ERROR CONTROL WITH BINARY CYCLIC CODES

A thesis submitted to the University of Manchester
for the degree of Doctor of Philosophy
in the Faculty of Engineering and Physical Sciences

2012

By
Martin-Thomas Grymel
School of Computer Science

Contents

Word Count: 39638

List of Tables

List of Figures

List of Algorithms

Abstract

Error-control codes provide a mechanism to increase the reliability of digital data being processed, transmitted, or stored under noisy conditions. Cyclic codes constitute an important class of error-control code, offering powerful error detection and correction capabilities. They can easily be generated and verified in hardware, which makes them particularly well suited to the practical use as error detecting codes.

A cyclic code is based on a generator polynomial which determines its properties including the specific error detection strength. The optimal choice of polynomial depends on many factors that may be influenced by the underlying application. It is therefore advantageous to employ programmable cyclic code hardware that allows a flexible choice of polynomial to be applied to different requirements. A novel method is presented in this thesis to realise programmable cyclic code circuits that are fast, energy-efficient and minimise implementation resources.

It can be shown that the correction of a single-bit error on the basis of a cyclic code is equivalent to the solution of an instance of the discrete logarithm problem. A new approach is proposed for computing discrete logarithms; this leads to a generic deterministic algorithm for analysed group orders that equal Mersenne numbers with an exponent of a power of two. The algorithm exhibits a worst-case runtime in the order of the square root of the group order and constant space requirements.

This thesis establishes new relationships for finite fields that are represented as the polynomial ring over the binary field modulo a primitive polynomial. With a subset of these properties, a novel approach is developed for the solution of the discrete logarithm in the multiplicative groups of these fields. This leads to a deterministic algorithm for small group orders that has linear space and linearithmic time requirements in the degree of defining polynomial, enabling an efficient correction of single-bit errors based on the corresponding cyclic codes.

Declaration

No portion of the work referred to in this thesis has been submitted in support of an application for another degree or qualification of this or any other university or other institute of learning.

Copyright

Acknowledgements

First of all, I would like to express my sincere gratitude to Prof. Stephen B. Furber for offering me the opportunity to join his research group in Manchester and for supervising my work. His confidence led me to explore daunting territory, which at the beginning seemed to be a hopeless endeavour. I'm thankful for the inspiration and guidance that I have received throughout my research project.

I would like to acknowledge Dr. Jim D. Garside for initially suggesting I investigate a CRC circuit for SpiNNaker and for his valuable advice on my hardware queries.

Special thanks go to Dr. Chiara Del Vescovo for sharing her invaluable insights on number theory and abstract algebra, and for being a very good friend.

Throughout the years, I have received great support from many members of the APT group and I am in particular very grateful to Dr. John V. Woods, Dr. Luis A. Plana, Dr. Steve Temple, Dr. David R. Lester, Dr. Simon Davidson, Dr. Barry Cheetham, Dr. Mikel Luján, and Jeff Pepper.

I would like to thank my brave proof-readers Dr. John V. Woods, Dr. Jim D. Garside, Dr. David R. Lester, Dr. Cameron Patterson, and Francesco Galluppi for their valuable input.

I greatly appreciate the help from the High Throughput Computing facility team in the Faculty of Engineering and Physical Sciences, in particular for permitting me to consume huge quantities of processor cycles for my computations.

I acknowledge the support of an Engineering and Physical Sciences Research Council (EPSRC) studentship.

Many thanks to the friends I've made in Manchester; all the fantastic pub and pizza nights will remain unforgotten.

I am very thankful to my family for their constant encouragement and support in all my endeavours.

Finally, I would like to thank Jelo for sweetening and spicing up my life, whilst keeping me sane(ish) during my work towards completing this thesis.

Part I

Introduction

Chapter 1

Introduction

The preservation of digital data integrity is of major concern for computer, communication, and storage systems. In all these applications digital data is susceptible to unintentional modification which may arise from electrical or magnetic disturbance, component failure, or the result of system design error. Depending on the specific application, data failures may result in severe consequences and thus their potential occurrence needs to be considered carefully in the underlying design of the system.

Data reliability can be enhanced through the employment of error-control codes [LC83; PW72; RF89], which provide mechanisms for the detection and correction of errors. These codes enrich the data with redundancy by forming *code words*, which initially can be used to detect inconsistencies in the received data. If the affected data can be retransmitted or recalculated, one simple error correction scheme is the Automatic Repeat Query (ARQ), where the receiver simply requests a retransmission once it detects data inconsistencies. However, where retransmission or recalculation is not feasible, being too slow or uneconomic, Forward Error Correction (FEC) techniques can correct errors on the basis of the corrupted received data and its inherent redundancy.

Cyclic codes form an important class of error-control code offering powerful error detection and correction capabilities. At the same time, their algebraic properties permit the use of simplified processing procedures when compared to non-cyclic codes. For instance, the encoding of data into cyclic code words can easily be achieved in hardware using a simple linear feedback shift register. Likewise, the same circuit can be used to validate code words, and thus detect errors. For these reasons, cyclic codes have been widely adopted as error-detecting codes and commonly deployed in combination with ARQ schemes. This thesis concentrates on cyclic codes with

symbols from the binary field.

If error correction is desired, the most basic case concerns the localisation of a single erroneous bit. It can be shown that for cyclic codes this problem is equivalent to the computation of the discrete logarithm in finite cyclic groups, for which it is widely believed that for the general case no efficient classical algorithm is devisable. This is one reason why the discrete logarithm forms the basis for many cryptographic applications such as the Diffie-Hellman-Merkle key exchange [DH76]. However, it has not been proven that the computation of the discrete logarithm is hard for all groups of practical interest [Odl85; Odl00; McC90; MVO96; Mos96; Buc01; Sti02; Gal12].

Cyclic codes are characterised by an underlying generator polynomial. Different generator polynomials exhibit different capabilities in regard to the detection and correction of errors [TW11; Koo02; KC04]. Certain polynomials, for example, may be particularly well suited to the practical realisation of the error correction process. The selection of polynomial is also dependent on the length of the data that is to be protected and the anticipated error patterns. In systems where diverse applications may favour different cyclic codes, and where the requirements may change over time or be unknown at the design stage, it may be advantageous to provide full flexibility in regard to the usable cyclic code generator polynomials. Efficient cyclic code processing relies on dedicated cyclic code hardware circuits, which may be programmable if the underlying generator polynomial is adaptable. Such a programmable circuit has been created for the SpiNNaker project [FB09; Fur12], a massively parallel spiking neural network simulator.

1.1 Research Objectives

Cyclic codes comprise a powerful class of error-control code; they have gained wide popularity in the field of error detection owing to their efficient hardware implementations and simultaneous effective error detection. Programmable cyclic code circuits have the benefit of flexible adaption of the underlying cyclic code to meet the requirements of a specific application. One goal of this research is to provide a method for the efficient realisation of parallel programmable cyclic code circuits, in hardware, to make them appealing for a wider range of applications.

The current predominating drawback of cyclic codes is the lack of efficient error correction techniques for them. To perform the correction of a single-bit error an instance of the generalised discrete logarithm problem needs to be solved. It is an

additional goal of this research to explore new methods in the quest for an efficient mechanism for computing discrete logarithms in relevant finite cyclic groups and, therefore, facilitate the efficient recovery from single-bit errors through cyclic codes.

1.2 Contribution

The contributions of this thesis are summarised as follows:

- A new scheme is proposed enabling the efficient calculation of the state transition and control matrix for the parallel operation of a cyclic code circuit in hardware. This circuit is used both for the data encoding step and the decoder error detection phase. With the incorporation of a programmable transition and control matrix, this adaptable circuit can be configured to use different cyclic codes. This added flexibility is a valuable enhancement, as the error detection and error correction performance of a particular cyclic code depends on parameters including the length of the data that is to be protected and the anticipated error patterns of an application. A simulation of the novel programmable cyclic code circuit produced shows significant improvements in terms of speed, area and energy efficiency when compared with previously published designs. The design of the new circuit has been published [GF11].
- A new approach is proposed for the computation of discrete logarithms; this leads to a generic deterministic algorithm for analysed group orders that equal a Mersenne number where the exponent is a power of two. It is shown that, for these groups, the worst-case running time is proportional to the square root of the group order, while the space requirements are constant. The scheme is further improved for particular cases where the discrete logarithm values occur with different probabilities, leading to reduced average and worst-case execution times. Furthermore, properties are derived that apply to the sequences that are used by the algorithm.
- A set of new relationships is developed for the field elements of the ring of polynomials over the binary field modulo a primitive polynomial. Based on a subset of these properties, a novel approach is proposed for the computation of discrete logarithms in the cyclic multiplicative group of the finite field. For at least all primitive polynomials up to degree 12 and the first evaluated primitive polynomials of degree 13 and 14, a deterministic algorithm with linear space and linearithmic time requirements in the degree of the polynomial results.

1.3 Thesis Organisation

The remainder of this thesis is structured as follows:

Part II: Background

Chapter 2

This chapter reviews the fundamental principles behind error-control codes and establishes important related terminology. Particular focus is directed towards linear block codes and their subclass of cyclic codes. The error detection and correction capabilities of cyclic codes are described, and it is shown how simple but inefficient error correction mechanisms are realised. Established error correction techniques such as the Meggitt decoder, error-trapping decoding, or the BCH code decoder are briefly reviewed, and their inefficiency concerning the correction of a single-bit error is highlighted. Moreover, it is shown which factors influence the selection of the optimal generator polynomial for a cyclic code.

Chapter 3

In this chapter, the discrete logarithm problem is defined. Information concerning the difficulty of the problem and properties of the underlying finite cyclic groups are presented. Important cryptographic applications that base their operating principle on the presumed difficulty of the discrete logarithm problem in certain groups are specified. Today's most important algorithms for computing discrete logarithms are presented detailing their execution overheads.

Chapter 4

This chapter presents an overview of the SpiNNaker project which targets the large-scale simulation of spiking neural networks. The SpiNNaker architecture facilitates a massively-parallel supercomputer with a million processors, which supports these neural simulations. The issue of anticipated memory faults in a SpiNNaker system of this scale is highlighted, and the usage of cyclic codes as a layer of protection against many of these errors is explained.

Part III: Contributions

Chapter 5

In this chapter, the equations for a parallel realisation of a cyclic code circuit are derived. It is then shown how a programmable version of such a circuit can be created which can be configured to use different generator polynomials. Furthermore, a new scheme is presented that allows the efficient computation of the state transition and control matrix necessary for the circuit. With this scheme a novel programmable parallel cyclic code circuit is proposed that is then compared to previous work. This chapter is based on a journal publication [GF11].

Chapter 6

This chapter describes a new generic approach for the computation of the discrete logarithm. Based on this approach an algorithm is presented for group sizes that equal a Mersenne number with an exponent of a power of two. It is shown how the scheme can be improved if the discrete logarithm values occur with unequal probabilities. Properties are derived for the sequences that are used by the algorithm and finally, the proposed scheme is compared with an established method for the evaluation of discrete logarithms.

Chapter 7

In this chapter, a new set of properties is derived for the elements of a finite field with binary characteristic, where the field is represented as a polynomial ring over the binary field modulo a primitive polynomial. For the multiplicative group of the finite field, a novel approach is presented to compute discrete logarithms based on a subset of the newly established field properties. Performance results are reported for the resulting algorithm for evaluated small group sizes.

Part IV: Conclusion

Chapter 8

This chapter draws conclusions and suggests future directions of research.

Part II

Background

Chapter 2

Error Control Coding

Digital data in computer, communication and storage systems is inevitably subjected to noise and may thus undergo undesired alterations, which may cause entire systems to fail. To lower the probability of such scenarios, it is possible to employ error-control codes that can increase the reliability of data [LC83; PW72; RF89]. For this purpose, the data is sent, in the first place, through an encoding process that will add redundancy to it, before it is processed, transported or stored later on, and thereby exposed to noise. Once the encoded and possibly corrupted data is received by a consumer, the decoding process will attempt to recover the original data. There are two different, but combinable approaches to the decoding process.

On the one hand, the decoder can perform a simple error detection and request a retransmission of the data if inconsistencies are detected; this is known as Automatic Repeat Query (ARQ). This is, however, infeasible if a feedback channel for a retransmission request is not available, or if the data is retrieved from a memory, where it is already stored erroneously. If an ARQ scheme is implementable, it may still be inefficient from a speed or energy point of view, and therefore impractical.

On the other hand, it is possible to employ Forward Error Correction (FEC) techniques that can not only detect, but also correct, errors in the decoder, on the basis of the corrupted data and redundancy only. Error correction proves to be typically more complex in its realisation than pure error detection, but has the significant advantage of managing to recover from certain faults without the need for retransmission.

The setup of an FEC system is shown in Figure 2.1. It is assumed that the source outputs a data message u, which is, in the next step, translated by the encoder into a code word $\bar{u}$, which carries the same information as u but in a redundant form

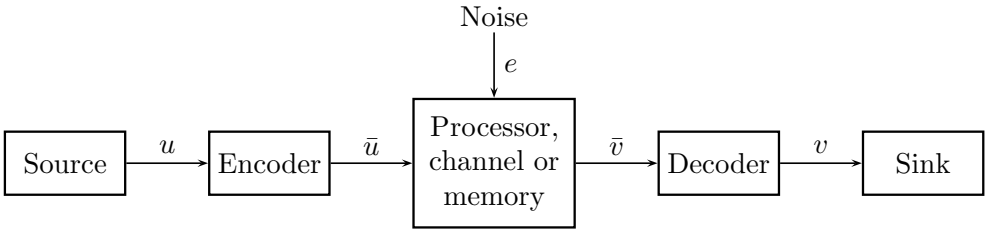

Figure 2.1: Forward error correction in computer, transmission or storage systems.

according to the employed error correction code. The code word $\bar{u}$ is then passed on to a processor, channel or memory, where noise may cause undesired alterations, modelled as e. In this scenario, the processor constitutes a special case as it is assumed to produce an output possibly different from $\bar{u}$ as part of its functionality, which may then, however, differ from the expected output in consequence of the noise exposure. However, the processor case is not further considered here. Instead, it is assumed that $\bar{u}$ is passed on to a communication channel or a storage medium. In both of these cases, it is expected that after a successful transmission or retrieval of the data, without any noise interference, the decoder will receive $\bar{v}$, which will equal $\bar{u}$. If errors occur, it follows that $e \neq 0$ and $\bar{v} = \bar{u} + e$. It is the task of the decoder to translate the received and possibly corrupted code word $\bar{v}$ into the most likely message v. In the ideal case no decoding error is made and it follows $v = u$.

From an information theoretic point of view, the only aspect that can be influenced in the described forward error correction system concerns the encoding and decoding procedure. It was Shannon who published groundbreaking work in this field in 1948 [Sha48a; Sha48b]. He postulated that by choosing an appropriate encoding and decoding strategy, it is possible to reduce the decoding error probability to an arbitrarily small value, as long as the information rate, i.e. the ratio of non-redundant bits to the total number of bits per code word, stays below a certain threshold—the capacity—that is specific to each channel or memory.

The model that is assumed for the transmission or memory channel is called the Binary Symmetric Channel (BSC) and is depicted in Figure 2.2. It has an error probability parameter $p \leq 0.5$, which indicates the probability of an erroneously received symbol.

In what follows, the considered codes are assumed to have symbols from the binary field $GF(2)$, although generalisations to non-binary alphabets can be made. The symbols are represented as 0 and 1, where the addition and multiplication is performed in modulo-2 arithmetic. There are two different types of codes: block and

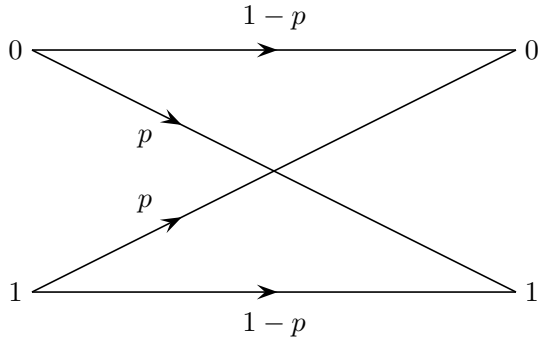

Figure 2.2: Binary symmetric channel with transition error probability p.

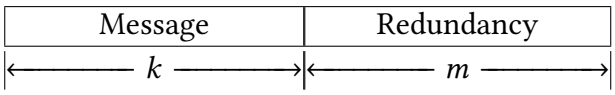

Figure 2.3: Code word in systematic form.

convolutional codes; both encode k-bit input messages into n-bit code words. The difference is that, for convolutional codes, the code word is not only made dependent on the current input message, but also on previous messages, by employing memory in the encoder. Subsequently, however, only block codes are considered.

A code in systematic form has the property that the message bits appear as one section of the code word and the bits for the redundancy as a second section as visualised in Figure 2.3. The number of redundancy bits is denoted by m such that $n = k + m$.

2.1 Linear Block Codes

An (n, k) linear block code is a block code that transforms k-bit message vectors into n-bit code word vectors with the additional property that all the 2^k different code vectors form a k-dimensional subspace of the n-dimensional vector space of all vectors of size n over the binary field. It follows from this definition that the linear combination of code vectors results in a code vector of the same code. An example of a $(7, 4)$ linear block code in systematic form is shown in Table 2.1.

2.1.1 Generator Matrix

Due to the fact that an (n, k) linear block code forms a k-dimensional vector space, it is possible to generate every code vector through a linear combination of k code basis

Table 2.1: A (7, 4) systematic linear block code.

