maximum degrees of parallelism, we determined that most of them are indeed the local optimum choices. Finally, we showed that the time-area product efficiency for difference degrees of parallelism generally tracks the CPD.

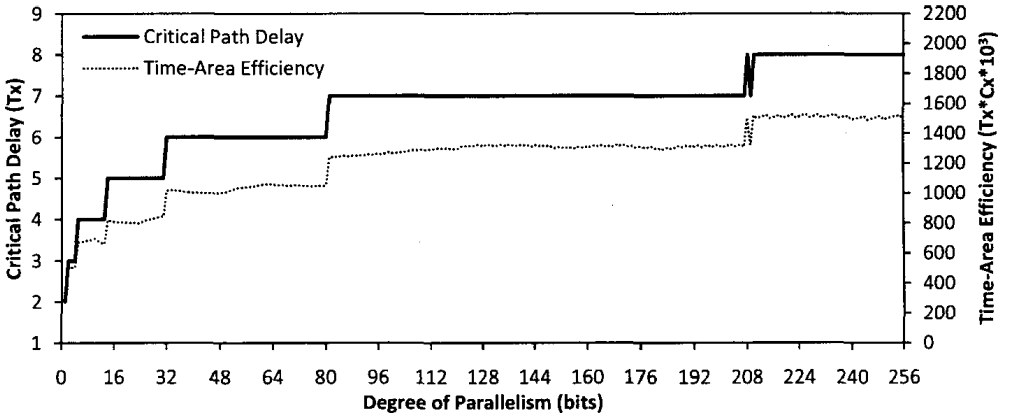


Figure 5.4: Plot of the CPD and time-area product, versus degree of parallelism, using CRC-32 for $1 \leq l \leq 256$.

Study: Optimum ps

This study concerns the selection of p to obtain the best LFSRp Architectures in terms of CPD and latency. The material is explored with the hope of reducing the time of the CRC computation. Recall from the previous chapter, where we have shown that all the LFSRp the implementations with $0 \leq p \leq m$ have equal hardware complexities and the computational latency is $\lceil \frac{k+p}{l} \rceil$.

We begin this study with an illustration of the effects of varying p for $0 \leq p \leq m$. Consider the generator polynomial CRC-32, Figure 5.5 shows plots of the CPD versus p for some useful degrees of parallelism, i.e., $l = 1$, $\frac{1}{4}m$, $\frac{1}{2}m$, m, and $2m$. Similar to the previous study, from this plots, the optimum p for a given l can be identified as the left-most point with a minimum CPD. Plots using other degrees of parallelism and/or generator polynomials can be constructed and they are similar.

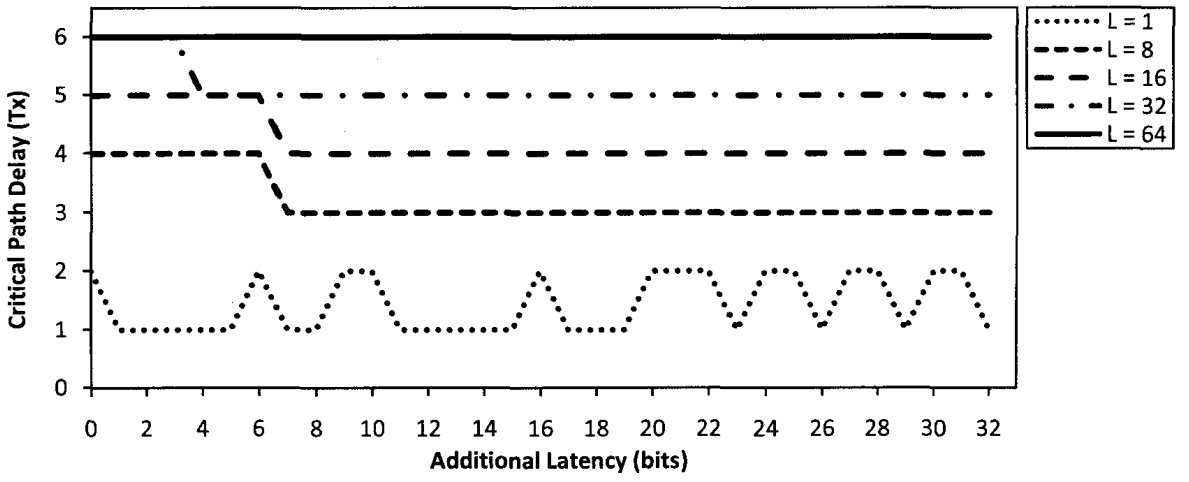
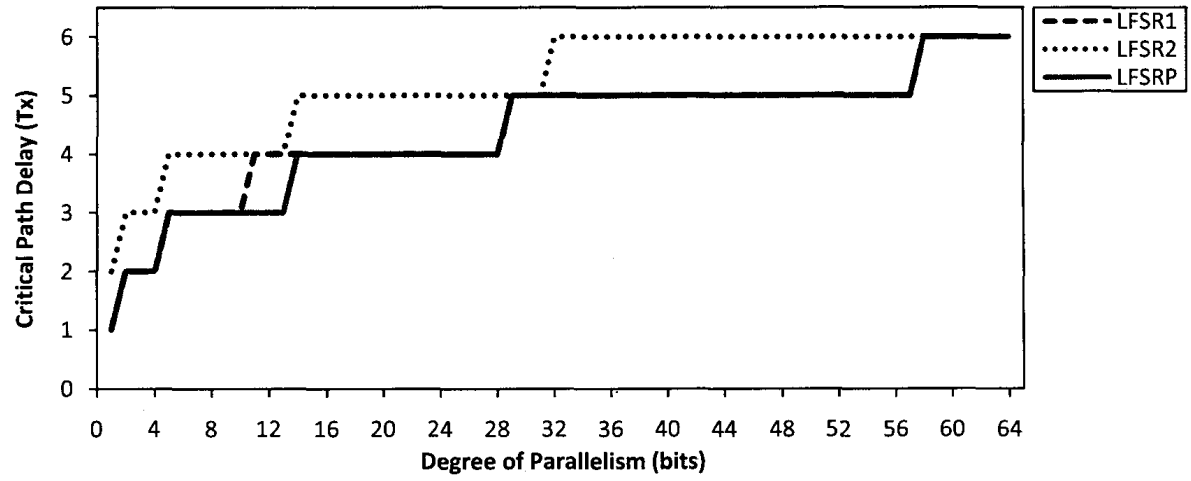
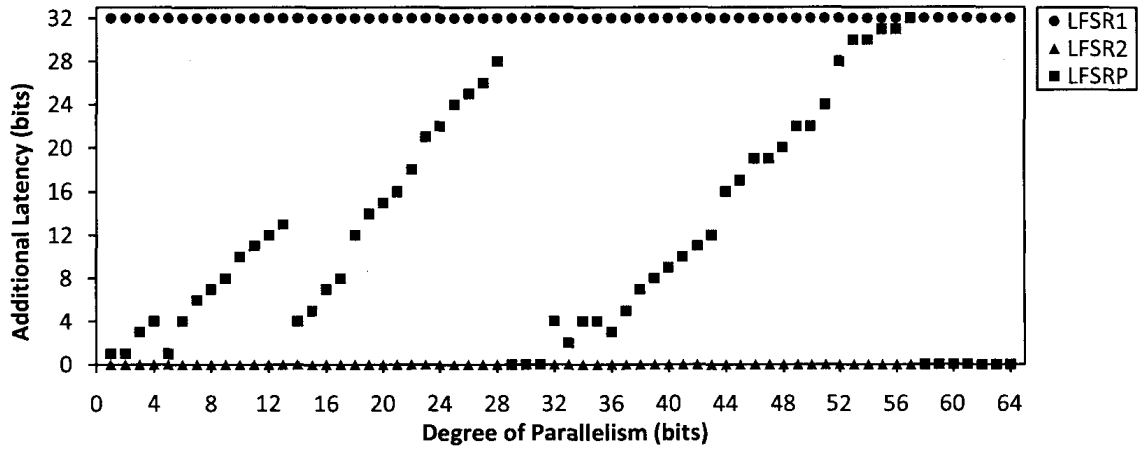


Figure 5.5: Plot of CPD versus p for some useful degrees of parallelism using CRC-32.



(a)



(b)

Figure 5.6: Plots of LFSR1, LFSR2, and LFSRp using CRC-32: (a) CPD versus degree of parallelism, (b) p versus degree of parallelism.

Next, using CRC-32, in Figures 5.6a and 5.6b, we show a plot of the Δ and p versus l for $1 \leq l \leq 2m$, respectively using the LFSR1, LFSR2, and LFSRp Architectures with the optimum p for $0 \leq p \leq m$. We note that for CRC-32 and the other frequently referenced generator polynomials, in terms of Δ, LFSRp is as good or better than LFSR1 for $1 \leq l \leq 2m$. Also, the latency of LFSRp is less than or equal to that of LFSR1. Thus, from our earlier discussion, one can conclude that LFSRp matches/outperforms LFSR1 in terms of time complexity, and it matches/outperforms LFSR2 as well.

Now, we provide the optimum p values for $0 \leq p \leq m$ with degrees of parallelism that are multiples of the generator polynomial degree in Table 5.4, and the corresponding Δ comparison is shown in Table 5.5. We first observe that there are only three instances where LFSR1 matches the performance of LFSRp, namely when

Table 5.4: LFSRp Architecture optimum p for $0 \leq p \leq m$ for frequently referenced generator polynomials with useful degrees of parallelism.

$G(x) \setminus l$	1	$\frac{1}{4}m$	$\frac{1}{2}m$	m	$2m$
CRC-12	2	4	6	7	0
CRC-16	2	2	2	16	14
CCITT-16	1	15	12	0	0
CRC16†	1	4	8	16	0
CCITT-16†	1	16	13	0	12
CRC-32	1	7	7	4	0

Table 5.5: LFSR hardware architecture critical path delay comparison for frequently referenced generator polynomials with useful degrees of parallelism.

l	1			$\frac{1}{4}m$			$\frac{1}{2}m$			m			$2m$		
$G(x) \setminus$ LFSR	1	2	p	1	2	p	1	2	p	1	2	p	1	2	p
CRC-12	1	2	1	2	3	2	3	4	3	4	5	4	5	5	5
CRC-16	1	2	1	3	4	3	4	5	4	4	5	4	5	6	5
CCITT-16	1	2	1	1	2	1	2	3	2	4	4	4	5	5	5
CRC-16†	1	2	1	3	3	2	4	4	3	4	5	4	5	5	5
CCITT-16†	1	2	1	1	2	1	2	3	2	4	4	4	4	5	4
CRC-32	1	2	1	3	4	3	4	5	4	5	6	5	6	6	6

$p = m$. Also, there are six instances where LFSR2 matches the performance of LFSRp, namely when $p = 0$. However, this result is not significant, the difference in terms of the number of cycles between LFSR1 and LFSR2 is rather small, and there are two cases in Table 5.5 where the Δ of LFSRp is less than that of LFSR1. Therefore, in most cases, LFSRp marginally improves upon LFSR1 by reducing the latency from $\lceil \frac{k+m}{l} \rceil$ to $\lceil \frac{k+p}{l} \rceil$ cycles.

Comparison: State-Space Transformed Architecture

Here, we present the results contained in our conference paper [53]. From the state-space transformed coupling matrices (3.27) obtained using the trivial and optimum $\mathbf{b}^*_{m\times 1}$ vectors, the number of XOR gates, FFs, and PSs are calculated. As noted in [24], the XOR gate count of an implementation can be obtained by summing the number of 1s in the rows of the coupling matrices and subtracting 1 from each row subtotal, plus the m additional XOR gates required to perform the addition

$$\bar{\mathbf{x}}'[i+1] = \bar{\mathbf{A}}'_{m\times m} \cdot \bar{\mathbf{x}}'_{m\times 1}[i] + \bar{\mathbf{B}}'_{m\times l} \cdot \bar{\mathbf{u}}_{l\times 1}[i]. \tag{5.1}$$

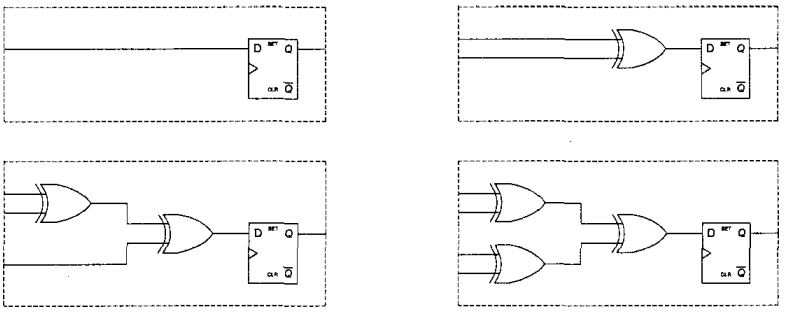


Figure 5.7: Pipelining blocks used in our implementations of the State-Space Transformed Architecture.

Recalling that $\bar{\mathbf{A}}'_{m\times m}$ is a companion matrix, the best CPD of the pipelined transformed system that one can obtain is $2{\cdot}T_X$. In other words, the maximum delay of the output wires from the $\bar{\mathbf{A}}'_{m\times m}\cdot\bar{\mathbf{x}}'_{m\times 1}[i]$ block is T_X, and another level of XOR gates is required to perform the addition (5.1) to form $\bar{\mathbf{x}}'_{m\times 1}[i+1]$. This results in wires with delays of $2\cdot T_X$. Thus, to pipeline the logic in the $\bar{\mathbf{B}}'_{m\times l}\cdot\mathbf{u}_{l\times 1}[i]$ and $\bar{\mathbf{C}}'_{m\times m}\cdot\bar{\mathbf{x}}'_{m\times 1}[i]$ blocks, one can use a combination of the pipelining blocks shown in Figure 5.7. We note that this pipelining approach is slightly different than the one reported in [24].

Tables 5.6a and 5.6b give the number of XOR gates, FFs, and PSs required to realize each of the transformed coupling matrices generated from the simple $\mathbf{b}^*_{m\times 1}$ compared to an optimum $\hat{\mathbf{b}}^*_{m\times 1}$, respectively. Also, Table 5.7 shows the total hardware complexity comparison. The optimum $\hat{\mathbf{b}}^*_{m\times 1}$ values that correspond to results in Tables 5.6 and 5.7 are marked with asterisks in Table 3.4. There is a case where multiple $\hat{\mathbf{b}}^*_{m\times 1}$ vectors result in optimum implementations when ranked by our metrics, and both vectors are marked. One observes significant reduction in the number of XOR gates and FFs when an optimum $\hat{\mathbf{b}}^*_{m\times 1}$ is used to construct the transformation matrix $\mathbf{T}_{m\times m}$ for all the frequently referenced generator polynomials. Finally, for all of these cases the number of PSs are equal for each generator polynomial.

Also, experimenting with other input sizes, we found that for small degrees of parallelism with the frequently referenced generator polynomials, it is possible to obtain transformed systems with $\bar{\mathbf{B}}'_{m\times r}$ matrices which have an entire row of zeros. In other words, the transformed system has states that are not coupled to the inputs. Those cases are advantageous because they reduce the number of retiming FFs and eliminate an XOR gate from the implementation of the addition in (5.1). An example of this instance is CRC-12 with $l=6$ and $\mathbf{b}^*_{12\times 1}=[0\text{x}255]^T$, where the states $\bar{x}_1[i]$ and $\bar{x}_{10}[i]$ are not coupled to any input. However, since the columns of $\mathbf{T}_{m\times m}$ are linearly independent and $\bar{\mathbf{C}}_{m\times m}=\mathbf{I}_{m\times m}$, all of the outputs are always coupled to a state. In other words, it is not possible to have an entire row of zeros in $\bar{\mathbf{C}}'_{m\times m}$.

Table 5.6: State-Space Transformed Architecture coupling matrix logic hardware requirements for frequently referenced generator polynomials when $l = m$ with: (a) simple $\mathbf{b}^*_{m\times 1} = \begin{bmatrix} 1 & 0 & 0 & \cdots & 0 \end{bmatrix}^T$, (b) optimum $\mathbf{b}^*_{m\times 1} = \hat{\mathbf{b}}^*_{m\times 1}$.

(a)

$l = m$	$\bar{\mathbf{B}}'_{m\times l}$			$\bar{\mathbf{A}}'_{m\times m}$			$\bar{\mathbf{C}}'_{m\times m}$		
$G(x)$	XOR	FF	PS	XOR	FF	PS	XOR	FF	PS
CRC-12	46	40	2	8	12	1	46	39	2
CRC-16	76	53	2	2	16	1	92	62	2
CCITT-16	88	68	2	2	16	1	100	68	2
CRC-16†	86	57	2	2	16	1	114	66	2
CCITT-16†	102	70	2	2	16	1	96	69	2
CRC-32	466	281	3	13	32	1	456	249	3

(b)

$l = m$	$\bar{\mathbf{B}}'_{m\times l}$			$\bar{\mathbf{A}}'_{m\times m}$			$\bar{\mathbf{C}}'_{m\times m}$		
$G(x)$	XOR	FF	PS	XOR	FF	PS	XOR	FF	PS
CRC-12	42	41	2	8	12	1	34	31	2
CRC-16	64	53	2	2	16	1	74	56	2
CCITT-16	90	65	2	2	16	1	86	58	2
CRC-16†	64	52	2	2	16	1	76	57	2
CCITT-16†	90	65	2	2	16	1	86	58	2
CRC-32	404	214	3	13	32	1	416	226	3

Table 5.7: State-Space Transformed Architecture hardware comparison for frequently referenced generator polynomials when $l = m$.

$l = m$	$\mathbf{b}^*_{n\times 1}$				$\hat{\mathbf{b}}^*_{n\times 1}$			
$G(x)$	1	XOR	FF	PS	1	XOR	FF	PS
CRC-12	136	112	91	5	120	96	84	5
CRC-16	218	186	131	5	188	156	125	5
CCITT-16	238	206	152	5	226	194	139	5
CRC-16†	250	218	139	5	190	158	125	5
CCITT-16†	248	216	155	5	226	194	139	5
CRC-32	1031	967	562	7	929	865	472	7

Table 5.8: Two-Step Architecture hardware comparisons for frequently referenced generator polynomials when $l = 8$.

Stage	First			Second			Total		
$G(x)$	XOR	FF	PS	XOR	FF	PS	XOR	FF	PS
CRC-12	32	56	1	182	213	5	214	269	6
CRC-16	48	76	1	290	347	6	338	423	7
CCITT-16	32	48	1	218	260	5	250	308	6
CRC-16†	48	76	1	558	661	6	606	687	7
CCITT-16†	32	48	1	192	216	4	224	264	5
CRC-32	56	123	1	1353	1444	6	1409	1567	7

Comparison: Two-Step Architecture

The hardware requirements of the Two-Step Architecture determined by simulation are listed in Table 5.8. Observing the results, one notices the large amount of hardware required to achieve a parallel implementation with T_X delay. Even though the theoretical CPD is less than that of the State-Space Transformed Architecture, the degree of parallelism is smaller and implementation of architectures this large are expected to have poor performance due to the wiring complexity. For these reasons, we felt that it was not necessary to implement this architecture in VHDL and it is excluded from our hardware implementations.

5.2.2 Implementations

Here, we present the data that was gathered through our hardware implementations. We opted to complete all of our implementations using VHDL and performed deployments on both ASIC and FPGA platforms. The State-Space Transformed Architecture implementations were completed on ASIC, while we compared the non-pipelined architectures on FPGA. Some examples of very-large-scale-integration (VLSI) implementations of the CRC computation found in the literature include [71], [72], [73], and [74]. Two examples of published works of FPGA implementations include [47], and [75].

We note that excluding the Cascade and Look-Ahead Architectures, all the other architectures in this thesis require XOR trees for their implementations. Describing these XOR trees in VHDL for all the implementations by hand would be a difficult, error prone, and tiresome task. Even more challenging is the pipelining of the State-Space Transformed Architecture. Therefore, we wrote C++ software that generates complete VHDL files, which describe an architecture for a given generator polynomial

and degree of parallelism. The generated VHDL files with XOR trees tended to be quite large $\sim$ 60 KB, and demonstrate the advantage of the Cascade [23] and Look-Ahead [25] Architectures, both of which are easily expressed with a hardware description language (HDL).

Proper care is taken to ensure that all of the inputs and outputs of an architecture are clocked. All the non-pipelined architectures have no logic between their CRC registers and outputs, therefore the output wires are clocked by default. However, all the architectures require their input wires to be clocked, which increases the hardware cost by l FFs. Finally, we note that the logic in the output coupling matrix of the State-Space Transformed Architecture was pipelined in such a manner that its output wires were clocked (see Figure 3.14).

ASIC: State-Space Transformed Architecture

To investigate how an optimum $\mathbf{b}^*_{m\times 1}$ vector affects the physical characteristics of the State-Space Transform Architecture, ASIC implementations are performed. The generated VHDL files are deployed on 0.18μ complementary metal-oxide semiconductor (CMOS) ASIC technology using the Synopsys® Design Analyzer®. The optimization effort is set to medium with a target period of 5.0 ns, and the area (μm^2) and timing (ns) are obtained for each of the designs.

The results of our ASIC experiments are summarized in Table 5.9. These results verify the expected reduction in area and also demonstrate improvements in timing. The improvements in timing could be attributed to the reduction in area, which reduces the wiring complexity and shortens the length of global wires in the design.

Table 5.9: State-Space Transformed Architecture ASIC implementation results for frequently referenced polynomials when $l = m$.

$l = m$	$\mathbf{b}^*_{n\times 1}$		$\hat{\mathbf{b}}^*_{n\times 1}$	
$G(x)$	area (μm^2)	delay (ns)	area (μm^2)	delay (ns)
CRC-12	11107	2.79	10156	2.79
CRC-16	16811	2.93	15299	2.72
CCITT-16	17905	2.73	17458	2.64
CRC-16†	18751	2.87	15445	2.71
CCITT-16†	18616	2.73	17470	2.49
CRC-32	72917	4.23	66469	3.20

Table 5.10: Non-pipelined architecture FPGA implementation results for frequently referenced polynomials when $l = m$.

$G(x)$	CRC-12		CRC-16		CCITT-16	
Arch	# slices	delay (ns)	# slices	delay (ns)	# slices	delay (ns)
[22]	11	1.756	17	2.072	20	1.816
[40]	24	1.280	15	1.619	18	1.315
[23]	12	1.706	18	1.752	14	1.868
[25]	10	1.543	15	1.865	16	1.971
ΛΓ	9	1.684	15	1.743	16	1.815

$G(x)$	CRC-16†		CCITT-16†		CRC-32	
Arch	# slices	delay (ns)	# slices	delay (ns)	# slices	delay (ns)
[22]	29	2.143	16	1.617	73	2.940
[40]	21	1.905	15	1.678	74	2.605
[23]	25	1.946	19	1.818	97	2.888
[25]	26	2.148	15	1.942	76	2.756
ΛΓ	30	2.053	17	2.002	70	3.446

FPGA: Non-Pipelined Architectures

For this study, we choose to perform FPGA experiments for the non-pipelined architectures. The generated VHDL files are deployed on a Virtex®-5 FPGA. The target device is xc5vlx30-3ff324 using Xilinx® USE Design Suite 10.1.02 - Web PACK with a design goal of balanced. The speed value is set to -3 and the synthesis tool is CST.

Table 5.10 shows the results of our FPGA experiments. One observes a weak correlation between the theoretical results in Table 5.1, and there is no clear cut winner amongst them. These results suggest that if designer wishes to realize the CRC computation on an FPGA, then they should select an architecture that is well understood and easily implemented. Because these experiments demonstrate that there is little to choose between them in terms of the performance on FPGA.

5.3 Software Experiments

In this section, we present the results of our software experiments. Before beginning our discussion, we remind the reader to recall the assumptions made in Chapter 3, which concern the properties of the datapath. The assumptions simply our analysis and allow the reader to better predict the performance of these algorithms on their datapath.

To properly interpret the simulation data, the most important assumption made is that all operations in these algorithms take equal processing time. We justify this assumption by noting that all the algorithms use similar operations and the memory accesses tend not to be slow. This is because the maximum LUT has 1024 words. Therefore, all the LUTs can fit inside the cache of a typical modern processor [7]. Moreover, for the implementation data, during the early stages of this study, we attempted to generalize the existing algorithms and investigate their performance by varying the degree of parallelism. We determined that each algorithm has an optimum degree of parallelism, and implemented the ones found to be optimal for our datapath. For instance, comparing the performance of CRCT(8) to CRCT(16), it is clear that CRCT(16) requires half of the number of LUT accesses, but the implementation performance is actually worse, due to slower LUT access times [61]. Finally, we note the program sizes and code complexities of all these algorithms are comparable when they are implemented using C++, and we expect similar results for assembly language implementations. For these reasons, we omit the code size from our memory comparison.

5.3.1 Simulations

The software simulation results consist of memory and time requirements of a given algorithm. The memory requirements are well defined for each algorithm regardless of the platform and they are reported in terms of the number of memory words required to store the look-up tables. We note that different packing schemes are possible for storing LUTs in memory. In our experiments, we have chosen to store one LUT entry per memory word, regardless of its length. In other words, on our 32-bit datapath, it is possible to store multiple LUT entries for degrees 12 and 16 generator polynomials in one memory word, and consequently reduce the LUT size. However, the trade-off is that the implementation becomes more complex and the execution time is increased.

For the theoretical results, the time requirement is defined to be the number of instructions necessary to process one word of the message, and can be identified from the algorithms presented throughout the thesis. Note that, we do not consider the message length in this case, therefore the values reported are normalized. Now, consider how one would obtain the execution time for CRCB(1) with CRC-32. Referring back to its implementation in Algorithm 3.1, we count the number of operations as follows: 1×(word fetch, loop initialization, pointer increment) + 32×(loop comparison, loop increment) + 32×(condition test) + 16×(condition true) + 16×(condition

false) + 32×(shift operation) = $1\times3+32\times2+32\times3+16\times2+16\times1+32\times1 = 243$ operations. Here, we assume an equal branch probability, and the results for the other existing algorithms can be obtained similarly. We note that, none of the determined time requirements for the existing algorithms depend on the degree or coefficients of the generator polynomial.