Message	Code word
[0, 0, 0, 0]	[0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 1]	[0, 0, 0, 1, 0, 1, 1]
[0, 0, 1, 0]	[0, 0, 1, 0, 1, 1, 0]
[0, 0, 1, 1]	[0, 0, 1, 1, 1, 0, 1]
[0, 1, 0, 0]	[0, 1, 0, 0, 1, 1, 1]
[0, 1, 0, 1]	[0, 1, 0, 1, 1, 0, 0]
[0, 1, 1, 0]	[0, 1, 1, 0, 0, 0, 1]
[0, 1, 1, 1]	[0, 1, 1, 1, 0, 1, 0]
[1, 0, 0, 0]	[1, 0, 0, 0, 1, 0, 1]
[1, 0, 0, 1]	[1, 0, 0, 1, 1, 1, 0]
[1, 0, 1, 0]	[1, 0, 1, 0, 0, 1, 1]
[1, 0, 1, 1]	[1, 0, 1, 1, 0, 0, 0]
[1, 1, 0, 0]	[1, 1, 0, 0, 0, 1, 0]
[1, 1, 0, 1]	[1, 1, 0, 1, 0, 0, 1]
[1, 1, 1, 0]	[1, 1, 1, 0, 1, 0, 0]
[1, 1, 1, 1]	[1, 1, 1, 1, 1, 1, 1]

vectors g_0 to g_{k-1}, which are aggregated as row vectors in the generator matrix G. A k-bit message vector $u = [u_{k-1}, \ldots, u_0]$ can now be encoded through multiplication with G as

$$\bar{u} = uG = u \begin{bmatrix} g_{k-1} \\ \vdots \\ g_0 \end{bmatrix}.$$

For the (7, 4) linear block code in Table 2.1, the generator matrix constitutes

$$G = \begin{bmatrix} 1 & 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 0 & 1 & 1 & 0 \\ \underbrace{0 & 0 & 0 & 1}_{\text{identity matrix}} & 0 & 1 & 1 \end{bmatrix}.$$

The identity matrix in the left part of the generator matrix indicates the systematic encoding property of this particular code. A message $u = [1, 0, 1, 0]$ can now be

encoded as

$$\bar{u} = uG = [1,0,1,0] \begin{bmatrix} 1 & 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 & 1 \end{bmatrix} = [\underbrace{1,0,1,0}_{u}, \underbrace{0,1,1}_{\text{redundancy}}].$$

2.1.2 Parity-check Matrix

For an (n, k) linear block code C with generator matrix G, it is possible to construct the dual code $C^{\perp}$. This code is an $(n, n-k)$ linear block code and it is the null space of G. The corresponding generator matrix H of $C^{\perp}$ is the *parity-check matrix* of C. It follows that

$$GH^T = 0.$$

Furthermore, for every code vector $\bar{u}$ of C,

$$\bar{u}H^T = 0. \tag{2.1}$$

The parity-check matrix H is thus also a different way of describing the code C. For the $(7, 4)$ linear block code in Table 2.1, the parity-check matrix takes thereby the following shape

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

It can easily be verified that $\bar{u} = [1,0,1,0,0,1,1]$ is a code vector of C since

$$\bar{u}H^T = [1,0,1,0,0,1,1] \begin{bmatrix} 1 & 1 & 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 1 & 0 & 0 & 1 \end{bmatrix}^T = 0.$$

2.1.3 Error Syndrome

The parity-check matrix H of a linear block code not only provides a method to validate code vectors, but also helps in the correction of errors that may have occurred. It is assumed that the code vector $\bar{u}$ gets corrupted with the error vector e such that

$$\bar{v} = \bar{u} + e \tag{2.2}$$

Table 2.2: Single-bit error syndromes.

Error pattern	Error syndrome
$[1, 0, 0, 0, 0, 0, 0]$	$[1, 0, 1]$
$[0, 1, 0, 0, 0, 0, 0]$	$[1, 1, 1]$
$[0, 0, 1, 0, 0, 0, 0]$	$[0, 1, 1]$
$[0, 0, 0, 1, 0, 0, 0]$	$[1, 1, 0]$
$[0, 0, 0, 0, 1, 0, 0]$	$[0, 0, 1]$
$[0, 0, 0, 0, 0, 1, 0]$	$[0, 1, 0]$
$[0, 0, 0, 0, 0, 0, 1]$	$[1, 0, 0]$

is received. The error syndrome of the vector $\bar{v}$ is now defined as

$$s = \bar{v}H^T.$$

With (2.1) and (2.2) it follows further that

$$s = \bar{v}H^T = (\bar{u} + e)H^T = eH^T.$$

The error syndrome s is thus entirely determined by the error vector e. If no error has occurred, $e = 0$, and therefore also the error syndrome $s = 0$. In the case that the error vector coincides with a code vector, the syndrome will also indicate no error. However, in any other case, the syndrome will be different from zero, indicating the occurrence of one or several errors.

Once a nonzero syndrome has been computed, an error correction can be performed if the syndrome can be associated with a most likely error pattern. This can be achieved with a simple but inefficient table-lookup mechanism. For the code in Table 2.1, the error syndromes for all the different single-bit error patterns are given in Table 2.2. It can be seen that the syndromes are all different, which means that every single-bit error can be corrected. Error patterns with more than one bit error are either falsely classified as single-bit errors or go undetected if they correspond to code vectors. The considered code is thus only able to correct all possible single-bit errors.

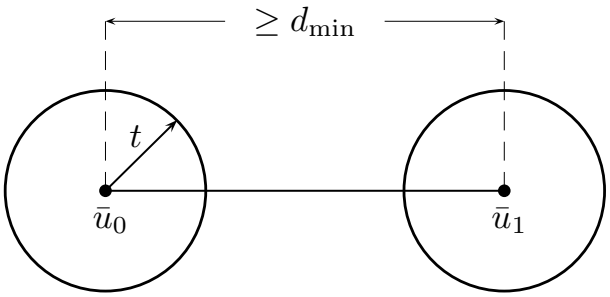

Figure 2.4: Minimum code distance.

2.1.4 Hamming Distance

For a block code C and code vectors $\bar{u}_0, \bar{u}_1 \in C$, the Hamming distance of $\bar{u}_0$ and $\bar{u}_1$, denoted by $d(\bar{u}_0, \bar{u}_1)$, corresponds to the number of components in which $\bar{u}_0$ and $\bar{u}_1$ differ. The Hamming weight of $\bar{u}$, indicated by $w(\bar{u})$, equals the Hamming distance between $\bar{u}$ and the zero code vector, such that $w(\bar{u}) = d(\bar{u}, 0)$. Another important measure is the minimum distance of C, which is defined as the smallest Hamming distance that is achievable between any two distinct code vectors of C and it is denoted by

$$d_{\min} = \min_{\bar{u}_0, \bar{u}_1 \in C} \{d(\bar{u}_0, \bar{u}_1) | \bar{u}_0 \neq \bar{u}_1\}.$$

A special case arises if C is a linear block code, since the minimum distance of C can then be indicated as the minimum weight over all nonzero code vectors of the code. This can be, for instance, attributed to the fact that for a linear block code the linear combination of code vectors results again in a code vector.

2.1.5 Error Detection and Correction Capabilities

The minimum distance $d_{\min}$ of a block code gives an indication of its error detection and correction properties as stated in the following theorem [RF89]:

Theorem 2.1. *For a block code with minimum distance $d_{\min}$, it is guaranteed that up to t errors can be corrected, while at the same time up to d errors can be detected, as long as $t + d + 1 \leq d_{\min}$ and $t \leq d$.*

As an example, it is assumed that the minimum code distance accounts for $d_{\min} = 4$, such that the Hamming distance between any two distinct code vectors $\bar{u}_0$ and $\bar{u}_1$ is at least $d_{\min}$ as illustrated in Figure 2.4. With Theorem 2.1, two different decoder choices can be made. Either a pure error detection is realised, which would be capable of

detecting all error patterns with up to three errors, or a combination of a single error correction with the detection of up to two errors can be targeted.

The error detection and correction capabilities of an (n, k) block code are in general better than stated by Theorem 2.1 on the basis of the minimum distance. If pure error detection is considered, then $2^n - 2^k$ different error patterns are detectable, which is the number of error patterns of length n that do not correspond to any code vectors. For the pure error correction case of a t-error correcting linear block code, the standard array decoding mechanism allows the correction of $2^{n-k} = 2^m$ error patterns, which include all combinations of t or fewer errors [LC83].

2.2 Hamming Codes

A special class of linear block code is formed by *Hamming codes*. These codes exhibit, in basic form, a minimum distance of three, which qualifies them to be used as either single-error correcting or double-error detecting codes. For every number of redundancy bits m with $m \geq 2$, it is possible to construct a Hamming code with a code word length of $n = 2^m - 1$ and $k = 2^m - m - 1$ information bits. An example of a $(7, 4)$ Hamming code is given in Table 2.1 as the minimal weight over all nonzero code word accounts for three.

Hamming codes are considered as perfect codes as they reach the Hamming bound, which means that for a t-error correcting (n, k) block code, the number of all error patterns of t or fewer errors equals the ratio of all possible words to valid code words such that

$$2^m = \sum_{i=0}^{t} \binom{n}{i}.$$

2.3 Cyclic Codes

An (n, k) linear block code C is considered to be *cyclic* if every cyclically shifted code vector results again in a code vector of the same code C [Pra57; PB61; PW72; LC83]. In other words, for every $\bar{u} \in C$ with $\bar{u} = [\bar{u}_{n-1}, \bar{u}_{n-2}, \ldots, \bar{u}_0]$, it must follow that $[\bar{u}_0, \bar{u}_{n-1}, \ldots, \bar{u}_1] \in C$. The $(7, 4)$ linear block code in Table 2.1 serves as an example of a cyclic code.

For the following considerations, it will be convenient to regard word vectors in an equivalent polynomial representation in the indeterminate X in such a way that a

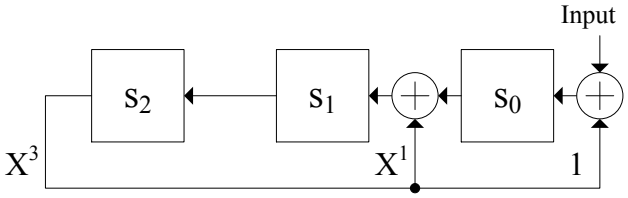

Figure 2.5: LFSR for $p(X) = X^3 + X + 1$.

vector $v = [v_{n-1}, v_{n-2}, \ldots, v_0]$ will correspond to the polynomial

$$v(X) = v_{n-1}X^{n-1} + \cdots + v_1X + v_0.$$

An (n, k) cyclic code C can be characterised through a *generator polynomial* $p(X)$ of degree $m = n - k$, which is a factor of $X^n + 1$ and takes the following shape

$$p(X) = X^m + p_{m-1}X^{m-1} + \cdots + p_1X + 1. \tag{2.3}$$

The code polynomials of C are comprised of all binary polynomials of degree less than n that are divisible by the generator polynomial $p(X)$. For the cyclic code in Table 2.1, the generator polynomial accounts for $p(X) = X^3 + X + 1$.

2.3.1 Encoding

For an (n, k) cyclic code with generator polynomial $p(X)$, a message polynomial $u(X)$ of degree $k - 1$ can be translated into a code polynomial $\bar{u}(X)$, by simply multiplying it with the generator polynomial, such that $\bar{u}(X) = u(X)p(X)$. This guarantees that the code polynomial is divisible by the generator polynomial. The drawback of this method is, however, that the code is not in systematic form. A simple remedy can be provided as follows. If the message polynomial $u(X)$ is premultiplied by X^m, and subsequently divided by $p(X)$ to obtain the remainder $r(X)$, code polynomials in systematic form are produced by adding $r(X)$ to $X^m u(X)$, such that

$$\bar{u}(X) = X^m u(X) + r(X).$$

The described systematic encoding process can easily be translated into hardware by employing a linear feedback shift register (LFSR) in Galois configuration as shown in Figure 2.5. It consists only of storage elements arranged into a circular shift register and XOR gates, which are used to couple the feedback path with different bits from the

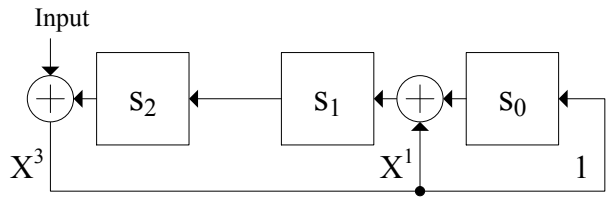

Figure 2.6: LFSR for $p(X) = X^3 + X + 1$ with automatic premultiplication by X^3.

register according to the underlying generator polynomial. To compute the remainder, the LFSR is reset to the zero state and the message is then fed in serially starting from the most significant bit u_{k-1}. After all the message bits have entered the circuit, m additional shifts are performed with the input set to zero to simulate the multiplication by X^m. As a result, the state of the LFSR corresponds to the remainder, which can then simply be appended to the message to form the code word.

The encoding process can be simplified by combining the input with the most significant register bit to form the feedback as shown in Figure 2.6. This modification allows the omission of the last m register shifts with zero input.

A different modification concerns the length of the generated code words. Based on an (n, k) cyclic code, it is possible to derive a *shortened cyclic code* with a length reduced by l. This can be achieved by considering the most significant message bits u_{k-l} to u_{k-1} to be always set to zero. The resulting code is, however, not cyclic anymore, but is an $(n - l, k - l)$ linear block code.

2.3.2 Decoding

The first step in the decoding of a cyclic code comprises the detection of errors. For this purpose the error syndrome is computed, which corresponds, in the case of a cyclic code, to the remainder of the division of the received polynomial and the generator polynomial. Since code polynomials are multiples of the generator polynomial $p(X)$, a syndrome, and therefore remainder, of zero indicates the detection of no errors. For all other syndrome cases, it is certain that errors have occurred.

The computation of the syndrome can easily be achieved by reusing the encoding circuit shown in Figure 2.5. Once the state of the circuit is reset to zero, the entire received polynomial is shifted into the circuit. The register will subsequently hold the syndrome. Alternatively, the circuit shown in Figure 2.6 can be used, with the difference that the obtained value will correspond to the syndrome after m further register shifts with zero input, due to the automatic premultiplication by X^m. This

does not present any complication as the modification to the syndrome is easily reversible if necessary. For the case that no error is detected, the modified syndrome will also be zero. The efficient generation and validation of cyclic codes, employing the same circuit, contributed, to a great extent, to the popularity of cyclic codes in the field of error detection. In this context, a cyclic code is often referred to as a Cyclic Redundancy Checksum (CRC).

If error correction is considered, the situation is slightly different. The most basic case concerns the localisation and correction of a single-bit error and is considered in what follows. Under the assumption of a single-bit error occurrence at bit position j, where $0 \leq j \leq n-1$, a sent code polynomial $\bar{u}(X)$ is received as

$$\bar{v}(X) = \bar{u}(X) + X^j.$$

Since $\bar{u}(X)$ is divisible by $p(X)$, the syndrome $s(X)$ of $\bar{v}(X)$ will equal the remainder of the division of X^j by $p(X)$, such that

$$X^j \equiv s(X) \pmod{p(X)}. \tag{2.4}$$

From (2.4) it is apparent that a maximum number of $2^m - 1$ single-bit errors for a cyclic code are correctable, if the sequence of powers of X modulo the generator polynomial $p(X)$ has a length of $2^m - 1$. This is, at the same time, the maximum length that such a sequence can achieve with the available m bits. Zero cannot be part of the sequence if $m > 1$ as the least significant coefficient of the generator polynomial is one according to (2.3). The sequence is referred to as a maximum-length sequence and the generator polynomials that induce such sequences are called primitive polynomials. Primitive polynomials of degree m can be used to generate $(2^m - 1, 2^m - m - 1)$ cyclic codes, as every irreducible polynomial of degree m is a factor of $X^{2^m-1} + 1$ [Gol82; LN86]. These codes form a subclass of Hamming codes, which are characterised through their cyclic property.

A single-bit-error-correction mechanism for a cyclic or shortened cyclic code needs to deduce the bit error location j from $s(X)$ in (2.4) to correct the erroneous bit. This computation is an instance of the generalised discrete logarithm problem, which is discussed in more detail with state-of-the-art algorithms in Chapter 3. There exist two very straightforward, but inefficient, solutions to this problem which form the basis of many error correction techniques. The first approach employs a table-lookup mechanism, which associates each single-bit error syndrome with the corresponding

bit error location j. This method is very fast, once it is set up, but impractical with regard to the memory requirements. At the other extreme, it is possible to search through all the powers of X in (2.4); this requires a minimum amount of memory but needs, in the worst case, n steps until completion. Predominantly qualified for this task is the LFSR, which traverses the power sequence through its shift operations.

Established error correction mechanisms for cyclic codes are based on the Meggitt decoder [Meg61]. It computes the error syndrome and uses it subsequently to decode the message bits in a serial manner. An error-pattern detection circuit is required that needs to sense all error syndromes that correspond to correctable error patterns that have been shifted to the high order bit positions. The complexity of the decoder is thus, on the one hand, determined by the number and characteristic of error patterns that are to be corrected. On the other hand, if a single-bit error is to be corrected, the Meggitt decoder requires, in the worst case, another n steps after the syndrome has been calculated, as it can easily be verified that its operation is equivalent to the exhaustive search method described earlier. It starts its search from the syndrome, which is shifted inside an LFSR until a predefined value in the sequence is reached—signaled by the error detection circuit—, so that the error position can be deduced and the error corrected.

An improvement to the Meggitt decoder is constituted by the error-trapping decoder [RM64; Kas64; LC83], which drastically reduces the complexity of the error detection circuit for some cases. Its operation is based on the fact that if the error bits can be cyclically shifted into the m least significant bit positions of the received vector, the corresponding syndrome will coincide with those m bits of the error pattern. It can be further shown that if the code is capable of correcting t errors, the weight of the syndrome will be t or less only if the error bits are located in the m least significant bits of the received vector. It follows that the error detection circuit can be simplified since it needs to look out only for syndromes with a weight of t or less. The drawback of the error-trapping method in comparison to the Meggitt decoder is, however, that the error bits of the correctable error patterns need to be confined to m consecutive bit positions, including the case where the bits wrap from the most significant to the least significant end of the error vector. Also, as is the case with the Meggitt decoder, in the worst case n steps are required to locate and correct a single-bit error.

2.3.3 Error Detection and Correction Capabilities

Since cyclic codes form a subclass of linear block codes, they inherit the error detection and correction capabilities from their superclass. These capabilities are described in Subsection 2.1.5. In addition, an (n, k) cyclic code generated by $p(X)$ with degree $m = n - k$ is capable of detecting any error pattern where the erroneous bits are confined to m or fewer consecutive bit positions, which includes the case of the erroneous bits wrapping from the most to the least significant bit position of the error vector [LC83; PW72]. These error patterns are referred to as cyclic burst errors of length m or less, which cover also the single-bit error case. The probability that a cyclic burst error of length $m + 1$ goes undetected is $2^{-(m-1)}$. For cyclic burst errors of length l with $l > m + 2$, the probability of undetectability is 2^{-m}.

If restrictions are put on the generator polynomial $p(X)$, further properties can be derived. The code generated by a generator polynomial $p(X)$ that does not divide $x^i + 1$ for any integer i with $1 \leq i \leq l$, detects all double bit errors that are not more than l bit positions apart [TW11]. In particular, all double bit errors are detected for $l = n - 1$. If $(x + 1)$ is a factor of $p(X)$, all error patterns with an odd number of bit errors are detected, at the expense of error patterns with an even number of bit errors [TW11; Koo02].

Cyclic codes possess, in general, further error detection capabilities that go beyond the ones stated above. These and their error correction capabilities need to be assessed in detail for each generator polynomial individually, as there are great variations among them [Koo02; KC04].

2.3.4 Generator Selection Considerations

There are many factors upon which the selection of a cyclic code generator polynomial for an application depends. First of all, there may be constraints on the degree of the generator polynomial, for instance, due to block length, information rate or system design specifications.

Secondly, only certain generator polynomials may be qualified for use in some cases, due to differences in the efficiency in which the corresponding encoder and decoder can be implemented. This concerns, particularly, the differences in speed or resource demand in hardware [CW94; Bra+96; Der01; CP06; KRM08; KRM09] or software [Ngu09].