For CRC$\Lambda\Gamma$(32), we obtain the closed form operation count as,

$$\begin{aligned}\text{time}\,(\text{CRC}\Lambda\Gamma\,(32)) &= 6+|\Lambda|\times4+|\Gamma|\times4+\{1\}\\ &= 6+(|\Lambda|+|\Gamma|)\times4+\{1\},\end{aligned}$$

where the 1 in braces is required when $m < w$. The memory requirements of this approach is found to be

$$\text{memory}\,(\text{CRC}\Lambda\Gamma\,(32)) = |\Lambda|+|\Gamma|\,.$$

We note that unlike the existing algorithms, the performance of CRC$\Lambda\Gamma(l)$ depends on the sizes of the sets Λ and Γ, which are fixed for a given generator polynomial and degree of parallelism.

The determined memory and timing values for the existing algorithms and the proposed CRC$\Lambda\Gamma$(32) using the frequently referenced generator polynomials are given in Tables 5.11a and 5.11b, respectively. Examining the results, one observes that CRC$\Lambda\Gamma$(32) is expected to outperform CRCR(32), with equal or less memory. We note that there is a strong correlation between the order of LUT size and timing performance in these results. That is, algorithms that have larger memory requirements generally deliver better speed performance until caching becomes and issue.

Table 5.11: Theoretical software algorithm comparison, memory in words and loop instruction counts: (a) existing algorithms, (b) CRC$\Lambda\Gamma$(32).

(a)

Algorithm	CRCB(1)	CRCR(32)	CRCT(8)	CRCS4(32)
Memory	1	32	256	1024
Timing	274	180	34	11

(b)

CRC$\Lambda\Gamma$	CRC-12	CRC-16	CCITT-16	CRC-16†	CCITT-16†	CRC-32
Memory	28	32	14	19	16	27
Timing	119	135	63	83	71	114

5.3.2 Implementations

Here, we outline our software benchmarking system and then provide our measured implementation results.

Benchmarking System

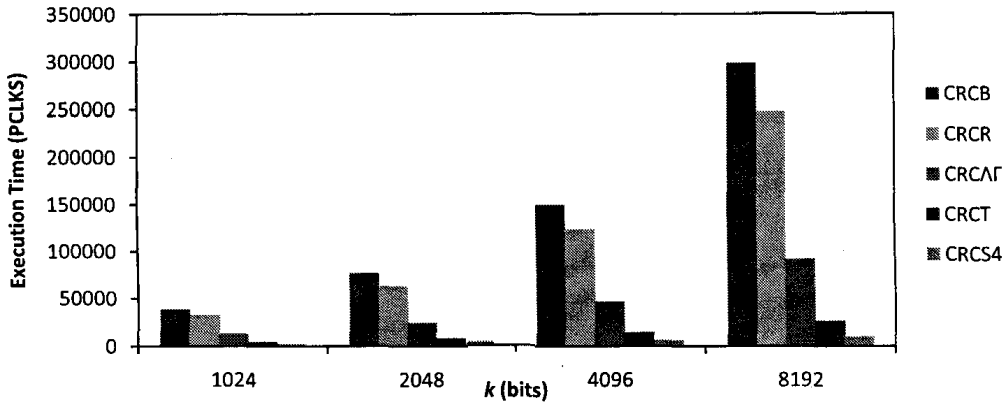
The software algorithms are benchmarked using a desktop PC with 2 GB RAM, Intel® 0x86 32-bit Pentium® IV processor operating at 1.5 GHz, running Microsoft® Windows® XP Professional, version 2002, and Service Pack 3. The implementations were completed using C++ and compiled under Microsoft® Visual Studio® 2005, version 8.0.50727.762, as a WIN32 console application with no optimization options selected. Finally, we note that all the implementations closely resemble the C++ code snippets presented throughout this thesis.

To obtain the timing results of the algorithms, we used a high-resolution counter class [76]. The counter class wraps the Pentium® specific time stamp counter, enabled one to measure the number of processor clock ticks (PCLKS) between two execution points in a program. When beginning the benchmarking experiments, we use the program EndItAll version 2.0.0.0 [77] to terminate many of the Windows® running processes, with the goal of reducing the amount of resource contention. We note that we cannot measure the percentage of CPU time that is consumed by other processes while our algorithms are being benchmarked. To combat this problem, we process relatively short messages and discard the fastest and slowest measured run times.

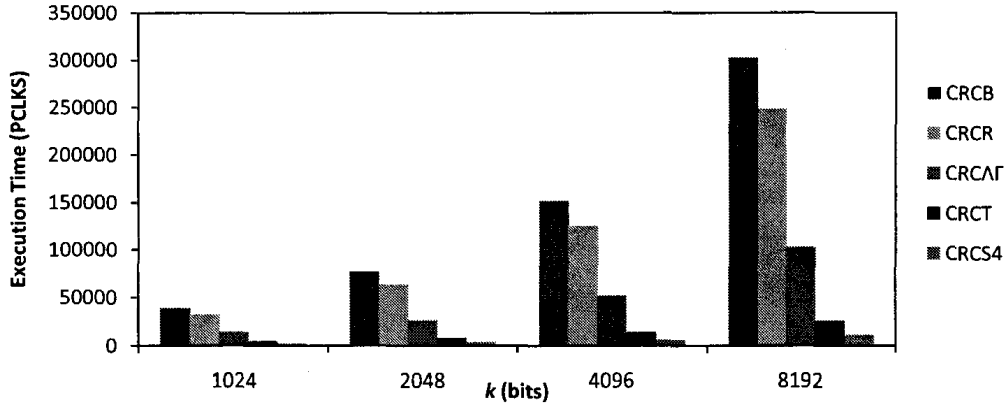
Results

We examine the impact that the message length, k, has on the speed of the algorithms for generator polynomials with different degrees. Figures 5.8a, 5.8b, and 5.8c, show the timing results for the generator polynomials CRC-12, CRC-16, and CRC-32, respectively, for message lengths $k = 1024$, 2048, 4096, and 8192 bits. In these figures, each point on a timing plot represents the average number of PCLKS for 12 runs, with each of the fastest and slowest measured times discarded. After the discard, 87.5% of the fastest and slowest remaining times are within 2.0 standard deviations of the mean time. Furthermore, 80% of the standard deviations are less than 1% of their respective means, with an overall average of 0.56%. These statistics demonstrate an acceptable amount of variance between run times. We note that we have selected CRC-16, which is predicted to have the worst case CRC$\Lambda\Gamma$(32) performance amongst

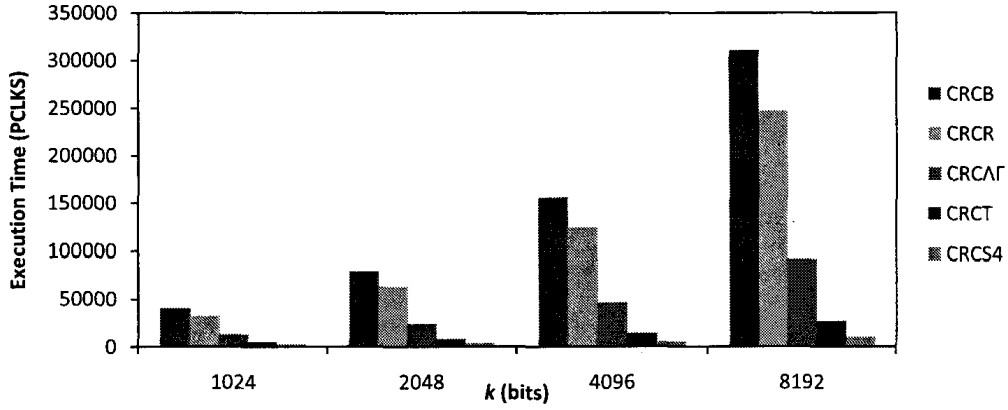
the degree 16 generator polynomials. These plots verify the simulation performance results presented in Tables 5.11a and 5.11b. Finally, we conclude that CRCΛΓ(32) outperforms the existing low-memory CRCR(32) on our system.



(a)



(b)



(c)

Figure 5.8: Algorithm timing plots for: (a) CRC-12, (b) CRC-16, (c) CRC-32.

5.4 Summary

In this chapter, we presented our simulation and implementation results for the hardware architectures and software algorithms contained in the thesis. The hardware simulation data was gathered from C++ software that was written by the author over the course of the study. To obtain the hardware implementation data, the simulation software was extended to generate full VHDL files, and some of these files were deployed on the ASIC and FPGA platforms. The software algorithms were theoretically analyzed by counting both the number of operations required to process a message word and the memory required to represent their LUTs. Actual algorithm run times were measured on a desktop PC with a high-resolution counter class.

For the hardware results, we compared our proposed Lambda Gamma Architecture against the non-pipelined architectures, and found for CRC-32 it achieves the best area performance of all the approaches, also in terms of time complexity it matches/outperforms the systematic approaches for all the frequently referenced generator polynomials. The study of the optimum degrees of parallelism for the parallel LFSR2 Architecture determined that the time-area product tracks the CPD delay plot, therefore one achieves diminishing time-area returns as the degree of parallelism is increased. The optimum p values were obtained for the LFSRp Architecture. However, we note that the improvements of this approach over LFSR1 are minor. Our pipelining strategy for the State-Space Transformed Architecture was outlined, and we found for optimum transformations improvements are observed in the number of XOR gates and FFs. The area complexity of the Two-Step Architecture was investigated, and we found it to be quite large for small degrees of parallelism. The ASIC implementations of the State-Space Transformed Architecture verified the expected reduction in area and showed improvements in timing. The FPGA implementations of the non-pipelined architectures demonstrated that there is little difference amongst them and suggest that a designer should implement a systematic architecture.

For the theoretical software results, we investigated the performance of the proposed CRC$\Lambda\Gamma$(32) algorithm for the frequently referenced generator polynomials. CRC$\Lambda\Gamma$(32) was shown to outperform CRCR(32) in terms of time complexity and requires equal or less LUT memory space. We neglected to study the program sizes, since they are all comparable in C++. Finally, benchmarking experiments were performed on a desktop PC. These experiments validated the theoretical timing values and we concluded that CRC$\Lambda\Gamma$ outperforms CRCR when the degree of the generator polynomial is less than or equal to the bus width.

Chapter 6

Contributions and Future Work

6.1 Preview

THE Cyclic Redundancy Check, is an EDC first proposed by Peterson and Brown in 1961 [1]. The two major areas of ongoing research concerning CRCs consists of developing new approaches to realize the computation and further investigations into its error detection properties. This thesis has focused exclusively on studying approaches to realize the CRC computation as an architecture or algorithm, in hardware or software, respectively.

In this final chapter, we review the contributions contained in this thesis and present potential directions for future work. As this thesis has shown, most of the realizations of the CRC computation are proposed from a presented formulation. Formulations are typically expressed using binary polynomials or a state-space representations. The existing formulations are derived by considering the computation of the CRC equation (2.2), reproduced here for convenience,

$$(x^m \cdot U(x)) \bmod G(x),$$

as either the reduction of the augmented $(x^m \cdot U(x))$ or extended $(\tilde{U}(x) = x^m \cdot U(x))$ message by the generator polynomial, called LFSR2 and LFSR1 formulations, respectively. All software algorithms are based on LFSR2 formulations, however realizing an LFSR1 formulation in hardware can result in an architecture that has lower a CPD, when compared to its LFSR2 counterpart. It is noted that a small increase in latency is incurred for architectures based on LFSR1 formulations.

6.1.1 Organization

The remainder of this chapter is organized as follows. In Section 6.2, the contributions contained in this thesis are reviewed. Our contributions are summarized in a chapter-by-chapter format for easy cross-referencing. In Section 6.3, some ideas for potential future studies are given and the section is divided into hardware and software subsections.

6.2 Contributions

This thesis has examined a wide range of aspects surrounding the CRC computation. We have generalized many of the existing formulations and derived some new ones. These new formulations have been utilized to propose novel CRC computation approaches that are realized as hardware architectures and software algorithms. These proposed approaches offer some improvement over the existing architectures and algorithms in terms of either computation time or area/memory requirements. Of the newly proposed schemes, we feel the most significant contribution is the software algorithm CRC$\Lambda\Gamma$ derived from Lambda Gamma formulation. This high-performance algorithm achieves good timing results with low memory usage and is easily implemented on most systems. In the following subsections, we provide a chapter-by-chapter summary of all the contributions contained in this thesis.

6.2.1 Chapter 3

In Chapter 3, many of the existing parallel CRC formulations were developed from first principles, generalized, and some minor contributions were presented.

We began by developing a rigorous binary polynomial derivation for partitioning the message into blocks. Afterward, we derived generalized binary polynomial and state-space formulations of the parallel CRC computation using both LFSR2 and LFSR1 approaches. Unlike the previous approaches, our formulations were constructed for all three cases of different degree of parallelism and generator polynomial degree. These generalized formulations form the foundation which allow us to derive all the approaches contained in the thesis.

The existing hardware architectures were examined next, each was derived and some minor extensions were performed. For the Two-Step Architecture, we presented a search methodology to find multiple polynomials and we found them for the frequently referenced generator polynomials. Also, we noted that it is possible to pipeline

the second stage of the architecture and obtain a CPD of $1 \cdot T_X$. For the Cascade Architecture, we showed that the LFSR1 formulation could be applied and result in an architecture with less CPD. For the State-Space Transformed Architecture, we performed an exhaustive search to obtain the optimum transformation, which results in a system with reduced hardware complexity. Finally, this chapter examined the existing software algorithms. We presented a straightforward derivation for the Slicing Algorithms which showed that this approach is a combination of the Table Look-up and Reduced Table Look-up Algorithms.

6.2.2 Chapter 4

In Chapter 4, some novel computation approaches were presented. The first major contribution of this thesis concerned the development of a binary polynomial to matrix-based formulation. Afterward, we investigated the detailed design of the parallel LFSR2 Architecture when the degree of parallelism is greater than the generator polynomial degree, and presented some optimizations specific to that case. Next, the matrix-based formulation was extended and a novel matrix decomposition was presented. We consider this decomposition, we call the Lambda Gamma decomposition, to be the central contribution of our study. The decomposition gave rise to a novel software algorithm and hardware architecture. Next, we generalized the CRC formulation introducing the parameter p. This formulation allows one to derive both the existing LFSR formulations from a common starting point. Finally, we investigated the Message Splitting Architecture and found that it may have promise for some cases.

6.2.3 Chapter 5

In Chapter 5, the simulation and implementation results of the various architectures and algorithms were gathered. Hardware experiments were performed first, followed by software experiments.

Our first hardware simulation compared all the non-pipelined architectures. The proposed approach outperformed the systematic approaches in terms of CPD, however at the cost of additional hardware. Next, we presented a study which obtains the optimum degrees of parallelism in terms of critical path delay for the LFSR2 Architecture. We also performed searches for the optimum p values for the LFSRp architecture that reduce the computation time. LFSRp was shown to match/outperform both LFSR2 and LFSR1, however most of the results offered only small improvements in the overall

computation time. The results of the state-space transformation simulations showed modest reductions in the number of XOR gates and FFs, when the optimum transform was selected over the trivial one. We found for the Two-Step Architecture, obtaining a circuit with $1 \cdot T_X$ delay requires a large amount of hardware and is likely not practical for implementations. Next, ASIC and FPGA implementations were performed for some selected architectures. We chose to verify our state-space transformation result in ASIC, and the implementations demonstrated improvement in both area and timing. FPGA implementations were performed for the non-pipelined architectures, and we concluded that it is best to implement a systematic architecture on FPGA, since there is little difference between the performance of these approaches.

For the software algorithms, we first conducted a theoretical analysis to obtain the expected performance. We determined that our proposed CRC$\Lambda\Gamma$ requires less or equal LUT memory compared to the existing low-memory CRCR algorithm, and offers better computation times. Finally, we performed software benchmarking experiments for all the algorithms and found that our measured results validated our predicted ones. We concluded that the Lambda Gamma Algorithm outperforms the Reduced Table Look-up Algorithm when the degree of the generator polynomial is less than or equal to the bus width, which is the case in many systems.

6.3 Future Work

In this final section of the thesis, we present possible future work and some open research questions surrounding the realization of the CRC computation in hardware and software.

6.3.1 Hardware Architectures

After completing this study, it is clear to us that the best CPD that can be achieved by retiming techniques for a CRC computation architecture is bounded by the complexity in its feedback loop. As the degree of parallelism is increased, inherently, the feedback loop complexity increases. Consequently, increasing the degree of parallelism does not always result in timing improvements. An interesting challenge would be to derive a systematic approach to obtain an architecture with CPD $1 \cdot T_X$, which does not rely on any assumptions that limit its ability to be deployed.

Currently, the fastest retimed architecture is the Two-Step Architecture, and it has a best-case CPD of $1 \cdot T_X$. However, one must find appropriate multiple polynomials,

which can be difficult. We suggested methods to shorten the search times from a naive method of testing every polynomial, but they are not suitable for larger degrees of parallelism. So we pose the following question: in addition to the Two-Step Architecture, are there any another approaches to obtain parallel CRC hardware realization of the CRC computation with $1 \cdot T_X$ delay?

As mentioned in Chapter 3, there is an approach based on manipulating the serial LFSR2 Architecture by unfolding, pipelining, and retiming it to obtain a fast parallel CRC architecture. It would be interesting to try to develop a mathematical approach to describe these manipulations. By taking this approach, one may have more flexibility and possibly be able to improve on the results in that paper. We note that other than [24], this is the only paper with an architecture that has a CPD that is less than CPD in the feedback loop of its primitive counterpart.

Generally, there has been little work attempted to the application of CRC onto higher order fields, i.e., $GF(q)$. Investigations into the extension of the Lambda Gamma decomposition onto $GF(q)$ could be performed, as the decomposition should also be valid in those fields.

Finally, simulations could be performed to determine if it is possible to outperform the parallel LFSR Architectures by using the Message Splitting Architecture. However, we note that this approach is also bounded by delays in its feedback loop.

6.3.2 Software Algorithms

This study verified that the speed performance of a given software algorithms is strongly influenced by the properties of datapath that it will be deployed on. Factors such as cache size, instruction set, memory access times, etc., will dictate the performance of an algorithm. With the recent rise in popularity of multicore processors, a future study could be undertaken on the performance of the existing algorithms when deployed on these systems. That is, investigations into the influence of different shared memory schemes for representing the message and the LUTs.

Finally, the desktop benchmarking experiments that were performed in this thesis could be followed up on a microcontroller that has build-in benchmarking utilities. We noted that the instruction counts provided in Chapter 5 were based on C++ implementations of the algorithms, consequently they will be compiled differently on different systems. Even though the variance was low between measured run times on our system, a more accurate result would likely be observed through assembly language implementations on a microcontroller.

Appendix A

CRC-32 Hardware Architecture Equations

IN this appendix, the hardware architecture equations for some implementations of the CRC computation using the generator polynomial CRC-32, $G(x) = 1 + x + x^2 + x^4 + x^5 + x^7 + x^8 + x^{10} + x^{11} + x^{12} + x^{16} + x^{22} + x^{23} + x^{26} + x^{32}$ [4] are presented. For the parallel LFSR2 and LFSR1, Lambda Gamma, State-Space Transformed, and Two-Step Architectures, we list the complete set of equations required for their implementation. The Cascade and Look-Ahead Architectures are based on the serial LFSR2 Architecture, therefore their implementation equations are trivial and are not provided.

Since we are discussing implementations, the iteration number in the formulation has no impact on the architecture. Therefore, we drop it from the equations. However, since all these formulations are recursive, we mark the next terms using primes. In other words, the primes denote terms which input to a storage element.

We note that, all of the equations in this appendix were obtained using modified versions of the LUT generation algorithms presented in this thesis. Therefore, the content contained in this appendix can be used by a hardware designer who wishes to realize the CRC-32 computation as an architecture and/or verify the correctness their equation generation code.

The remainder of this appendix is organized as follows. In Section A.1, the equations for the parallel LFSR2 and LFSR1 Architectures are presented. In Section A.2, the equations for the Lambda Gamma Architecture are presented. In Section A.3, the equations for the State-Space Transformed Architecture are presented. In Section A.4, the equations for the Two-Step Architecture are presented.

A.1 Parallel LFSR Architectures

An illustration of the parallel LFSR2 and LFSR1 Architectures are shown in Figures 3.5 and 3.6, respectively. Here, the implementation equations are provided for $l = 32$. We note that for LFSR2, $t_i = s_i + u_i$ see (3.5). Figures A.1 and A.2, display the logic equations for the parallel LFSR2 and LFSR1 Architectures, respectively.

$$
\begin{aligned}
s'_0 &= t_0 + t_6 + t_9 + t_{10} + t_{12} + t_{16} + t_{24} + t_{25} + t_{26} + t_{28} + t_{29} + t_{30} + t_{31} \\
s'_1 &= t_0 + t_1 + t_6 + t_7 + t_9 + t_{11} + t_{12} + t_{13} + t_{16} + t_{17} + t_{24} + t_{27} + t_{28} \\
s'_2 &= t_0 + t_1 + t_2 + t_6 + t_7 + t_8 + t_9 + t_{13} + t_{14} + t_{16} + t_{17} + t_{18} + t_{24} + t_{26} + t_{30} + t_{31} \\
s'_3 &= t_1 + t_2 + t_3 + t_7 + t_8 + t_9 + t_{10} + t_{14} + t_{15} + t_{17} + t_{18} + t_{19} + t_{25} + t_{27} + t_{31} \\
s'_4 &= t_0 + t_2 + t_3 + t_4 + t_6 + t_8 + t_{11} + t_{12} + t_{15} + t_{18} + t_{19} + t_{20} + t_{24} + t_{25} + t_{29} + t_{30} + t_{31} \\
s'_5 &= t_0 + t_1 + t_3 + t_4 + t_5 + t_6 + t_7 + t_{10} + t_{13} + t_{19} + t_{21} + t_{24} + t_{28} + t_{29} \\
s'_6 &= t_1 + t_2 + t_4 + t_5 + t_6 + t_7 + t_8 + t_{11} + t_{14} + t_{20} + t_{21} + t_{22} + t_{25} + t_{29} + t_{30} \\
s'_7 &= t_0 + t_2 + t_3 + t_5 + t_7 + t_8 + t_{10} + t_{15} + t_{16} + t_{21} + t_{22} + t_{23} + t_{24} + t_{25} + t_{28} + t_{29}
\end{aligned}
$$

$$
\begin{aligned}
s'_8 &= t_0 + t_1 + t_3 + t_4 + t_8 + t_{10} + t_{11} + t_{12} + t_{17} + t_{22} + t_{23} + t_{28} + t_{31} \\
s'_9 &= t_1 + t_2 + t_4 + t_5 + t_9 + t_{11} + t_{12} + t_{13} + t_{18} + t_{23} + t_{24} + t_{29} \\
s'_{10} &= t_0 + t_2 + t_3 + t_5 + t_9 + t_{13} + t_{14} + t_{16} + t_{19} + t_{26} + t_{28} + t_{29} + t_{31} \\
s'_{11} &= t_0 + t_1 + t_3 + t_4 + t_9 + t_{12} + t_{14} + t_{15} + t_{16} + t_{17} + t_{20} + t_{24} + t_{25} + t_{26} + t_{27} + t_{28} + t_{31} \\
s'_{12} &= t_0 + t_1 + t_2 + t_4 + t_5 + t_6 + t_9 + t_{12} + t_{13} + t_{15} + t_{17} + t_{18} + t_{21} + t_{24} + t_{27} + t_{30} + t_{31} \\
s'_{13} &= t_1 + t_2 + t_3 + t_5 + t_6 + t_7 + t_{10} + t_{13} + t_{14} + t_{16} + t_{18} + t_{19} + t_{22} + t_{25} + t_{28} + t_{31} \\
s'_{14} &= t_2 + t_3 + t_4 + t_6 + t_7 + t_8 + t_{11} + t_{14} + t_{15} + t_{17} + t_{19} + t_{20} + t_{23} + t_{26} + t_{29} \\
s'_{15} &= t_3 + t_4 + t_5 + t_7 + t_8 + t_9 + t_{12} + t_{15} + t_{16} + t_{18} + t_{20} + t_{21} + t_{24} + t_{27} + t_{30}
\end{aligned}
$$

$$
\begin{aligned}
s'_{16} &= t_0 + t_4 + t_5 + t_8 + t_{12} + t_{13} + t_{17} + t_{19} + t_{21} + t_{22} + t_{24} + t_{26} + t_{29} + t_{30} \\
s'_{17} &= t_1 + t_5 + t_6 + t_9 + t_{13} + t_{14} + t_{18} + t_{20} + t_{22} + t_{23} + t_{25} + t_{27} + t_{30} + t_{31} \\
s'_{18} &= t_2 + t_6 + t_7 + t_{10} + t_{14} + t_{15} + t_{19} + t_{21} + t_{23} + t_{24} + t_{26} + t_{28} + t_{31} \\
s'_{19} &= t_3 + t_7 + t_8 + t_{11} + t_{15} + t_{16} + t_{20} + t_{22} + t_{24} + t_{25} + t_{27} + t_{29} \\
s'_{20} &= t_4 + t_8 + t_9 + t_{12} + t_{16} + t_{17} + t_{21} + t_{23} + t_{25} + t_{26} + t_{28} + t_{30} \\
s'_{21} &= t_5 + t_9 + t_{10} + t_{13} + t_{17} + t_{18} + t_{22} + t_{24} + t_{26} + t_{27} + t_{29} + t_{31} \\
s'_{22} &= t_0 + t_9 + t_{11} + t_{12} + t_{14} + t_{16} + t_{18} + t_{19} + t_{23} + t_{24} + t_{26} + t_{27} + t_{29} + t_{31} \\
s'_{23} &= t_0 + t_1 + t_6 + t_9 + t_{13} + t_{15} + t_{16} + t_{17} + t_{19} + t_{20} + t_{26} + t_{27} + t_{29} + t_{31}
\end{aligned}
$$

$$
\begin{aligned}
s'_{24} &= t_1 + t_2 + t_7 + t_{10} + t_{14} + t_{16} + t_{17} + t_{18} + t_{20} + t_{21} + t_{27} + t_{28} + t_{30} \\
s'_{25} &= t_2 + t_3 + t_8 + t_{11} + t_{15} + t_{17} + t_{18} + t_{19} + t_{21} + t_{22} + t_{28} + t_{29} + t_{31} \\
s'_{26} &= t_0 + t_3 + t_4 + t_6 + t_{10} + t_{18} + t_{19} + t_{20} + t_{22} + t_{23} + t_{24} + t_{25} + t_{26} + t_{28} + t_{31} \\
s'_{27} &= t_1 + t_4 + t_5 + t_7 + t_{11} + t_{19} + t_{20} + t_{21} + t_{23} + t_{24} + t_{25} + t_{26} + t_{27} + t_{29} \\
s'_{28} &= t_2 + t_5 + t_6 + t_8 + t_{12} + t_{20} + t_{21} + t_{22} + t_{24} + t_{25} + t_{26} + t_{27} + t_{28} + t_{30} \\
s'_{29} &= t_3 + t_6 + t_7 + t_9 + t_{13} + t_{21} + t_{22} + t_{23} + t_{25} + t_{26} + t_{27} + t_{28} + t_{29} + t_{31} \\
s'_{30} &= t_4 + t_7 + t_8 + t_{10} + t_{14} + t_{22} + t_{23} + t_{24} + t_{26} + t_{27} + t_{28} + t_{29} + t_{30} \\
s'_{31} &= t_5 + t_8 + t_9 + t_{11} + t_{15} + t_{23} + t_{24} + t_{25} + t_{27} + t_{28} + t_{29} + t_{30} + t_{31}
\end{aligned}
$$

Figure A.1: Parallel LFSR2 Architecture $\mathbf{G}_{32\times 32}$ equations.

$$s'_0 = u_0 + s_0 + s_6 + s_9 + s_{10} + s_{12} + s_{16} + s_{24} + s_{25} + s_{26} + s_{28} + s_{29} + s_{30} + s_{31}$$
$$s'_1 = u_1 + s_0 + s_1 + s_6 + s_7 + s_9 + s_{11} + s_{12} + s_{13} + s_{16} + s_{17} + s_{24} + s_{27} + s_{28}$$
$$s'_2 = u_2 + s_0 + s_1 + s_2 + s_6 + s_7 + s_8 + s_9 + s_{13} + s_{14} + s_{16} + s_{17} + s_{18} + s_{24} + s_{26} + s_{30} + s_{31}$$
$$s'_3 = u_3 + s_1 + s_2 + s_3 + s_7 + s_8 + s_9 + s_{10} + s_{14} + s_{15} + s_{17} + s_{18} + s_{19} + s_{25} + s_{27} + s_{31}$$
$$s'_4 = u_4 + s_0 + s_2 + s_3 + s_4 + s_6 + s_8 + s_{11} + s_{12} + s_{15} + s_{18} + s_{19} + s_{20} + s_{24} + s_{25} + s_{29} + s_{30} + s_{31}$$
$$s'_5 = u_5 + s_0 + s_1 + s_3 + s_4 + s_5 + s_6 + s_7 + s_{10} + s_{13} + s_{19} + s_{21} + s_{24} + s_{28} + s_{29}$$
$$s'_6 = u_6 + s_1 + s_2 + s_4 + s_5 + s_6 + s_7 + s_8 + s_{11} + s_{14} + s_{20} + s_{21} + s_{22} + s_{25} + s_{29} + s_{30}$$
$$s'_7 = u_7 + s_0 + s_2 + s_3 + s_5 + s_7 + s_8 + s_{10} + s_{15} + s_{16} + s_{21} + s_{22} + s_{23} + s_{24} + s_{25} + s_{28} + s_{29}$$

$$s'_8 = u_8 + s_0 + s_1 + s_3 + s_4 + s_8 + s_{10} + s_{11} + s_{12} + s_{17} + s_{22} + s_{23} + s_{28} + s_{31}$$
$$s'_9 = u_9 + s_1 + s_2 + s_4 + s_5 + s_9 + s_{11} + s_{12} + s_{13} + s_{18} + s_{23} + s_{24} + s_{29}$$
$$s'_{10} = u_{10} + s_0 + s_2 + s_3 + s_5 + s_9 + s_{13} + s_{14} + s_{16} + s_{19} + s_{26} + s_{28} + s_{29} + s_{31}$$
$$s'_{11} = u_{11} + s_0 + s_1 + s_3 + s_4 + s_9 + s_{12} + s_{14} + s_{15} + s_{16} + s_{17} + s_{20} + s_{24} + s_{25} + s_{26} + s_{27} + s_{28} + s_{31}$$
$$s'_{12} = u_{12} + s_0 + s_1 + s_2 + s_4 + s_5 + s_6 + s_9 + s_{12} + s_{13} + s_{15} + s_{17} + s_{18} + s_{21} + s_{24} + s_{27} + s_{30} + s_{31}$$
$$s'_{13} = u_{13} + s_1 + s_2 + s_3 + s_5 + s_6 + s_7 + s_{10} + s_{13} + s_{14} + s_{16} + s_{18} + s_{19} + s_{22} + s_{25} + s_{28} + s_{31}$$
$$s'_{14} = u_{14} + s_2 + s_3 + s_4 + s_6 + s_7 + s_8 + s_{11} + s_{14} + s_{15} + s_{17} + s_{19} + s_{20} + s_{23} + s_{26} + s_{29}$$
$$s'_{15} = u_{15} + s_3 + s_4 + s_5 + s_7 + s_8 + s_9 + s_{12} + s_{15} + s_{16} + s_{18} + s_{20} + s_{21} + s_{24} + s_{27} + s_{30}$$

$$s'_{16} = u_{16} + s_0 + s_4 + s_5 + s_8 + s_{12} + s_{13} + s_{17} + s_{19} + s_{21} + s_{22} + s_{24} + s_{26} + s_{29} + s_{30}$$
$$s'_{17} = u_{17} + s_1 + s_5 + s_6 + s_9 + s_{13} + s_{14} + s_{18} + s_{20} + s_{22} + s_{23} + s_{25} + s_{27} + s_{30} + s_{31}$$
$$s'_{18} = u_{18} + s_2 + s_6 + s_7 + s_{10} + s_{14} + s_{15} + s_{19} + s_{21} + s_{23} + s_{24} + s_{26} + s_{28} + s_{31}$$
$$s'_{19} = u_{19} + s_3 + s_7 + s_8 + s_{11} + s_{15} + s_{16} + s_{20} + s_{22} + s_{24} + s_{25} + s_{27} + s_{29}$$
$$s'_{20} = u_{20} + s_4 + s_8 + s_9 + s_{12} + s_{16} + s_{17} + s_{21} + s_{23} + s_{25} + s_{26} + s_{28} + s_{30}$$
$$s'_{21} = u_{21} + s_5 + s_9 + s_{10} + s_{13} + s_{17} + s_{18} + s_{22} + s_{24} + s_{26} + s_{27} + s_{29} + s_{31}$$
$$s'_{22} = u_{22} + s_0 + s_9 + s_{11} + s_{12} + s_{14} + s_{16} + s_{18} + s_{19} + s_{23} + s_{24} + s_{26} + s_{27} + s_{29} + s_{31}$$
$$s'_{23} = u_{23} + s_0 + s_1 + s_6 + s_9 + s_{13} + s_{15} + s_{16} + s_{17} + s_{19} + s_{20} + s_{26} + s_{27} + s_{29} + s_{31}$$

$$s'_{24} = u_{24} + s_1 + s_2 + s_7 + s_{10} + s_{14} + s_{16} + s_{17} + s_{18} + s_{20} + s_{21} + s_{27} + s_{28} + s_{30}$$
$$s'_{25} = u_{25} + s_2 + s_3 + s_8 + s_{11} + s_{15} + s_{17} + s_{18} + s_{19} + s_{21} + s_{22} + s_{28} + s_{29} + s_{31}$$
$$s'_{26} = u_{26} + s_0 + s_3 + s_4 + s_6 + s_{10} + s_{18} + s_{19} + s_{20} + s_{22} + s_{23} + s_{24} + s_{25} + s_{26} + s_{28} + s_{31}$$
$$s'_{27} = u_{27} + s_1 + s_4 + s_5 + s_7 + s_{11} + s_{19} + s_{20} + s_{21} + s_{23} + s_{24} + s_{25} + s_{26} + s_{27} + s_{29}$$
$$s'_{28} = u_{28} + s_2 + s_5 + s_6 + s_8 + s_{12} + s_{20} + s_{21} + s_{22} + s_{24} + s_{25} + s_{26} + s_{27} + s_{28} + s_{30}$$
$$s'_{29} = u_{29} + s_3 + s_6 + s_7 + s_9 + s_{13} + s_{21} + s_{22} + s_{23} + s_{25} + s_{26} + s_{27} + s_{28} + s_{29} + s_{31}$$
$$s'_{30} = u_{30} + s_4 + s_7 + s_8 + s_{10} + s_{14} + s_{22} + s_{23} + s_{24} + s_{26} + s_{27} + s_{28} + s_{29} + s_{30}$$
$$s'_{31} = u_{31} + s_5 + s_8 + s_9 + s_{11} + s_{15} + s_{23} + s_{24} + s_{25} + s_{27} + s_{28} + s_{29} + s_{30} + s_{31}$$

Figure A.2: Parallel LFSR1 Architecture $\mathbf{G}_{32\times32}$ equations.

A.2 Lambda Gamma Architecture

An illustration of the Lambda Gamma Architecture is shown in Figure 4.5. Here, the implementation equations are provided for $l = 32$, and similar to the parallel LFSR2 equations, $t_i = s_i + u_i$. Figures A.3 and A.4 display the logic equations for the Lambda and Gamma matrices, respectively.

$$
\begin{aligned}
\lambda_0 &= t_0 + t_6 + t_9 + t_{10} + t_{12} + t_{16} + t_{24} + t_{25} + t_{26} + t_{28} + t_{29} + t_{30} + t_{31} \\
\lambda_1 &= t_1 + t_7 + t_{10} + t_{11} + t_{13} + t_{17} + t_{25} + t_{26} + t_{27} + t_{29} + t_{30} + t_{31} \\
\lambda_2 &= t_2 + t_8 + t_{11} + t_{12} + t_{14} + t_{18} + t_{26} + t_{27} + t_{28} + t_{30} + t_{31} \\
\lambda_3 &= t_3 + t_9 + t_{12} + t_{13} + t_{15} + t_{19} + t_{27} + t_{28} + t_{29} + t_{31} \\
\lambda_4 &= t_4 + t_{10} + t_{13} + t_{14} + t_{16} + t_{20} + t_{28} + t_{29} + t_{30} \\
\lambda_5 &= t_5 + t_{11} + t_{14} + t_{15} + t_{17} + t_{21} + t_{29} + t_{30} + t_{31} \\
\lambda_6 &= t_6 + t_{12} + t_{15} + t_{16} + t_{18} + t_{22} + t_{30} + t_{31} \\
\lambda_7 &= t_7 + t_{13} + t_{16} + t_{17} + t_{19} + t_{23} + t_{31}
\end{aligned}
$$

$$
\begin{aligned}
\lambda_8 &= t_8 + t_{14} + t_{17} + t_{18} + t_{20} + t_{24} \\
\lambda_9 &= t_9 + t_{15} + t_{18} + t_{19} + t_{21} + t_{25} \\
\lambda_{10} &= t_{10} + t_{16} + t_{19} + t_{20} + t_{22} + t_{26} \\
\lambda_{11} &= t_{11} + t_{17} + t_{20} + t_{21} + t_{23} + t_{27} \\
\lambda_{12} &= t_{12} + t_{18} + t_{21} + t_{22} + t_{24} + t_{28} \\
\lambda_{13} &= t_{13} + t_{19} + t_{22} + t_{23} + t_{25} + t_{29} \\
\lambda_{14} &= t_{14} + t_{20} + t_{23} + t_{24} + t_{26} + t_{30} \\
\lambda_{15} &= t_{15} + t_{21} + t_{24} + t_{25} + t_{27} + t_{31}
\end{aligned}
$$

$$
\begin{aligned}
\lambda_{16} &= t_{16} + t_{22} + t_{25} + t_{26} + t_{28} \\
\lambda_{17} &= t_{17} + t_{23} + t_{26} + t_{27} + t_{29} \\
\lambda_{18} &= t_{18} + t_{24} + t_{27} + t_{28} + t_{30} \\
\lambda_{19} &= t_{19} + t_{25} + t_{28} + t_{29} + t_{31} \\
\lambda_{20} &= t_{20} + t_{26} + t_{29} + t_{30} \\
\lambda_{21} &= t_{21} + t_{27} + t_{30} + t_{31} \\
\lambda_{22} &= t_{22} + t_{28} + t_{31} \\
\lambda_{23} &= t_{23} + t_{29}
\end{aligned}
$$

$$
\begin{aligned}
\lambda_{24} &= t_{24} + t_{30} \\
\lambda_{25} &= t_{25} + t_{31} \\
\lambda_{26} &= t_{26} \\
\lambda_{27} &= t_{27} \\
\lambda_{28} &= t_{28} \\
\lambda_{29} &= t_{29} \\
\lambda_{30} &= t_{30} \\
\lambda_{31} &= t_{31}
\end{aligned}
$$

Figure A.3: Lambda Gamma Architecture $\Lambda_{32\times32}$ equations.

$$
\begin{aligned}
s'_0 &= \lambda_0 \\
s'_1 &= \lambda_0 + \lambda_1 \\
s'_2 &= \lambda_0 + \lambda_1 + \lambda_2 \\
s'_3 &= \lambda_1 + \lambda_2 + \lambda_3 \\
s'_4 &= \lambda_0 + \lambda_2 + \lambda_3 + \lambda_4 \\
s'_5 &= \lambda_0 + \lambda_1 + \lambda_3 + \lambda_4 + \lambda_5 \\
s'_6 &= \lambda_1 + \lambda_2 + \lambda_4 + \lambda_5 + \lambda_6 \\
s'_7 &= \lambda_0 + \lambda_2 + \lambda_3 + \lambda_5 + \lambda_6 + \lambda_7
\end{aligned}
$$

$$
\begin{aligned}
s'_8 &= \lambda_0 + \lambda_1 + \lambda_3 + \lambda_4 + \lambda_6 + \lambda_7 + \lambda_8 \\
s'_9 &= \lambda_1 + \lambda_2 + \lambda_4 + \lambda_5 + \lambda_7 + \lambda_8 + \lambda_9 \\
s'_{10} &= \lambda_0 + \lambda_2 + \lambda_3 + \lambda_5 + \lambda_6 + \lambda_8 + \lambda_9 + \lambda_{10} \\
s'_{11} &= \lambda_0 + \lambda_1 + \lambda_3 + \lambda_4 + \lambda_6 + \lambda_7 + \lambda_9 + \lambda_{10} + \lambda_{11} \\
s'_{12} &= \lambda_0 + \lambda_1 + \lambda_2 + \lambda_4 + \lambda_5 + \lambda_7 + \lambda_8 + \lambda_{10} + \lambda_{11} + \lambda_{12} \\
s'_{13} &= \lambda_1 + \lambda_2 + \lambda_3 + \lambda_5 + \lambda_6 + \lambda_8 + \lambda_9 + \lambda_{11} + \lambda_{12} + \lambda_{13} \\
s'_{14} &= \lambda_2 + \lambda_3 + \lambda_4 + \lambda_6 + \lambda_7 + \lambda_9 + \lambda_{10} + \lambda_{12} + \lambda_{13} + \lambda_{14} \\
s'_{15} &= \lambda_3 + \lambda_4 + \lambda_5 + \lambda_7 + \lambda_8 + \lambda_{10} + \lambda_{11} + \lambda_{13} + \lambda_{14} + \lambda_{15}
\end{aligned}
$$

$$
\begin{aligned}
s'_{16} &= \lambda_0 + \lambda_4 + \lambda_5 + \lambda_6 + \lambda_8 + \lambda_9 + \lambda_{11} + \lambda_{12} + \lambda_{14} + \lambda_{15} + \lambda_{16} \\
s'_{17} &= \lambda_1 + \lambda_5 + \lambda_6 + \lambda_7 + \lambda_9 + \lambda_{10} + \lambda_{12} + \lambda_{13} + \lambda_{15} + \lambda_{16} + \lambda_{17} \\
s'_{18} &= \lambda_2 + \lambda_6 + \lambda_7 + \lambda_8 + \lambda_{10} + \lambda_{11} + \lambda_{13} + \lambda_{14} + \lambda_{16} + \lambda_{17} + \lambda_{18} \\
s'_{19} &= \lambda_3 + \lambda_7 + \lambda_8 + \lambda_9 + \lambda_{11} + \lambda_{12} + \lambda_{14} + \lambda_{15} + \lambda_{17} + \lambda_{18} + \lambda_{19} \\
s'_{20} &= \lambda_4 + \lambda_8 + \lambda_9 + \lambda_{10} + \lambda_{12} + \lambda_{13} + \lambda_{15} + \lambda_{16} + \lambda_{18} + \lambda_{19} + \lambda_{20} \\
s'_{21} &= \lambda_5 + \lambda_9 + \lambda_{10} + \lambda_{11} + \lambda_{13} + \lambda_{14} + \lambda_{16} + \lambda_{17} + \lambda_{19} + \lambda_{20} + \lambda_{21} \\
s'_{22} &= \lambda_0 + \lambda_6 + \lambda_{10} + \lambda_{11} + \lambda_{12} + \lambda_{14} + \lambda_{15} + \lambda_{17} + \lambda_{18} + \lambda_{20} + \lambda_{21} + \lambda_{22} \\
s'_{23} &= \lambda_0 + \lambda_1 + \lambda_7 + \lambda_{11} + \lambda_{12} + \lambda_{13} + \lambda_{15} + \lambda_{16} + \lambda_{18} + \lambda_{19} + \lambda_{21} + \lambda_{22} + \lambda_{23}
\end{aligned}
$$

$$
\begin{aligned}
s'_{24} &= \lambda_1 + \lambda_2 + \lambda_8 + \lambda_{12} + \lambda_{13} + \lambda_{14} + \lambda_{16} + \lambda_{17} + \lambda_{19} + \lambda_{20} + \lambda_{22} + \lambda_{23} + \lambda_{24} \\
s'_{25} &= \lambda_2 + \lambda_3 + \lambda_9 + \lambda_{13} + \lambda_{14} + \lambda_{15} + \lambda_{17} + \lambda_{18} + \lambda_{20} + \lambda_{21} + \lambda_{23} + \lambda_{24} + \lambda_{25} \\
s'_{26} &= \lambda_0 + \lambda_3 + \lambda_4 + \lambda_{10} + \lambda_{14} + \lambda_{15} + \lambda_{16} + \lambda_{18} + \lambda_{19} + \lambda_{21} + \lambda_{22} + \lambda_{24} + \lambda_{25} + \lambda_{26} \\
s'_{27} &= \lambda_1 + \lambda_4 + \lambda_5 + \lambda_{11} + \lambda_{15} + \lambda_{16} + \lambda_{17} + \lambda_{19} + \lambda_{20} + \lambda_{22} + \lambda_{23} + \lambda_{25} + \lambda_{26} + \lambda_{27} \\
s'_{28} &= \lambda_2 + \lambda_5 + \lambda_6 + \lambda_{12} + \lambda_{16} + \lambda_{17} + \lambda_{18} + \lambda_{20} + \lambda_{21} + \lambda_{23} + \lambda_{24} + \lambda_{26} + \lambda_{27} + \lambda_{28} \\
s'_{29} &= \lambda_3 + \lambda_6 + \lambda_7 + \lambda_{13} + \lambda_{17} + \lambda_{18} + \lambda_{19} + \lambda_{21} + \lambda_{22} + \lambda_{24} + \lambda_{25} + \lambda_{27} + \lambda_{28} + \lambda_{29} \\
s'_{30} &= \lambda_4 + \lambda_7 + \lambda_8 + \lambda_{14} + \lambda_{18} + \lambda_{19} + \lambda_{20} + \lambda_{22} + \lambda_{23} + \lambda_{25} + \lambda_{26} + \lambda_{28} + \lambda_{29} + \lambda_{30} \\
s'_{31} &= \lambda_5 + \lambda_8 + \lambda_9 + \lambda_{15} + \lambda_{19} + \lambda_{20} + \lambda_{21} + \lambda_{23} + \lambda_{24} + \lambda_{26} + \lambda_{27} + \lambda_{29} + \lambda_{30} + \lambda_{31}
\end{aligned}
$$

Figure A.4: Lambda Gamma Architecture $\Gamma_{32\times32}$ equations.

A.3 State-Space Transformed Architecture

An illustration of the State-Space Transformed Architecture is shown in Figure 3.14. Here, the implementation equations are provided for the state-space transformed system when it is constructed using $\mathbf{b}^{\star}_{32\times 1} = \begin{bmatrix} 1 & 0 & 0 & \cdots & 0 \end{bmatrix}^T$ with $l = 32$. Figures A.5, A.6, and A.7 display the logic equations of the input, state, and output coupling matrices, respectively.

$$
\begin{array}{llll}
a_0 = x_{31} & a_8 = x_7 + x_{31} & a_{16} = x_{15} + x_{31} & a_{24} = x_{23} \\
a_1 = x_0 + x_{31} & a_9 = x_8 & a_{17} = x_{16} & a_{25} = x_{24} \\
a_2 = x_1 + x_{31} & a_{10} = x_9 + x_{31} & a_{18} = x_{17} & a_{26} = x_{25} + x_{31} \\
a_3 = x_2 & a_{11} = x_{10} + x_{31} & a_{19} = x_{18} & a_{27} = x_{26} \\
a_4 = x_3 + x_{31} & a_{12} = x_{11} + x_{31} & a_{20} = x_{19} & a_{28} = x_{27} \\
a_5 = x_4 + x_{31} & a_{13} = x_{12} & a_{21} = x_{20} & a_{29} = x_{28} \\
a_6 = x_5 & a_{14} = x_{13} & a_{22} = x_{21} + x_{31} & a_{30} = x_{29} \\
a_7 = x_6 + x_{31} & a_{15} = x_{14} & a_{23} = x_{22} + x_{31} & a_{31} = x_{30}
\end{array}
$$

Figure A.5: State-Space Transformed Architecture $\mathbf{A}'_{32\times 32}$ equations.

$$b_0 = u_6 + u_9 + u_{13} + u_{15} + u_{17} + u_{20} + u_{21} + u_{22} + u_{23} + u_{24} + u_{25} + u_{29} + u_{30} + u_{31}$$
$$b_1 = u_0 + u_1 + u_5 + u_6 + u_7 + u_8 + u_{10} + u_{12} + u_{13} + u_{14} + u_{15} + u_{16} + u_{17} + u_{18} + u_{20} + u_{21} + u_{22} + u_{23} + u_{24} + u_{25} + u_{27}$$
$$b_2 = u_2 + u_5 + u_8 + u_{11} + u_{13} + u_{15} + u_{16} + u_{17} + u_{18} + u_{20} + u_{23} + u_{24} + u_{25} + u_{26} + u_{27} + u_{29} + u_{30}$$
$$b_3 = u_1 + u_3 + u_5 + u_6 + u_{13} + u_{14} + u_{15} + u_{17} + u_{20} + u_{21} + u_{22} + u_{23} + u_{24} + u_{29} + u_{30} + u_{31}$$
$$b_4 = u_3 + u_4 + u_6 + u_7 + u_9 + u_{10} + u_{12} + u_{13} + u_{18} + u_{20} + u_{22} + u_{24} + u_{26} + u_{27} + u_{28} + u_{29}$$
$$b_5 = u_1 + u_3 + u_5 + u_6 + u_7 + u_9 + u_{10} + u_{11} + u_{13} + u_{15} + u_{17} + u_{18} + u_{19} + u_{20} + u_{21} + u_{22} + u_{23} + u_{25} + u_{26} + u_{27} + u_{28} + u_{29}$$
$$b_6 = u_3 + u_4 + u_5 + u_6 + u_7 + u_8 + u_{10} + u_{12} + u_{16} + u_{23} + u_{24} + u_{25} + u_{31}$$
$$b_7 = u_2 + u_3 + u_9 + u_{12} + u_{13} + u_{18} + u_{19} + u_{20} + u_{21} + u_{23} + u_{25} + u_{26}$$

$$b_8 = u_6 + u_7 + u_9 + u_{14} + u_{17} + u_{18} + u_{19} + u_{21} + u_{25} + u_{26} + u_{27} + u_{31}$$
$$b_9 = u_1 + u_2 + u_3 + u_4 + u_5 + u_7 + u_{14} + u_{15} + u_{16} + u_{19} + u_{20} + u_{21} + u_{28} + u_{29}$$
$$b_{10} = u_6 + u_7 + u_9 + u_{10} + u_{11} + u_{12} + u_{14} + u_{16} + u_{17} + u_{19} + u_{20} + u_{21} + u_{22} + u_{23} + u_{26} + u_{27} + u_{29} + u_{30} + u_{31}$$
$$b_{11} = u_1 + u_{10} + u_{12} + u_{13} + u_{16} + u_{20} + u_{21} + u_{24} + u_{28} + u_{31}$$
$$b_{12} = u_2 + u_7 + u_9 + u_{10} + u_{11} + u_{13} + u_{22} + u_{23} + u_{30} + u_{31}$$
$$b_{13} = u_2 + u_4 + u_8 + u_{10} + u_{12} + u_{17} + u_{18} + u_{19} + u_{21} + u_{23} + u_{25} + u_{27} + u_{28} + u_{29}$$
$$b_{14} = u_1 + u_3 + u_4 + u_7 + u_9 + u_{13} + u_{18} + u_{20} + u_{22} + u_{23} + u_{25} + u_{26} + u_{27}$$
$$b_{15} = u_2 + u_5 + u_6 + u_7 + u_8 + u_9 + u_{10} + u_{11} + u_{12} + u_{13} + u_{14} + u_{15} + u_{16} + u_{17} + u_{19} + u_{20} + u_{22} + u_{23} + u_{27} + u_{31}$$

$$b_{16} = u_3 + u_4 + u_5 + u_7 + u_8 + u_9 + u_{10} + u_{11} + u_{12} + u_{14} + u_{15} + u_{17} + u_{18} + u_{19} + u_{25} + u_{26} + u_{27} + u_{28} + u_{30} + u_{31}$$
$$b_{17} = u_2 + u_8 + u_{12} + u_{15} + u_{18} + u_{20} + u_{21} + u_{24} + u_{25} + u_{26} + u_{28} + u_{29} + u_{31}$$
$$b_{18} = u_1 + u_3 + u_7 + u_{10} + u_{11} + u_{13} + u_{14} + u_{15} + u_{29} + u_{30}$$
$$b_{19} = u_1 + u_4 + u_6 + u_7 + u_8 + u_9 + u_{11} + u_{12} + u_{13} + u_{14} + u_{16} + u_{18} + u_{20} + u_{21} + u_{22} + u_{23} + u_{24} + u_{26} + u_{28}$$
$$b_{20} = u_1 + u_2 + u_3 + u_4 + u_7 + u_{12} + u_{14} + u_{15} + u_{18} + u_{19} + u_{21} + u_{26} + u_{27} + u_{28} + u_{29} + u_{30}$$
$$b_{21} = u_6 + u_7 + u_9 + u_{10} + u_{11} + u_{12} + u_{13} + u_{16} + u_{22} + u_{24} + u_{27} + u_{29} + u_{30}$$
$$b_{22} = u_2 + u_3 + u_6 + u_8 + u_9 + u_{10} + u_{12} + u_{13} + u_{14} + u_{16} + u_{19} + u_{20} + u_{21} + u_{25} + u_{26} + u_{27} + u_{31}$$
$$b_{23} = u_1 + u_2 + u_3 + u_5 + u_6 + u_7 + u_8 + u_{11} + u_{17} + u_{20} + u_{22} + u_{23} + u_{27} + u_{30}$$

$$b_{24} = u_3 + u_6 + u_7 + u_9 + u_{13} + u_{14} + u_{15} + u_{17} + u_{18} + u_{19} + u_{20} + u_{21} + u_{24} + u_{26} + u_{27} + u_{28} + u_{30}$$
$$b_{25} = u_1 + u_2 + u_5 + u_7 + u_8 + u_{14} + u_{15} + u_{18} + u_{19} + u_{20} + u_{21} + u_{22} + u_{24} + u_{27} + u_{28} + u_{29}$$
$$b_{26} = u_1 + u_3 + u_5 + u_7 + u_8 + u_{10} + u_{13} + u_{15} + u_{17} + u_{18} + u_{19} + u_{22} + u_{23} + u_{25} + u_{26} + u_{30}$$
$$b_{27} = u_4 + u_5 + u_7 + u_9 + u_{10} + u_{11} + u_{12} + u_{13} + u_{14} + u_{15} + u_{16} + u_{17} + u_{20} + u_{21} + u_{23} + u_{24} + u_{25} + u_{27} + u_{28} + u_{31}$$
$$b_{28} = u_2 + u_3 + u_5 + u_6 + u_7 + u_8 + u_{11} + u_{13} + u_{14} + u_{15} + u_{16} + u_{18} + u_{19} + u_{20} + u_{21} + u_{25} + u_{27} + u_{31}$$
$$b_{29} = u_2 + u_3 + u_4 + u_5 + u_6 + u_7 + u_9 + u_{11} + u_{16} + u_{17} + u_{18} + u_{19} + u_{21} + u_{22} + u_{25} + u_{26} + u_{27} + u_{28} + u_{29}$$
$$b_{30} = u_2 + u_{11} + u_{15} + u_{16} + u_{18} + u_{19} + u_{22} + u_{23} + u_{24} + u_{26} + u_{29} + u_{30} + u_{31}$$
$$b_{31} = u_2 + u_4 + u_5 + u_6 + u_9 + u_{11} + u_{17} + u_{20} + u_{22} + u_{23} + u_{27} + u_{28} + u_{29} + u_{30}$$