Thirdly, different generator polynomials exhibit different error detection and correction capabilities as outlined in Subsection 2.1.5 and Subsection 2.3.3. In particular, if a generator polynomial is to be used for different sized codes, i.e. different shortened cyclic codes derived from a single cyclic code, it is important to analyse and weigh up the properties of different generator polynomials to select the most advantageous one [Koo02; KC04]. The capabilities of the code need also to be considered in conjunction with the error patterns that are anticipated. Errors that are very likely to occur should be manageable by the selected code. The choice of polynomial can also be influenced by the data that is to be protected, as it may exhibit redundancy that could assist in the error detection and correction process.

Finally, there exist cyclic code classes for which simplified error correction procedures have been devised as, for instance, is the case with BCH codes which are briefly outlined in Section 2.4, or codes that take advantage of the special factorisations of the sequence length in (2.4) [CW94]. Where decoding efficiency is of interest, it may be necessary to restrict the cyclic code search to such classes.

2.4 BCH Codes

An important class of cyclic code, the Bose-Chaudhuri-Hocquenghem (BCH) codes [Hoc59; BRC60], offer a simplified decoding mechanism for random bit errors. A t-error-correcting binary primitive BCH code for a positive integer m, has a code word length of $n = 2^m - 1$ and a number of redundancy bits that does not exceed mt. The generator polynomial $p(X)$ is defined as the polynomial of lowest degree that has $2t$ consecutive powers of a primitive element α of the Galois field $GF(2^m)$ as its roots, such that $g(\alpha^i) = 0$ for $1 \leq i \leq 2t$ [LC83; PW72]. With $\phi_i(X)$ being the minimal polynomial of α^i, the generator polynomial can be computed as the least common multiple of the minimum polynomials of the roots α^{2j+1}, where $0 \leq j \leq t-1$, so that

$$p(X) = \text{lcm}(\phi_1(X), \phi_3(X), \ldots, \phi_{2t-1}(X)).$$

The encoding of a BCH code with generator polynomial $p(X)$ can be accomplished in the same way as for general cyclic codes described in Subsection 2.3.1.

2.4.1 Decoding

The decoding process for BCH codes takes advantage of the roots of the generator polynomial $p(X)$. It is assumed that a code polynomial $\bar{u}(X)$ is corrupted with an error polynomial $e(X)$, such that $\bar{v}(X) = \bar{u}(X) + e(X)$ is received. The errors are assumed to be located at bit positions j_0 to $j_{\nu-1}$ with $0 \leq j_0 < j_1 < \cdots < j_{\nu-1} \leq n-1$, so that

$$e(X) = X^{j_{\nu-1}} + X^{j_{\nu-2}} + \cdots + X^{j_0}.$$

If $\bar{v}(X)$ is evaluated at the roots α^i of $p(X)$ for $1 \leq i \leq 2t$, it is essentially the error polynomial $e(X)$, which is evaluated, since the α^i are also roots of code polynomials passed down from the generator. Alternatively, the syndrome $s(X)$ of $\bar{v}(X)$ can be evaluated at the roots α^i leading to the same result [CS09]. With $S_i = \bar{v}(\alpha^i) = e(\alpha^i) = s(\alpha^i)$, a set of $2t$ equations can be obtained:

$$\begin{aligned}
S_1 &= (\alpha^{j_{\nu-1}})^1 + (\alpha^{j_{\nu-2}})^1 + \cdots + (\alpha^{j_0})^1 \\
S_2 &= (\alpha^{j_{\nu-1}})^2 + (\alpha^{j_{\nu-2}})^2 + \cdots + (\alpha^{j_0})^2 \\
S_3 &= (\alpha^{j_{\nu-1}})^3 + (\alpha^{j_{\nu-2}})^3 + \cdots + (\alpha^{j_0})^3 \\
&\vdots \\
S_{2t} &= (\alpha^{j_{\nu-1}})^{2t} + (\alpha^{j_{\nu-2}})^{2t} + \cdots + (\alpha^{j_0})^{2t}.
\end{aligned}$$

Under the assumption that $\nu \leq t$, i.e. t or fewer errors occurred, a unique solution for the error-location numbers $\beta_i = \alpha^{j_i}$ with $0 \leq i \leq \nu - 1$, for the above set of nonlinear equations is computable. From an error-location number β_i, the actual bit error location j_i can be derived by computing its discrete logarithm, which is touched on in Subsection 2.3.2, and for which more details are provided in Chapter 3.

In the standard decoding procedure for BCH codes, an *error-location polynomial* $\sigma(X)$ is constructed, which is defined as

$$\begin{aligned}
\sigma(X) &= (\beta_{\nu-1}X + 1)(\beta_{\nu-2}X + 1)\cdots(\beta_0 X + 1) \\
&= \sigma_\nu X^\nu + \sigma_{\nu-1}X^{\nu-1} + \cdots + \sigma_0 X^0.
\end{aligned}$$

The polynomial coefficients σ_i for $0 \leq i \leq \nu$ can be computed from the values of S_j, where $1 \leq j \leq 2t$, with the help of the Berlekamp-Massey algorithm [Ber65; Mas69]. Once the error-location polynomial $\sigma(X)$ is determined, its roots will point to the error-location numbers, since a root is just the inverse of an error-location number.

An established procedure for determining the roots of $\sigma(X)$ is Chien's search [Pet60; Chi64]. Its operating principle consists in the successive evaluation of $\sigma(X)$ at all potential error-location numbers α^1 to α^n. For a discovered root α^i in step i with $1 \leq i \leq n$, the error-location number corresponds to α^{n-i}, and it follows that the erroneous bit is located at bit position $n - i$.

If the correction of a single-bit error at bit position j on the basis of the described decoding procedures for BCH codes is considered, the computationally expensive step is performing Chien's search. The error-location number for a single-bit error is readily given by $S_1 = s(\alpha) = \alpha^j$, from which the error-location polynomial $\sigma(X)$ can directly be assembled as $\sigma(X) = (\alpha^j X + 1)$. Chien's search algorithm is then used to determine j essentially by searching through all possible values, which is one of the inefficient methods of determining the discrete logarithm of α^j as described in Subsection 2.3.2. More details on the computation of discrete logarithms are given in Chapter 3.

2.5 Conclusion

Error-control codes provide a means to enhance the reliability of digital data that is exposed to noise during its processing, transmission, or storage. To employ a code, before transmission the sender of the data enriches it with redundancy to form a code word according to the code specification. Once the possibly corrupted code word is received by a consumer, it can use the redundancy to detect and possibly also correct errors.

Cyclic codes from an important class of error-control code, since their algebraic properties allow the use of simple LFSRs to generate and verify code words. Additionally, cyclic codes offer powerful error detection and correction capabilities. The specific capabilities depend not only on the cyclic code generator polynomial, but also on the length of the data that is to be protected. To exploit the full potential of cyclic codes, it may be necessary to provide the flexibility of an adaptable generator polynomial, as in the case of SpiNNaker which is described in more detail in Chapter 4. A novel solution to the creation of efficient programmable cyclic code circuits is proposed in Chapter 5.

The most basic case of error correction concerns the localisation of a single-bit error in the data. If cyclic codes serve as the basis for error control, the correction of a single-bit error requires the computation of the discrete logarithm in relevant groups.

This can be achieved with, for instance, a table-lookup mechanism or the exhaustive search method. However, neither of these methods is efficient. More details on the discrete logarithm problem together with more efficient state-of-the-art algorithms are found in Chapter 3, and two new solutions to this problem for particular groups are proposed in Chapter 6 and Chapter 7.

Chapter 3

Discrete Logarithms

For a generator element α of a finite cyclic group $(G, \cdot)$ of order q, and an element $\beta \in G$, the discrete logarithm is defined as the integer k with $0 \leq k \leq q-1$, such that

$$\alpha^k = \beta, \tag{3.1}$$

and denoted by $k = \log_a \beta$. Finding the unique integer k in the above setting is described by the generalised discrete logarithm problem [MVO96], and is referred to as the discrete logarithm problem in what follows. For general finite cyclic groups G, no efficient classical method for solving the discrete logarithm problem has been reported yet [Odl85; Odl00; McC90; MVO96; Mos96; Buc01; Sti02; Gal12].

It is, however, the case that groups exist for which the discrete logarithm is easily obtainable. If the finite cyclic group $(\mathbb{Z}_n, +)$ under addition modulo n is considered for instance, exponentiation on the basis of the group operation as in (3.1) translates into multiplication in $\mathbb{Z}_n$, such that for a generator $\alpha \in \mathbb{Z}_n$, $\beta \in \mathbb{Z}_n$, and $0 \leq k \leq n-1$, the problem takes the following shape

$$\alpha k \equiv \beta \pmod{n}.$$

Since α is a generator, it has a multiplicative inverse element α^{-1}, and it follows

$$k \equiv \alpha^{-1}\beta \pmod{n}.$$

The computation of the inverse element α^{-1} can easily be accomplished with the help of the extended Euclidean algorithm, since the relation $\gcd(\alpha, n) = 1$ holds for every generator element α of the group $\mathbb{Z}_n$.

As a matter of fact, cyclic groups of the same order are isomorphic to each other, so that every finite cyclic group of order n can be transformed into $\mathbb{Z}_n$ [Gal02; BM77]. This means that, if the isomorphism between a finite cyclic group G of order n and $\mathbb{Z}_n$ can be computed efficiently, the discrete logarithm can also be computed efficiently for G. However, no method is known for determining this isomorphism efficiently for arbitrary groups [Sti02].

The difficulty of the discrete logarithm problem is independent of the choice of the generator element. For a cyclic group G, with generator elements α and $\bar{\alpha}$, it is assumed that logarithms can easily be computed to the base $\bar{\alpha}$. In this case, the logarithm of a $\beta \in G$ to the base α can simply be obtained as

$$\log_\alpha \beta = (\log_{\bar{\alpha}} \beta)(\log_{\bar{\alpha}} \alpha)^{-1} \pmod{q},$$

where q denotes the order of G [MVO96].

A popular choice for a finite cyclic group in computer system applications is the multiplicative group of a finite field of binary characteristic, as its arithmetic can be implemented rather easily in hardware and software [Odl85]. The group can be represented as the polynomial ring over $GF(2)$ modulo an irreducible polynomial $f(X)$ over $GF(2)$, and indicated as $\mathbb{Z}_2[X]/\langle f(X)\rangle$. Different irreducible polynomials $f(X)$ of the same degree induce different representations of one and the same group. However, similarly to the choice of the primitive element, the choice of the irreducible polynomial $f(X)$ does not have any effect on the difficulty of the discrete logarithm problem, as the group representation can be changed in polynomial time [Zie74; Len91].

The inverse operation of the discrete logarithm, discrete exponentiation, is always computable in polynomial time with the square-and-multiply algorithm [Knu97]. This fact, and the wide belief that the computation of the discrete logarithm is impractical for groups of certain representation and order, has led to a number of cryptographic applications that base their operating principle on the discrete logarithm, such as the Diffie-Hellman-Merkle key exchange [DH76], the ElGamal public-key cryptosystem and signature scheme [Elg85], and its variant the Schnorr signature and identification scheme [Sch91].

Algorithms for computing discrete logarithms are subdivided into generic algorithms that do not rely on any special representation of the group elements, and special algorithms that work best only for certain group representations. Furthermore, some algorithms are dependent on the factorisation of the group order q. It has been

shown that the fastest constructable generic algorithm that assumes that each group element has a unique encoding, cannot improve on the time complexity $\Omega(\sqrt{p})$, where p is the largest prime factor of the group order [Nec94; Sho97].

In what follows, the main ideas for the fastest algorithms for the discrete logarithm that have been published are briefly outlined. The algorithms are all of the generic type apart from the index-calculus method.

3.1 Shanks' Algorithm

This algorithm is also known as the baby-step giant-step algorithm [Sha71] and is essentially a time-space trade-off. A group $(G, \cdot)$ of order q with generator element α and an element $\beta \in G$ is considered. The algorithm is based on the fact that a α^k, $0 \leq k \leq q-1$, can be expressed for an m, $1 \leq m \leq q$, as

$$\alpha^k = \alpha^{im+j}$$

with $0 \leq i \leq \left\lceil \frac{q}{m} \right\rceil - 1$ and $0 \leq j \leq m-1$. The algorithm generates, initially, a list of 'giant steps' α^{im} for $0 \leq i \leq \left\lceil \frac{q}{m} \right\rceil - 1$, which are stored sorted in memory and associated with their corresponding exponent factor i. If the logarithm of β to the base α is to be computed, the *baby steps* $\beta\alpha^{-j}$ are evaluated and searched for in the stored list, for each j that satisfies $0 \leq j \leq m-1$. Once a match is found so that $\beta\alpha^{-j} = \alpha^{im}$, the discrete logarithm can be computed as

$$\log_\alpha \beta \equiv im + j \pmod{q}.$$

The pseudocode for the initialisation of Shanks' algorithm is shown in Algorithm 3.1, whereas the pseudocode for the evaluation of a specific discrete logarithm can be found in Algorithm 3.2.

Algorithm 3.1 Shanks' precomputation.

```
1: procedure SHANKS_PRECOMPUTATION(α, q, m)
2:     for i = 0 to ⌈q/m⌉ − 1 do
3:         insert (α^{im}, i) into the memory, sorting pairs by their first component
4:     end for
5: end procedure
```

Algorithm 3.2 Shanks' algorithm.

```
1: function SHANKS(α, β, q, m)
2:     for j = 0 to m − 1 do
3:         if (x, i) ∈ memory with x = βα^{−j} then
4:             return im + j (mod q)
5:         end if
6:     end for
7: end function
```

Considering that a search in the sorted list requires in the worst case $\lceil \log_2 \frac{q}{m} \rceil$ steps, and that in the worst case a maximum number of m *baby steps* need to be evaluated until a match is found, the worst-case running time can be indicated as $m\lceil \log_2 \frac{q}{m} \rceil$. In terms of memory requirements, $\lceil \frac{q}{m} \rceil$ list entries need to be stored. If m is chosen to be set to $m = 1$, all possible powers of α are stored in memory, so that essentially a table-lookup mechanism is realised. The other extreme is achieved if m is set to its upper bound q, which basically results in an exhaustive search algorithm with minimal memory requirements. For the case that $m = \lceil \sqrt{q} \rceil$, the time complexity accounts for $O(\sqrt{q} \log_2 \sqrt{q})$, while the memory complexity accounts for $O(\sqrt{q})$.

3.2 Pollard's Rho Algorithm

Pollard's rho algorithm for discrete logarithms [Pol78] is considered for a group $(G, \cdot)$ of order q with generator element α. For an element $\beta \in G$, the discrete logarithm to the base α is to be determined. The algorithm partitions G into three sets S_0 to S_2 of comparable size, and defines a recursive sequence of elements in G with starting value $x_0 = 1$ and

$$x_{i+1} = \begin{cases} x_i \alpha & \text{if } x_i \in S_0 \\ x_i^2 & \text{if } x_i \in S_1 \\ x_i \beta & \text{if } x_i \in S_2. \end{cases}$$

It needs to be ensured that $1 \notin S_1$, as otherwise all sequence elements would equal 1. To keep track of the manipulations that are applied to the starting value x_0 during the traversal of the sequence, it is possible to record the exponents of α and β in a_i and b_i, respectively, so that x_i can be expressed as

$$x_i = \alpha^{a_i} \beta^{b_i}.$$

The a_i and b_i can be defined recursively as a function of x_i with $a_0 = 0$ and $b_0 = 0$ as follows

$$a_{i+1} = \begin{cases} a_i + 1 \bmod q & \text{if } x_i \in S_0 \\ 2a_i \bmod q & \text{if } x_i \in S_1 \\ a_i & \text{if } x_i \in S_2, \end{cases}$$

and

$$b_{i+1} = \begin{cases} b_i & \text{if } x_i \in S_0 \\ 2b_i \bmod q & \text{if } x_i \in S_1 \\ b_i + 1 \bmod q & \text{if } x_i \in S_2. \end{cases}$$

The algorithm searches now the sequence for a collision of the form $x_i = x_j$, for $i \neq j$, which is simplified for practical reasons to cases where

$$x_i = x_{2i},$$

for $i > 0$. Once such a collision is found, it follows that

$$\alpha^{a_i} \beta^{b_i} = \alpha^{a_{2i}} \beta^{b_{2i}},$$

which can be rewritten as

$$\beta^{b_i - b_{2i}} = \alpha^{a_{2i} - a_i}.$$

Taking the logarithm to the base α on both sides leads to

$$(b_i - b_{2i}) \log_\alpha \beta \equiv a_{2i} - a_i \pmod{q}.$$

If $\gcd(b_i - b_{2i}, q) = 1$, the logarithm can be solved as

$$\log_\alpha \beta \equiv (b_i - b_{2i})^{-1} (a_{2i} - a_i) \pmod{q}.$$

Otherwise, the linear congruence exhibits $\gcd(b_i - b_{2i}, q)$ solutions. If the greatest common divisor is not too large, all the different solutions can be tested until the unique solution is found that corresponds to the discrete logarithm. In the case that the number of solution to the linear congruence is too large, Pollard's rho algorithm can be repeated with a different starting value $x_0 = \alpha^{a_0} \beta^{b_0}$ with $a_0, b_0 \in \mathbb{Z}_q$ [MVO96]. The pseudocode for the algorithm is shown in Algorithm 3.3.

Under the assumption that the sequence behaves like a random mapping on G,

Algorithm 3.3 Pollard's rho algorithm for discrete logarithms.

```
1:  function POLLARD(α, β, q, S_0, S_1, S_2)
2:      globalise α, β, q, S_0, S_1, S_2
3:      {x, a, b} = {1, 0, 0}
4:      {x̄, ā, b̄} = SEQ(x, a, b)
5:      while x ≠ x̄ do
6:          {x, a, b} = SEQ(x, a, b)
7:          {x̄, ā, b̄} = SEQ(x̄, ā, b̄)
8:          {x̄, ā, b̄} = SEQ(x̄, ā, b̄)
9:      end while
10:     if GCD(b − b̄, q) is small then
11:         determine k satisfying (b − b̄)k ≡ (ā − a) (mod q) and α^k = β
12:         return k
13:     else
14:         restart with different starting values {x, a, b}
15:     end if
16: end function
17:
18: function SEQ(x, a, b)
19:     switch x ∈
20:         case S_0 : return { xα, a + 1 mod q, b }
21:         case S_1 : return { x^2, 2a mod q, 2b mod q }
22:         case S_2 : return { xβ, a, b + 1 mod q }
23:     end switch
24: end function
```

it has been shown that the number of operations that the algorithm requires until a collision is found, has an expectation value close to

$$\sqrt{\frac{\pi^5 q}{288}} \approx 1.0308\sqrt{q}.$$

Alternative sequences that improve on the described sequence above have been suggested [Tes00].