Figure A.6: State-Space Transformed Architecture $\mathbf{B}'_{32\times 32}$ equations.

$$
\begin{aligned}
c_0 &= x_0 + x_1 + x_2 + x_4 + x_5 + x_7 + x_8 + x_{10} + x_{11} + x_{14} + x_{15} + x_{16} + x_{17} + x_{21} + x_{22} + x_{23} + x_{24} + x_{26} + x_{27} + x_{29} + x_{31} \\
c_1 &= x_1 + x_3 + x_8 + x_9 + x_{14} + x_{15} + x_{19} + x_{20} + x_{21} + x_{22} + x_{23} + x_{24} + x_{25} + x_{26} + x_{27} + x_{28} + x_{29} \\
c_2 &= x_1 + x_2 + x_3 + x_4 + x_5 + x_6 + x_7 + x_8 + x_9 + x_{10} + x_{11} + x_{15} + x_{17} + x_{18} + x_{21} + x_{24} + x_{25} + x_{27} + x_{28} + x_{29} + x_{31} \\
c_3 &= x_2 + x_5 + x_6 + x_9 + x_{11} + x_{12} + x_{13} + x_{14} + x_{18} + x_{19} + x_{26} + x_{29} \\
c_4 &= x_1 + x_5 + x_7 + x_8 + x_9 + x_{11} + x_{13} + x_{14} + x_{16} + x_{17} + x_{25} + x_{27} \\
c_5 &= x_1 + x_3 + x_7 + x_8 + x_{10} + x_{12} + x_{13} + x_{19} + x_{20} + x_{23} + x_{24} + x_{25} + x_{26} + x_{29} + x_{31} \\
c_6 &= x_3 + x_5 + x_6 + x_{10} + x_{11} + x_{13} + x_{15} + x_{17} + x_{18} + x_{19} + x_{20} + x_{21} + x_{23} + x_{25} + x_{26} + x_{27} + x_{28} \\
c_7 &= x_1 + x_2 + x_7 + x_8 + x_{10} + x_{12} + x_{14} + x_{15} + x_{19} + x_{20} + x_{21} + x_{24} + x_{25} + x_{26} + x_{27} + x_{28} + x_{29} + x_{31} \ .
\end{aligned}
$$

$$
\begin{aligned}
c_8 &= x_1 + x_2 + x_5 + x_6 + x_9 + x_{13} + x_{16} + x_{18} + x_{21} + x_{22} + x_{24} + x_{25} + x_{27} + x_{30} \\
c_9 &= x_2 + x_3 + x_4 + x_8 + x_9 + x_{13} + x_{14} + x_{15} + x_{16} + x_{23} + x_{26} + x_{27} + x_{29} + x_{30} \\
c_{10} &= x_1 + x_2 + x_4 + x_6 + x_7 + x_8 + x_9 + x_{15} + x_{17} + x_{20} + x_{21} + x_{22} + x_{24} + x_{26} + x_{27} + x_{28} + x_{31} \\
c_{11} &= x_1 + x_3 + x_6 + x_{11} + x_{12} + x_{13} + x_{14} + x_{15} + x_{16} + x_{18} + x_{22} + x_{23} + x_{24} + x_{26} + x_{31} \\
c_{12} &= x_1 + x_4 + x_5 + x_7 + x_9 + x_{11} + x_{13} + x_{14} + x_{18} + x_{21} + x_{22} + x_{23} + x_{30} + x_{31} \\
c_{13} &= x_2 + x_3 + x_5 + x_9 + x_{12} + x_{14} + x_{18} + x_{19} + x_{24} + x_{26} + x_{27} + x_{28} + x_{29} + x_{30} + x_{31} \\
c_{14} &= x_2 + x_4 + x_6 + x_7 + x_8 + x_{11} + x_{14} + x_{15} + x_{17} + x_{18} + x_{21} + x_{23} + x_{25} + x_{26} + x_{27} + x_{31} \\
c_{15} &= x_3 + x_6 + x_9 + x_{16} + x_{19} + x_{20} + x_{21} + x_{22} + x_{24} + x_{25} + x_{26} + x_{27} + x_{28} + x_{31}
\end{aligned}
$$

$$
\begin{aligned}
c_{16} &= x_1 + x_2 + x_5 + x_6 + x_{10} + x_{11} + x_{13} + x_{14} + x_{18} + x_{19} + x_{24} + x_{27} + x_{29} + x_{31} \\
c_{17} &= x_5 + x_8 + x_{10} + x_{11} + x_{13} + x_{14} + x_{16} + x_{18} + x_{20} + x_{21} + x_{22} + x_{25} + x_{26} + x_{27} + x_{28} + x_{31} \\
c_{18} &= x_2 + x_4 + x_7 + x_8 + x_{10} + x_{13} + x_{14} + x_{19} + x_{23} + x_{26} + x_{27} + x_{28} + x_{30} + x_{31} \\
c_{19} &= x_2 + x_6 + x_7 + x_8 + x_{12} + x_{13} + x_{14} + x_{17} + x_{20} + x_{22} + x_{24} + x_{29} + x_{30} + x_{31} \\
c_{20} &= x_5 + x_6 + x_8 + x_{10} + x_{11} + x_{12} + x_{14} + x_{15} + x_{18} + x_{19} + x_{20} + x_{21} + x_{29} \\
c_{21} &= x_4 + x_6 + x_8 + x_{12} + x_{13} + x_{14} + x_{16} + x_{17} + x_{18} + x_{21} + x_{23} + x_{25} + x_{31} \\
c_{22} &= x_1 + x_5 + x_9 + x_{11} + x_{13} + x_{15} + x_{21} + x_{22} + x_{23} + x_{24} + x_{25} + x_{27} + x_{29} + x_{30} \\
c_{23} &= x_1 + x_4 + x_5 + x_6 + x_7 + x_8 + x_{10} + x_{14} + x_{17} + x_{19} + x_{21} + x_{22} + x_{23} + x_{27} + x_{29} + x_{30}
\end{aligned}
$$

$$
\begin{aligned}
c_{24} &= x_2 + x_5 + x_6 + x_7 + x_8 + x_9 + x_{15} + x_{17} + x_{19} + x_{20} + x_{24} + x_{25} \\
c_{25} &= x_3 + x_5 + x_9 + x_{10} + x_{13} + x_{15} + x_{16} + x_{17} + x_{19} + x_{22} + x_{25} + x_{26} + x_{27} + x_{28} + x_{30} + x_{31} \\
c_{26} &= x_1 + x_5 + x_6 + x_7 + x_8 + x_{10} + x_{12} + x_{13} + x_{14} + x_{16} + x_{17} + x_{21} + x_{23} + x_{24} + x_{25} + x_{26} + x_{29} + x_{30} \\
c_{27} &= x_2 + x_4 + x_7 + x_9 + x_{12} + x_{18} + x_{20} + x_{24} + x_{27} + x_{28} + x_{29} + x_{30} \\
c_{28} &= x_3 + x_5 + x_8 + x_{10} + x_{13} + x_{15} + x_{17} + x_{19} + x_{20} + x_{21} + x_{23} + x_{24} + x_{26} + x_{28} + x_{29} \\
c_{29} &= x_3 + x_4 + x_8 + x_9 + x_{13} + x_{14} + x_{16} + x_{20} + x_{22} + x_{23} + x_{26} + x_{27} + x_{28} + x_{29} + x_{30} \\
c_{30} &= x_2 + x_3 + x_4 + x_6 + x_7 + x_8 + x_{10} + x_{11} + x_{13} + x_{14} + x_{15} + x_{16} + x_{17} + x_{19} + x_{20} + x_{21} + x_{22} + x_{26} + x_{27} + x_{29} \\
c_{31} &= x_3 + x_4 + x_6 + x_7 + x_9 + x_{11} + x_{12} + x_{15} + x_{16} + x_{18} + x_{20} + x_{22} + x_{25} + x_{29}
\end{aligned}
$$

Figure A.7: State-Space Transformed Architecture $\mathbf{C}'_{32\times 32}$ equations.

A.4 Two-Step Architecture

An illustration of the Two-Step Architecture is shown in Figure 3.7. Here, the implementation equations are given for $l = 8$ using the CRC-32 multiple polynomial $M(x) = 1 + x^{23} + x^{46} + x^{64} + x^{84} + x^{92} + x^{111} + x^{123}$. Figure A.8 displays the first step equations, and Figures A.9, A.10, A.11, and A.12 display the second step equations.

$$
\begin{aligned}
m'_0 &= u_0 + m_{115} & m'_{32} &= m_{24} & m'_{64} &= m_{56} & m'_{96} &= m_{88} + m_{119} \\
m'_1 &= u_1 + m_{116} & m'_{33} &= m_{25} & m'_{65} &= m_{57} & m'_{97} &= m_{89} + m_{120} \\
m'_2 &= u_2 + m_{117} & m'_{34} &= m_{26} & m'_{66} &= m_{58} & m'_{98} &= m_{90} + m_{121} \\
m'_3 &= u_3 + m_{118} & m'_{35} &= m_{27} & m'_{67} &= m_{59} & m'_{99} &= m_{91} + m_{122} \\
m'_4 &= u_4 + m_{119} & m'_{36} &= m_{28} & m'_{68} &= m_{60} & m'_{100} &= m_{92} \\
m'_5 &= u_5 + m_{120} & m'_{37} &= m_{29} & m'_{69} &= m_{61} & m'_{101} &= m_{93} \\
m'_6 &= u_6 + m_{121} & m'_{38} &= m_{30} & m'_{70} &= m_{62} & m'_{102} &= m_{94} \\
m'_7 &= u_7 + m_{122} & m'_{39} &= m_{31} & m'_{71} &= m_{63} & m'_{103} &= m_{95}
\end{aligned}
$$

$$
\begin{aligned}
m'_8 &= m_0 & m'_{40} &= m_{32} & m'_{72} &= m_{64} & m'_{104} &= m_{96} \\
m'_9 &= m_1 & m'_{41} &= m_{33} & m'_{73} &= m_{65} & m'_{105} &= m_{97} \\
m'_{10} &= m_2 & m'_{42} &= m_{34} & m'_{74} &= m_{66} & m'_{106} &= m_{98} \\
m'_{11} &= m_3 & m'_{43} &= m_{35} & m'_{75} &= m_{67} & m'_{107} &= m_{99} \\
m'_{12} &= m_4 & m'_{44} &= m_{36} & m'_{76} &= m_{68} & m'_{108} &= m_{100} \\
m'_{13} &= m_5 & m'_{45} &= m_{37} & m'_{77} &= m_{69} & m'_{109} &= m_{101} \\
m'_{14} &= m_6 & m'_{46} &= m_{38} + m_{115} & m'_{78} &= m_{70} & m'_{110} &= m_{102} \\
m'_{15} &= m_7 & m'_{47} &= m_{39} + m_{116} & m'_{79} &= m_{71} & m'_{111} &= m_{103} + m_{115}
\end{aligned}
$$

$$
\begin{aligned}
m'_{16} &= m_8 & m'_{48} &= m_{40} + m_{117} & m'_{80} &= m_{72} & m'_{112} &= m_{104} + m_{116} \\
m'_{17} &= m_9 & m'_{49} &= m_{41} + m_{118} & m'_{81} &= m_{73} & m'_{113} &= m_{105} + m_{117} \\
m'_{18} &= m_{10} & m'_{50} &= m_{42} + m_{119} & m'_{82} &= m_{74} & m'_{114} &= m_{106} + m_{118} \\
m'_{19} &= m_{11} & m'_{51} &= m_{43} + m_{120} & m'_{83} &= m_{75} & m'_{115} &= m_{107} + m_{119} \\
m'_{20} &= m_{12} & m'_{52} &= m_{44} + m_{121} & m'_{84} &= m_{76} + m_{115} & m'_{116} &= m_{108} + m_{120} \\
m'_{21} &= m_{13} & m'_{53} &= m_{45} + m_{122} & m'_{85} &= m_{77} + m_{116} & m'_{117} &= m_{109} + m_{121} \\
m'_{22} &= m_{14} & m'_{54} &= m_{46} & m'_{86} &= m_{78} + m_{117} & m'_{118} &= m_{110} + m_{122} \\
m'_{23} &= m_{15} + m_{115} & m'_{55} &= m_{47} & m'_{87} &= m_{79} + m_{118} & m'_{119} &= m_{111}
\end{aligned}
$$

$$
\begin{aligned}
m'_{24} &= m_{16} + m_{116} & m'_{56} &= m_{48} & m'_{88} &= m_{80} + m_{119} & m'_{120} &= m_{112} \\
m'_{25} &= m_{17} + m_{117} & m'_{57} &= m_{49} & m'_{89} &= m_{81} + m_{120} & m'_{121} &= m_{113} \\
m'_{26} &= m_{18} + m_{118} & m'_{58} &= m_{50} & m'_{90} &= m_{82} + m_{121} & m'_{122} &= m_{114} \\
m'_{27} &= m_{19} + m_{119} & m'_{59} &= m_{51} & m'_{91} &= m_{83} + m_{122} & & \\
m'_{28} &= m_{20} + m_{120} & m'_{60} &= m_{52} & m'_{92} &= m_{84} + m_{115} & & \\
m'_{29} &= m_{21} + m_{121} & m'_{61} &= m_{53} & m'_{93} &= m_{85} + m_{116} & & \\
m'_{30} &= m_{22} + m_{122} & m'_{62} &= m_{54} & m'_{94} &= m_{86} + m_{117} & & \\
m'_{31} &= m_{23} & m'_{63} &= m_{55} & m'_{95} &= m_{87} + m_{118} & &
\end{aligned}
$$

Figure A.8: Two-Step Architecture first step equations.

$$s'_0 = m_0 + m_{32} + m_{38} + m_{41} + m_{42} + m_{44} + m_{48} + m_{56} + m_{57} + m_{58} + m_{60} + m_{61} + m_{62} + m_{63} + m_{64} + m_{66}$$
$$+m_{69} + m_{76} + m_{77} + m_{79} + m_{80} + m_{82} + m_{85} + m_{86} + m_{87} + m_{90} + m_{92} + m_{93} + m_{95} + m_{97} + m_{98} + m_{99}$$
$$+m_{100} + m_{104} + m_{105} + m_{111} + m_{113} + m_{114} + m_{115} + m_{116} + m_{117} + m_{119}$$
$$s'_1 = m_1 + m_{32} + m_{33} + m_{38} + m_{39} + m_{41} + m_{43} + m_{44} + m_{45} + m_{48} + m_{49} + m_{56} + m_{59} + m_{60} + m_{65} + m_{66}$$
$$+m_{67} + m_{69} + m_{70} + m_{76} + m_{78} + m_{79} + m_{81} + m_{82} + m_{83} + m_{85} + m_{88} + m_{90} + m_{91} + m_{92} + m_{94} + m_{95}$$
$$+m_{96} + m_{97} + m_{101} + m_{104} + m_{106} + m_{111} + m_{112} + m_{113} + m_{118} + m_{119} + m_{120}$$
$$s'_2 = m_2 + m_{32} + m_{33} + m_{34} + m_{38} + m_{39} + m_{40} + m_{41} + m_{45} + m_{46} + m_{48} + m_{49} + m_{50} + m_{56} + m_{58} + m_{62}$$
$$+m_{63} + m_{64} + m_{67} + m_{68} + m_{69} + m_{70} + m_{71} + m_{76} + m_{83} + m_{84} + m_{85} + m_{87} + m_{89} + m_{90} + m_{91} + m_{96}$$
$$+m_{99} + m_{100} + m_{102} + m_{104} + m_{107} + m_{111} + m_{112} + m_{115} + m_{116} + m_{117} + m_{120} + m_{121}$$
$$s'_3 = m_3 + m_{33} + m_{34} + m_{35} + m_{39} + m_{40} + m_{41} + m_{42} + m_{46} + m_{47} + m_{49} + m_{50} + m_{51} + m_{57} + m_{59} + m_{63}$$
$$+m_{64} + m_{65} + m_{68} + m_{69} + m_{70} + m_{71} + m_{72} + m_{77} + m_{84} + m_{85} + m_{86} + m_{88} + m_{90} + m_{91} + m_{92} + m_{97}$$
$$+m_{100} + m_{101} + m_{103} + m_{105} + m_{108} + m_{112} + m_{113} + m_{116} + m_{117} + m_{118} + m_{121} + m_{122}$$

$$s'_4 = m_4 + m_{32} + m_{34} + m_{35} + m_{36} + m_{38} + m_{40} + m_{43} + m_{44} + m_{47} + m_{50} + m_{51} + m_{52} + m_{56} + m_{57} + m_{61}$$
$$+m_{62} + m_{63} + m_{65} + m_{70} + m_{71} + m_{72} + m_{73} + m_{76} + m_{77} + m_{78} + m_{79} + m_{80} + m_{82} + m_{89} + m_{90} + m_{91}$$
$$+m_{95} + m_{97} + m_{99} + m_{100} + m_{101} + m_{102} + m_{105} + m_{106} + m_{109} + m_{111} + m_{115} + m_{116} + m_{118} + m_{122}$$
$$s'_5 = m_5 + m_{32} + m_{33} + m_{35} + m_{36} + m_{37} + m_{38} + m_{39} + m_{42} + m_{45} + m_{51} + m_{52} + m_{53} + m_{56} + m_{60} + m_{61} + m_{69}$$
$$+m_{71} + m_{72} + m_{73} + m_{74} + m_{76} + m_{78} + m_{81} + m_{82} + m_{83} + m_{85} + m_{86} + m_{87} + m_{91} + m_{93} + m_{95} + m_{96} +$$
$$+m_{97} + m_{99} + m_{101} + m_{102} + m_{103} + m_{104} + m_{105} + m_{106} + m_{107} + m_{110} + m_{111} + m_{112} + m_{113} + m_{114} + m_{115}$$
$$s'_6 = m_6 + m_{33} + m_{34} + m_{36} + m_{37} + m_{38} + m_{39} + m_{40} + m_{43} + m_{46} + m_{52} + m_{53} + m_{54} + m_{57} + m_{61} + m_{62} + m_{70}$$
$$+m_{72} + m_{73} + m_{74} + m_{75} + m_{77} + m_{79} + m_{82} + m_{83} + m_{84} + m_{86} + m_{87} + m_{88} + m_{92} + m_{94} + m_{96} + m_{97}$$
$$+m_{98} + m_{100} + m_{102} + m_{103} + m_{104} + m_{105} + m_{106} + m_{107} + m_{108} + m_{111} + m_{112} + m_{113} + m_{114} + m_{115} + m_{116}$$
$$s'_7 = m_7 + m_{32} + m_{34} + m_{35} + m_{37} + m_{39} + m_{40} + m_{42} + m_{47} + m_{48} + m_{53} + m_{54} + m_{55} + m_{56} + m_{57} + m_{60}$$
$$+m_{61} + m_{64} + m_{66} + m_{69} + m_{71} + m_{73} + m_{74} + m_{75} + m_{77} + m_{78} + m_{79} + m_{82} + m_{83} + m_{84} + m_{86} + m_{88}$$
$$+m_{89} + m_{90} + m_{92} + m_{100} + m_{101} + m_{103} + m_{106} + m_{107} + m_{108} + m_{109} + m_{111} + m_{112} + m_{119}$$

Figure A.9: Two-Step Architecture second step equations (1 of 4).

$$
\begin{aligned}
s'_8 &= m_8 + m_{32} + m_{33} + m_{35} + m_{36} + m_{40} + m_{42} + m_{43} + m_{44} + m_{49} + m_{54} + m_{55} + m_{60} + m_{63} + m_{64} + m_{65} + m_{66} + m_{67} \\
&\quad + m_{69} + m_{70} + m_{72} + m_{74} + m_{75} + m_{77} + m_{78} + m_{82} + m_{83} + m_{84} + m_{86} + m_{89} + m_{91} + m_{92} + m_{95} + m_{97} + m_{98} + m_{99} \\
&\quad + m_{100} + m_{101} + m_{102} + m_{105} + m_{107} + m_{108} + m_{109} + m_{110} + m_{111} + m_{112} + m_{114} + m_{115} + m_{116} + m_{117} + m_{119} + m_{120} \\
s'_9 &= m_9 + m_{33} + m_{34} + m_{36} + m_{37} + m_{41} + m_{43} + m_{44} + m_{45} + m_{50} + m_{55} + m_{56} + m_{61} + m_{64} + m_{65} + m_{66} + m_{67} + m_{68} \\
&\quad + m_{70} + m_{71} + m_{73} + m_{75} + m_{76} + m_{78} + m_{79} + m_{83} + m_{84} + m_{85} + m_{87} + m_{90} + m_{92} + m_{93} + m_{96} + m_{98} + m_{99} + m_{100} \\
&\quad + m_{101} + m_{102} + m_{103} + m_{106} + m_{108} + m_{109} + m_{110} + m_{111} + m_{112} + m_{113} + m_{115} + m_{116} + m_{117} + m_{118} + m_{120} + m_{121} \\
s'_{10} &= m_{10} + m_{32} + m_{34} + m_{35} + m_{37} + m_{41} + m_{45} + m_{46} + m_{48} + m_{51} + m_{58} + m_{60} + m_{61} + m_{63} + m_{64} + m_{65} \\
&\quad + m_{67} + m_{68} + m_{71} + m_{72} + m_{74} + m_{82} + m_{84} + m_{87} + m_{88} + m_{90} + m_{91} + m_{92} + m_{94} + m_{95} + m_{98} + m_{101} \\
&\quad + m_{102} + m_{103} + m_{105} + m_{107} + m_{109} + m_{110} + m_{112} + m_{115} + m_{118} + m_{121} + m_{122} \\
s'_{11} &= m_{11} + m_{32} + m_{33} + m_{35} + m_{36} + m_{41} + m_{44} + m_{46} + m_{47} + m_{48} + m_{49} + m_{52} + m_{56} + m_{57} + m_{58} + m_{59} + m_{60} \\
&\quad + m_{63} + m_{65} + m_{68} + m_{72} + m_{73} + m_{75} + m_{76} + m_{77} + m_{79} + m_{80} + m_{82} + m_{83} + m_{86} + m_{87} + m_{88} + m_{89} + m_{90} \\
&\quad + m_{91} + m_{96} + m_{97} + m_{98} + m_{100} + m_{102} + m_{103} + m_{105} + m_{106} + m_{108} + m_{110} + m_{114} + m_{115} + m_{117} + m_{122}
\end{aligned}
$$

$$
\begin{aligned}
s'_{12} &= m_{12} + m_{32} + m_{33} + m_{34} + m_{36} + m_{37} + m_{38} + m_{41} + m_{44} + m_{45} + m_{47} + m_{49} + m_{50} + m_{53} + m_{56} + m_{59} \\
&\quad + m_{62} + m_{63} + m_{73} + m_{74} + m_{78} + m_{79} + m_{81} + m_{82} + m_{83} + m_{84} + m_{85} + m_{86} + m_{88} + m_{89} + m_{91} + m_{93} \\
&\quad + m_{95} + m_{100} + m_{101} + m_{103} + m_{105} + m_{106} + m_{107} + m_{109} + m_{113} + m_{114} + m_{117} + m_{118} + m_{119} \\
s'_{13} &= m_{13} + m_{33} + m_{34} + m_{35} + m_{37} + m_{38} + m_{39} + m_{42} + m_{45} + m_{46} + m_{48} + m_{50} + m_{51} + m_{54} + m_{57} + m_{60} \\
&\quad + m_{63} + m_{64} + m_{74} + m_{75} + m_{79} + m_{80} + m_{82} + m_{83} + m_{84} + m_{85} + m_{86} + m_{87} + m_{89} + m_{90} + m_{92} + m_{94} \\
&\quad + m_{96} + m_{101} + m_{102} + m_{104} + m_{106} + m_{107} + m_{108} + m_{110} + m_{114} + m_{115} + m_{118} + m_{119} + m_{120} \\
s'_{14} &= m_{14} + m_{34} + m_{35} + m_{36} + m_{38} + m_{39} + m_{40} + m_{43} + m_{46} + m_{47} + m_{49} + m_{51} + m_{52} + m_{55} + m_{58} + m_{61} \\
&\quad + m_{64} + m_{65} + m_{75} + m_{76} + m_{80} + m_{81} + m_{83} + m_{84} + m_{85} + m_{86} + m_{87} + m_{88} + m_{90} + m_{91} + m_{93} + m_{95} \\
&\quad + m_{97} + m_{102} + m_{103} + m_{105} + m_{107} + m_{108} + m_{109} + m_{111} + m_{115} + m_{116} + m_{119} + m_{120} + m_{121} \\
s'_{15} &= m_{15} + m_{35} + m_{36} + m_{37} + m_{39} + m_{40} + m_{41} + m_{44} + m_{47} + m_{48} + m_{50} + m_{52} + m_{53} + m_{56} + m_{59} + m_{62} \\
&\quad + m_{65} + m_{66} + m_{76} + m_{77} + m_{81} + m_{82} + m_{84} + m_{85} + m_{86} + m_{87} + m_{88} + m_{89} + m_{91} + m_{92} + m_{94} + m_{96} \\
&\quad + m_{98} + m_{103} + m_{104} + m_{106} + m_{108} + m_{109} + m_{110} + m_{112} + m_{116} + m_{117} + m_{120} + m_{121} + m_{122}
\end{aligned}
$$