3.3 Silver-Pohlig-Hellman Algorithm

The group $(G, \cdot)$ of order q with generator element α is considered. For an element $\beta \in G$, the discrete logarithm $k = \log_\alpha \beta$ is to be determined. The Silver-Pohlig-Hellman algorithm [PH78] simplifies the task by taking into account the factorisation of the group order q. It is assumed that q decomposes into distinct primes p_i, such

that

$$q = \prod_{i=1}^{N} p_i^{e_i}.$$

With this factorisation, the algorithm computes the discrete logarithm k modulo each of the prime powers $p_i^{e_i}$ for $1 \leq i \leq N$. All those values can then be easily combined with the Chinese remainder theorem to the overall solution k.

To compute k modulo $p_i^{e_i}$, the result is considered in the radix p_i expansion as

$$k \equiv \sum_{j=0}^{e_i-1} k_j p_i^j \pmod{p_i^{e_i}},$$

where $0 \leq k_j \leq p_i - 1$. The values k_j are determined one by one in increasing significance starting from k_0. It is now assumed that k_0 to k_{j-1} with $j < e_i$, have already been computed, so that k_j will be determined in the next step. For this reason β_j is introduced as follows

$$\begin{aligned}
\beta_j &= \left(\beta\alpha^{-(k_{j-1}p_i^{j-1}+k_{j-2}p_i^{j-2}+\cdots+k_0)}\right)^{q/p_i^{j+1}} \\
&= \left(\alpha^k\alpha^{-(k_{j-1}p_i^{j-1}+k_{j-2}p_i^{j-2}+\cdots+k_0)}\right)^{q/p_i^{j+1}} \\
&= \left(\alpha^{cp_i^{e_i}+k_{e_i-1}p_i^{e_i-1}+k_{e_i-2}p_i^{e_i-2}+\cdots+k_jp_i^j}\right)^{q/p_i^{j+1}} \\
&= \alpha^{\bar{c}q}\alpha^{(q/p_i)k_j} \\
&= \alpha^{(q/p_i)k_j} \\
&= \alpha_j^{k_j},
\end{aligned}$$

where c and $\bar{c}$ are integers. The coefficient k_j can thus be computed as $k_j = \log_{\alpha_j}\beta_j$, whereby it is apparent from the transformation that α_j has an order of p_i.

It follows that with the Silver-Pohlig-Hellman algorithm, finding the discrete logarithm k in a group G of order q, is reduced to finding discrete logarithms in subgroups of G, whose orders correspond to the different prime factors p_i of q. More precisely, for every prime factor p_i with multiplicity e_i of the group order q, e_i instances of the discrete logarithm need to be solved in the subgroup of order p_i. If the group order q decomposes into different prime powers $p_i^{e_i}$, the Chinese remainder theorem needs to be employed in a last step to combine the intermediate results to the overall solution k.

The operation of the algorithm is summarised in the pseudocode given in Algorithm 3.4. Obvious optimisations to the code have been omitted for enhanced

Algorithm 3.4 Silver-Pohlig-Hellman algorithm.

function SILVER-POHLIG-HELLMAN($\alpha, \beta, q = \prod_{i=1}^{N} p_i^{e_i}$)
  for $i = 1$ **to** N **do**
    for $j = 0$ **to** $e_i - 1$ **do**
      $\beta_j = (\beta \alpha^{-(k_{j-1} p_i^{j-1} + k_{j-2} p_i^{j-2} + \cdots + k_0)})^{q/p_i^{j+1}}$
      $\alpha_j = \alpha^{q/p_i}$
      $k_j = \log_{\alpha_j} \beta_j$
    end for
    $\bar{k}_i = k_{e_i-1} p_i^{e_i-1} + k_{e_i-2} p_i^{e_i-2} + \cdots + k_0$
  end for
  compute k such that $k \equiv \bar{k}_i \pmod{p_i^{e_i}}$ for $1 \leq i \leq N$
  return k
end function

readability. The runtime of the algorithm is on the one hand influenced by the time T_{p_i} that is necessary to evaluate the discrete logarithm in the subgroup of order p_i, for which for instance, Shanks' or Pollard's rho algorithm can be used. On the other hand it is necessary to dedicate time in the order of $\log_2 q$ to compute β_j. Certain calculations for the Chinese remainder theorem can be precomputed for the factorisation of q, so that its execution time can be neglected. In summary, the overall time complexity of the algorithm is given by $O(\sum_{i=1}^{N} e_i(\log_2 q + T_{p_i}))$.

The Silver-Pohlig-Hellman is particularly efficient if the group order q is smooth, that is to say its prime factors are all below a certain threshold, such that the discrete logarithms can easily be evaluated in the corresponding subgroups. However, if the group order q is prime, the method is without effect.

3.4 Index-Calculus Algorithm

The index-calculus method [Odl85; Odl00; Sch+96; MVO96; Sti02; Sch08; Gal12] is a special algorithm for solving discrete logarithms, as its effectiveness depends on properties of the underlying group representation. It is probabilistic and requires a substantial amount of memory but it is, at the same time, the most powerful algorithm that has been devised for computing discrete logarithms. Its main idea is sketched in what follows.

A cyclic group G of order q with the primitive element α is considered. Furthermore, a factor base $\mathcal{B} = \{b_1, b_2, \ldots, b_{|\mathcal{B}|}\}$ is selected, which is a rather small subset of elements in G, such that a large portion of elements in G factorise over $\mathcal{B}$. If an

element factorises over $\mathcal{B}$, it is considered to be smooth with respect to $\mathcal{B}$. During the precomputation phase of the index-calculus method, which is discussed in more detail below, the discrete logarithms to the base α of the elements in the factor base $\mathcal{B}$ are determined.

Under the assumption that the precomputation phase has been completed, a particular discrete logarithm to the base α for an element $\beta \in G$ can be computed as follows. An integer r is chosen at random with $0 \leq r \leq q-1$, until $\beta\alpha^r$ is found to factorise over $\mathcal{B}$, such that

$$\beta\alpha^r = \prod_{i=1}^{|\mathcal{B}|} b_i^{e_i}.$$

If the logarithm to the base α is taken on both sides of the equation, it follows that

$$\log_\alpha \beta \equiv \sum_{i=1}^{|\mathcal{B}|} e_i \log_\alpha b_i - r \pmod{q}.$$

Thus, $\log_\alpha \beta$ can be computed since the logarithms of the elements in the factor base are known.

The precomputation phase is subdivided into a sieving and a linear algebra stage. During the sieving stage, random integers r with $0 \leq r \leq q-1$ are chosen, and used to test if α^r can be composed of elements in $\mathcal{B}$, such that

$$\alpha^r = \prod_{i=1}^{|\mathcal{B}|} b_i^{e_i},$$

which leads to the linear congruence

$$r \equiv \sum_{i=1}^{|\mathcal{B}|} e_i \log_\alpha b_i \pmod{q}.$$

Once enough of these linear relations modulo q have been collected such that the resulting linear system has a unique solution, the linear algebra stage is initiated to compute this solution and thus the logarithms of the elements in the factor base.

If the underlying group is the multiplicative group $\mathbb{F}_q^*$ of a finite field $\mathbb{F}_q$, a more general variant of the index-calculus method is usually considered [Sch+96; Sch08]. Let α be a primitive element of $\mathbb{F}_q^*$. The logarithm to the base α for an element $\beta \in \mathbb{F}_q^*$ is to be determined. Two Dedekind domains R_0 and R_1 need to be selected with corresponding surjective ring homomorphisms $\phi_0 : R_0 \to \mathbb{F}_q^*$ and $\phi_1 : R_1 \to \mathbb{F}_q^*$.

Furthermore, two factor bases $\mathcal{A} = \{\mathfrak{a}_1, \mathfrak{a}_2, \ldots, \mathfrak{a}_{|\mathcal{A}|}\}$ and $\mathcal{B} = \{\mathfrak{b}_1, \mathfrak{b}_2, \ldots, \mathfrak{b}_{|\mathcal{B}|}\}$ are defined that consist of prime ideals of R_0 and R_1, respectively. For α and β, the preimages $a \in R_0$ and $b \in R_1$ under the corresponding ring homomorphisms need to be obtained, such that $\phi_0(a) = \alpha$, $\phi_1(b) = \beta$ and the ideals generated by a and b are smooth in respect to their corresponding factor bases $\mathcal{A}$ and $\mathcal{B}$. It follows that the ideals can be expressed as

$$(a) = \prod_{j=1}^{|\mathcal{A}|} \mathfrak{a}_j{}^{v_j} \quad \text{and} \quad (b) = \prod_{j=1}^{|\mathcal{B}|} \mathfrak{b}_j{}^{w_j}.$$

The algorithm searches now for pairs $(a_i, b_i) \in R_0 \times R_1$, such that

$$\phi_0(a_i) = \phi_1(b_i),$$

$$(a_i) = \prod_{j=1}^{|\mathcal{A}|} \mathfrak{a}_j{}^{v_{i,j}} \quad \text{and} \quad (b_i) = \prod_{j=1}^{|\mathcal{B}|} \mathfrak{b}_j{}^{w_{i,j}}.$$

With a sufficiently large collection of N pairs (a_i, b_i) and the corresponding factorisation of their ideals, the following linear system modulo $q - 1$ is constructed

$$\begin{bmatrix} v_1 & v_{1,1} & v_{2,1} & \cdots & v_{N,1} \\ v_2 & v_{1,2} & v_{2,2} & \cdots & v_{N,2} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ v_{|\mathcal{A}|} & v_{1,|\mathcal{A}|} & v_{2,|\mathcal{A}|} & \cdots & v_{N,|\mathcal{A}|} \\ 0 & w_{1,1} & w_{2,1} & \cdots & w_{N,1} \\ 0 & w_{1,2} & w_{2,2} & \cdots & w_{N,2} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & w_{1,|\mathcal{B}|} & w_{2,|\mathcal{B}|} & \cdots & w_{N,|\mathcal{B}|} \end{bmatrix} X \equiv - \begin{bmatrix} 0 \\ 0 \\ \vdots \\ 0 \\ w_1 \\ w_2 \\ \vdots \\ w_{|\mathcal{B}|} \end{bmatrix} \pmod{q-1}.$$

With a solution $X = [x_0, x_1, x_2, \ldots, x_N]^T$ to the system of linear congruences, the elements

$$\bar{a} = a^{x_0} \prod_{i=1}^{N} a_i^{x_i} \quad \text{and} \quad \bar{b} = b \prod_{i=1}^{N} b_i^{x_i}$$

are defined. Furthermore,

$$\beta\phi_0(\bar{a}) = \beta\alpha^{x_0}\phi_0(\textstyle\prod_{i=1}^{N} a_i^{x_i}) = \alpha^{x_0}\beta\phi_1(\prod_{i=1}^{N} b_i^{x_i}) = \alpha^{x_0}\phi_1(\bar{b}). \tag{3.2}$$

If additionally, both $\bar{a}$ and $\bar{b}$ are assumed to be $(q-1)$st powers, the corresponding ring homomorphisms would map them to the multiplicative identity element in $\mathbb{F}_q$, such that (3.2) would yield

$$\alpha^{x_0} = \beta,$$

where x_0 is the discrete logarithm that is being searched for.

Instead of using R_0 and R_1 for the preimages of α and β, respectively, it is possible to modify the algorithm slightly to locate the preimages in only one of the two Dedekind domains [Sch08]. Also, in contrast to the former described index-calculus variant, the precomputation phase has been integrated with the main computation, however, they can be split if more than one discrete logarithm needs to be computed for the same setup [JL02; Sch05].

It will be convenient to introduce the following notation for the characterisation of the time complexity of the index-calculus method

$$L_q(a, c) = \exp\left((c + o(1))(\ln q)^a(\ln \ln q)^{1-a}\right),$$

which interpolates between polynomial and exponential behaviour as a varies from zero to one. Different variants of the index-calculus method have been devised to deal with different sizes of the multiplicative group of a finite field $\mathbb{F}_q$ with $q = p^m$ and p being a prime number. The time complexity for some of the variants has been proven rigorously, in other cases plausible assumptions are made that lead to algorithms with only conjectured, but better, time complexities.

If rigorously proven algorithms are considered, for $m = 1$ and $p \to \infty$, an expected subexponential running time of $L_q(1/2, \sqrt{2})$ is achievable [Pom87]. The same time complexity can be achieved if $p \leq m^{o(m)}$ for $q \to \infty$ [LP98]. For the special case of $m = 2$ and $p \to \infty$, an index-calculus variant with expected running time of $L_q(1/2, 3/2)$ is constructable [Lov92].

If unproven assumptions are allowed for the runtime analysis, versions of the index-calculus method have been devised that yield a better performance than in the case of the rigorously proven ones. Adleman and DeMarrais have proposed for $p > m$ as $q \to \infty$ an algorithm with conjectured expected running time of $L_q(1/2, 2)$ [Adl79; AD93; AD94]. Thus, if this result is combined with the rigorously proven algorithm for $m = 1$ and $p \to \infty$, a subexponential algorithm for all finite fields with a time complexity of the form $L_q(1/2, c)$ for $q \to \infty$ with $\sqrt{2} \leq c \leq 2$ is obtained.

The most prominent variants of the index-calculus method, which lead to even

better conjectured time complexities, are the number field sieve and the function field sieve. Initially, the number field sieve was introduced for the factorisation of integers [Len+93; Rie94], but has been adapted to deal with the discrete logarithm problem by Gordon [Gor93]. It has been subsequently further studied and improved [Sch93; Sch+96; JL03; Sch05; Jou+06; Sch08]. As long as m does not grow too fast, such that $m \leq o(\log q / \log\log q)^{1/3}$, it has been shown that the following conjectured expected time complexity is achievable

$$L_q(1/3, (64/9)^{1/3}),$$

for $q \to \infty$. For special cases of prime fields, where $p = 2^n \pm 1$, a conjectured expected time complexity of $L_p(1/3, (32/9)^{1/3})$ is obtainable [Sch10].

The function field sieve has been proposed by Adleman [Adl94] and further improved since then [AH99; Sch02; JL02; Gra+04; JL06; Sch08]. For the case that $p \leq m^{o(\sqrt{m})}$ and $q \to \infty$, it has been conjectured that the running time has an expectancy value of

$$L_q(1/3, (32/9)^{1/3}).$$

As a special case of the function field sieve the Coppersmith algorithm [Cop84; Odl85] is considered. It has been devised for finite fields with binary characteristic so that $p = 2$. The conjectured expected value for the running time accounts for $L_{2^m}(1/3, c)$, where $(32/9)^{1/3} \leq c \leq (4)^{1/3}$.

More recently, it has been shown that the gap that existed between the number field sieve and the function field sieve in terms of field sizes, has been closed, so that it is now possible to construct, for all finite fields, an algorithm that runs in conjectured expected running time of $L_q(1/3, O(1))$ for $q \to \infty$ [JL06; Jou+06; Sch08].

The size of the factor base for all of the variants of the index calculus method, is approximately in the order of the corresponding running times, and thus is an important factor that cannot be neglected [Odl85; Sch08; Gal12].

3.5 Conclusion

The interest in the discrete logarithm is two-sided. On one hand, it is assumed that the computation of the discrete logarithm is hard in certain groups and this is exploited in many cryptographic applications. On the other hand, the efficient correction of single-bit errors based on cyclic codes, as described in Chapter 2, requires the discrete

logarithm to be easily computable in relevant groups. These two applications have conflicting interests and research progress in favour of either may have negative implications for the other.

The discrete logarithm problem can efficiently be solved for the finite cyclic group $(\mathbb{Z}_n, +)$ under addition modulo n. However, it is currently unclear as to whether this may also apply to certain other groups. It has been shown that if no special properties of the group element encodings are assumed, the fastest constructable algorithm requires $\Omega(\sqrt{p})$ group operations to compute the discrete logarithm, where p is the largest prime factor of the group order. However, this assumption may not hold for all groups of practical interest, leaving open the possibility of algorithms that improve on the lower bound.

Shank's algorithm is a time-space trade-off and thus is only practically applicable to groups of small order. Pollard's rho algorithm has constant space requirements and an expected runtime in the order of the square root of the group order, if the simulated walk in the underlying group is assumed to be random. The Silver-Pohlig-Hellman algorithm reduces the initial discrete logarithm problem effectively into subgroups, whose orders correspond to the prime factors of the initial group order. However, the most powerful algorithm for the computation of discrete logarithms is the index-calculus method which can be applied to multiplicative groups of finite fields. It is a probabilistic algorithm with an expected subexponential running time and space requirements in the same order.

Two new solutions are proposed to the discrete logarithm problem for certain groups in Chapter 6 and Chapter 7.

Chapter 4

SpiNNaker

The functioning of the human brain still remains a mystery and is under constant investigation [FT07; Fur12]. Within the brain the main cell type that processes information is believed to be the neuron and an average human brain consists of about 10^{11} of these neurons interconnected to form a large neural network [Kan+12; DA01]. Neurons receive input signals from other neurons in the form of electrical impulses; these are often modelled as spike events in artificial neural networks. Within a neuron received spikes may trigger the emission of a new outgoing spike, or a sequence of spikes, which are transmitted to a set of downstream neurons to contribute to their input. It is believed that the *timing* with which neurons emit spikes (fire) is one of the key principles that underpins how information is represented and processed in the brain [TFM96; Mao+01].

A connection between two neurons is established through a synapse, permitting the transmission of signals from one neuron to the other. Each synapse has characteristics which determine how signals are relayed, including their specific timing. The strength of a connection is a further important synaptic property, describing the magnitude of a relayed signal on the receiving neuron, and is often modelled as a single numerical value: the synaptic weight. It is estimated that about 10^{15} synapses can be found in the human brain [Kan+12]; these are dynamic, strengthening or weakening neural connections over time. This dynamism extends to synaptic connections which may be completely removed, or new connections which may arise between neuron pairs. This whole dynamic process that governs the neural network is known as synaptic plasticity and it is believed that it is central to biological processes of a higher level, such as learning or memory.

The understanding of the operating mechanisms of the brain is presently far from

complete. For this reason, the SpiNNaker project [FT07; FB09; Fur12; Fur+12] has been initiated to support the quest to understand how the brain works, and in the hope of finding new and more efficient ways of computing—inspired by biology. It aims to deliver a research platform for the large-scale simulation of arbitrary spiking neural networks and operate in biological real-time. SpiNNaker is a spiking neural supercomputer architecture that is tailored to support the efficient simulation of real-time networks of a billion neurons; the scale and complexity of the simulation depends on the neuron and synapse models used. In the billion neuron case each neuron receives a biologically plausible average input from 1,000 other neurons with a mean neuron firing rate of 10 Hz. Although simulation of spiking neural networks is a fairly specific task, SpiNNaker is not limited to this computational scope, as the following architectural description will make clear.

4.1 Architecture

The SpiNNaker architecture has been devised to facilitate a massively-parallel computing platform consisting of a million processors, primarily for the real-time simulation of large-scale spiking neural networks. With respect to the target machine, particular emphasis has been laid upon a power-efficient and fault-tolerant design. The central element of the architecture is the SpiNNaker chip, which is a custom-designed Multi-Processor System-on-Chip (MPSoC), fabricated on a 130 nm CMOS process [Fur+12]. A 128 MiB off-die mobile Double Data Rate (DDR) Synchronous Dynamic Random Access Memory (SDRAM) is stacked and connected on top of the MPSoC, forming a System-in-Package (SiP), photographed in Figure 4.1. A SpiNNaker die is depicted in Figure 4.2 and a schematic of the SpiNNaker MPSoC in Figure 4.3.

The MPSoC incorporates 18 processing subsystems that are built around low-power ARM968 processing cores with tightly-coupled local memories of 32 KiB for instructions, and 64 KiB for data storage as illustrated in Figure 4.4. Each processing subsystem is also equipped with a Direct Memory Access (DMA) controller that manages block transfers between the local subsystem data and the SDRAMs. In addition to using the SDRAM as the source or target for the DMA data transfers, other shared resources on the chip can transparently be selected such as the System RAM or the System ROM. Each processing subsystem is complemented by an interrupt controller, a communication interface, and two timers/counters.

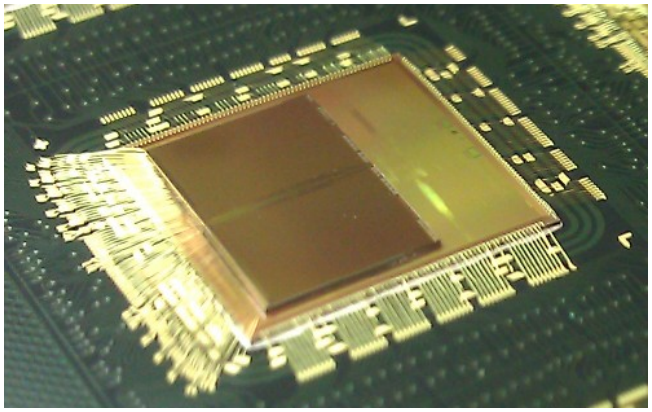

Figure 4.1: SpiNNaker MPSoC with stacked SDRAM on top. 3D packaging by UNISEM (Europe) Ltd.

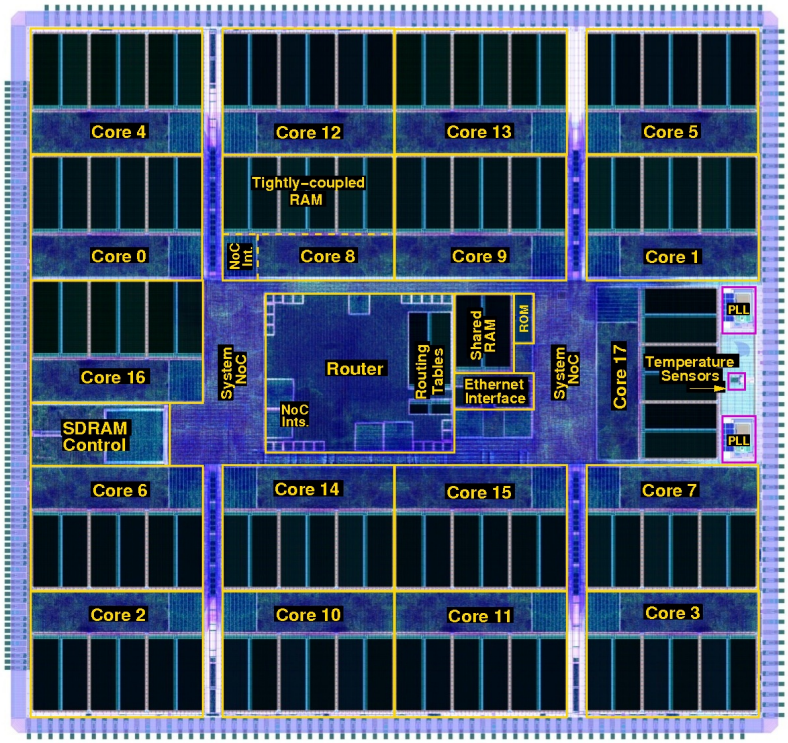

Figure 4.2: SpiNNaker Die.

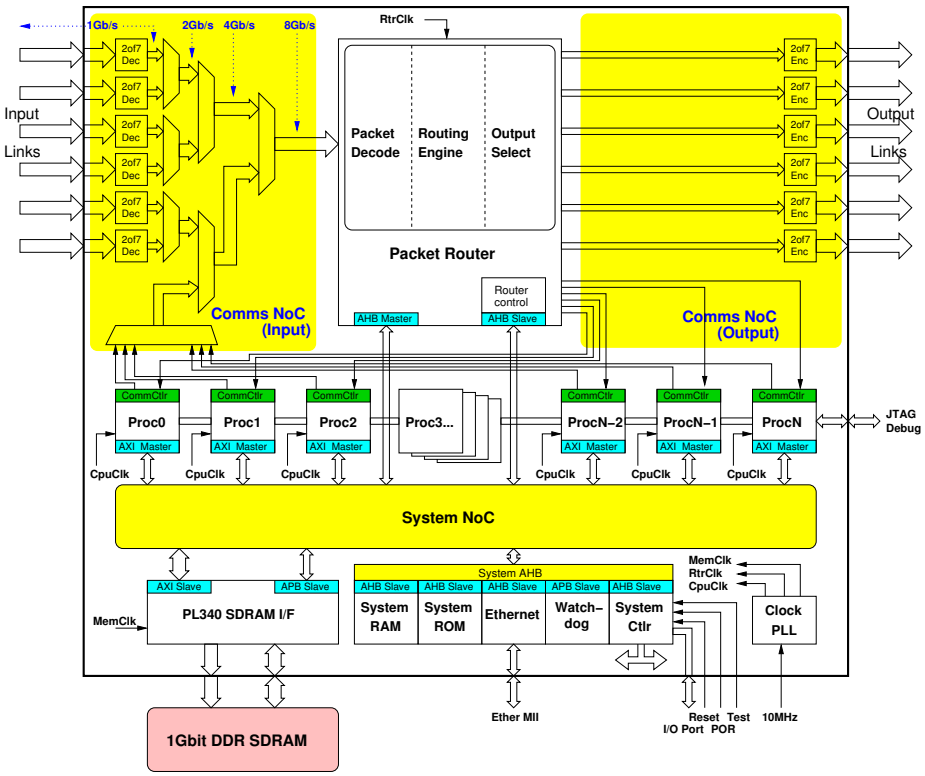

Figure 4.3: SpiNNaker chip schematic.

An important and innovative feature of the SpiNNaker chip is its custom communication system. At its heart is the multicast packet router, which relays packets between the processors on the chip, and to the routers of six neighbouring chips through external links. The interconnect fabric that ties together both on-chip processors and external links to the router is a self-timed Network-on-Chip (NoC), referred to as the Communications NoC. Similarly, the System NoC, a second, independent self-timed interconnect fabric, is used to allow the processor subsystems access to shared resources on the chip including the SDRAM. By employing asynchronous interconnection systems, the SpiNNaker chip follows the Globally Asynchronous, Locally Synchronous (GALS) design paradigm which eliminates the requirement to distribute a global synchronous clock signal across all the cores in a system [BF02; Pla+07; Pla+11]. Benefits also arise within the design of chip, as the processor subsystems are decoupled from one other, and from the rest of the on-chip system. The implementation process of the chip is therefore eased, as timing issues are limited to smaller areas of the chip. A further communication interface integrated into the SpiNNaker chip is an optional Ethernet link. It is primarily intended for the connection of a host system for configuration and monitoring purposes, and is deployed to a limited

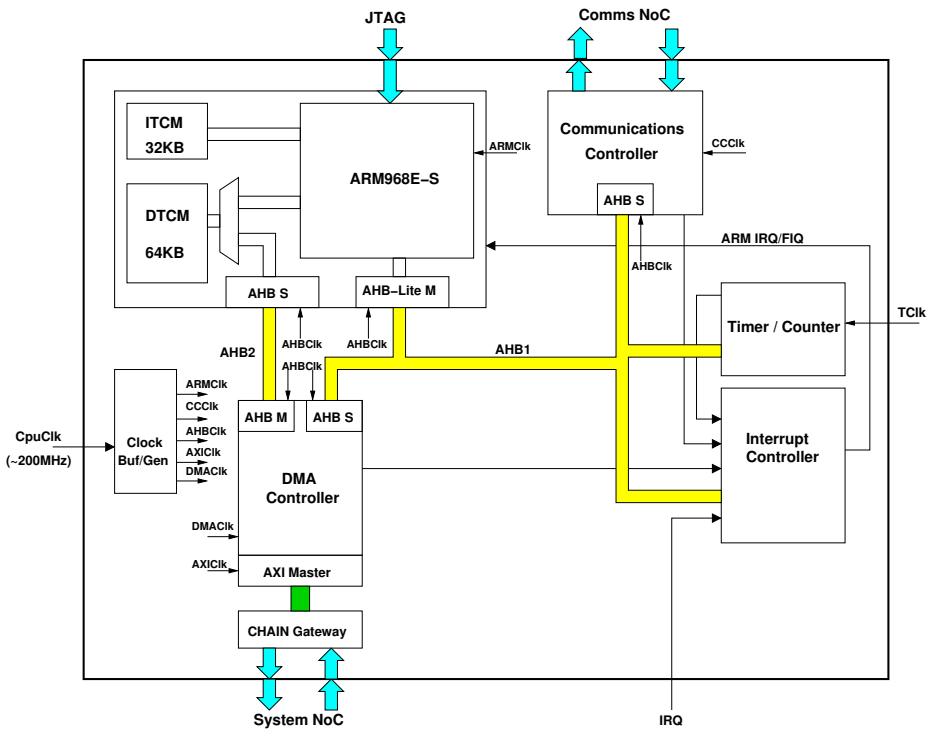

Figure 4.4: Processor subsystem schematic.

number of nodes as depicted in Figure 4.5.

4.2 System

SpiNNaker chips will be used to compose large programmable computing systems, in the first instance for the simulation of spiking neural networks at biological real-time. Since each chip can be interfaced to six others, it is possible to form a triangular grid of chips as one configuration. It is then intended to connect this grid into a toroid as illustrated in Figure 4.5. The advantage of such a constellation is that packets can be rerouted with only one additional hop around links that are congested or that have been detected as broken. This fault-tolerance feature of the SpiNNaker architecture is known as emergency routing.

It is intended to build a SpiNNaker system consisting of 57,600 chips that will include more than a million processing cores [Fur12]. Depending on the neuron and the synapse models selected, this should be sufficient to model approximately a billion biologically plausible neurons. Neural simulations on a SpiNNaker system operate with processing cores executing neuron simulations according to the required neuron

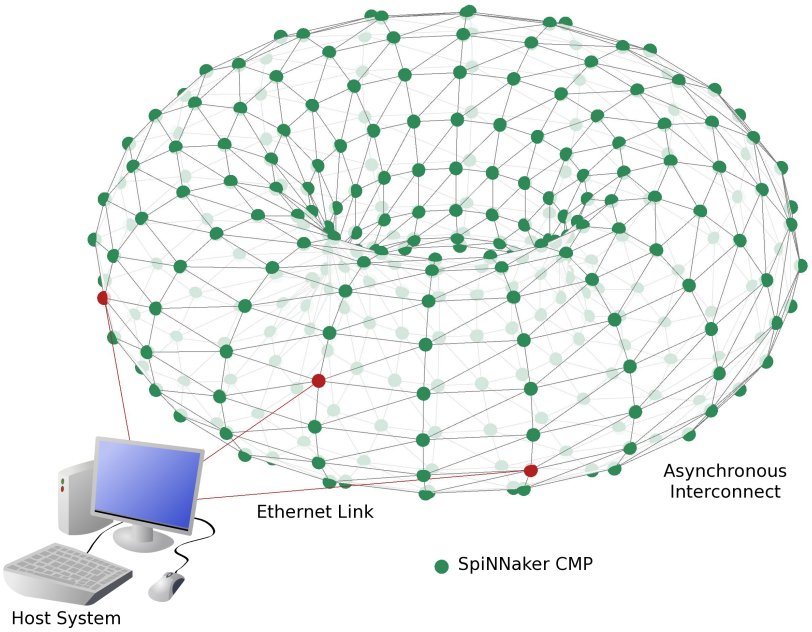

Figure 4.5: SpiNNaker system.

model. A neuron may, if certain conditions are met, emit a spike which is represented in the system as a short packet of 40 bits. 32 bits of the packet are reserved for the identification of the neuron emitting the spike, and 8 bits are used to carry further routing control information. Such spike packets are released from the processing cores into the routing fabric that handles the distribution and bifurcation to all target neurons connected to the emitting neuron, according to information stored in the routing tables of the routers. If a spike has been transmitted to a processing core modelling a destination neuron, then a DMA transfer is initiated to retrieve the corresponding synaptic parameters for that connection including its synaptic weight and the associated connection delay. This data is transferred by DMA from SDRAM to the local memory of the processing core. It is then possible to calculate the effect of the input spike on the target neuron via its synaptic connection completing the cycle of a basic neural simulation.

4.3 Memory

The large SpiNNaker system in its envisaged configuration of 57,600 nodes will incorporate in the order of 7 TiB of SDRAM. With such an immense amount of memory, the effect of data bit errors is significant during the operation of the machine.

Memory bit errors are subdivided into two different classes: hard and soft errors

[ZL79; Zie96; Zie+96]. A hard error is characterised by a permanent hardware fault in a memory cell that will result in a consistent reliability issue. For instance, it may be the case that a memory cell will always provide one particular bit value during readout, no matter what value has been written to it. Soft errors are transient faults that occur randomly and may, for example, be induced through cosmic rays or the decay of radioactive atoms in the memory packaging materials. Also, a soft error may arise either directly in the memory, or along the data path during the memory read or write phase.

Recently, a large-scale study has been conducted to investigate statistics for error rates in Dynamic Random Access Memory (DRAM) in production systems [SPW09]. It suggests that the average error rate ranges from 25,000 to 75,000 FIT (failures in time per billion hours of operation) per Mibit, however a distinction between hard and soft errors is not made. If these numbers are applied to the SpiNNaker system of 57,600 nodes, 25 to 74 bit errors on average can be expected to occur within the SDRAM per minute, roughly approximated as one bit error per second.

It may be the case that the number of expected bit errors in the SpiNNaker system will not have a significant impact on particular applications such as neural network simulations. However, it is not known to what extent neural network simulations can compensate for memory faults, and other potential applications may not tolerate bit errors at all, so appropriate measures need to be taken to deal with them in the SpiNNaker system. For this reason error-control codes are employed within SpiNNaker to provide a layer of protection against memory faults.

4.4 CRC Unit

The DMA controller of each processing subsystem has been equipped with a CRC unit that allows the generation and verification of error-control codes. The circuit primarily supports cyclic codes as they offer powerful error detection and correction capabilities and as they are, at the same time, easily implementable in hardware [LC83]. If, for instance, a processor initiates a DMA transfer to copy a data block from the local memory to the SDRAM, the CRC unit can be instructed to calculate (transparently and in parallel) the redundancy part for a cyclic code and, automatically, append this to the SDRAM data block. The CRC unit can be used to calculate the error syndrome for a data block retrieved from memory and signal the corresponding processing core if an integrity issue arose. The program that is executed on the

processing core has to decide what action is to be taken in the event of a detected data inconsistency. A simple retransmission of the data block could correct the error if it occurred along the data path during the readout phase, however even this may not be fast enough for the 'real-time' operation of a SpiNNaker neural simulation. Therefore it is necessary to consider appropriate error correction procedures in software, to recover from memory faults based on the obtained error syndrome, including when they are uncorrectable. These can range from a simple disregard of the error, through a localisation and correction of the error, to a shutdown of the relevant SpiNNaker system components for replacement if hard errors are involved.

In the choice of employed cyclic code, many factors need to be taken into account for the selection of the generator polynomial as outlined in Subsection 2.3.4. For instance, certain undiscovered subclasses of cyclic codes may allow the realisation of very efficient error correction procedures in software, or data blocks of different lengths may be stored in the SDRAM so that a polynomial offering best combined error protection for all of the block lengths should be selected. To offer maximal flexibility within SpiNNaker, a programmable CRC circuit has been incorporated that permits switching the generator polynomial to any of degree 32 or lower whenever required. A direct advantage is that the polynomial is adaptable to the length of the data block that is to be protected, which means that the best choice of offered error protection can be made. Another feature of the CRC circuit is that several cyclic codes of a smaller degree can be generated based on different bits of the data stream. For example, for each half-word of the data stream, a cyclic code based on a generator polynomial of degree 16 can be computed.

The width of the data bus that traverses the DMA controller in SpiNNaker is 32 bits, and the CRC unit has been designed to process this number of bits in parallel to avoid being a bottleneck to DMA data transfers. To configure the unit for the usage of a cyclic code or any other supported error-control code, one Kibit of configuration data needs to be supplied by the corresponding processing core to the appropriate registers inside the unit. Since the data bus is used to provide this configuration data, the transfer takes place as a series of 32 bit words. The registers are realised as latches to reduce the hardware demand, as each SpiNNaker chip accommodates one CRC unit for each of the 18 DMA controllers (one per processor subsystem). A detailed description of the SpiNNaker CRC circuit together with its derivation and capabilities can be found in the next chapter.

4.5 Conclusion

The SpiNNaker architecture has been created to support large-scale simulations of spiking neural networks in biological real-time. It has been dimensioned to scale up to machines consisting of a million processors with SDRAM totalling about 7 TiB. With such a vast amount of memory, it is estimated that, on average, about 1 bit error per second will occur.

To improve the reliability of memory transfers, SpiNNaker employs cyclic codes for error control due to the efficient realisation of the code generation and verification circuit. Once inconsistencies are detected for a block of data, software procedures may be triggered to attempt an error recovery. The correction of a single-bit error on the basis of a cyclic code, essentially requires the computation of the discrete logarithm in relevant groups as described in Chapter 2. However, no efficient algorithm is known for the computation of this type of discrete logarithm as discussed in Chapter 3. Two new solutions for the computation of the discrete logarithm in certain groups are proposed in Chapter 6 and Chapter 7.

The optimal choice of cyclic code to employ is influenced by many factors including the length of the data that is to be protected and the desired error control capabilities. Therefore, programmable cyclic code circuits are employed within SpiNNaker to maximise flexibility in the choice of cyclic code for different scenarios. A novel method for the generation of efficient programmable cyclic code circuits is proposed in Chapter 5.