Figure A.10: Two-Step Architecture second step equations (2 of 4).

$$s'_{16} = m_{16} + m_{32} + m_{36} + m_{37} + m_{40} + m_{44} + m_{45} + m_{49} + m_{51} + m_{53} + m_{54} + m_{56} + m_{58} + m_{61} + m_{62} + m_{64}$$
$$+m_{67} + m_{69} + m_{76} + m_{78} + m_{79} + m_{80} + m_{83} + m_{88} + m_{89} + m_{98} + m_{100} + m_{107} + m_{109} + m_{110} + m_{114}$$
$$+m_{115} + m_{116} + m_{118} + m_{119} + m_{121} + m_{122}$$
$$s'_{17} = m_{17} + m_{33} + m_{37} + m_{38} + m_{41} + m_{45} + m_{46} + m_{50} + m_{52} + m_{54} + m_{55} + m_{57} + m_{59} + m_{62} + m_{63} + m_{65}$$
$$+m_{68} + m_{70} + m_{77} + m_{79} + m_{80} + m_{81} + m_{84} + m_{89} + m_{90} + m_{99} + m_{101} + m_{108} + m_{110} + m_{111} + m_{115}$$
$$+m_{116} + m_{117} + m_{119} + m_{120} + m_{122}$$
$$s'_{18} = m_{18} + m_{34} + m_{38} + m_{39} + m_{42} + m_{46} + m_{47} + m_{51} + m_{53} + m_{55} + m_{56} + m_{58} + m_{60} + m_{63} + m_{64} + m_{66}$$
$$+m_{69} + m_{71} + m_{78} + m_{80} + m_{81} + m_{82} + m_{85} + m_{90} + m_{91} + m_{100} + m_{102} + m_{109} + m_{111} + m_{112} + m_{116}$$
$$+m_{117} + m_{118} + m_{120} + m_{121}$$
$$s'_{19} = m_{19} + m_{35} + m_{39} + m_{40} + m_{43} + m_{47} + m_{48} + m_{52} + m_{54} + m_{56} + m_{57} + m_{59} + m_{61} + m_{64} + m_{65} + m_{67}$$
$$+m_{70} + m_{72} + m_{79} + m_{81} + m_{82} + m_{83} + m_{86} + m_{91} + m_{92} + m_{101} + m_{103} + m_{110} + m_{112} + m_{113} + m_{117}$$
$$+m_{118} + m_{119} + m_{121} + m_{122}$$

$$s'_{20} = m_{20} + m_{36} + m_{40} + m_{41} + m_{44} + m_{48} + m_{49} + m_{53} + m_{55} + m_{57} + m_{58} + m_{60} + m_{62} + m_{65} + m_{66} + m_{68}$$
$$+m_{71} + m_{73} + m_{80} + m_{82} + m_{83} + m_{84} + m_{87} + m_{92} + m_{93} + m_{102} + m_{104} + m_{111} + m_{113} + m_{114} + m_{118}$$
$$+m_{119} + m_{120} + m_{122}$$
$$s'_{21} = m_{21} + m_{37} + m_{41} + m_{42} + m_{45} + m_{49} + m_{50} + m_{54} + m_{56} + m_{58} + m_{59} + m_{61} + m_{63} + m_{66} + m_{67} + m_{69}$$
$$+m_{72} + m_{74} + m_{81} + m_{83} + m_{84} + m_{85} + m_{88} + m_{93} + m_{94} + m_{103} + m_{105} + m_{112} + m_{114} + m_{115} + m_{119}$$
$$+m_{120} + m_{121}$$
$$s'_{22} = m_{22} + m_{32} + m_{41} + m_{43} + m_{44} + m_{46} + m_{48} + m_{50} + m_{51} + m_{55} + m_{56} + m_{58} + m_{59} + m_{61} + m_{63} + m_{66}$$
$$+m_{67} + m_{68} + m_{69} + m_{70} + m_{73} + m_{75} + m_{76} + m_{77} + m_{79} + m_{80} + m_{84} + m_{87} + m_{89} + m_{90} + m_{92} + m_{93}$$
$$+m_{94} + m_{97} + m_{98} + m_{99} + m_{100} + m_{105} + m_{106} + m_{111} + m_{114} + m_{117} + m_{119} + m_{120} + m_{121} + m_{122}$$
$$s'_{23} = m_{23} + m_{32} + m_{33} + m_{38} + m_{41} + m_{45} + m_{47} + m_{48} + m_{49} + m_{51} + m_{52} + m_{58} + m_{59} + m_{61} + m_{63} + m_{66} + m_{67}$$
$$+m_{68} + m_{70} + m_{71} + m_{74} + m_{78} + m_{79} + m_{81} + m_{82} + m_{86} + m_{87} + m_{88} + m_{91} + m_{92} + m_{94} + m_{97} + m_{101}$$
$$+m_{104} + m_{105} + m_{106} + m_{107} + m_{111} + m_{112} + m_{113} + m_{114} + m_{116} + m_{117} + m_{118} + m_{119} + m_{120} + m_{121} + m_{122}$$

Figure A.11: Two-Step Architecture second step equations (3 of 4).

$$
\begin{aligned}
s'_{24} &= m_{24} + m_{33} + m_{34} + m_{39} + m_{42} + m_{46} + m_{48} + m_{49} + m_{50} + m_{52} + m_{53} + m_{59} + m_{60} + m_{62} + m_{64} + m_{67} \\
&\quad + m_{68} + m_{69} + m_{71} + m_{72} + m_{75} + m_{79} + m_{80} + m_{82} + m_{83} + m_{87} + m_{88} + m_{89} + m_{92} + m_{93} + m_{95} + m_{98} \\
&\quad + m_{102} + m_{105} + m_{106} + m_{107} + m_{108} + m_{112} + m_{113} + m_{114} + m_{115} + m_{117} + m_{118} + m_{119} + m_{120} + m_{121} + m_{122} \\
s'_{25} &= m_{25} + m_{34} + m_{35} + m_{40} + m_{43} + m_{47} + m_{49} + m_{50} + m_{51} + m_{53} + m_{54} + m_{60} + m_{61} + m_{63} + m_{65} + m_{68} \\
&\quad + m_{69} + m_{70} + m_{72} + m_{73} + m_{76} + m_{80} + m_{81} + m_{83} + m_{84} + m_{88} + m_{89} + m_{90} + m_{93} + m_{94} + m_{96} + m_{99} \\
&\quad + m_{103} + m_{106} + m_{107} + m_{108} + m_{109} + m_{113} + m_{114} + m_{115} + m_{116} + m_{118} + m_{119} + m_{120} + m_{121} + m_{122} \\
s'_{26} &= m_{26} + m_{32} + m_{35} + m_{36} + m_{38} + m_{42} + m_{50} + m_{51} + m_{52} + m_{54} + m_{55} + m_{56} + m_{57} + m_{58} + m_{60} + m_{63} \\
&\quad + m_{70} + m_{71} + m_{73} + m_{74} + m_{76} + m_{79} + m_{80} + m_{81} + m_{84} + m_{86} + m_{87} + m_{89} + m_{91} + m_{92} + m_{93} + m_{94} \\
&\quad + m_{98} + m_{99} + m_{105} + m_{107} + m_{108} + m_{109} + m_{110} + m_{111} + m_{113} + m_{120} + m_{121} + m_{122} \\
s'_{27} &= m_{27} + m_{33} + m_{36} + m_{37} + m_{39} + m_{43} + m_{51} + m_{52} + m_{53} + m_{55} + m_{56} + m_{57} + m_{58} + m_{59} + m_{61} + m_{64} \\
&\quad + m_{71} + m_{72} + m_{74} + m_{75} + m_{77} + m_{80} + m_{81} + m_{82} + m_{85} + m_{87} + m_{88} + m_{90} + m_{92} + m_{93} + m_{94} + m_{95} \\
&\quad + m_{99} + m_{100} + m_{106} + m_{108} + m_{109} + m_{110} + m_{111} + m_{112} + m_{114} + m_{121} + m_{122}
\end{aligned}
$$

$$
\begin{aligned}
s'_{28} &= m_{28} + m_{34} + m_{37} + m_{38} + m_{40} + m_{44} + m_{52} + m_{53} + m_{54} + m_{56} + m_{57} + m_{58} + m_{59} + m_{60} + m_{62} + m_{65} \\
&\quad + m_{72} + m_{73} + m_{75} + m_{76} + m_{78} + m_{81} + m_{82} + m_{83} + m_{86} + m_{88} + m_{89} + m_{91} + m_{93} + m_{94} + m_{95} + m_{96} \\
&\quad + m_{100} + m_{101} + m_{107} + m_{109} + m_{110} + m_{111} + m_{112} + m_{113} + m_{115} + m_{122} \\
s'_{29} &= m_{29} + m_{35} + m_{38} + m_{39} + m_{41} + m_{45} + m_{53} + m_{54} + m_{55} + m_{57} + m_{58} + m_{59} + m_{60} + m_{61} + m_{63} + m_{66} \\
&\quad + m_{73} + m_{74} + m_{76} + m_{77} + m_{79} + m_{82} + m_{83} + m_{84} + m_{87} + m_{89} + m_{90} + m_{92} + m_{94} + m_{95} + m_{96} + m_{97} \\
&\quad + m_{101} + m_{102} + m_{108} + m_{110} + m_{111} + m_{112} + m_{113} + m_{114} + m_{116} \\
s'_{30} &= m_{30} + m_{36} + m_{39} + m_{40} + m_{42} + m_{46} + m_{54} + m_{55} + m_{56} + m_{58} + m_{59} + m_{60} + m_{61} + m_{62} + m_{64} + m_{67} \\
&\quad + m_{74} + m_{75} + m_{77} + m_{78} + m_{80} + m_{83} + m_{84} + m_{85} + m_{88} + m_{90} + m_{91} + m_{93} + m_{95} + m_{96} + m_{97} + m_{98} \\
&\quad + m_{102} + m_{103} + m_{109} + m_{111} + m_{112} + m_{113} + m_{114} + m_{115} + m_{117} \\
s'_{31} &= m_{31} + m_{37} + m_{40} + m_{41} + m_{43} + m_{47} + m_{55} + m_{56} + m_{57} + m_{59} + m_{60} + m_{61} + m_{62} + m_{63} + m_{65} + m_{68} \\
&\quad + m_{75} + m_{76} + m_{78} + m_{79} + m_{81} + m_{84} + m_{85} + m_{86} + m_{89} + m_{91} + m_{92} + m_{94} + m_{96} + m_{97} + m_{98} + m_{99} \\
&\quad + m_{103} + m_{104} + m_{110} + m_{112} + m_{113} + m_{114} + m_{115} + m_{116} + m_{118}
\end{aligned}
$$

Figure A.12: Two-Step Architecture second step equations (4 of 4).

Appendix B

CRC-32 Software Algorithm Data

IN this appendix, the software algorithm data for the various implementations of the CRC computation using the generator polynomial CRC-32 $G(x) = 1 + x + x^2 + x^4 + x^5 + x^7 + x^8 + x^{10} + x^{11} + x^{12} + x^{16} + x^{22} + x^{23} + x^{26} + x^{32}$ [4] are presented. For each of the look-up based software algorithms studied in this thesis, we list the complete LUTs in hexadecimal notation. In addition, we provide the CRCF equations in a form similar to what is shown in [13] for CRC-16.

Many of the assumptions stated in Chapter 3 concerning the bit ordering of the software CRC computation are maintained, the two most important being

- the leading x^m term of the generator polynomial is considered implicit; and
- reverse Endianness convention is employed.

Since our datapath is 32-bit, i.e., $w = 32$, the CRC-32 LUT entries each occupy a single memory word. However, the LUT entries contained in this appendix are easily generalized for use on smaller datapaths.

We note that, all of the LUT entries were obtained using the generation algorithms presented in this thesis. Therefore, the content contained in this appendix can be used by a designer who wishes to realize the CRC-32 computation as a software algorithm and/or to verify the correctness of their LUT generation code.

The remainder of this appendix is organized as follows. In Section B.1, the LUT entries of CRCT(8) are listed. In Section B.2, the LUT entries of CRCR(32) are listed. In Section B.3, the LUT entries of CRCS4(32) are listed. In Section B.4, the LUT entries of CRCΛΓ(32) are listed. Finally, in Section B.5, the equations for CRCF(8) are illustrated.

B.1 Table Look-up Algorithm

An implementation of the Table Look-up Algorithm (CRCT) is shown in Algorithm 3.2. Figures B.1 and B.2 display the LUT contents generated by Algorithm 3.5.

```
lut[0x00] = 0x00000000  lut[0x20] = 0x3b6e20c8  lut[0x40] = 0x76dc4190  lut[0x60] = 0x4db26158
lut[0x01] = 0x77073096  lut[0x21] = 0x4c69105e  lut[0x41] = 0x01db7106  lut[0x61] = 0x3ab551ce
lut[0x02] = 0xee0e612c  lut[0x22] = 0xd56041e4  lut[0x42] = 0x98d220bc  lut[0x62] = 0xa3bc0074
lut[0x03] = 0x990951ba  lut[0x23] = 0xa2677172  lut[0x43] = 0xefd5102a  lut[0x63] = 0xd4bb30e2
lut[0x04] = 0x076dc419  lut[0x24] = 0x3c03e4d1  lut[0x44] = 0x71b18589  lut[0x64] = 0x4adfa541
lut[0x05] = 0x706af48f  lut[0x25] = 0x4b04d447  lut[0x45] = 0x06b6b51f  lut[0x65] = 0x3dd895d7
lut[0x06] = 0xe963a535  lut[0x26] = 0xd20d85fd  lut[0x46] = 0x9fbfe4a5  lut[0x66] = 0xa4d1c46d
lut[0x07] = 0x9e6495a3  lut[0x27] = 0xa50ab56b  lut[0x47] = 0xe8b8d433  lut[0x67] = 0xd3d6f4fb

lut[0x08] = 0x0edb8832  lut[0x28] = 0x35b5a8fa  lut[0x48] = 0x7807c9a2  lut[0x68] = 0x4369e96a
lut[0x09] = 0x79dcb8a4  lut[0x29] = 0x42b2986c  lut[0x49] = 0x0f00f934  lut[0x69] = 0x346ed9fc
lut[0x0a] = 0xe0d5e91e  lut[0x2a] = 0xdbbbc9d6  lut[0x4a] = 0x9609a88e  lut[0x6a] = 0xad678846
lut[0x0b] = 0x97d2d988  lut[0x2b] = 0xacbcf940  lut[0x4b] = 0xe10e9818  lut[0x6b] = 0xda60b8d0
lut[0x0c] = 0x09b64c2b  lut[0x2c] = 0x32d86ce3  lut[0x4c] = 0x7f6a0dbb  lut[0x6c] = 0x44042d73
lut[0x0d] = 0x7eb17cbd  lut[0x2d] = 0x45df5c75  lut[0x4d] = 0x086d3d2d  lut[0x6d] = 0x33031de5
lut[0x0e] = 0xe7b82d07  lut[0x2e] = 0xdcd60dcf  lut[0x4e] = 0x91646c97  lut[0x6e] = 0xaa0a4c5f
lut[0x0f] = 0x90bf1d91  lut[0x2f] = 0xabd13d59  lut[0x4f] = 0xe6635c01  lut[0x6f] = 0xdd0d7cc9

lut[0x10] = 0x1db71064  lut[0x30] = 0x26d930ac  lut[0x50] = 0x6b6b51f4  lut[0x70] = 0x5005713c
lut[0x11] = 0x6ab020f2  lut[0x31] = 0x51de003a  lut[0x51] = 0x1c6c6162  lut[0x71] = 0x270241aa
lut[0x12] = 0xf3b97148  lut[0x32] = 0xc8d75180  lut[0x52] = 0x856530d8  lut[0x72] = 0xbe0b1010
lut[0x13] = 0x84be41de  lut[0x33] = 0xbfd06116  lut[0x53] = 0xf262004e  lut[0x73] = 0xc90c2086
lut[0x14] = 0x1adad47d  lut[0x34] = 0x21b4f4b5  lut[0x54] = 0x6c0695ed  lut[0x74] = 0x5768b525
lut[0x15] = 0x6ddde4eb  lut[0x35] = 0x56b3c423  lut[0x55] = 0x1b01a57b  lut[0x75] = 0x206f85b3
lut[0x16] = 0xf4d4b551  lut[0x36] = 0xcfba9599  lut[0x56] = 0x8208f4c1  lut[0x76] = 0xb966d409
lut[0x17] = 0x83d385c7  lut[0x37] = 0xb8bda50f  lut[0x57] = 0xf50fc457  lut[0x77] = 0xce61e49f

lut[0x18] = 0x136c9856  lut[0x38] = 0x2802b89e  lut[0x58] = 0x65b0d9c6  lut[0x78] = 0x5edef90e
lut[0x19] = 0x646ba8c0  lut[0x39] = 0x5f058808  lut[0x59] = 0x12b7e950  lut[0x79] = 0x29d9c998
lut[0x1a] = 0xfd62f97a  lut[0x3a] = 0xc60cd9b2  lut[0x5a] = 0x8bbeb8ea  lut[0x7a] = 0xb0d09822
lut[0x1b] = 0x8a65c9ec  lut[0x3b] = 0xb10be924  lut[0x5b] = 0xfcb9887c  lut[0x7b] = 0xc7d7a8b4
lut[0x1c] = 0x14015c4f  lut[0x3c] = 0x2f6f7c87  lut[0x5c] = 0x62dd1ddf  lut[0x7c] = 0x59b33d17
lut[0x1d] = 0x63066cd9  lut[0x3d] = 0x58684c11  lut[0x5d] = 0x15da2d49  lut[0x7d] = 0x2eb40d81
lut[0x1e] = 0xfa0f3d63  lut[0x3e] = 0xc1611dab  lut[0x5e] = 0x8cd37cf3  lut[0x7e] = 0xb7bd5c3b
lut[0x1f] = 0x8d080df5  lut[0x3f] = 0xb6662d3d  lut[0x5f] = 0xfbd44c65  lut[0x7f] = 0xc0ba6cad
```

Figure B.1: CRCT(8) LUT entries (1 of 2).

```
lut[0x80] = 0xedb88320  lut[0xa0] = 0xd6d6a3e8  lut[0xc0] = 0x9b64c2b0  lut[0xe0] = 0xa00ae278
lut[0x81] = 0x9abfb3b6  lut[0xa1] = 0xa1d1937e  lut[0xc1] = 0xec63f226  lut[0xe1] = 0xd70dd2ee
lut[0x82] = 0x03b6e20c  lut[0xa2] = 0x38d8c2c4  lut[0xc2] = 0x756aa39c  lut[0xe2] = 0x4e048354
lut[0x83] = 0x74b1d29a  lut[0xa3] = 0x4fdff252  lut[0xc3] = 0x026d930a  lut[0xe3] = 0x3903b3c2
lut[0x84] = 0xead54739  lut[0xa4] = 0xd1bb67f1  lut[0xc4] = 0x9c0906a9  lut[0xe4] = 0xa7672661
lut[0x85] = 0x9dd277af  lut[0xa5] = 0xa6bc5767  lut[0xc5] = 0xeb0e363f  lut[0xe5] = 0xd06016f7
lut[0x86] = 0x04db2615  lut[0xa6] = 0x3fb506dd  lut[0xc6] = 0x72076785  lut[0xe6] = 0x4969474d
lut[0x87] = 0x73dc1683  lut[0xa7] = 0x48b2364b  lut[0xc7] = 0x05005713  lut[0xe7] = 0x3e6e77db

lut[0x88] = 0xe3630b12  lut[0xa8] = 0xd80d2bda  lut[0xc8] = 0x95bf4a82  lut[0xe8] = 0xaed16a4a
lut[0x89] = 0x94643b84  lut[0xa9] = 0xaf0a1b4c  lut[0xc9] = 0xe2b87a14  lut[0xe9] = 0xd9d65adc
lut[0x8a] = 0x0d6d6a3e  lut[0xaa] = 0x36034af6  lut[0xca] = 0x7bb12bae  lut[0xea] = 0x40df0b66
lut[0x8b] = 0x7a6a5aa8  lut[0xab] = 0x41047a60  lut[0xcb] = 0x0cb61b38  lut[0xeb] = 0x37d83bf0
lut[0x8c] = 0xe40ecf0b  lut[0xac] = 0xdf60efc3  lut[0xcc] = 0x92d28e9b  lut[0xec] = 0xa9bcae53
lut[0x8d] = 0x9309ff9d  lut[0xad] = 0xa867df55  lut[0xcd] = 0xe5d5be0d  lut[0xed] = 0xdebb9ec5
lut[0x8e] = 0x0a00ae27  lut[0xae] = 0x316e8eef  lut[0xce] = 0x7cdcefb7  lut[0xee] = 0x47b2cf7f
lut[0x8f] = 0x7d079eb1  lut[0xaf] = 0x4669be79  lut[0xcf] = 0x0bdbdf21  lut[0xef] = 0x30b5ffe9

lut[0x90] = 0xf00f9344  lut[0xb0] = 0xcb61b38c  lut[0xd0] = 0x86d3d2d4  lut[0xf0] = 0xbdbdf21c
lut[0x91] = 0x8708a3d2  lut[0xb1] = 0xbc66831a  lut[0xd1] = 0xf1d4e242  lut[0xf1] = 0xcabac28a
lut[0x92] = 0x1e01f268  lut[0xb2] = 0x256fd2a0  lut[0xd2] = 0x68ddb3f8  lut[0xf2] = 0x53b39330
lut[0x93] = 0x6906c2fe  lut[0xb3] = 0x5268e236  lut[0xd3] = 0x1fda836e  lut[0xf3] = 0x24b4a3a6
lut[0x94] = 0xf762575d  lut[0xb4] = 0xcc0c7795  lut[0xd4] = 0x81be16cd  lut[0xf4] = 0xbad03605
lut[0x95] = 0x806567cb  lut[0xb5] = 0xbb0b4703  lut[0xd5] = 0xf6b9265b  lut[0xf5] = 0xcdd70693
lut[0x96] = 0x196c3671  lut[0xb6] = 0x220216b9  lut[0xd6] = 0x6fb077e1  lut[0xf6] = 0x54de5729
lut[0x97] = 0x6e6b06e7  lut[0xb7] = 0x5505262f  lut[0xd7] = 0x18b74777  lut[0xf7] = 0x23d967bf

lut[0x98] = 0xfed41b76  lut[0xb8] = 0xc5ba3bbe  lut[0xd8] = 0x88085ae6  lut[0xf8] = 0xb3667a2e
lut[0x99] = 0x89d32be0  lut[0xb9] = 0xb2bd0b28  lut[0xd9] = 0xff0f6a70  lut[0xf9] = 0xc4614ab8
lut[0x9a] = 0x10da7a5a  lut[0xba] = 0x2bb45a92  lut[0xda] = 0x66063bca  lut[0xfa] = 0x5d681b02
lut[0x9b] = 0x67dd4acc  lut[0xbb] = 0x5cb36a04  lut[0xdb] = 0x11010b5c  lut[0xfb] = 0x2a6f2b94
lut[0x9c] = 0xf9b9df6f  lut[0xbc] = 0xc2d7ffa7  lut[0xdc] = 0x8f659eff  lut[0xfc] = 0xb40bbe37
lut[0x9d] = 0x8ebeeff9  lut[0xbd] = 0xb5d0cf31  lut[0xdd] = 0xf862ae69  lut[0xfd] = 0xc30c8ea1
lut[0x9e] = 0x17b7be43  lut[0xbe] = 0x2cd99e8b  lut[0xde] = 0x616bffd3  lut[0xfe] = 0x5a05df1b
lut[0x9f] = 0x60b08ed5  lut[0xbf] = 0x5bdeae1d  lut[0xdf] = 0x166ccf45  lut[0xff] = 0x2d02ef8d
```

Figure B.2: CRCT(8) LUT entries (2 of 2).

B.2 Reduced Table Look-up Algorithm

An implementation of the Reduced Table Look-up Algorithm (CRCR) is shown in Algorithm 3.3. Figure B.3 displays the LUT contents generated by Algorithm 3.6.

```
lut[0x00] = 0xedb88320  lut[0x08] = 0x3b83984b  lut[0x10] = 0xe1351b80  lut[0x18] = 0xed59b63b
lut[0x01] = 0x76dc4190  lut[0x09] = 0xf0794f05  lut[0x11] = 0x709a8dc0  lut[0x19] = 0x9b14583d
lut[0x02] = 0x3b6e20c8  lut[0x0a] = 0x958424a2  lut[0x12] = 0x384d46e0  lut[0x1a] = 0xa032af3e
lut[0x03] = 0x1db71064  lut[0x0b] = 0x4ac21251  lut[0x13] = 0x1c26a370  lut[0x1b] = 0x5019579f
lut[0x04] = 0x0edb8832  lut[0x0c] = 0xc8d98a08  lut[0x14] = 0x0e1351b8  lut[0x1c] = 0xc5b428ef
lut[0x05] = 0x076dc419  lut[0x0d] = 0x646cc504  lut[0x15] = 0x0709a8dc  lut[0x1d] = 0x8f629757
lut[0x06] = 0xee0e612c  lut[0x0e] = 0x32366282  lut[0x16] = 0x0384d46e  lut[0x1e] = 0xaa09c88b
lut[0x07] = 0x77073096  lut[0x0f] = 0x191b3141  lut[0x17] = 0x01c26a37  lut[0x1f] = 0xb8bc6765
```

Figure B.3: CRCR(32) LUT entries.

B.3 Slicing-by-4 Algorithm

An implementation of the Slicing-by-4 Algorithm (CRCS4) is shown in Algorithm 3.4. Figures B.4, B.5, B.6, B.7, B.8, and B.9 display the LUT contents for LUT_56, LUT_48, and LUT_40, all generated by Algorithm 3.7. The entries for LUT_32 can be found in Figures B.1 and B.2.

```
lut[0x00] = 0x00000000  lut[0x20] = 0xa032af3e  lut[0x40] = 0x9b14583d  lut[0x60] = 0x3b26f703
lut[0x01] = 0xb8bc6765  lut[0x21] = 0x188ec85b  lut[0x41] = 0x23a83f58  lut[0x61] = 0x839a9066
lut[0x02] = 0xaa09c88b  lut[0x22] = 0x0a3b67b5  lut[0x42] = 0x311d90b6  lut[0x62] = 0x912f3f88
lut[0x03] = 0x12b5afee  lut[0x23] = 0xb28700d0  lut[0x43] = 0x89a1f7d3  lut[0x63] = 0x299358ed
lut[0x04] = 0x8f629757  lut[0x24] = 0x2f503869  lut[0x44] = 0x1476cf6a  lut[0x64] = 0xb4446054
lut[0x05] = 0x37def032  lut[0x25] = 0x97ec5f0c  lut[0x45] = 0xaccaa80f  lut[0x65] = 0x0cf80731
lut[0x06] = 0x256b5fdc  lut[0x26] = 0x8559f0e2  lut[0x46] = 0xbe7f07e1  lut[0x66] = 0x1e4da8df
lut[0x07] = 0x9dd738b9  lut[0x27] = 0x3de59787  lut[0x47] = 0x06c36084  lut[0x67] = 0xa6f1cfba

lut[0x08] = 0xc5b428ef  lut[0x28] = 0x658687d1  lut[0x48] = 0x5ea070d2  lut[0x68] = 0xfe92dfec
lut[0x09] = 0x7d084f8a  lut[0x29] = 0xdd3ae0b4  lut[0x49] = 0xe61c17b7  lut[0x69] = 0x462eb889
lut[0x0a] = 0x6fbde064  lut[0x2a] = 0xcf8f4f5a  lut[0x4a] = 0xf4a9b859  lut[0x6a] = 0x549b1767
lut[0x0b] = 0xd7018701  lut[0x2b] = 0x7733283f  lut[0x4b] = 0x4c15df3c  lut[0x6b] = 0xec277002
lut[0x0c] = 0x4ad6bfb8  lut[0x2c] = 0xeae41086  lut[0x4c] = 0xd1c2e785  lut[0x6c] = 0x71f048bb
lut[0x0d] = 0xf26ad8dd  lut[0x2d] = 0x525877e3  lut[0x4d] = 0x697e80e0  lut[0x6d] = 0xc94c2fde
lut[0x0e] = 0xe0df7733  lut[0x2e] = 0x40edd80d  lut[0x4e] = 0x7bcb2f0e  lut[0x6e] = 0xdbf98030
lut[0x0f] = 0x58631056  lut[0x2f] = 0xf851bf68  lut[0x4f] = 0xc377486b  lut[0x6f] = 0x6345e755

lut[0x10] = 0x5019579f  lut[0x30] = 0xf02bf8a1  lut[0x50] = 0xcb0d0fa2  lut[0x70] = 0x6b3fa09c
lut[0x11] = 0xe8a530fa  lut[0x31] = 0x48979fc4  lut[0x51] = 0x73b168c7  lut[0x71] = 0xd383c7f9
lut[0x12] = 0xfa109f14  lut[0x32] = 0x5a22302a  lut[0x52] = 0x6104c729  lut[0x72] = 0xc1366817
lut[0x13] = 0x42acf871  lut[0x33] = 0xe29e574f  lut[0x53] = 0xd9b8a04c  lut[0x73] = 0x798a0f72
lut[0x14] = 0xdf7bc0c8  lut[0x34] = 0x7f496ff6  lut[0x54] = 0x446f98f5  lut[0x74] = 0xe45d37cb
lut[0x15] = 0x67c7a7ad  lut[0x35] = 0xc7f50893  lut[0x55] = 0xfcd3ff90  lut[0x75] = 0x5ce150ae
lut[0x16] = 0x75720843  lut[0x36] = 0xd540a77d  lut[0x56] = 0xee66507e  lut[0x76] = 0x4e54ff40
lut[0x17] = 0xcdce6f26  lut[0x37] = 0x6dfcc018  lut[0x57] = 0x56da371b  lut[0x77] = 0xf6e89825

lut[0x18] = 0x95ad7f70  lut[0x38] = 0x359fd04e  lut[0x58] = 0x0eb9274d  lut[0x78] = 0xae8b8873
lut[0x19] = 0x2d111815  lut[0x39] = 0x8d23b72b  lut[0x59] = 0xb6054028  lut[0x79] = 0x1637ef16
lut[0x1a] = 0x3fa4b7fb  lut[0x3a] = 0x9f9618c5  lut[0x5a] = 0xa4b0efc6  lut[0x7a] = 0x048240f8
lut[0x1b] = 0x8718d09e  lut[0x3b] = 0x272a7fa0  lut[0x5b] = 0x1c0c88a3  lut[0x7b] = 0xbc3e279d
lut[0x1c] = 0x1acfe827  lut[0x3c] = 0xbafd4719  lut[0x5c] = 0x81dbb01a  lut[0x7c] = 0x21e91f24
lut[0x1d] = 0xa2738f42  lut[0x3d] = 0x0241207c  lut[0x5d] = 0x3967d77f  lut[0x7d] = 0x99557841
lut[0x1e] = 0xb0c620ac  lut[0x3e] = 0x10f48f92  lut[0x5e] = 0x2bd27891  lut[0x7e] = 0x8be0d7af
lut[0x1f] = 0x087a47c9  lut[0x3f] = 0xa848e8f7  lut[0x5f] = 0x936e1ff4  lut[0x7f] = 0x335cb0ca
```

Figure B.4: CRCS4(32) LUT_56 entries (1 of 2).

```
lut[0x80] = 0xed59b63b  lut[0xa0] = 0x4d6b1905  lut[0xc0] = 0x764dee06  lut[0xe0] = 0xd67f4138
lut[0x81] = 0x55e5d15e  lut[0xa1] = 0xf5d77e60  lut[0xc1] = 0xcef18963  lut[0xe1] = 0x6ec3265d
lut[0x82] = 0x47507eb0  lut[0xa2] = 0xe762d18e  lut[0xc2] = 0xdc44268d  lut[0xe2] = 0x7c7689b3
lut[0x83] = 0xffec19d5  lut[0xa3] = 0x5fdeb6eb  lut[0xc3] = 0x64f841e8  lut[0xe3] = 0xc4caeed6
lut[0x84] = 0x623b216c  lut[0xa4] = 0xc2098e52  lut[0xc4] = 0xf92f7951  lut[0xe4] = 0x591dd66f
lut[0x85] = 0xda874609  lut[0xa5] = 0x7ab5e937  lut[0xc5] = 0x41931e34  lut[0xe5] = 0xe1a1b10a
lut[0x86] = 0xc832e9e7  lut[0xa6] = 0x680046d9  lut[0xc6] = 0x5326b1da  lut[0xe6] = 0xf3141ee4
lut[0x87] = 0x708e8e82  lut[0xa7] = 0xd0bc21bc  lut[0xc7] = 0xeb9ad6bf  lut[0xe7] = 0x4ba87981

lut[0x88] = 0x28ed9ed4  lut[0xa8] = 0x88df31ea  lut[0xc8] = 0xb3f9c6e9  lut[0xe8] = 0x13cb69d7
lut[0x89] = 0x9051f9b1  lut[0xa9] = 0x3063568f  lut[0xc9] = 0x0b45a18c  lut[0xe9] = 0xab770eb2
lut[0x8a] = 0x82e4565f  lut[0xaa] = 0x22d6f961  lut[0xca] = 0x19f00e62  lut[0xea] = 0xb9c2a15c
lut[0x8b] = 0x3a58313a  lut[0xab] = 0x9a6a9e04  lut[0xcb] = 0xa14c6907  lut[0xeb] = 0x017ec639
lut[0x8c] = 0xa78f0983  lut[0xac] = 0x07bda6bd  lut[0xcc] = 0x3c9b51be  lut[0xec] = 0x9ca9fe80
lut[0x8d] = 0x1f336ee6  lut[0xad] = 0xbf01c1d8  lut[0xcd] = 0x842736db  lut[0xed] = 0x241599e5
lut[0x8e] = 0x0d86c108  lut[0xae] = 0xadb46e36  lut[0xce] = 0x96929935  lut[0xee] = 0x36a0360b
lut[0x8f] = 0xb53aa66d  lut[0xaf] = 0x15080953  lut[0xcf] = 0x2e2efe50  lut[0xef] = 0x8e1c516e

lut[0x90] = 0xbd40e1a4  lut[0xb0] = 0x1d724e9a  lut[0xd0] = 0x2654b999  lut[0xf0] = 0x866616a7
lut[0x91] = 0x05fc86c1  lut[0xb1] = 0xa5ce29ff  lut[0xd1] = 0x9ee8defc  lut[0xf1] = 0x3eda71c2
lut[0x92] = 0x1749292f  lut[0xb2] = 0xb77b8611  lut[0xd2] = 0x8c5d7112  lut[0xf2] = 0x2c6fde2c
lut[0x93] = 0xaff54e4a  lut[0xb3] = 0x0fc7e174  lut[0xd3] = 0x34e11677  lut[0xf3] = 0x94d3b949
lut[0x94] = 0x322276f3  lut[0xb4] = 0x9210d9cd  lut[0xd4] = 0xa9362ece  lut[0xf4] = 0x090481f0
lut[0x95] = 0x8a9e1196  lut[0xb5] = 0x2aacbea8  lut[0xd5] = 0x118a49ab  lut[0xf5] = 0xb1b8e695
lut[0x96] = 0x982bbe78  lut[0xb6] = 0x38191146  lut[0xd6] = 0x033fe645  lut[0xf6] = 0xa30d497b
lut[0x97] = 0x2097d91d  lut[0xb7] = 0x80a57623  lut[0xd7] = 0xbb838120  lut[0xf7] = 0x1bb12e1e

lut[0x98] = 0x78f4c94b  lut[0xb8] = 0xd8c66675  lut[0xd8] = 0xe3e09176  lut[0xf8] = 0x43d23e48
lut[0x99] = 0xc048ae2e  lut[0xb9] = 0x607a0110  lut[0xd9] = 0x5b5cf613  lut[0xf9] = 0xfb6e592d
lut[0x9a] = 0xd2fd01c0  lut[0xba] = 0x72cfaefe  lut[0xda] = 0x49e959fd  lut[0xfa] = 0xe9dbf6c3
lut[0x9b] = 0x6a4166a5  lut[0xbb] = 0xca73c99b  lut[0xdb] = 0xf1553e98  lut[0xfb] = 0x516791a6
lut[0x9c] = 0xf7965e1c  lut[0xbc] = 0x57a4f122  lut[0xdc] = 0x6c820621  lut[0xfc] = 0xccb0a91f
lut[0x9d] = 0x4f2a3979  lut[0xbd] = 0xef189647  lut[0xdd] = 0xd43e6144  lut[0xfd] = 0x740cce7a
lut[0x9e] = 0x5d9f9697  lut[0xbe] = 0xfdad39a9  lut[0xde] = 0xc68bceaa  lut[0xfe] = 0x66b96194
lut[0x9f] = 0xe523f1f2  lut[0xbf] = 0x45115ecc  lut[0xdf] = 0x7e37a9cf  lut[0xff] = 0xde0506f1
```

Figure B.5: CRCS4(32) LUT_56 entries (2 of 2).

```
lut [0x00] = 0x00000000  lut [0x20] = 0x384d46e0  lut [0x40] = 0x709a8dc0  lut [0x60] = 0x48d7cb20
lut [0x01] = 0x01c26a37  lut [0x21] = 0x398f2cd7  lut [0x41] = 0x7158e7f7  lut [0x61] = 0x4915a117
lut [0x02] = 0x0384d46e  lut [0x22] = 0x3bc9928e  lut [0x42] = 0x731e59ae  lut [0x62] = 0x4b531f4e
lut [0x03] = 0x0246be59  lut [0x23] = 0x3a0bf8b9  lut [0x43] = 0x72dc3399  lut [0x63] = 0x4a917579
lut [0x04] = 0x0709a8dc  lut [0x24] = 0x3f44ee3c  lut [0x44] = 0x7793251c  lut [0x64] = 0x4fde63fc
lut [0x05] = 0x06cbc2eb  lut [0x25] = 0x3e86840b  lut [0x45] = 0x76514f2b  lut [0x65] = 0x4e1c09cb
lut [0x06] = 0x048d7cb2  lut [0x26] = 0x3cc03a52  lut [0x46] = 0x7417f172  lut [0x66] = 0x4c5ab792
lut [0x07] = 0x054f1685  lut [0x27] = 0x3d025065  lut [0x47] = 0x75d59b45  lut [0x67] = 0x4d98dda5

lut [0x08] = 0x0e1351b8  lut [0x28] = 0x365e1758  lut [0x48] = 0x7e89dc78  lut [0x68] = 0x46c49a98
lut [0x09] = 0x0fd13b8f  lut [0x29] = 0x379c7d6f  lut [0x49] = 0x7f4bb64f  lut [0x69] = 0x4706f0af
lut [0x0a] = 0x0d9785d6  lut [0x2a] = 0x35dac336  lut [0x4a] = 0x7d0d0816  lut [0x6a] = 0x45404ef6
lut [0x0b] = 0x0c55efe1  lut [0x2b] = 0x3418a901  lut [0x4b] = 0x7ccf6221  lut [0x6b] = 0x448224c1
lut [0x0c] = 0x091af964  lut [0x2c] = 0x3157bf84  lut [0x4c] = 0x798074a4  lut [0x6c] = 0x41cd3244
lut [0x0d] = 0x08d89353  lut [0x2d] = 0x3095d5b3  lut [0x4d] = 0x78421e93  lut [0x6d] = 0x400f5873
lut [0x0e] = 0x0a9e2d0a  lut [0x2e] = 0x32d36bea  lut [0x4e] = 0x7a04a0ca  lut [0x6e] = 0x4249e62a
lut [0x0f] = 0x0b5c473d  lut [0x2f] = 0x331101dd  lut [0x4f] = 0x7bc6cafd  lut [0x6f] = 0x438b8c1d

lut [0x10] = 0x1c26a370  lut [0x30] = 0x246be590  lut [0x50] = 0x6cbc2eb0  lut [0x70] = 0x54f16850
lut [0x11] = 0x1de4c947  lut [0x31] = 0x25a98fa7  lut [0x51] = 0x6d7e4487  lut [0x71] = 0x55330267
lut [0x12] = 0x1fa2771e  lut [0x32] = 0x27ef31fe  lut [0x52] = 0x6f38fade  lut [0x72] = 0x5775bc3e
lut [0x13] = 0x1e601d29  lut [0x33] = 0x262d5bc9  lut [0x53] = 0x6efa90e9  lut [0x73] = 0x56b7d609
lut [0x14] = 0x1b2f0bac  lut [0x34] = 0x23624d4c  lut [0x54] = 0x6bb5866c  lut [0x74] = 0x53f8c08c
lut [0x15] = 0x1aed619b  lut [0x35] = 0x22a0277b  lut [0x55] = 0x6a77ec5b  lut [0x75] = 0x523aaabb
lut [0x16] = 0x18abdfc2  lut [0x36] = 0x20e69922  lut [0x56] = 0x68315202  lut [0x76] = 0x507c14e2
lut [0x17] = 0x1969b5f5  lut [0x37] = 0x2124f315  lut [0x57] = 0x69f33835  lut [0x77] = 0x51be7ed5

lut [0x18] = 0x1235f2c8  lut [0x38] = 0x2a78b428  lut [0x58] = 0x62af7f08  lut [0x78] = 0x5ae239e8
lut [0x19] = 0x13f798ff  lut [0x39] = 0x2bbade1f  lut [0x59] = 0x636d153f  lut [0x79] = 0x5b2053df
lut [0x1a] = 0x11b126a6  lut [0x3a] = 0x29fc6046  lut [0x5a] = 0x612bab66  lut [0x7a] = 0x5966ed86
lut [0x1b] = 0x10734c91  lut [0x3b] = 0x283e0a71  lut [0x5b] = 0x60e9c151  lut [0x7b] = 0x58a487b1
lut [0x1c] = 0x153c5a14  lut [0x3c] = 0x2d711cf4  lut [0x5c] = 0x65a6d7d4  lut [0x7c] = 0x5deb9134
lut [0x1d] = 0x14fe3023  lut [0x3d] = 0x2cb376c3  lut [0x5d] = 0x6464bde3  lut [0x7d] = 0x5c29fb03
lut [0x1e] = 0x16b88e7a  lut [0x3e] = 0x2ef5c89a  lut [0x5e] = 0x662203ba  lut [0x7e] = 0x5e6f455a
lut [0x1f] = 0x177ae44d  lut [0x3f] = 0x2f37a2ad  lut [0x5f] = 0x67e0698d  lut [0x7f] = 0x5fad2f6d
```

Figure B.6: CRCS4(32) LUT_48 entries (1 of 2).

```
lut[0x80] = 0xe1351b80  lut[0xa0] = 0xd9785d60  lut[0xc0] = 0x91af9640  lut[0xe0] = 0xa9e2d0a0
lut[0x81] = 0xe0f771b7  lut[0xa1] = 0xd8ba3757  lut[0xc1] = 0x906dfc77  lut[0xe1] = 0xa820ba97
lut[0x82] = 0xe2b1cfee  lut[0xa2] = 0xdafc890e  lut[0xc2] = 0x922b422e  lut[0xe2] = 0xaa6604ce
lut[0x83] = 0xe373a5d9  lut[0xa3] = 0xdb3ee339  lut[0xc3] = 0x93e92819  lut[0xe3] = 0xaba46ef9
lut[0x84] = 0xe63cb35c  lut[0xa4] = 0xde71f5bc  lut[0xc4] = 0x96a63e9c  lut[0xe4] = 0xaeeb787c
lut[0x85] = 0xe7fed96b  lut[0xa5] = 0xdfb39f8b  lut[0xc5] = 0x976454ab  lut[0xe5] = 0xaf29124b
lut[0x86] = 0xe5b86732  lut[0xa6] = 0xddf521d2  lut[0xc6] = 0x9522eaf2  lut[0xe6] = 0xad6fac12
lut[0x87] = 0xe47a0d05  lut[0xa7] = 0xdc374be5  lut[0xc7] = 0x94e080c5  lut[0xe7] = 0xacadc625

lut[0x88] = 0xef264a38  lut[0xa8] = 0xd76b0cd8  lut[0xc8] = 0x9fbcc7f8  lut[0xe8] = 0xa7f18118
lut[0x89] = 0xeee4200f  lut[0xa9] = 0xd6a966ef  lut[0xc9] = 0x9e7eadcf  lut[0xe9] = 0xa633eb2f
lut[0x8a] = 0xeca29e56  lut[0xaa] = 0xd4efd8b6  lut[0xca] = 0x9c381396  lut[0xea] = 0xa4755576
lut[0x8b] = 0xed60f461  lut[0xab] = 0xd52db281  lut[0xcb] = 0x9dfa79a1  lut[0xeb] = 0xa5b73f41
lut[0x8c] = 0xe82fe2e4  lut[0xac] = 0xd062a404  lut[0xcc] = 0x98b56f24  lut[0xec] = 0xa0f829c4
lut[0x8d] = 0xe9ed88d3  lut[0xad] = 0xd1a0ce33  lut[0xcd] = 0x99770513  lut[0xed] = 0xa13a43f3
lut[0x8e] = 0xebab368a  lut[0xae] = 0xd3e6706a  lut[0xce] = 0x9b31bb4a  lut[0xee] = 0xa37cfdaa
lut[0x8f] = 0xea695cbd  lut[0xaf] = 0xd2241a5d  lut[0xcf] = 0x9af3d17d  lut[0xef] = 0xa2be979d

lut[0x90] = 0xfd13b8f0  lut[0xb0] = 0xc55efe10  lut[0xd0] = 0x8d893530  lut[0xf0] = 0xb5c473d0
lut[0x91] = 0xfcd1d2c7  lut[0xb1] = 0xc49c9427  lut[0xd1] = 0x8c4b5f07  lut[0xf1] = 0xb40619e7
lut[0x92] = 0xfe976c9e  lut[0xb2] = 0xc6da2a7e  lut[0xd2] = 0x8e0de15e  lut[0xf2] = 0xb640a7be
lut[0x93] = 0xff5506a9  lut[0xb3] = 0xc7184049  lut[0xd3] = 0x8fcf8b69  lut[0xf3] = 0xb782cd89
lut[0x94] = 0xfa1a102c  lut[0xb4] = 0xc25756cc  lut[0xd4] = 0x8a809dec  lut[0xf4] = 0xb2cddb0c
lut[0x95] = 0xfbd87a1b  lut[0xb5] = 0xc3953cfb  lut[0xd5] = 0x8b42f7db  lut[0xf5] = 0xb30fb13b
lut[0x96] = 0xf99ec442  lut[0xb6] = 0xc1d382a2  lut[0xd6] = 0x89044982  lut[0xf6] = 0xb1490f62
lut[0x97] = 0xf85cae75  lut[0xb7] = 0xc011e895  lut[0xd7] = 0x88c623b5  lut[0xf7] = 0xb08b6555

lut[0x98] = 0xf300e948  lut[0xb8] = 0xcb4dafa8  lut[0xd8] = 0x839a6488  lut[0xf8] = 0xbbd72268
lut[0x99] = 0xf2c2837f  lut[0xb9] = 0xca8fc59f  lut[0xd9] = 0x82580ebf  lut[0xf9] = 0xba15485f
lut[0x9a] = 0xf0843d26  lut[0xba] = 0xc8c97bc6  lut[0xda] = 0x801eb0e6  lut[0xfa] = 0xb853f606
lut[0x9b] = 0xf1465711  lut[0xbb] = 0xc90b11f1  lut[0xdb] = 0x81dcdad1  lut[0xfb] = 0xb9919c31
lut[0x9c] = 0xf4094194  lut[0xbc] = 0xcc440774  lut[0xdc] = 0x8493cc54  lut[0xfc] = 0xbcde8ab4
lut[0x9d] = 0xf5cb2ba3  lut[0xbd] = 0xcd866d43  lut[0xdd] = 0x8551a663  lut[0xfd] = 0xbd1ce083
lut[0x9e] = 0xf78d95fa  lut[0xbe] = 0xcfc0d31a  lut[0xde] = 0x8717183a  lut[0xfe] = 0xbf5a5eda
lut[0x9f] = 0xf64fffcd  lut[0xbf] = 0xce02b92d  lut[0xdf] = 0x86d5720d  lut[0xff] = 0xbe9834ed
```

Figure B.7: CRCS4(32) LUT_48 entries (2 of 2).