Part III

Contributions

Chapter 5

Programmable CRC Hardware

Cyclic codes constitute a powerful class of error-control code as set out in Section 2.3. They offer effective error detection and can easily be realised in hardware, which makes them a popular choice for many applications that require the detection of errors, including Ethernet [TW11]. In the context of pure error detection, a cyclic code is often referred to as a Cyclic Redundancy Checksum (CRC) and the popularity of cyclic codes has led to a number of different software and hardware implementations [LC83; RG88]. Speed requirements usually make software schemes impractical and dedicated hardware is needed. The generic hardware approach uses an inexpensive Linear Feedback Shift Register (LFSR), which assumes serial data input. In the presence of wide data buses, the serial computation has been extended to parallel versions that process whole data words based on derived equations [AS90; PZ92; CPR03; Shi+01] and on cascading the LFSR [Spr01]. Various optimisation techniques have been developed that target resource reduction [Bra+96] and speed increase [Der01; CP06; KRM08; KRM09].

A wide range of factors influence the selection of an appropriate CRC generator polynomial for a particular application as outlined in Subsection 2.3.4. This range includes the error detection and correction capabilities of a generator polynomial, which depend on the length of the data that is to be protected. For instance, in scenarios where data blocks of different length are used, or where the final requirements of the generator polynomial are not known at the time of the hardware implementation, it is beneficial to employ programmable CRC circuits that can be configured to different generator polynomials.

This applies precisely to the SpiNNaker project, whose target is to provide a research platform for the simulation of arbitrary spiking neural networks as described

in Chapter 4. The planned large SpiNNaker system will incorporate a substantial amount of SDRAM to hold relevant data for the neural network simulations; in this context, CRCs are used to reduce the effect of data errors arising in the memory. Since the length of the data blocks may vary between different neural network simulations, for instance, it has been decided to incorporate programmable CRC circuits into the SpiNNaker system to offer maximal flexibility.

With the design of a circuit, there is usually a tradeoff between speed and area. In this particular scenario, there are two dimensions to the speed of a programmable CRC circuit: the time necessary to process a data word, and the time required to reconfigure to a new polynomial. This chapter directly extends the parallel CRC circuit by Campobello *et al.* [CPR03] based on state space representation in several ways. On the one hand, restrictions between the width of the data processed in parallel and the order of the polynomial are lifted. On the other hand, a novel scheme is presented allowing the inexpensive computation of the CRC transition and control matrix in hardware. This leads to a programmable parallel CRC implementation that offers an improved balance between area and both dimensions of speed.

5.1 From Serial to Parallel

Where systems use wide data buses, it is advantageous for CRC circuits to operate on data words and many approaches have been made to address this issue. Albertengo and Sisto [AS90] derived equations, in 1990, for a parallel CRC circuit with automatic premultiplication based on the Z-transform. A simpler method utilising state space transformation leading to basically the same circuit was published two years later by Pei and Zukowski [PZ92]. In 2003, Campobello *et al.* [CPR03] developed a similar proof for the parallel CRC circuit without automatic premultiplication, under the assumption that the order of the polynomial and the length of the message are both multiples of the number of bits to be processed in parallel, and reported a recursive formula for calculating powers of the state transition matrix.

In this section, the equations for the circuit with automatic premultiplication are derived; the principle of the derivation is very similar to the variant without premultiplication. Furthermore, the proof is extended in such a way that there will be no restriction on the order m of the polynomial or the number of bits w that are to be processed in parallel. The parameters are unrelated; it is only assumed that the k-bit message that is to be encoded can be split into data words of w bits, which will

usually be the case in computer systems.

The starting point for the derivation of the parallel CRC circuit for a generator polynomial $p(X)$ is the LFSR with automatic premultiplication by X^m for a serial data input as shown in Figure 2.6. The m least significant coefficients of $p(X)$ are aggregated into the coefficient vector $p = [p_{m-1}, \ldots, p_1, 1]^T$. Since the LFSR is a discrete time-invariant linear system, it can be expressed as:

$$s[i+1] = Ts[i] + p\bar{d}[i] \tag{5.1}$$

where

$$T = \left[p \middle| \frac{I_{m-1}}{0} \right] = \begin{bmatrix} p_{m-1} & 1 & 0 & \cdots & 0 \\ p_{m-2} & 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ p_1 & 0 & 0 & \cdots & 1 \\ 1 & 0 & 0 & \cdots & 0 \end{bmatrix}. \tag{5.2}$$

I_{m-1} denotes the identity matrix of size $m-1$ and $s[i]$ is the state of the system at time step i, which is equivalent to the corresponding m-bit LFSR value. The scalar input to the system is denoted by $\bar{d}[i]$. In each time step i, one bit of the k-bit message u is shifted into the system, starting from the most significant bit, and thus $\bar{d}[0] = u_{k-1}$, $\bar{d}[1] = u_{k-2}$ and so forth. It can be verified that the solution for system (5.1) takes the following shape:

$$s[i] = T^i s[0] + [T^{i-1}p, \ldots, Tp, p][\bar{d}[0], \ldots, \bar{d}[i-1]]^T, \tag{5.3}$$

with $s[0]$ being the initial state of the LFSR.

A simplified way exists to obtain T^i from T^{i-1}:

$$\begin{aligned} T^i &= T^{i-1}T \\ &= T^{i-1}\left[p \middle| \frac{I_{m-1}}{0} \right] \\ &= \left[T^{i-1}p \middle| T^{i-1}\frac{I_{m-1}}{0} \right], \end{aligned} \tag{5.4}$$

where i starts from 2. Expanding T to the power of w with the help of (5.4) leads to:

$$T^w = \begin{cases} \left[[T^{w-1}p, \ldots, Tp, p] \middle| \frac{I_{m-w}}{0} \right] & \text{if } w \leq m \\ \left[T^{w-1}p, \ldots, T^{w-m}p \right] & \text{otherwise.} \end{cases} \tag{5.5}$$

The columns of T^w that drop out in the case where w exceeds m are combined in the auxiliary rectangular matrix

$$T_w := \left[T^{w-m-1}p, \ldots, Tp, p\right].$$

In order to obtain the LFSR value after w bits have been processed, $s[w]$ simply needs to be evaluated. For this purpose the w-dimensional data input vector $d[t] = [u_{k-1-t}, \ldots, u_{k-w-t}]^T$ is introduced. Then (5.3) becomes

$$s[w] = T^w s[0] + \left[T^{w-1}p, \ldots, Tp, p\right] d[0]. \tag{5.6}$$

Two basic cases can be differentiated:

Case $w \leq m$:

$$\begin{aligned} s[w] &= T^w s[0] + \left[[T^{w-1}p, \ldots, Tp, p]\middle|\frac{I_{m-w}}{0}\right]\left[\frac{d[0]}{0}\right] \\ &= T^w s[0] + T^w\left[\frac{d[0]}{0}\right]. \end{aligned}$$

Considering additionally that the system is time-invariant, the behaviour of the circuit can be described as:

$$s[i+w] = T^w\left(s[i] + \left[\frac{d[i]}{0}\right]\right). \tag{5.7}$$

The special case of $w = m$ leads to the compact form:

$$s[i+w] = T^w(s[i] + d[i]). \tag{5.8}$$

Case $w > m$:

$$s[i+w] = T^w s[i] + [T^w|T_w]d[i]. \tag{5.9}$$

The result of (5.7) and (5.9) can be condensed into a single equation:

$$s[i+w] = [T^w|T_w]\left(\left[\frac{s[i]}{0}\right] + \left[\frac{d[i]}{0}\right]\right). \tag{5.10}$$

As an example, generator polynomial $p(X) = X^4 + X^3 + X + 1$ is selected. It is

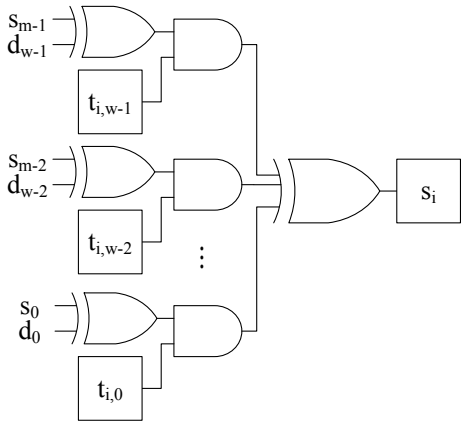

Figure 5.1: Programmable parallel CRC circuit with $w = m$ for CRC bit s_i utilising control latches.

intended to process four bits in parallel ($w = 4$). Consequently

$$T = \begin{bmatrix} 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \end{bmatrix}, \quad T^4 = \begin{bmatrix} 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 1 \end{bmatrix}. \tag{5.11}$$

According to (5.8), the necessary logic can be directly assembled with the help of (5.11). The time step indices will be dropped in the following where not needed for simplification. Matrix entries of T^w are numbered from $m-1$ to 0, where the top left most element is denoted by $(m-1, m-1)$. Thus, an entry $t_{i,j}$ in matrix T^w indicates that s_j XOR d_j is an input to the XOR forming the new value of s_i one clock cycle later.

5.2 From Static to Programmable

The parallel CRC architecture from the previous section can be transformed into a programmable entity that is no longer bound to a specific CRC generator polynomial $p(X)$. A polynomial directly affects the transition and control matrix $[T^w|T_w]$ of the linear system (5.10). Programmability can be achieved by introducing an AND gate with a controlling latch for each signal that may be a potential input to an XOR function as illustrated in Figure 5.1. Flipflops can be utilised as well, but will have 'in general' higher demands in terms of area, which may become crucial as $m \max(m, w)$ bits need to be stored.

The derivation of the matrix necessary to set up all the latches can be performed

in software. As the data bus width imposes a limitation on the transferable data, the matrix may need to be communicated line by line, which requires m clock cycles. Additionally, the software function itself relies on a processor. Many scenarios may even imply a dedicated core for this task, if the polynomial needs to be changed frequently and faster than the matrix can be communicated over the data bus.

In the case of the SpiNNaker architecture which is described in more detail in Chapter 4, it has been decided to employ this variant of the programmable CRC circuit, as it offers wide flexibility in terms of codes to which it can be configured. The number of cyclic codes that this circuit supports accounts for 2^{m-1}. This can easily be seen from the fact that a cyclic code generator polynomial $p(X)$ needs to divide $X^n + 1$ for an integer n. Such an n exists, as long as $p(0) \neq 0$ [GG05], which is fulfilled for every $p(X)$ due to its constant term, so that the degree of freedom for the configurable cyclic codes equals $(m - 1)$. However, the matrix that is supplied to the circuit exhibits $m \max(m, w)$ degrees of freedom, which is clearly more than for the case of the cyclic codes that the circuit supports.

To see how the redundancy part $s[k]$ for a k-bit data message is calculated by the circuit for an arbitrary binary matrix $[T^w|T_w]$, the effect of the first w message bits is considered at first, so that $s[w]$ can be computed according to (5.10) as

$$s[w] = [T^w|T_w]\left(\begin{bmatrix} s[0] \\ 0 \end{bmatrix} + \begin{bmatrix} d[0] \\ 0 \end{bmatrix}\right) = T^w s[0] + [T^w|T_w]\begin{bmatrix} d[0] \\ 0 \end{bmatrix},$$

with $s[0]$ being the initial state of the LFSR. If this process is continued with the subsequent input data words, $s[k]$ can be obtained as follows

$$s[k] = T^k s[0] + \sum_{i=0}^{k/w-1} T^{k-(i+1)w}[T^w|T_w]\begin{bmatrix} d[iw] \\ 0 \end{bmatrix}.$$

For the case that $w \leq m$, the formula can be simplified to

$$s[k] = T^k s[0] + \sum_{i=0}^{k/w-1} T^{k-iw}\begin{bmatrix} d[iw] \\ 0 \end{bmatrix}.$$

One alternative sensible configuration of the circuit is obtained if several cyclic codes are calculated for independent bits of the data stream. For example, it is possible to employ two cyclic codes, each with a generator polynomial of degree $m/2$ and each designated for a half-word of the data input. If the general case is considered, the

configuration matrix $[T^w|T_w]$ takes the following shape

$$[T^w|T_w] = \begin{bmatrix} [\bar{T}^{\bar{w}}|\bar{T}_{\bar{w}}] & 0 & 0 & \cdots & 0 \\ 0 & [\tilde{T}^{\tilde{w}}|\tilde{T}_{\tilde{w}}] & 0 & \cdots & 0 \\ 0 & 0 & [\breve{T}^{\breve{w}}|\breve{T}_{\breve{w}}] & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & [\hat{T}^{\hat{w}}|\hat{T}_{\hat{w}}] \end{bmatrix}.$$

The sum of the degrees of the individual cyclic codes cannot exceed m. Similarly, the sum of the numbers of the bits that are to be processed in parallel for the cyclic codes is bounded by w. Furthermore, it needs to be ensured that the top left element of each small configuration matrix is aligned with the diagonal of $[T^w|T_w]$, as the relevant state bits need to be fed back to the leftmost columns of the matrix.

5.2.1 Proposed Method

The chosen approach to construct efficient programmable cyclic code circuits is to outsource the matrix derivation into hardware if the circuit is supposed to be used exclusively for cyclic codes. As there is no computational effort involved to obtain the identity matrix in (5.5) for the case $w < m$, it is assumed that $w \geq m$, and thus

$$[T^w|T_w] = \left[T^{w-1}p, \ldots, Tp, p\right].$$

A new recursive formula for a column $T^i p$ with $i \geq 1$ is established as follows

$$\begin{aligned} T^i p &= TT^{i-1}p \\ &= \left[p \middle| \frac{I_{m-1}}{0}\right] T^{i-1}p \\ &= pe_1^T T^{i-1}p + \left[0 \middle| \frac{I_{m-1}}{0}\right] T^{i-1}p, \end{aligned} \tag{5.12}$$

where $e_1 = [1, 0 \ldots, 0]^T$ is the unit vector. Hence, each column element $(T^i p)_j$ can easily be computed with the help of the previous column $T^{i-1}p$ and p:

$$(T^i p)_j = \begin{cases} p_j (T^{i-1}p)_{m-1} + (T^{i-1}p)_{j-1} & \text{if } j \neq 0, \\ p_j (T^{i-1}p)_{m-1} & \text{otherwise.} \end{cases} \tag{5.13}$$

For an implementation in hardware, an $(m-1)$-bit register needs to be provided

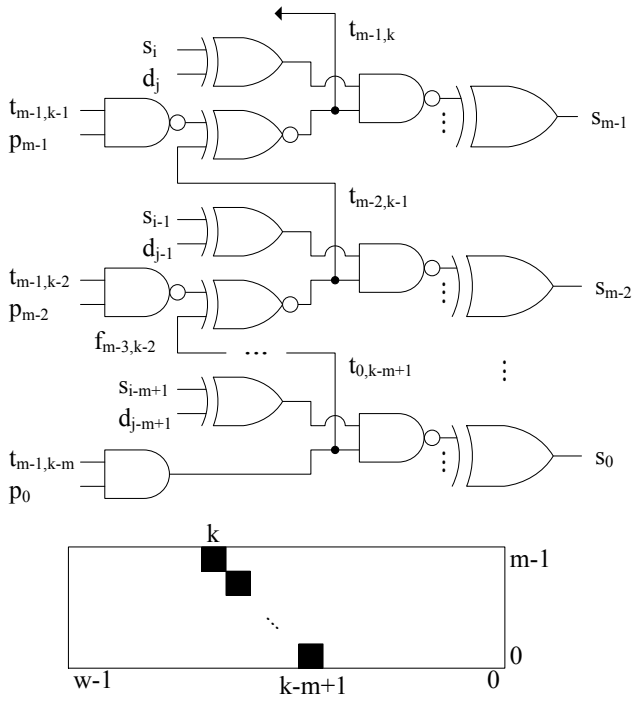

Figure 5.2: Programmable parallel CRC circuit for the case $w > m$. Corresponding elements of $[T^w|T_w]$ are indicated with black pixels in the rectangular representing the matrix.

to hold the coefficients of the polynomial; this register already forms the rightmost column of $[T^w|T_w]$. Each other column can then be obtained with m AND gates and $m-1$ XOR gates. The column element $(T^i p)_0$ of a column i can be obtained through an AND gate that takes as inputs the polynomial coefficient p_0, and the column element $(T^{i-1}p)_{m-1}$ of the previous column. For every other element $(T^i p)_j$ of column i, polynomial coefficient p_j and $(T^{i-1}p)_{m-1}$ need to be fed into an AND gate, before combining its result with column element $(T^{i-1}p)_{j-1}$ in an XOR gate.

It is possible to reduce the area with equivalent logic as illustrated in Figure 5.2, which additionally shows the attached CRC circuitry. Apart from the column storing the polynomial, each other column requires $m-1$ NAND gates, $m-1$ XNOR gates and 1 AND gate. The controlling AND gates have been replaced with NAND gates under the assumption that w is even. Inverting an even number of inputs to an XOR function does not affect the result of the function.

A circuit dimensioned for a certain polynomial degree m can be used to calculate CRCs for a polynomial of smaller degree r. This can be achieved by providing the polynomial premultiplied by X^{m-r}. Additional multiplexing circuitry is required to switch between different data input widths, as the most significant bits of the data d and the state of the system s need to be aligned, when being combined by the bitwise

XOR function according to (5.10).

The proposed circuit allows a further improvement if only generator polynomials $p(X)$ of degree m, for which the circuit is dimensioned, are used. In this case $p_0 = 1$, since every cyclic code generator polynomial exhibits a constant term. This implies that the $(w-1)$ AND gates can be omitted in the circuit by setting the value for the matrix element $(T^i p)_0$ to $(T^{i-1} p)_{m-1}$, for $i > 0$.

5.3 From Theory to Silicon

With the scheme from the previous section it is possible to replace each latch of the programmable CRC circuit (except for the first column that holds the generator polynomial) with a NAND and XNOR gate, which can compute the necessary value of the latch. The $w-1$ latches corresponding to the least significant LFSR bit can each be replaced with only an AND gate. If the circuit is only intended for the use of generator polynomials of degree m, not even the AND gates will be required. Several 130-nm standard cell libraries indicate a saving of about $6-7\%$ in logic gate area for a NAND plus XNOR gate in comparison to a latch. Further savings arise from the much smaller AND gate area, if required at all, and irrelevant latch select logic.

To assess the performance of the new circuit in Figure 5.2, it is necessary to consider two different paths. Assuming that the generator polynomial $p(X)$ is already set up, the logic gate delay for the data (from d to s) adds up to

$$T_{DS} = (\lceil \log_2 w \rceil + 1)\, T_{XOR} + T_{NAND}, \tag{5.14}$$

where T is a logic gate or path delay. Changing the polynomial, on the other hand, affects a longer path. The difference between T_P and T_{DS} accounts for

$$\triangle\, T_P = T_{LATCH} + (w-1)(T_{XNOR} + T_{NAND}) - T_{XOR}. \tag{5.15}$$

Appending a CRC value to a message will typically require a data stream to stall for at least one clock cycle. Hence, this clock cycle can be used to provide a new generator polynomial for the subsequent message. This means that the polynomial has two clock cycles to propagate through the entire circuit.