lut [0x00] = 0x00000000	lut [0x20] = 0x958424a2	lut [0x40] = 0xf0794f05	lut [0x60] = 0x65fd6ba7
lut [0x01] = 0x191b3141	lut [0x21] = 0x8c9f15e3	lut [0x41] = 0xe9627e44	lut [0x61] = 0x7ce65ae6
lut [0x02] = 0x32366282	lut [0x22] = 0xa7b24620	lut [0x42] = 0xc24f2d87	lut [0x62] = 0x57cb0925
lut [0x03] = 0x2b2d53c3	lut [0x23] = 0xbea97761	lut [0x43] = 0xdb541cc6	lut [0x63] = 0x4ed03864
lut [0x04] = 0x646cc504	lut [0x24] = 0xf1e8e1a6	lut [0x44] = 0x94158a01	lut [0x64] = 0x0191aea3
lut [0x05] = 0x7d77f445	lut [0x25] = 0xe8f3d0e7	lut [0x45] = 0x8d0ebb40	lut [0x65] = 0x188a9fe2
lut [0x06] = 0x565aa786	lut [0x26] = 0xc3de8324	lut [0x46] = 0xa623e883	lut [0x66] = 0x33a7cc21
lut [0x07] = 0x4f4196c7	lut [0x27] = 0xdac5b265	lut [0x47] = 0xbf38d9c2	lut [0x67] = 0x2abcfd60
lut [0x08] = 0xc8d98a08	lut [0x28] = 0x5d5daeaa	lut [0x48] = 0x38a0c50d	lut [0x68] = 0xad24e1af
lut [0x09] = 0xd1c2bb49	lut [0x29] = 0x44469feb	lut [0x49] = 0x21bbf44c	lut [0x69] = 0xb43fd0ee
lut [0x0a] = 0xfaefe88a	lut [0x2a] = 0x6f6bcc28	lut [0x4a] = 0x0a96a78f	lut [0x6a] = 0x9f12832d
lut [0x0b] = 0xe3f4d9cb	lut [0x2b] = 0x7670fd69	lut [0x4b] = 0x138d96ce	lut [0x6b] = 0x8609b26c
lut [0x0c] = 0xacb54f0c	lut [0x2c] = 0x39316bae	lut [0x4c] = 0x5ccc0009	lut [0x6c] = 0xc94824ab
lut [0x0d] = 0xb5ae7e4d	lut [0x2d] = 0x202a5aef	lut [0x4d] = 0x45d73148	lut [0x6d] = 0xd05315ea
lut [0x0e] = 0x9e832d8e	lut [0x2e] = 0x0b07092c	lut [0x4e] = 0x6efa628b	lut [0x6e] = 0xfb7e4629
lut [0x0f] = 0x87981ccf	lut [0x2f] = 0x121c386d	lut [0x4f] = 0x77e153ca	lut [0x6f] = 0xe2657768
lut [0x10] = 0x4ac21251	lut [0x30] = 0xdf4636f3	lut [0x50] = 0xbabb5d54	lut [0x70] = 0x2f3f79f6
lut [0x11] = 0x53d92310	lut [0x31] = 0xc65d07b2	lut [0x51] = 0xa3a06c15	lut [0x71] = 0x362448b7
lut [0x12] = 0x78f470d3	lut [0x32] = 0xed705471	lut [0x52] = 0x888d3fd6	lut [0x72] = 0x1d091b74
lut [0x13] = 0x61ef4192	lut [0x33] = 0xf46b6530	lut [0x53] = 0x91960e97	lut [0x73] = 0x04122a35
lut [0x14] = 0x2eaed755	lut [0x34] = 0xbb2af3f7	lut [0x54] = 0xded79850	lut [0x74] = 0x4b53bcf2
lut [0x15] = 0x37b5e614	lut [0x35] = 0xa231c2b6	lut [0x55] = 0xc7cca911	lut [0x75] = 0x52488db3
lut [0x16] = 0x1c98b5d7	lut [0x36] = 0x891c9175	lut [0x56] = 0xece1fad2	lut [0x76] = 0x7965de70
lut [0x17] = 0x05838496	lut [0x37] = 0x9007a034	lut [0x57] = 0xf5facb93	lut [0x77] = 0x607eef31
lut [0x18] = 0x821b9859	lut [0x38] = 0x179fbcfb	lut [0x58] = 0x7262d75c	lut [0x78] = 0xe7e6f3fe
lut [0x19] = 0x9b00a918	lut [0x39] = 0x0e848dba	lut [0x59] = 0x6b79e61d	lut [0x79] = 0xfefdc2bf
lut [0x1a] = 0xb02dfadb	lut [0x3a] = 0x25a9de79	lut [0x5a] = 0x4054b5de	lut [0x7a] = 0xd5d0917c
lut [0x1b] = 0xa936cb9a	lut [0x3b] = 0x3cb2ef38	lut [0x5b] = 0x594f849f	lut [0x7b] = 0xcccba03d
lut [0x1c] = 0xe6775d5d	lut [0x3c] = 0x73f379ff	lut [0x5c] = 0x160e1258	lut [0x7c] = 0x838a36fa
lut [0x1d] = 0xff6c6c1c	lut [0x3d] = 0x6ae848be	lut [0x5d] = 0x0f152319	lut [0x7d] = 0x9a9107bb
lut [0x1e] = 0xd4413fdf	lut [0x3e] = 0x41c51b7d	lut [0x5e] = 0x243870da	lut [0x7e] = 0xb1bc5478
lut [0x1f] = 0xcd5a0e9e	lut [0x3f] = 0x58de2a3c	lut [0x5f] = 0x3d23419b	lut [0x7f] = 0xa8a76539

Figure B.8: CRCS4(32) LUT_40 entries (1 of 2).

```
lut[0x80] = 0x3b83984b  lut[0xa0] = 0xae07bce9  lut[0xc0] = 0xcbfad74e  lut[0xe0] = 0x5e7ef3ec
lut[0x81] = 0x2298a90a  lut[0xa1] = 0xb71c8da8  lut[0xc1] = 0xd2e1e60f  lut[0xe1] = 0x4765c2ad
lut[0x82] = 0x09b5fac9  lut[0xa2] = 0x9c31de6b  lut[0xc2] = 0xf9ccb5cc  lut[0xe2] = 0x6c48916e
lut[0x83] = 0x10aecb88  lut[0xa3] = 0x852aef2a  lut[0xc3] = 0xe0d7848d  lut[0xe3] = 0x7553a02f
lut[0x84] = 0x5fef5d4f  lut[0xa4] = 0xca6b79ed  lut[0xc4] = 0xaf96124a  lut[0xe4] = 0x3a1236e8
lut[0x85] = 0x46f46c0e  lut[0xa5] = 0xd37048ac  lut[0xc5] = 0xb68d230b  lut[0xe5] = 0x230907a9
lut[0x86] = 0x6dd93fcd  lut[0xa6] = 0xf85d1b6f  lut[0xc6] = 0x9da070c8  lut[0xe6] = 0x0824546a
lut[0x87] = 0x74c20e8c  lut[0xa7] = 0xe1462a2e  lut[0xc7] = 0x84bb4189  lut[0xe7] = 0x113f652b

lut[0x88] = 0xf35a1243  lut[0xa8] = 0x66de36e1  lut[0xc8] = 0x03235d46  lut[0xe8] = 0x96a779e4
lut[0x89] = 0xea412302  lut[0xa9] = 0x7fc507a0  lut[0xc9] = 0x1a386c07  lut[0xe9] = 0x8fbc48a5
lut[0x8a] = 0xc16c70c1  lut[0xaa] = 0x54e85463  lut[0xca] = 0x31153fc4  lut[0xea] = 0xa4911b66
lut[0x8b] = 0xd8774180  lut[0xab] = 0x4df36522  lut[0xcb] = 0x280e0e85  lut[0xeb] = 0xbd8a2a27
lut[0x8c] = 0x9736d747  lut[0xac] = 0x02b2f3e5  lut[0xcc] = 0x674f9842  lut[0xec] = 0xf2cbbce0
lut[0x8d] = 0x8e2de606  lut[0xad] = 0x1ba9c2a4  lut[0xcd] = 0x7e54a903  lut[0xed] = 0xebd08da1
lut[0x8e] = 0xa500b5c5  lut[0xae] = 0x30849167  lut[0xce] = 0x5579fac0  lut[0xee] = 0xc0fdde62
lut[0x8f] = 0xbc1b8484  lut[0xaf] = 0x299fa026  lut[0xcf] = 0x4c62cb81  lut[0xef] = 0xd9e6ef23

lut[0x90] = 0x71418a1a  lut[0xb0] = 0xe4c5aeb8  lut[0xd0] = 0x8138c51f  lut[0xf0] = 0x14bce1bd
lut[0x91] = 0x685abb5b  lut[0xb1] = 0xfdde9ff9  lut[0xd1] = 0x9823f45e  lut[0xf1] = 0x0da7d0fc
lut[0x92] = 0x4377e898  lut[0xb2] = 0xd6f3cc3a  lut[0xd2] = 0xb30ea79d  lut[0xf2] = 0x268a833f
lut[0x93] = 0x5a6cd9d9  lut[0xb3] = 0xcfe8fd7b  lut[0xd3] = 0xaa1596dc  lut[0xf3] = 0x3f91b27e
lut[0x94] = 0x152d4f1e  lut[0xb4] = 0x80a96bbc  lut[0xd4] = 0xe554001b  lut[0xf4] = 0x70d024b9
lut[0x95] = 0x0c367e5f  lut[0xb5] = 0x99b25afd  lut[0xd5] = 0xfc4f315a  lut[0xf5] = 0x69cb15f8
lut[0x96] = 0x271b2d9c  lut[0xb6] = 0xb29f093e  lut[0xd6] = 0xd7626299  lut[0xf6] = 0x42e6463b
lut[0x97] = 0x3e001cdd  lut[0xb7] = 0xab84387f  lut[0xd7] = 0xce7953d8  lut[0xf7] = 0x5bfd777a

lut[0x98] = 0xb9980012  lut[0xb8] = 0x2c1c24b0  lut[0xd8] = 0x49e14f17  lut[0xf8] = 0xdc656bb5
lut[0x99] = 0xa0833153  lut[0xb9] = 0x350715f1  lut[0xd9] = 0x50fa7e56  lut[0xf9] = 0xc57e5af4
lut[0x9a] = 0x8bae6290  lut[0xba] = 0x1e2a4632  lut[0xda] = 0x7bd72d95  lut[0xfa] = 0xee530937
lut[0x9b] = 0x92b553d1  lut[0xbb] = 0x07317773  lut[0xdb] = 0x62cc1cd4  lut[0xfb] = 0xf7483876
lut[0x9c] = 0xddf4c516  lut[0xbc] = 0x4870e1b4  lut[0xdc] = 0x2d8d8a13  lut[0xfc] = 0xb809aeb1
lut[0x9d] = 0xc4eff457  lut[0xbd] = 0x516bd0f5  lut[0xdd] = 0x3496bb52  lut[0xfd] = 0xa1129ff0
lut[0x9e] = 0xefc2a794  lut[0xbe] = 0x7a468336  lut[0xde] = 0x1fbbe891  lut[0xfe] = 0x8a3fcc33
lut[0x9f] = 0xf6d996d5  lut[0xbf] = 0x635db277  lut[0xdf] = 0x06a0d9d0  lut[0xff] = 0x9324fd72
```

Figure B.9: CRCS4(32) LUT_40 entries (2 of 2).

B.4 Lambda Gamma Algorithm

An implementation of the Lambda Gamma Algorithm (CRCΛΓ) is shown in Algorithm 4.1. Figures B.10a and B.10a display the Lambda LUT and Gamma LUT contents, respectively. The Lambda LUT entries are generated by Algorithm 4.2, whereas the Gamma LUT entries are obtained directly from the coefficients of the generator polynomial.

```
lut[0x00] = 0x00
lut[0x01] = 0x06
lut[0x02] = 0x09
lut[0x03] = 0x0a
lut[0x04] = 0x0c
lut[0x05] = 0x10
lut[0x06] = 0x18
lut[0x07] = 0x19
lut[0x08] = 0x1a
lut[0x09] = 0x1c
lut[0x0a] = 0x1d
lut[0x0b] = 0x1e
lut[0x0c] = 0x1f
```

(a)

```
lut[0x00] = 0x00
lut[0x01] = 0x01
lut[0x02] = 0x02
lut[0x03] = 0x04
lut[0x04] = 0x05
lut[0x05] = 0x07
lut[0x06] = 0x08
lut[0x07] = 0x0a
lut[0x08] = 0x0b
lut[0x09] = 0x0c
lut[0x0a] = 0x10
lut[0x0b] = 0x16
lut[0x0c] = 0x17
lut[0x0d] = 0x1a
```

(b)

Figure B.10: CRCΛΓ(32) LUT entries: (a) Λ LUT, (b) Γ LUT.

B.5 On-the-Fly Algorithm

To implement the On-the-Fly Algorithm (CRCF), the designer must use bit operations to realize the parallel expressions in software. Figure B.11 illustrates the parallel expressions for CRCF(8).

s'_0	s'_1	s'_2	s'_3	s'_4	s'_5	s'_6	s'_7	s'_8	s'_9	s'_{10}	s'_{11}	s'_{12}	s'_{13}	s'_{14}	s'_{15}	s'_{16}	s'_{17}	s'_{18}	s'_{19}	s'_{20}	s'_{21}	s'_{22}	s'_{23}	s'_{24}	s'_{25}	s'_{26}	s'_{27}	s'_{28}	s'_{29}	s'_{30}	s'_{31}
								s_0	s_1	s_2	s_3	s_4	s_5	s_6	s_7	s_8	s_9	s_{10}	s_{11}	s_{12}	s_{13}	s_{14}	s_{15}	s_{16}	s_{17}	s_{18}	s_{19}	s_{20}	s_{21}	s_{22}	s_{23}
t_0	t_0	t_0	t_1	t_0	t_0	t_1	t_0	t_0	t_1	t_0	t_0	t_0	t_1	t_2	t_3	t_0	t_1	t_2	t_3	t_4	t_5	t_0	t_0	t_1	t_2	t_0	t_1	t_2	t_3	t_4	t_5
t_6	t_1	t_1	t_2	t_2	t_1	t_2	t_2	t_1	t_2	t_2	t_1	t_1	t_2	t_3	t_4	t_4	t_5	t_6	t_7				t_1	t_2	t_3	t_3	t_4	t_5	t_6	t_7	
	t_6	t_2	t_3	t_3	t_3	t_4	t_3	t_3	t_4	t_3	t_3	t_2	t_3	t_4	t_5	t_5	t_6	t_7					t_6	t_7		t_4	t_5	t_6	t_7		
	t_7	t_6	t_7	t_4	t_4	t_5	t_5	t_4	t_5	t_5	t_4	t_4	t_5	t_6	t_7											t_6	t_7				
		t_7		t_6	t_5	t_6	t_7					t_5	t_6	t_7																	
					t_6	t_7						t_6	t_7																		
					t_7																										

Figure B.11: CRCF(8) equations.

Appendix C

z-Transform Approach

In this appendix, we review the approach taken in [21] to obtain parallel LFSR2 equations. It consists of first modeling the serial LFSR2 Architecture (see Figure 2.3b) as a discrete-time system to obtain the serial-input/serial-output transfer function using z-transforms. Then the serial-input/multiple-output transfer functions are obtained. The serial-input/multiple-output transfer functions are then generalized to solve for the multiple-input/multiple-output transfer functions and the parallel LFSR2 expressions are subsequently extracted.

The derivation presented in this section takes a slightly different approach than the one presented by the authors of [21]. They introduce a parallel architecture and some additional notations that we have found not to be necessary and tend to distract the reader. By omitting this parallel architecture, we feel the presentation of this approach is more straightforward and easily understood.

Serial Input/Serial Output Transfer Function

In Figure C.1 we reproduce the discrete-time representation of the serial LFSR2 Architecture that is illustrated in Figure 2 in [21] with some small notational changes. In this case, i denotes the iteration number, g_j for $0 \leq j \leq m-1$ are the coefficients of the generator polynomial, and $x[i]$ and $y[i]$ represent the input, and output sequences,

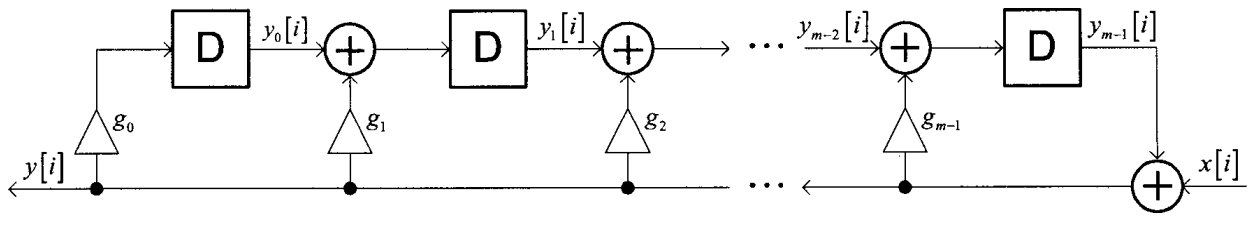


Figure C.1: Discrete-time system illustration of the serial LFSR2 Architecture.

respectively.

From the Figure C.1, one can write equations for the outputs of the delay elements $y_j[i]$ for $0 \leq j \leq m-1$, as

$$\begin{aligned} y_0[i] &= g_0 \cdot y[i-1] \\ y_1[i] &= g_0 \cdot y[i-2] + g_1 \cdot y[i-1] \\ &\vdots \\ y_{m-2}[i] &= g_0 \cdot y[i-(m-1)] + g_1 \cdot y[i-(m-2)] + \cdots + g_{m-2} \cdot y[i-1] \\ y_{m-1}[i] &= g_0 \cdot y[i-m] + g_1 \cdot y[i-(m-1)] + \cdots + g_{m-1} \cdot y[i-1], \end{aligned} \tag{C.1}$$

which can be generalized as[1]

$$y_j[i] = \sum_{k=0}^{j} g_k \cdot y[i-(j+1)+k].$$

Observing that

$$y[i] = x[i] + y_{m-1}[i],$$

and the difference equation of the system is obtained as

$$y[i] + g_{m-1} \cdot y[i-1] + g_{m-2} \cdot y[i-2] + \cdots + g_0 \cdot y[i-m] = x[i]. \tag{C.2}$$

After obtaining the difference equation of the serial LFSR2 Architecture in [21], the authors then proceed to show that the system is linear time-invariant (LTI), and they note that LTI systems can be completely described by their impulse response $h[i]$. The modulo-2 convolution and z-transform operations are then defined as

$$x[i] * h[i] = \sum_{k=-\infty}^{\infty} n[i] \cdot h[i-k]$$

and

$$\mathcal{Z}\{h[i]\} = H(z) = \sum_{i=-\infty}^{\infty} h[i] z^{-i},$$

respectively.

[1] A similar equation in [21] on page 66 is developed using a different approach and has a typo. It should be $y_i(n) = \sum_{k=0}^{i} g_k j(n-i-1+k)$.

Now, taking the z-transform of (C.2), and one obtains

$$\begin{aligned} \mathcal{Z}\{y[i] + g_{m-1} \cdot y[i-1] + g_{m-2} \cdot y[i-2] + \cdots + g_0 \cdot y[i-m]\} &= \mathcal{Z}\{x[i]\} \\ Y(z) + g_{m-1}z^{-1} \cdot Y(z) + g_{m-2}z^{-2} \cdot Y(z) + \cdots + g_0 z^{-m} \cdot Y(z) &= X(z) \\ Y(z) \cdot \left(1 + g_{m-1}z^{-1} + g_{m-2}z^{-2} + \cdots + g_0 z^{-m}\right) &= X(z). \end{aligned} \tag{C.3}$$

From (C.3), the transfer function of the system is obtained as

$$\begin{aligned} H(z) &= \frac{Y(z)}{X(z)} \\ &= \frac{z^m}{z^m + g_{m-1}z^{m-1} + g_{m-2}z^{m-2} + \cdots + g_0} \\ &= \frac{z^m}{G(z)}, \end{aligned} \tag{C.4}$$

where $G(z)$ is the generator polynomial of the system.

Next, consider the k-bit input sequence, $\{x[i]\} = \{x_0, x_1, \cdots, x_{k-1}\}$[2], whose z-transform is

$$\begin{aligned} \mathcal{Z}\{x[i]\} &= X(z) \\ &= x_0 + x_1 z^{-1} + \cdots + x_{k-1} z^{-(k-1)}. \end{aligned}$$

Assuming that the initial content of all the delay elements to be 0s, then the output sequence of the system $y[n]$ with the input $x[n]$ applied is computed from (C.4) as,

$$\begin{aligned} Y(z) &= X(z) \cdot H(z) \\ &= \frac{z^m \cdot \left(x_0 + x_1 z^{-1} + \cdots + x_{k-1} z^{-k+1}\right)}{G(z)}, \end{aligned}$$

which is the quotient of $z^m \cdot \left(x_0 + x_1 z^{-1} + \cdots + x_{k-1} z^{-k+1}\right)$ divided by the generator polynomial, and the remainder is stored in the delay elements [21].

[2]Note that $x[i]$ corresponds to the message bit being inputted to the LFSR at the i-th iteration, i.e., the coefficient of the x^{k-1-i} term of the message polynomial.

Serial Input/Parallel Output Transfer Functions

Define $H_j(z)$ for $0 \leq j \leq m-1$ to be the group of transfer functions from the serial input $x[i]$ in Figure C.1 to the output of the j-th delay element $y_j[i]$, i.e.,

$$\begin{aligned} H_j(z) &= \frac{Y_j(z)}{X(z)} \\ &= \frac{g_0 z^{-(j+1)} \cdot Y(z) + g_1 z^{-(j+1)+1} \cdot Y(z) + \cdots + g_j z^{-1} \cdot Y(z)}{X(z)} \\ &= \left(g_0 z^{-(j+1)} + g_1 z^{-(j+1)+1} + \cdots + g_j z^{-1}\right) \cdot \frac{Y(z)}{X(z)} \\ &= \left(g_0 z^{-(j+1)} + g_1 z^{-(j+1)+1} + \cdots + g_j z^{-1}\right) \cdot \frac{z^m}{G(z)}, \end{aligned} \tag{C.5}$$

where $Y_j(z) = \mathcal{Z}\{y_j[i]\}$ and $y_j[i]$ was defined in (C.1). To find the inverse z-transform of $H_j(z)$, i.e., $h_j[i]$, one can perform the division in (C.5), and to illustrate this concept the authors of [21] provide an example for CRC-16 and $j = 2$. For convenience we reproduce it here:

$$\begin{aligned} H_2(z) &= \frac{z^{15} + z^{13}}{z^{16} + z^{15} + z^2 + 1} \\ &= z^{-1} + z^{-2} + z^{-15} + z^{-29} + \cdots . \end{aligned}$$

Thus, the first 16 values of $h_2[i]$ are

$$\{h_2[i]\} = \{0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1\},$$

and later it will be shown that the first l values of the $H_j(z)$ transfer functions for $0 \leq j \leq m-1$ are sufficient for finding the parallel CRC equations.

Parallel Input/Parallel Output Transfer Functions

Now that the transfer functions $h_j[i]$ have been obtained for the serial input $x[i]$ to the outputs of each delay element $y_j[i]$, the next step is to generalize those transfer functions after l input bits have been processed. Then, the parallel input/parallel output transfer functions can be obtained along with the desired parallel CRC equations.

From the theory of recursive equations, the expression of the z-transform of $y_j[i]$

is

$$\begin{aligned} Y_j(z) &= (X(z) + I(z)) \cdot H_j(z) \\ &= W(z) \cdot H_j(z), \end{aligned} \tag{C.6}$$

where $I(z)$ represents the contribution due to the initial contents of the delay elements and $W(z) = X(z) + I(z)$. It can be determined by observing Figure C.1 that

$$I(z) = \sum_{k=0}^{m-1} y_{m-1-k}[0] z^{-k}.$$

Thus, one can obtain $W(z)$ as[3]

$$W(z) = \sum_{k=0}^{m-1} (x[k] + y_{m-1-k}[0]) z^{-k} + \sum_{k=m}^{\infty} x[k] z^{-k},$$

and the inverse z-transform of $W(z)$ can be computed as

$$\begin{aligned} \mathcal{Z}^{-1}\{W(z)\} &= w[i] \\ &= \begin{cases} x[i] + y_{m-1-i}[0], & i \leq m \\ x[i], & i > m. \end{cases} \end{aligned}$$

Finally, the discrete-time convolution operation is used to compute the inverse z-transform of (C.6), and the following result is obtained

$$\begin{aligned} y_j[i] &= w[i] * h_j[i] \\ &= \sum_{k=-\infty}^{\infty} w[k] \cdot h_j[i-k]. \end{aligned}$$

Considering the fact that $h_j[i] = 0$ for $i < 0$ (causality) and evaluating the case when $i = l-1$, the parallel CRC equations for degree of parallelism l can be obtained from

$$y_i[l-1] = \sum_{k=0}^{l} w[k] \cdot h_i[l-k].$$

Note that $h_j[i]$ must be developed for $0 \leq i \leq l-1$ using (C.5).

[3] A similar equation in [21] on page 66 is developed using a different approach and has a typo. It should be $W(z) = \sum_{q=0}^{r-1} [x(q) \oplus y_{r-1-q}(0)] z^{-q} \oplus \sum_{q=r}^{\infty} x(q) z^{-q}$.

The approach in [21] does not place any restrictions on the degree of parallelism and results in primitive equations. It is conceivable that this approach could be developed for an LFSR1 formulation. One could go about this by modifying the discrete time system in Figure C.1 for the serial LFSR1 Architecture by swapping the input to the left side and the output to the right side of the delay elements, and following similar steps to obtain the transfer functions.

Appendix D

Literature Errata

In this appendix, we identify and correct the various errors and typos that we noticed in the literature throughout this research project. Most of these mistakes are rather trivial and do not warrant a comment to be published. They are ordered chronologically, with the most recently published articles first.

Complexity Reduction of Constant Matrix Computations

In [65], a complexity reduction reduction scheme is proposed. There exist two small typos in the provided example on page 109 of the paper. The formation of the set C_3 should be

$$C_3 = C_2 \cup \left\{ \begin{array}{ll} (1\,1\,1\,1\,0\,0), & (1\,1\,1\,0\,1\,0) \\ (1\,1\,1\,0\,0\,1), & (0\,1\,0\,0\,0\,0) \\ (1\,1\,0\,0\,1\,0), & (\mathbf{1\,1\,0\,1\,1\,0}) \\ (1\,0\,1\,1\,1\,0), & (0\,1\,1\,1\,1\,0) \\ (0\,0\,1\,1\,1\,1), & (1\,0\,0\,1\,1\,0) \\ (\mathbf{0\,0\,0\,1\,0\,0}) & \end{array} \right\},$$

with the typos marked in boldface. The authors omitted the coordinate $(0\ 0\ 0\ 1\ 0\ 0)$ which is formed from the addition of $(0\ 0\ 1\ 1\ 1\ 0) \in C_2 \cap R_2$ and $(0\ 0\ 1\ 0\ 1\ 0) \in V_3$. The coordinate $(1\ 1\ 0\ 1\ 1\ 0)$ was included in the final position of C_3. However, to be consistent with the other C_i sets, it should appear in the marked position.