The same behaviour can be achieved on the message receiving side. Instead of inputting the received CRC as the last data word into the circuit, and checking the result for 0, the received and calculated CRC value can be compared directly. Again,

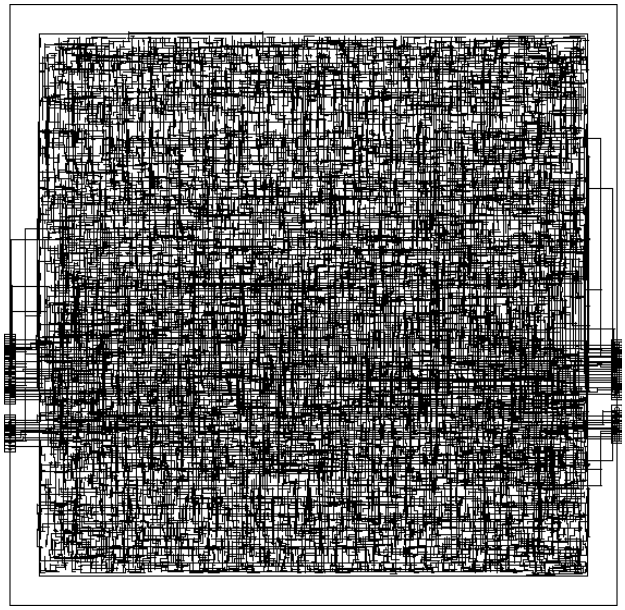

Figure 5.3: Fully routed layout of the proposed programmable parallel CRC circuit.

this would free up an extra clock cycle that can be used to set up a new polynomial.

Without any further clock cycles, the frequency of the circuit would be limited to $f_{worst} = 1/\max(T_{DS}, \triangle T_P)$. To achieve best case frequency $f_{best} = 1/T_{DS}$, additional $\lceil \triangle T_P f_{best} \rceil - 1$ clock cycles would be necessary between messages to propagate $p(X)$. Alternatively, the number of clock cycles can be reduced by providing c columns of matrix $[T^w | T_w]$ instead of only one to split up $\triangle T_P$. This requires more area as these columns need to be stored in latches again. In the extreme case the individual paths have a length of $\triangle T_{Pi} = iT_{DS}$, for, $i = 1, \ldots, c$.

The design has been simulated for $m = w = 32$ targeting 130-nm high-speed standard cell technology using Synopsys, Inc., synthesis tools with the resulting fully routed layout shown in Figure 5.3. The simulation results in Table 5.1 were obtained assuming a typical-typical process corner and operating conditions of 1.2 V and 25 °C. These are compared to a previous design [Toa+09], which will be discussed in the following section. Better performance is anticipated with a full-custom design that will further exploit the regular structure of the circuit and the omission of the AND gates as described earlier.

Table 5.1: Programmable CRC circuit implementation comparison

	Cell Array [Toa+09]	Novel Circuit	Result
Clock Frequency	154 MHz	481 MHz	
Data Throughput	4.92 Gbps	15.38 Gbps	+212.70%
Reconfiguration	33 clock cycles	4 clock cycles	
	214.29 ns	8.32 ns	−96.12%
Core Area	0.150 mm^2	0.033 mm^2	−78.00%
Core Utilization	Not specified	96.13%	
Total Power	5.70 mW	6.37 mW	
Internal Power	3.42 mW	3.69 mW	
Switching Power	2.19 mW	2.67 mW	
Leakage Power	0.0896 mW	0.0077 mW	
Energy[a]	63 pJ/word	14 pJ/word	−77.78%

[a] Based on the setup of a polynomial with a subsequent CRC calculation for 47 data words.

5.4 Comparison

A programmable parallel CRC architecture was recently proposed [Toa+09], which is referred to as the cell array architecture. It incorporates additional circuitry to switch between two different data input widths, which is considered in the following critical path and area analysis.

The main component of the cell array is a configurable array of $m \max(m, w)$ cells, each consisting of an XOR, two multiplexers, and a configuration register. A preliminary stage of XOR gates combines data with the current state of the system, which is then fed into the array. Furthermore, a configuration processor is integrated, which performs matrix multiplications to obtain the state transition matrix for a provided generator polynomial. The matrix is transferred row-wise into the configuration registers of the array.

For the basic CRC calculation, both architectures require the same number of two-input XOR gates. The present work however, also allows the utilisation of wider and proportionally smaller XOR gates that will assemble m trees each with w inputs.

Considering the logic that is necessary for programmability, each cell in the array constitutes two multiplexers and one register. In the new design, this corresponds in the general case to two NAND gates and one XNOR gate, in m cases to one NAND gate and one latch, and depending on the implementation, in $w - 1$ cases to only one

NAND gate and one AND gate or just one NAND gate. In all cases this is typically less than for a cell in the cell array design. More area is saved as there is no need for a processor.

The worst-case data path in the cell array can be specified as follows:

$$T_{DS2} = w(T_{XOR} + T_{MUX}).$$

This linear growth is inferior to the logarithmic growth of T_{DS} (5.14), which has also a reduced scaling factor by T_{MUX}.

The reconfiguration time of the new circuit accounts for $\triangle T_P$ (5.15). For the cell array, a reconfiguration time of $w+1$ clock cycles is indicated. The operating frequency of the circuit is limited to $f_{best2} = 1/T_{DS2}$. This means that

$$\triangle T_{P2} \approx w(w + 1)(T_{XOR} + T_{MUX}).$$

Consequently

$$\frac{\triangle T_{P2}}{\triangle T_P} \in O(w).$$

This suggests that the new design reconfigures in the order of approximately w times faster than the cell array with the configuration processor.

Both designs have been implemented targeting 130-nm standard cell technology, and are compared in Table 5.1. The new design can be operated at a frequency more than three times higher than the cell array, and has a correspondingly increased data throughput. It can reconfigure to a new generator polynomial 25 times faster than the cell array, while occupying only 22% of its area. Similarly, the energy consumption dropped by about 78%.

An alternative approach in realising, at least partial, programmability is to multiplex between several CRC modules dedicated to fixed generator polynomials. This method is beneficial if only a few polynomials come into consideration, for which each module can be specifically optimised in terms of speed. Beyond a certain number of different polynomials however, which depends on the polynomials and their realisation, the area requirements will exceed those for the proposed architecture. Furthermore, the multiplexing overhead will offset the speed advantage if too many polynomials are involved.

5.5 Conclusion

An existing proof [CPR03] for the derivation of parallel CRC circuits has been extended to any generator polynomial size m and data width w. The proof has been conducted for the LFSR realisation with automatic premultiplication, which avoids inserting a final zero data word.

A simple method has been presented to incorporate programmability into the circuit through latches thus allowing the generator polynomial to be changed during runtime.

Furthermore, a novel scheme has been proposed to compute the state transition and control matrix of the CRC circuit easily in hardware. The scheme is based on a new recursive formula and offers a range of advantages over existing techniques.

Firstly, it is necessary to provide only the desired generator polynomial, instead of a complete matrix for the CRC core; a preliminary matrix calculation in software is no longer required. Secondly, the logic area requirements are lower than those for a realisation that stores the matrix in latches. A recently proposed architecture [Toa+09] has significantly higher demands in terms of area as it incorporates a configuration processor, and more core logic in comparison to the latch variant. Thirdly, the data path grows only logarithmically with w in contrast to the existing architecture where it grows linearly with w with a higher scaling factor; this implies a faster CRC calculation. Most importantly however, the new circuit reconfigures approximately w times faster than the previous circuit.

Implementation figures support the theoretical results showing a significant improvement in speed, area and energy efficiency.

Chapter 6

Logarithms: A Generic Algorithm

The question as to whether it is possible to compute discrete logarithms efficiently in certain cyclic groups is of major interest for many applications, notably in the cryptographic field as outlined in Chapter 3.

For some cryptographic algorithms that are in wide use, the intractability of the discrete logarithm problem in certain groups is a key requirement, as the presumed security could otherwise be compromised easily. Other applications such as event counters based on the LFSR [CW94] or the use of cyclic codes for error control would benefit greatly from a method that allows the evaluation of discrete logarithms in polynomial time.

In the case of cyclic codes, an easy computation of discrete logarithms would enable the efficient correction of single-bit errors as presented in Chapter 2. This would be extremely useful for the operation of SpiNNaker machines, as the hardware provides support for cyclic codes to protect stored data in the SDRAM. The codes can transparently be generated and verified in hardware, but in the event of a detected error occurrence, software procedures will need to step in to attempt a data recovery as described in Chapter 4.

This chapter presents a new approach for determining discrete logarithms. For analysed groups where the order equals a Mersenne number with an exponent of a power of two, a generic algorithm is obtained that can be used with any group representation, requiring execution time in the order of the square root of the size of the group and negligible space. The operating principle of the algorithm is based on size differences of cyclotomic cosets which is explained below.

6.1 Computing Discrete Logarithms

It is assumed that G is a finite cyclic group of order q with a primitive element α. The discrete logarithm k of an element $\beta \in G$ is to be determined. The sequence of the form

$$(\beta)^{c^i}, \tag{6.1}$$

for an integer c with $2 \leq c \leq q-1$ and index i starting from zero, is considered. Since G is a finite group, the elements in the sequence will start recurring at some point according to the pigeonhole principle, so that two indices i, j of smallest possible value with $0 \leq i < j$ can be identified, where

$$(\beta)^{c^i} = (\beta)^{c^j},$$

holds. From the fact that $\beta = \alpha^k$, it follows that

$$kc^i \equiv kc^j \pmod{q},$$

which is equivalent to $q|k(c^j - c^i)$. It may very well be the case that q and k share common factors, which leads to

$$\frac{q}{\gcd(q,k)} \Big| \frac{k(c^j - c^i)}{\gcd(q,k)}.$$

Since $q/\gcd(q,k)$ is not a factor of $k/\gcd(q,k)$, the following simplification is obtained

$$c^i \equiv c^j \pmod{q/\gcd(q,k)}. \tag{6.2}$$

It can be seen that the indices i and j are influenced through k, and more precisely through the greatest common divisor of k and q. Thus, if all possible values of k are taken into account, the factorisation of the group order q has a vital impact on the solutions of the congruences. The idea is now to draw interferences from the values of i and j about k, which will be described in what follows.

Group orders q that equal a Mersenne number, where the exponent is restricted to a power of two, are now considered. This type of number is denoted by

$$M_{2^t} = 2^{2^t} - 1.$$

Another similar type of number that will be used hereafter is the Fermat number,

which has the form

$$F_t = 2^{2^t} + 1.$$

Attention is drawn to the special groups of order $q = M_{2^t}$ as properties can be derived that lead to a new algorithm for the solution of the discrete logarithm based on the relations obtained from (6.2). A second reason for the focus on these groups is that they are generated by primitive polynomials, where the degree is a power of two, which is a typical setting in computer systems. They also include the largest group that needs to be dealt with if the potential cyclic codes within SpiNNaker are considered. Since SpiNNaker permits cyclic code generator polynomials up to degree 32, the largest group in which the discrete logarithm needs to be solved has an order of M_{32}. The following theorem gives useful information about the structure of the considered M_{2^t} numbers.

Theorem 6.1. *M_{2^t} can be factored into Fermat numbers F_0 to F_{t-1} for $t > 0$. Furthermore, F_i and F_j are coprime for $i \neq j$.*

Proof. The first statement is true for $t = 1$, since $M_2 = 3 = F_0$. Now it is assumed that the statement is true for t. It follows that $M_{2^{t+1}} = 2^{2^{t+1}} - 1 = (2^{2^t} - 1)(2^{2^t} + 1) = M_{2^t} F_t$. Therefore, by induction the statement is true for all $t > 0$. The second statement is Goldbach's theorem [KLS02; Ros93]. □

It is the case that two and M_{2^t} are coprime, so that two has finite multiplicative order modulo M_{2^t}. With $c = 2$ in (6.1), it follows from (6.2) with $i = 0$ that

$$2^i \equiv 2^0 \equiv 1 \equiv 2^j \pmod{M_{2^t} / \gcd(M_{2^t}, k)},$$

for a $j > 0$. This means that in the sequence given by (6.1), the starting value β will reoccur after j steps. Before information is provided about the period of the sequence, a few supporting theorems are introduced at first.

Theorem 6.2. *The smallest integer $x > 0$ that satisfies $2^x \equiv 1 \pmod{F_t}$ is $x = 2^{t+1}$.*

Proof. A solution for the congruence is provided by $\bar{x} = 2^{t+1}$, for

$$\frac{2^{\bar{x}} - 1}{F_t} = \frac{2^{2^{t+1}} - 1}{2^{2^t} + 1} = 2^{2^t} - 1 = M_{2^t}.$$

Furthermore, x can be restricted to $2^t < x \leq 2^{t+1}$, as $2^x \not\equiv 1 \pmod{F_t}$ for $1 < 2^x < F_t$. It is assumed that $\tilde{x}$ is the smallest solution and different from $\bar{x}$, such that $2^t < \tilde{x} < \bar{x}$.

Then

$$2^{\bar{x}} \equiv 2^{\tilde{x}} 2^{\bar{x}-\tilde{x}} \equiv 2^{\bar{x}-\tilde{x}} \equiv 1 \mod F_t,$$

which implies that $\bar{x} - \tilde{x} \geq \tilde{x}$, as $\tilde{x}$ is the smallest solution. This means that $\tilde{x} \leq \bar{x}/2 = 2^t$, which contradicts that $2^t < \tilde{x}$, and therefore $\bar{x} = 2^{t+1}$ must be the smallest solution. □

The proof of the following theorem can be found in the literature [BS96].

Theorem 6.3. *For every factor f of F_t, the smallest integer $x > 0$ that satisfies $2^x \equiv 1 \pmod{f}$ is $x = 2^{t+1}$.*

It is now possible to specify the period of the considered sequence as follows.

Theorem 6.4. *The smallest positive integer j for which $2^j \equiv 1 \pmod{M_{2^t} / \gcd(M_{2^t}, k)}$ is satisfied, accounts for $j = 2^{u+1}$ if F_u is the largest Fermat number that does not divide k, where $u < t$. If such a Fermat number does not exist, $j = 1$.*

Proof. With Theorem 6.1 and the premise F_u being the largest Fermat number of M_{2^t} with $\gcd(F_u, k) \neq F_u$, the congruence can be transformed into

$$2^j \equiv 1 \mod \left(\frac{F_0}{\gcd(F_0, k)} \frac{F_1}{\gcd(F_1, k)} \cdots \frac{F_u}{\gcd(F_u, k)} \right).$$

According to Theorem 6.1, the congruence holds for $j = 2^{u+1}$. Since at least one Fermat factor of F_0 to F_u remains in the modulus, Theorem 6.3 guarantees that $j = 2^{u+1}$ is the smallest positive solution fulfilling the congruence.

If k is divisible by all the Fermat numbers F_0 to F_u, which means it is divisible by M_{2^t}, it follows that $k = 0$. The congruence simplifies to $2^j \equiv 1 \pmod{1}$, so that the smallest positive j becomes $j = 1$. □

With the result of Theorem 6.4 it is clear that the period of the sequence is influenced by the largest Fermat number F_0 to F_{t-1} that is not a factor of k. This means in particular that if F_u is the largest Fermat number with $u < t$ that is not a factor of k, the period of the corresponding sequence can be indicated as $j = 2^{u+1}$. The period for the special case, where $k = 0$, accounts to $j = 1$. In other words, the period length equals $j = 2^t$, unless k is a multiple of Fermat numbers F_{u+1} to F_{t-1}, but not of F_u, in which case the period shrinks to 2^{u+1}.

The group order $q = M_4$ is considered as an example. For the sequence of group elements $\beta^{2^j} = \alpha^{k2^j}$ with starting index of $j = 0$, the resulting sequence of exponents

Table 6.1: Sequences of the form $k2^j$ modulo $(M_4/\gcd(M_4, k))$.

k	j			
	0	1	2	3
0	0			
1	1	2	4	8
2	2	4	8	1
3	3	6	12	9
4	4	8	1	2
5	5	10		
6	6	12	9	3
7	7	14	13	11
8	8	1	2	4
9	9	3	6	12
10	10	5		
11	11	7	14	13
12	12	9	3	6
13	13	11	7	14
14	14	13	11	7

of the generator element, $k2^j$ modulo $(M_4/\gcd(M_4, k))$, is listed in Table 6.1 for an entire period as a function of k. The group order factors into $q = M_4 = F_0F_1$, which explains why the period of the sequence is $j = 2$ if k is a multiple of F_1, but not of F_0. If k is a multiple of both, F_0 and F_1, the period equals $j = 1$. In every other case, the period exhibits a maximum length of $j = 4$.

A different way of looking at the sequences is to consider cyclotomic cosets C_k modulo q with respect to c, where c and q are coprime [GG05]. These are defined as

$$C_k = \{k, kc, \ldots, kc^j\},$$

with j being the smallest positive integer so that $k \equiv kc^j \pmod{q}$. The coset leaders k are chosen such that they are smallest nonnegative integers in their corresponding sets C_k. For the example in Table 6.1, where $q = 15$ and $c = 2$, the cyclotomic cosets are

$$C_0 = \{0\},$$

$$C_1 = \{1, 2, 4, 8\},$$
$$C_3 = \{3, 6, 12, 9\},$$
$$C_5 = \{5, 10\},$$
$$C_7 = \{7, 14, 13, 11\}.$$

It is possible to exploit the regular structure governing the differences in sequence period lengths or size of cyclotomic cosets, to deduce the discrete logarithm $k = \log_\alpha \beta$. First of all, β can be examined to determine if it corresponds to a certain period $j \leq 2^u$. Since all the periods are powers of two, smaller periods divide larger periods, and so a simple verification of $\beta = \beta^{2^{u+1}}$ would imply $j \leq 2^u$. Secondly, periods of the length $j \leq 2^u$ occur if k is a multiple of $F_u F_{u+1} \cdots F_{t-1}$, which implies that smaller periods $\bar{j}$ with $\bar{j} < j$ occur only if k is additionally a multiple of F_{u-1}.

Under the assumption that β belongs to a period of length $j \leq 2^u$, it is possible to test for the tighter period length bound $i \leq 2^{u-1}$, and modify β through multiplication with $\alpha^{F_u \cdots F_{t-1}}$ if the test fails. This process is repeated until the period length of β adheres to the bound i, which will require a maximum number of F_{u-1} tests. With the new β, the bound i can now be reduced to half of its value and the search restarted. The process stops once β corresponds to a period of length one, where it is known that the corresponding k equals $k = 0$. If track has been kept of the modifications that were applied to β, it is possible to work out the original value of k. The method is summarised through the pseudocode shown in Algorithm 6.1.