High-Speed Parallel CRC Implementation Based on Unfolding, Pipelining, and Retiming

In [10], the claim is made in the introduction that the two parallel architectures, derived from LFSR1 [26] and LFSR2 [22] have equal CPD. We have shown in Chapter 5 that this is not the case, and the CPD of the LFSR1 Architecture is less than or equal to the CPD of the LFSR2 Architecture.

Also, a comparison table is presented at the end of the paper (Table V), and the CPD entries for LFSR2 [22] using the CRC-12, CRC-16, CRC-16 Reverse, and CRC-32 generator polynomials should be increased by one (see Table 5.1 for the correct CPD values). This effectively improves their results. Finally, the column reporting the CPD before applying the tree structure is misleading, because all the LFSR2 wires have delay T_X, therefore all entries in that column should be reduced by one.

Parallel CRC Realization

In [26], the formulation does not match the illustration of the hardware architecture. This point is mentioned in our conference paper [41], and elaborated here. Recalling the example provided in [26]: $P = \{1, 0, 0, 1, 1\} \Longleftrightarrow G(x) = 1 + x^3 + x^4$, and

$$F^4 = \begin{bmatrix} 0 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 \\ 1 & 1 & 1 & 1 \end{bmatrix}, \; X' = \begin{bmatrix} x'_3 \\ x'_2 \\ x'_1 \\ x'_0 \end{bmatrix}, \; X = \begin{bmatrix} x_3 \\ x_2 \\ x_1 \\ x_0 \end{bmatrix}, \; D = \begin{bmatrix} d_3 \\ d_2 \\ d_1 \\ d_0 \end{bmatrix}.$$

From the formulation in the paper $X' = F^4 \cdot X + D$, shown in (4), and one obtains,

$$\begin{aligned} x'_3 &= x_2 + x_1 + x_0 + d_3 \\ x'_2 &= x_3 + x_2 + d_2 \\ x'_1 &= x_3 + x_2 + x_1 + d_1 \\ x'_0 &= x_3 + x_2 + x_1 + x_0 + d_0, \end{aligned} \tag{D.1}$$

which is verified to be correct by our implementations. However, writing the parallel equations from the illustrated architecture (Figure 4 in the paper), one obtains

$$\begin{aligned} x'_0 &= e_{0,0} \cdot x_0 + e_{0,1} \cdot x_1 + e_{0,2} \cdot x_2 + e_{0,3} \cdot x_3 \\ x'_1 &= e_{1,0} \cdot x_0 + e_{1,1} \cdot x_1 + e_{1,2} \cdot x_2 + e_{1,3} \cdot x_3 \\ x'_2 &= e_{2,0} \cdot x_0 + e_{2,1} \cdot x_1 + e_{2,2} \cdot x_2 + e_{2,3} \cdot x_3 \\ x'_3 &= e_{3,0} \cdot x_0 + e_{3,1} \cdot x_1 + e_{2,2} \cdot x_2 + e_{3,3} \cdot x_3, \end{aligned} \tag{D.2}$$

where $e_{r,c}$ is defined to correspond to the entry in F^4 at the r-th row and c-th column [26]. Substituting $e_{r,c}$ into (D.2) and one obtains

$$\begin{aligned} x'_0 &= x_1 + x_2 + x_3 + d_0 \\ x'_1 &= x_0 + x_1 + d_1 \\ x'_2 &= x_0 + x_1 + x_2 + d_2 \\ x'_3 &= x_0 + x_1 + x_2 + x_3 + d_3. \end{aligned} \tag{D.3}$$

The set of equations obtained from the Figure 4 in [26] and shown in (D.3) are different than the set (D.1) obtained from the formulation. We note that Figure 4 in the paper can be used if one adopts our matrix $\mathbf{G}$ conventions, or one can vertically flip the coordinates in the F^4 matrix as well as the X', X, and D vectors.

Generation of Parallel Circuits

In [23], an illustration error is made in Figure 3b. Here, in Figure D.1, we provide the error and its correction. Writing the equations from Figure 2b in [23] and one obtains,

$$\begin{aligned} s'_2 &= s_1 + (s_2 + u) \\ s'_1 &= s_0 \\ s'_0 &= s_2 + u. \end{aligned}$$

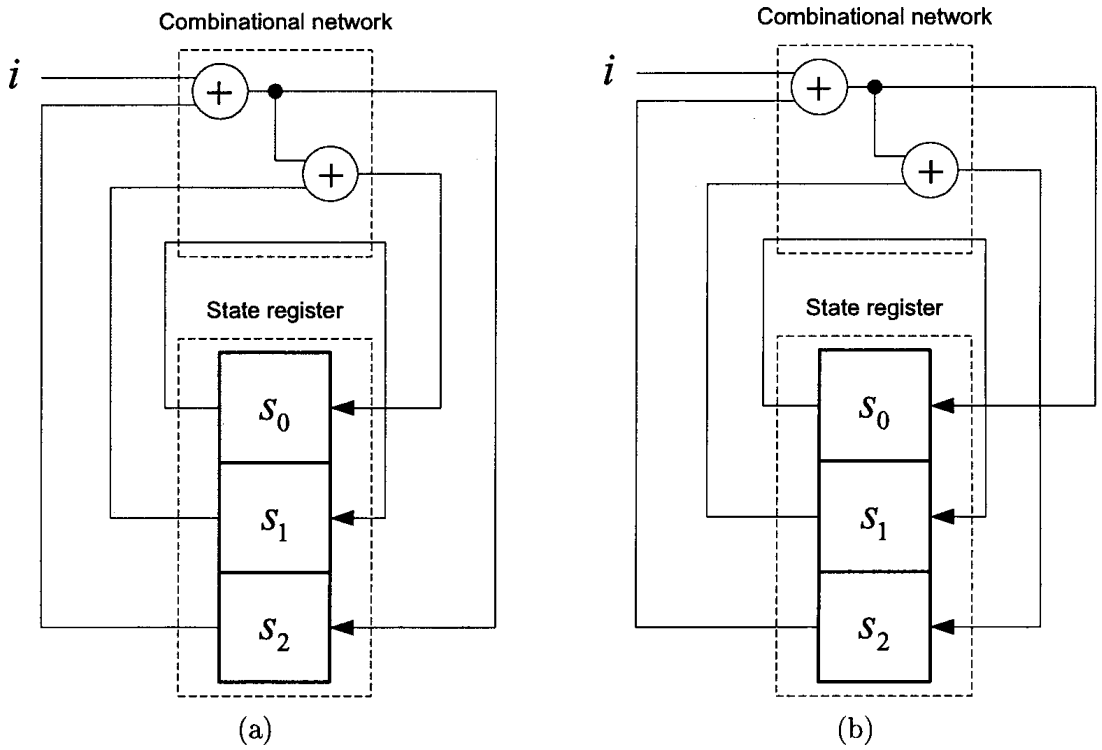


Figure D.1: Cascade literature error: (a) typo, (b) correction.

However, writing the equations from Figure D.1a and one obtains,

$$\begin{aligned} s_2' &= s_2 + i \\ s_1' &= s_0 \\ s_0' &= s_1 + (s_2 + i)\,. \end{aligned}$$

Therefore it is concluded that there exists a wiring typo in that Figure D.1a, and it is corrected in Figure D.1b. We note that the later figures in [23] are correct.

High-Speed CRC Computation Using State-Space Transformations

In [24], the inverse transformation matrix presented at the end of the paper is a typo. This is noted in our conference paper [53], and the correction is also provided in Figure D.2. The reader can easily verify this claim by inputting the coefficients of the matrices into a CAS and performing the matrix multiplication mod2.

$$\begin{bmatrix}
1 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 1 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 1 \\
0 & 0 & 1 & 0 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 1 \\
0 & 1 & 0 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 1 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 1 \\
0 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 1 & 1 & 1 & 0 & 1 & 1 \\
0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 1 & 0 & 1 & 1 \\
0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 1 \\
0 & 0 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 1 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 1 & 1 & 1 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 0 \\
0 & 1 & 1 & 1 & 1 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 0 & 1 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 0 & 0 & 1 & 1 & 1 & 0 \\
0 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 1 & 0 & 0 \\
0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 1 & 1 & 0 & 0 \\
0 & 1 & 0 & 1 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 \\
0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 1 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \\
0 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 \\
0 & 1 & 0 & 0 & 1 & 0 & 1 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 1 & 1 & 1 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 \\
0 & 1 & 1 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 1 & 0 \\
0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 1 & 1 & 1 & 0 & 1 & 0 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 0 & 1 & 0 \\
0 & 1 & 1 & 1 & 0 & 1 & 0 & 1 & 1 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 0 & 1 & 0 & 1 & 0 & 1 \\
0 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 1 & 0 & 1 & 1 & 1 & 0 & 1 & 0 \\
0 & 1 & 1 & 0 & 0 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 & 1 & 0 & 0 & 1 & 1 & 1 & 0 & 0 \\
0 & 1 & 0 & 1 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 1 & 0 & 1 \\
0 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & 0 & 0 & 1 \\
0 & 0 & 1 & 1 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 1 & 0 & 1 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 1 \\
0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 1 & 0 & 1 & 0 & 0 & 1 & 1 & 1 \\
0 & 0 & 1 & 0 & 1 & 1 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1
\end{bmatrix}$$

Figure D.2: State-Space Transformation literature correction: matrix $\mathbf{T}^{-1}$.

Bibliography

[1] W. W. Peterson and D. T. Brown, "Cyclic Codes for Error Detection," *Proceedings of the IRE*, vol. 49, no. 1, pp. 228–235, 1961.

[2] S. Lin and D. J. Costello, *Error Control Coding: Fundamentals and Applications*. Englewood Cliffs, NJ: Prentice-Hall, 1983.

[3] *B-ISDN ATM Layer Specification*. ITU-T Recommendation I.361, 1999.

[4] *IEEE Standard for Information Technology: Carrier Sense Multiple Access with Collision Detection (CSMA/CD) Access Method and Physical Layer Specifications*. ANSI/IEEE Std 802.3-2005.

[5] *Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications*. ANSI/IEEE Std 802.11-1999.

[6] *IEEE Standard for Local and Metropolitan Area Networks: Air Interface for Fixed and Mobile Broadband Wireless Access Systems*. ANSI/IEEE Std 802.16-2004.

[7] M. E. Kounavis and F. L. Berry, "Novel Table Lookup-Based Algorithms for High-Performance CRC Generation," *IEEE Transactions on Computers*, vol. 57, no. 11, pp. 1550–1560, 2008.

[8] T. S. Baicheva, "Determination of the Best CRC Codes with up to 10-bit Redundancy," *IEEE Transactions on Communications*, vol. 56, no. 8, pp. 1214–1220, 2008.

[9] S.-L. Shieh, P.-N. Chen, and Y. Han, "Flip CRC Modification for Message Length Detection," *IEEE Transactions on Communications*, vol. 55, no. 9, pp. 1747–1756, 2007.

[10] C. Cheng and K. K. Parhi, "High-Speed Parallel CRC Implementation Based on Unfolding, Pipelining, and Retiming," *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 53, no. 10, pp. 1017–1021, 2006.

[11] J. Satran, D. Sheinwald, and I. Shimony, "Out of Order Incremental CRC Computation," *IEEE Transactions on Computers*, vol. 54, no. 9, pp. 1178–1181, 2005.

[12] G. D. Nguyen, "Error-Detection Codes: Algorithms and Fast Implementation," *IEEE Transactions on Computers*, vol. 54, no. 1, pp. 1–11, 2005.

[13] T. V. Ramabadran and S. S. Gaitonde, "A Tutorial on CRC Computations," *IEEE Micro*, vol. 8, no. 4, pp. 62–75, 1988.

[14] R. Lidl and H. Niederreiter, *Introduction to Finite Fields and Their Applications*. New York, NY: Cambridge University Press, 1994.

[15] J. Ray and P. Koopman, "Efficient High Hamming Distance CRCs for Embedded Networks," in *the Proceedings of The 2006 International Conference on Dependable Systems and Networks (DSN'06)*, 2006, pp. 3–12.

[16] S. M. Joshi, P. K. Dubey, and M. A. Kaplan, "A New Parallel Algorithm for CRC Generation," in *the Proceedings of The 2000 IEEE International Conference on Communications (ICC 2000)*, vol. 3, 2000, pp. 1764–1768.

[17] M. A. Patel, "A Multi-Channel CRC Register," in *the Proceedings of The Spring Joint Computer Conference*, 1971, pp. 11–14.

[18] A. W. Maholick and R. B. Freeman, "A Universal Cyclic Division Circuit," in *the Proceedings of The Fall Joint Computer Conference*, vol. 39, 1971, pp. 1–8.

[19] A. K. Pandeya and T. J. Cassa, "Parallel CRC Lets Many Lines Use One Circuit," *Computer Design*, vol. 14, no. 9, pp. 87–97, 1975.

[20] M. C. Nielson, "Method for High Speed CRC Computation," *IBM Technical Disclosure Bulletin*, vol. 27, no. 6, pp. 3572–3576, 1984.

[21] G. Albertengo and R. Sisto, "Parallel CRC Generation," *IEEE Micro*, vol. 10, no. 5, pp. 63–71, 1990.

[22] T.-B. Pei and C. Zukowski, "High-Speed Parallel CRC Circuits in VLSI," *IEEE Transactions on Communications*, vol. 40, no. 4, pp. 653–657, 1992.

[23] M. Sprachmann, "Automatic Generation of Parallel CRC Circuits," *IEEE Design and Test of Computers*, vol. 18, no. 3, pp. 108–114, 2001.

[24] J. H. Derby, "High-Speed CRC Computation Using State-Space Transformations," in *the Proceedings of The 2001 IEEE Global Telecommunications Conference (GLOBECOM '01)*, vol. 1, 2001, pp. 166–170.

[25] M.-D. Shieh, M.-H. Sheu, C.-H. Chen, and H.-F. Lo, "A Systematic Approach for Parallel CRC Computations," *Journal of Information Science and Engineering*, vol. 17, no. 3, pp. 445–461, 2001.

[26] G. Campobello, G. Patane, and M. Russo, "Parallel CRC Realization," *IEEE Transactions on Computers*, vol. 52, no. 1, pp. 1312–1319, 2003.

[27] P. Koopman and T. Chakravarty, "Cyclic Redundancy Code (CRC) Polynomial Selection for Embedded Networks," in *the Proceedings of The 2004 International Conference on Dependable Systems and Networks (DSN'04)*, 2004, pp. 145–154.

[28] J. J. Kong and K. K. Parhi, "Interleaved Cyclic Redundancy Check (CRC) Code," in *the Proceedings of The 37th Asilomar Conference on Signals, Systems, and Computers*, vol. 2, 2003, pp. 2137–2141.

[29] G. Castagnoli, J. Ganz, and P. Graber, "Optimum Cyclic Redundancy-Check Codes with 16-bit Redundancy," *IEEE Transactions on Communications*, vol. 38, no. 1, pp. 111–114, 1990.

[30] G. Castagnoli, S. Brauer, and M. Herrmann, "Optimization of Cyclic Redundancy-Check Codes with 24 and 32 Parity Bits," *IEEE Transactions on Communications*, vol. 41, no. 6, pp. 883–892, 1993.

[31] D. Chun and J. K. Wolf, "Special Hardware for Computing the Probability of Undetected Error for Certain Binary CRC Codes and Test Results," *IEEE Transactions on Communications*, vol. 42, no. 10, pp. 2769–2772, 1994.

[32] B. Baicheva, S. Dodunekov, and P. Kazakov, "On the Cyclic Redundancy-Check Codes with 8-bit Redundancy," *Computer Communications*, vol. 21, no. 11, pp. 1030–1033, 1998.

[33] T. Baicheva, S. Dodunekov, and P. Kazakov, "Undetected Error Probability Performance of Cyclic Redundancy-Check Codes of 16-bit Redundancy," *IEE Proceedings Communications*, vol. 147, no. 5, pp. 253–256, 2000.

[34] P. Koopman, "32-bit Cyclic Redundancy Codes for Internet Applications," in *the Proceedings of The 2002 International Conference on Dependable Systems and Networks (DSN'02)*, 2002, pp. 459–468.

[35] D. R. Irvin, "Cyclic Redundancy Checks with Factorable Generators," *IEE Proceedings Communications*, vol. 150, no. 1, pp. 17–20, 2003.

[36] J. E. Mazo and B. R. Saltzberg, "Error-Burst Detection With Tandem CRC's," *IEEE Transactions on Communications*, vol. 39, no. 8, pp. 1175–1178, 1991.

[37] M.-L. Yin and B. Orenstein, "Assessment on Undetectable Burst Errors in Tandem CRCs," in *the Proceedings of The 12th IEEE Pacific Rim International Symposium on Dependable Computing (PRDC'06)*, 2006, pp. 89–96.

[38] T. Mattes, J. Pfahler, F. Schiller, and T. Honold, "Analysis of Combinations of CRC in Industrial Communication," in *the Proceedings of The 26th International Conference on Computer Safety, Reliability, and Security (SAFECOMP 2007)*, vol. 4680. Springer, 2007, pp. 329–341.

[39] S. Shukla and N. W. Bergmann, "Single Bit Error Correction Implementation in CRC-16 on FPGA," in *the Proceedings of The 2004 IEEE International Conference on Field-Programmable Technology (FPT'04)*, 2004, pp. 319–322.

[40] R. J. Glaise and X. Jacquart, "Fast CRC Calculation," in *the Proceedings of The IEEE International Conference on Computer Design: VLSI in Computers and Processors (ICCD'93)*, 1993, pp. 602–605.

[41] C. Kennedy and A. Reyhani-Masoleh, "High-Speed Parallel CRC Circuits," in *in the Proceedings of The 42nd Annual Asilomar Conference on Signals, Systems, and Computers*, 2008.

[42] M. Walma, "Pipelined Cyclic Redundancy Check (CRC) Calculation," in *the Proceedings of The 16th IEEE International Conference on Computer Communications and Networks (ICCCN 2007)*, 2007, pp. 365–370.

[43] Z. Xu, K. Yi, and Z. Liu, "A Universal Algorithm for Parallel CRC Computation and its Implementation," *Journal of Electronics (China)*, vol. 23, no. 4, pp. 528–531, 2006.

[44] M. S. Santina, A. R. Stubberud, and G. H. Hostetter, *Digital Control System Design.* Orlando, FL: Saunders College Publishing, 1994.

[45] R. F. Hobson and K. L. Cheung, "A High-Performance CMOS 32-bit Parallel CRC Engine," *IEEE Journal of Solid-State Circuits*, vol. 34, no. 2, pp. 233–235, 1999.

[46] R. J. Glaise, "A Two-Step Computation of Cyclic Redundancy Code CRC-32 for ATM Networks," *IBM Journal of Research and Development*, vol. 41, no. 6, pp. 705–709, 1997.

[47] F. Monteiro, A. Dandache, A. M'sir, and B. Lepley, "A Fast CRC Implementation on FPGA Using a Pipelined Architecture for the Polynomial Division," in *the Proceedings of The 8th International IEEE Conference on Electronics, Circuits, and Systems (ICECS 2001)*, vol. 3, 2001, pp. 1231–1234.

[48] J. M. N. Serrano, "5x4 Gbps 0.35 Micron CMOS CRC Generator Designed with Standard Cells," in *the Proceedings of The 11th IEEE Mediterranean Electrotechnical Conference (MELECON'02)*, 2002, pp. 215–219.

[49] C. Toal, S. Sezer, X. Yang, K. McLaughlin, D. Burns, and T. Seceleanu, "Progammable CRC Circuit Architecture," in *the Proceedings of The 20th IEEE International SOC Conference (SOCC 2007)*, 2007, pp. 123–126.

[50] A. Sobski and A. Albicki, "Parallel Encoder, Decoder, Detector, Corrector for Cyclic Redundancy Checking," in *the Proceedings of The 1992 IEEE International Symposium on Circuits and Systems (ISCAS 1992)*, vol. 6, 1992, pp. 2945–2948.

[51] ——, "Partitioned and Parallel Cyclic Redundancy Checking," in *the Proceedings of The 36th Midwest Symposium on Circuits and Systems (MWSCAS'93)*, vol. 1, 1993, pp. 538–541.

[52] ——, "High Throughput Error Control Using Parallel CRC," *VLSI Design*, vol. 2, no. 1, pp. 33–50, 1994.

[53] C. Kennedy and A. Reyhani-Masoleh, "High-Speed CRC Computations Using Improved State-Space Transformations," in *the Proceedings of The 2009 IEEE International Conference on Electro/Information Technology (EIT 2009)*, 2009.

[54] J.-S. Lin, C.-K. Lee, M.-D. Shieh, and J.-H. Chen, "High-Speed CRC Design for 10 Gbps Applications," in *the Proceedings of The 2006 IEEE International Symposium on Circuits and Systems (ISCAS 2006)*, 2006, pp. 3177–3180.

[55] R. Lee, "Cyclic Code Redundancy," *Digital Design*, vol. 11, no. 7, pp. 77–85, 1981.

[56] A. Perez, "Byte-Wise CRC Calculations," *IEEE Micro*, vol. 3, no. 3, pp. 40–46, 1983.

[57] G. Griffiths and G. C. Stones, "The Tea-Leaf Reader Algorithm: An Efficient Implementation of CRC-16 and CRC-32," *Communications of the ACM*, vol. 30, no. 7, pp. 617–620, 1987.

[58] D. V. Sarwate, "Computation of Cyclic Redundancy Checks Via Table Look-Up," *Communications of the ACM*, vol. 31, no. 8, pp. 1008–1013, 1988.

[59] J. Crenshaw, "Implementing CRCs," *Embedded Systems Programming*, vol. 5, no. 1, pp. 18–45, 1992.

[60] Y. Do, S.-R. Yoon, T. Kim, K. E. Pyun, and S.-C. Park, "High-speed Parallel Architecture for Software-based CRC," in *the Proceedings of The Fifth Annual IEEE Consumer Communications and Networking Conference (CCNC 2008)*, 2008, pp. 74–78.

[61] D. C. Feldmeier, "Fast Software Implementation of Error Detection Codes," *IEEE/ACM Transactions on Networking*, vol. 3, no. 6, pp. 640–651, 1995.

[62] K. K. Saluja and C.-F. See, "An Efficient Signature Computation Method," *IEEE Design and Test of Computers*, vol. 9, no. 4, pp. 22–26, 1992.

[63] C.-F. See and K. K. Saluja, "An Efficient Method for Computation of Signatures," in *the Proceedings of The 5th International Conference of VLSI Design*, 1992, pp. 245–250.

[64] B. Narendran, M. Franklin, and K. K. Saluja, "Parallel Computation of LFSR Signatures," in *the Proceedings of The 2nd Asian Test Symposium (ATS'93)*, 1993, pp. 75–80.

[65] O. Gustafsson and M. Olofsson, "Complexity Reduction of Constant Matrix Computations over the Binary Field," in *International Workshop on The Arithmetic of Finite Fields (WAIFI 2007)*, vol. 4547. Springer, 2007, pp. 103–115.

[66] C. Paar, "A New Architecture for a Parallel Finite Field Multiplier with Low Complexity Based on Composite Fields," *IEEE Transactions on Computers*, vol. 45, no. 7, pp. 856–861, 1996.

[67] ——, "Optimized Arithmetic for Reed-Solomon Encoders," in *the Proceedings of The 1997 IEEE International Symposium on Information Theory (ISIT 1997)*, 1997, p. 250.

[68] H. Yi, J. Song, S. Park, and C. Park, "Parallel CRC Logic Optimization Algorithm for High Speed Communication Systems," in *the Proceedings of The 10th IEEE International Conference on Communication Systems (ICCS 2006)*, 2006, pp. 1–5.

[69] R. Pasko, P. Schaumont, V. Derudder, S. Vernalde, and D. Durackova, "A New Algorithm for Elimination of Common Subexpressions," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 18, no. 1, pp. 58–68, 1999.

[70] A. Reyhani-Masoleh, "A New Bit-Serial Architecture for Field Multiplication Using Polynomial Bases," in *the Proceedings of Workshop on Cryptographic Hardware and Embedded Systems (CHES 2008)*, vol. 5154. Springer, 2008, pp. 300–314.

[71] S. M. Sait and M. S. K. Tanvir, "VLSI Layout Generation of a Programmable CRC Chip," *IEEE Transactions on Consumer Electronics*, vol. 39, no. 4, pp. 911–916, 1993.

[72] S. M. Sait and W. Hasan, "Hardware Design and VLSI Implementation of a Byte-Wise CRC Generator Chip," *IEEE Transactions on Consumer Electronics*, vol. 41, no. 1, pp. 195–200, 1995.

[73] T. Henriksson, H. Eriksson, U. Nordqvist, P. Larsson-Edefors, and D. Liu, "VLSI Implementation of CRC-32 for 10 Gigabit Ethernet," in *the Proceedings of The 8th International IEEE Conference on Electronics, Circuits, and Systems (ICECS 2001)*, vol. 3, 2001, pp. 1215–1218.

[74] S. H. Li and C. A. Zukowski, "A Self-Timed Cyclic Redundancy Check (CRC) in VLSI," in *the Proceedings of The 40th Midwest Symposium on Circuits and Systems (MWSCAS'97)*, vol. 2, 1997, pp. 1021–1025.

[75] M. Braun, J. Friedrich, T. Grun, and J. Lembert, "Parallel CRC Computations in FPGAs," in *the Proceedings of The 6th International Workshop on Field-Programmable Logic, Smart Applications, New Paradigms and Compilers (FPL'96)*, 1996, pp. 156–165.

[76] J. M. McGuiness and P. J. Naughter, "The Ultimate System Timer v1.2," 10/25 1997. [Online]. Available: http://www.hussar.demon.co.uk/mathszon.htm

[77] N. J. Rubenking, "EndItAll 2: A True Killer App," 10/16 2001. [Online]. Available: http://www.pcmag.com/article2/0,2817,1935,00.asp