Algorithm 6.1 Discrete logarithm computation based on cyclotomic cosets.

```
1:  function LOG(α, β, q = M_{2^t})
2:      k = 0
3:      for i = t − 1 downto 0 do
4:          β̄ = β^{2^{i+1}}
5:          while β ≠ β̄ do
6:              β = β α^{M_{2^t}/M_{2^{i+1}}}
7:              β̄ = β̄ α^{M_{2^t}/M_{2^{i+1}}}
8:              k = k − M_{2^t}/M_{2^{i+1}} mod q
9:          end while
10:     end for
11:     return k
12: end function
```

The worst-case running time of the algorithm for $q = M_{2^t}$ can be expressed as

$$O(t \log_2 q + \sum_{i=0}^{t-1} F_i),$$

where the first term takes into account the time necessary to compute $\bar{\beta}$ and $\alpha^{M_{2^t}/M_{2^{i+1}}}$, and the second term to traverse the while loop. Since $F_{t-1} \approx \sqrt{q}$, the time complexity can be simplified to $O(\sqrt{q})$. The space requirements account for $O(1)$.

6.2 Improvement

The running time of the algorithm can be improved if the probabilities of occurrence of the discrete logarithm values k with $0 \leq k \leq q-1$ are unequal. This is, for instance, the case if the discrete logarithm is restricted to lie in a certain interval.

Instead of multiplying both β and $\bar{\beta}$ in Algorithm 6.1 by $\alpha^{M_{2^t}/M_{2^{i+1}}}$ for a specific iteration i, it is possible to multiply with the inverse element likewise. This would require a corresponding advancement of k by $M_{2^t}/M_{2^{i+1}}$ rather than by $(-M_{2^t}/M_{2^{i+1}})$. If the first iteration of the algorithm for $i = t-1$ is considered, the following two sets of discrete logarithms k can be defined

$$S_0 = \{k | 1 \leq k + rF_{t-1} \leq (F_{t-1} - 1)/2\}$$
$$S_1 = \{k | (F_{t-1} - 1)/2 + 1 \leq k + rF_{t-1} \leq F_{t-1}\},$$

where r is an integer. Let $p(k)$ be the probability of occurrence for the discrete logarithm k. If

$$\sum_{k \in S_0} p(k) > \sum_{k \in S_1} p(k),$$

then the alternative method as suggested in this section could have an improved average or even improved worst-case running time, as a sequence length of $j \leq 2^{t-1}$ could be reached on average and, possibly, also in the worst case faster than with the standard method. The same principle can then be applied to each subsequent iteration step with $i \leq t-2$, by determining the corresponding probabilities and deciding on the appropriate method.

6.3 Properties

An interesting property can be derived for the considered sequences, which is summarised in the following theorem.

Theorem 6.5. *Let G be a cyclic group of order $q = M_{2^t}$ with primitive element α. If $k \equiv 1 \pmod{F_{t-1}}$, it follows that*

$$\alpha^k = (\alpha^{k+M_{2^t-1}})^{F_{t-1}-1}.$$

Proof. Starting from the left hand side of the equation, the logarithm to the base α is taken and k is replaced by $(rF_{t-1} + 1)$, where r is an integer and which leads to the corresponding logarithm of the right hand side as follows

$$\begin{aligned}
k &\equiv rF_{t-1} + 1 \\
&\equiv rF_{t-1} + 1 + M_{2^t}(r + 1) \\
&\equiv r(F_{t-1} + M_{2^t}) + (M_{2^t} + 1) \\
&\equiv rF_{t-1}(F_{t-1} - 1) + (M_{2^{t-1}} + 1)(F_{t-1} - 1) \\
&\equiv (rF_{t-1} + 1 + M_{2^{t-1}})(F_{t-1} - 1) \\
&\equiv (k + M_{2^{t-1}})(F_{t-1} - 1) \pmod{M_{2^t}}.
\end{aligned}$$

□

Theorem 6.5 can be applied directly to the example sequences shown in Table 6.1, which belong to the group order $q = M_{2^t}$ with $t = 2$. It can be seen that for every integer r, the sequence for $k \equiv F_1 r + 1 \equiv 5r + 1 \pmod{15}$ repeats at $\bar{k} \equiv F_1 r + M_{2^{t-1}} + 1 \equiv 5r + 4 \pmod{15}$ shifted by $t - 1 = 2$ elements. The involved sequences for k and $\bar{k}$ are just the sequences that are adjacent to sequences with a period length of $j \leq 2^{t-1}$, since $F_{t-1} | k - 1$ and $F_{t-1} | \bar{k} + 1$.

For the finite field $GF(2^m)$, the trace function is defined as

$$Tr(x) = \sum_{i=0}^{m-1} x^{2^i},$$

where $x \in GF(2^m)$. $Tr(x)$ defines a mapping from $GF(2^m)$ to $GF(2)$ [GG05]. This result can be refined for the considered sequences in the following theorem.

Theorem 6.6. *Let x be an element of the finite field $GF(2^m)$ and j the smallest positive integer such that $x = x^{2^j}$. It follows that*

$$\overline{Tr}(x) = \sum_{i=0}^{j-1} x^{2^i} \in GF(2).$$

Proof. The same principle is applied as for the mentioned result of the trace function [GG05]. Let $\beta \in GF(2^m)$ and j be the smallest positive integer such that $\beta = \beta^{2^j}$. Therefore,

$$\left(\sum_{i=0}^{j-1} \beta^{2^i}\right)^2 = \sum_{i=0}^{j-1} \beta^{2^{i+1}} = \sum_{i=0}^{j-1} \beta^{2^i}.$$

Since $\beta = \beta^2$ is equivalent to $\beta \in GF(2)$, it follows that

$$\sum_{i=0}^{j-1} \beta^{2^i} \in GF(2).$$

□

The theorem can be applied to the considered sequences if the underlying cyclic group is the multiplicative group of a finite field. To construct an example corresponding to the one given in Table 6.1, the finite field $GF(2^4)$ is considered, which will be represented as the polynomial ring over $GF(2)$ modulo the primitive polynomial $p(X) = X^4 + X^3 + 1$. Furthermore, $\alpha = X$ is a root of $p(X)$. The sequences of the form $(\alpha^k)^{2^j}$ to which the trace function $Tr(x)$ and its variant $\overline{Tr}(x)$ from Theorem 6.6 are applied, are shown in Table 6.2.

For the finite field $GF(M_{2^t}+1)$, where the order of the multiplicative group accounts for $q = M_{2^t}$, it has been shown that for a group element β, the sequence β^{2^i} with starting index $i = 0$ has a period length of a power of two. This means that the trace function $Tr(x)$ can be synthesised from the trace function variant $\overline{Tr}(x)$. If a group element β has a period length j, it follows

$$Tr(\beta) = \sum_{i=1}^{2^t/j} \overline{Tr}(\beta).$$

Therefore, if the group element β has a period length $j < 2^t$, it can be stated that $Tr(\beta) = 0$. In other words, whenever the period length of β is below the maximum value of 2^t, the trace function will result in zero. This applies to all $\beta = \alpha^k$, where α is

Table 6.2: Sequences of the form $(X^k)^{2^j}$ modulo $X^4 + X^3 + 1$ are shown for an entire period. The trace function $Tr(x)$ and its variant $\overline{Tr}(x)$ are applied to the sequences. Furthermore, the sum of the sequence elements X^k and X^{4k} is specified. Sequence elements are indicated in 4-tuple and 1-tuple representation, where the leftmost bit is the most significant bit.

k	j				$\overline{Tr}(X^k)$	$Tr(X^k)$	$X^k + X^{4k}$
	0	1	2	3			
0	0001				1	0	0000
1	0010	0100	1001	1110	0	1	1011
2	0100	1001	1110	0010	0	1	1010
3	1000	1111	0011	0101	1	1	1011
4	1001	1110	0010	0100	0	1	1011
5	1011	1010			1	0	0000
6	1111	0011	0101	1000	1	1	1010
7	0111	1100	0110	1101	1	0	0001
8	1110	0010	0100	1001	0	1	1010
9	0101	1000	1111	0011	1	1	1010
10	1010	1011			1	0	0000
11	1101	0111	1100	0110	1	0	0001
12	0011	0101	1000	1111	1	1	1011
13	0110	1101	0111	1100	1	0	0001
14	1100	0110	1101	0111	1	0	0001

a primitive element of the group and $F_{t-1}|k$ as can be observed in Table 6.2.

An irreducible polynomial $p(X) = X^m + p_{m-1}X^{m-1} + p_{m-2}X^{m-2} + \cdots + p_1X + p_0$ over $GF(2)$ of degree m defines a linear recursive sequence over $GF(2)$ with

$$s_{k+m} = \sum_{i=0}^{m-1} p_i s_{k+i},$$

where the index k starts from $k = 0$ and where s_0 to s_{m-1} form the initial state [GG05]. It can be shown that if α is a root of $p(X)$, then an element $\beta \in GF(2^m)$ exists, such that the linear recursive sequence can equivalently be described as

$$s_k = Tr(\beta\alpha^k),$$

for $k \geq 0$. The initial state is encoded through β in this case. Thus, the sequence that is obtained through the trace function in the example in Table 6.2 corresponds to the

linear recursive sequence defined by $p(X)$.

An additional property that can be derived concerns the summation of field elements of $\alpha \in GF(2^m)$. If two sets of indices K and $\bar{K}$ can be identified such that

$$\sum_{k \in K} \alpha^k = \sum_{k \in \bar{K}} \alpha^k,$$

then the equation will also hold if each index within K and $\bar{K}$ is multiplied by a power of two, since the characteristic of the field is two. The property can also be observed in the example that is given in Table 6.2, where $\alpha = X$ is a primitive element of $GF(2^4)$. It can be seen that $\alpha^k + \alpha^{4k}$ is equal for $k \in \{1, 3, 4, 12\}$. This implies that $\alpha^k + \alpha^{4k}$ has the same value for $k \in \{2, 6, 8, 9\}$ as can be verified in the example.

6.4 Comparison

As long as the involved Fermat numbers F_0 to F_{t-1} are all prime, the proposed algorithm competes with the Silver-Pohlig-Hellman algorithm, due to the similar running time. Composite Fermat numbers are split by the Silver-Pohlig-Hellman algorithm into its prime factors, and for the corresponding group sizes the discrete logarithms are solved individually to be combined to the overall solution as outlined in Section 3.3. Therefore, the Silver-Pohlig-Hellman algorithm has a time-complexity advantage if composite Fermat numbers are present in the group order factorisation. Currently, the only known Fermat primes range from F_0 to F_4 [CMP03; Kel]. The subsequent Fermat numbers F_5 to F_{32} have been proven to be composite. At present, the status of F_{33} is unknown.

Another aspect concerns the combinability of the Silver-Pohlig-Hellman algorithm with other methods. The algorithm breaks down the group order q into its prime factors and computes the discrete logarithm in the corresponding subgroups for which more efficient methods can be used than the exhaustive search. It is currently not obvious how this approach could be carried over to the proposed method.

6.5 Conclusion

A new approach for the computation of discrete logarithms has been proposed. For analysed cyclic groups with orders of the form $q = M_{2^t}$, a novel deterministic generic algorithm based on size differences of cyclotomic cosets has been obtained. The

algorithm requires $O(\sqrt{q})$ time and $O(1)$ space. It may very well be the case that a similar or better running time can be achieved for other classes of group orders. There is no speed advantage if the algorithm is applied to groups of prime order, since the factorisation of the group order plays a key role in the operation of the algorithm.

It has been shown how the average and worst-case running time of the algorithm can be improved if not all discrete logarithm values are equally likely to occur.

Furthermore, a set of properties has been derived that applies to the sequences that form the basis of the algorithm. Some of these properties assume a finite field within which the discrete logarithm is to be solved. These and possibly also other properties may allow for an improvement of the algorithm.

Chapter 7

Logarithms: Reduce-Map Algorithm

This chapter considers finite fields $GF(2^m)$, represented as the polynomial ring over $GF(2)$ modulo a primitive polynomial $p(X)$ of degree m over $GF(2)$. For each of these fields, new properties are presented that establish relationships between its elements.

On the basis of some of these properties, a novel approach is proposed for computing discrete logarithms in the multiplicative groups of the fields that can be used for the correction of single-bit errors based on cyclic codes as described in Chapter 2. This approach has been evaluated for all primitive polynomials up to degree 12 and the first primitive polynomials of degree 13 and 14, for which a deterministic algorithm is obtained that is efficient in time and space. The algorithm requires a number of parameters, where currently the optimal set can only be determined in exponential time in the degree of the defining polynomial. This is the reason why only partial results are provided for the time requirements of the algorithm for polynomials of higher degree up to 32.

7.1 Preliminaries

The finite field $GF(2^m)$ is represented as the polynomial ring over $GF(2)$ modulo a primitive polynomial $p(X) = X^m + p_{m-1}X^{m-1} + \cdots + p_1X + p_0$ of degree m, whose coefficients belong to $GF(2)$. The m least significant coefficients of $p(X)$ will be combined in the coefficient vector $p = [p_{m-1}, p_{m-2}, \ldots, p_0]^T$.

It will be convenient to treat the field elements as the states of a Linear Feedback Shift Register (LFSR) that is configured to the generator polynomial $p(X)$ as described in Section 2.3. An example of such an LFSR is shown in Figure 2.5 for the primitive polynomial $p(X) = X^3 + X + 1$. The register content is regarded as the state polynomial

$s(X) = s_{m-1}X^{m-1} + s_{m-2}X^{m-2} + \cdots + s_0$ with the corresponding vector notation $s = [s_{m-1}, s_{m-2}, \ldots, s_0]^T$. A single shift operation of the register corresponds to a multiplication of $s(X)$ by X modulo the generator polynomial $p(X)$. If the LFSR is initialised to a nonzero state s, it will traverse, by continuous shift operations, all of its $q = 2^m - 1$ possible nonzero states as illustrated in Table 7.1 for the primitive polynomial $p(X) = X^5 + X^2 + 1$. The sequence of states produced by such an LFSR is referred to as a maximum-length sequence, for which new relationships are developed in the next section.

In this context the following discrete logarithm problem is considered. The polynomial X is a root of $p(X)$ and thus a primitive element. Then, for a nonzero $s(X)$, the discrete logarithm k with $0 \leq k \leq q - 1$ to the base X is to be determined, such that

$$X^k \equiv s(X) \pmod{p(X)}. \tag{7.1}$$

From the perspective of the LFSR, the discrete logarithm k corresponds to the number of shifts that the register, initialised to the state vector $[0, \ldots, 0, 1]^T$, needs to perform to reach the corresponding coefficient vector s of $s(X)$.

7.2 Shift Register Sequences

This section establishes new facts about maximum-length shift register sequences. In what follows, all possible state vectors of the LFSR are aggregated into the set S^*; the set S includes additionally the zero vector. The 2^m elements of S form an m-dimensional vector space over $GF(2)$.

7.2.1 Sequence Numbering

The states of the maximum-length LFSR sequence will be numbered for convenience, and without loss of generality, in a certain way. A state at the sequence position i will be denoted by $s[i]$. The first state $s[0]$ is defined to equal $s[0] = [0, \ldots, 0, 1]^T$. Consequently, the polynomial vector p appears m states later, such that $s[m] = p$, as can be seen in the example in Table 7.1. With this definition, the position of a state in the sequence is equivalent to the discrete logarithm of that state as described by (7.1).

7.2.2 T Transformation

Within the sequence of LFSR states, it is possible to advance a state $s[i]$ by any number of j positions with a simple invertible linear transformation T, where $s[i+j] = T^j s[i]$. The transformation T, which simulates a single LFSR shift, takes, thereby, the following shape

$$T = \begin{bmatrix} p_{m-1} & 1 & 0 & \cdots & 0 \\ p_{m-2} & 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ p_1 & 0 & 0 & \cdots & 1 \\ 1 & 0 & 0 & \cdots & 0 \end{bmatrix}. \tag{7.2}$$

With this information the transformation T^j that advances a state by j steps, can be expressed as

$$\begin{aligned} T^j &= T^{j-1}T \\ &= T^{j-1}[s[m], s[m-1], \ldots, s[1]] \\ &= [s[j+m-1], s[j+m-2], \ldots, s[j]]. \end{aligned} \tag{7.3}$$

The transformation T^j is thus composed of the m consecutive state vectors $s[j+m-1]$ to $s[j]$.

7.2.3 Parity Vector Spaces

The elements of S can be grouped into different parity sets $P_{l,j}$, which are characterised by the parity level l, where $0 \leq l \leq m-1$, and the parity $j \in \{0,1\}$. For a vector $x \in S$ and a parity level l, the following sub-parity function is introduced

$$b_{l,i}(x) = \sum_{j=0}^{\lceil \frac{m-i}{l+1} \rceil - 1} x_{m-1-i-j(l+1)},$$

where $0 \leq i \leq l$. As an example, the calculation of the sub-parities $b_{2,i}$, where $0 \leq i \leq 2$, for a vector s of length $m = 9$ is shown in Figure 7.1. A parity set $P_{l,j}$ is now defined as

$$P_{l,j} = \{s | b_{l,i}(s) = j, 0 \leq i \leq l, s \in S\}. \tag{7.4}$$

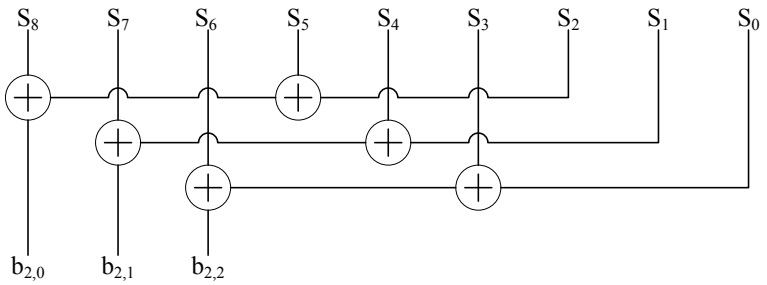

Figure 7.1: Calculation of all sub-parities $b_{2,i}$, where $0 \leq i \leq 2$, for parity level 2 for a vector s of length $m = 9$.

$P_{l,0}$ is considered to be of even parity, whereas $P_{l,1}$ will be of odd parity for each parity level l. It will be convenient to define P_l as

$$P_l = P_{l,0} \cup P_{l,1}.$$

A sample parity set categorisation is shown in the left part of Table 7.1.

In what follows a few properties are established that are inherent to the defined parity sets.

Theorem 7.1. *For $0 \leq l \leq m-1$ and $j \in \{0,1\}$, the number of elements in each parity set $P_{l,j}$ is*

$$|P_{l,j}| = 2^{m-l-1}.$$

Proof. The base set S from which the parity sets are constructed, consists of all possible vectors of length m over $GF(2)$. A single sub-parity $b_{l,i}(s)$ computes a simple parity over a set of bits of a vector $s \in S$. The elements of S are thus divided into two equal sized sets of even and odd sub-parity $b_{l,i}$. The $l+1$ different sub-parities $b_{l,i}$ for a specific parity level l are computed from disjoint bit subsets of s, and are thus independent of each other. For $P_{l,j}$ requires all $l+1$ sub-parities $b_{l,i}$ to equal j, the number of elements satisfying this condition equals $|S|2^{-l-1} = 2^{m-l-1}$. □

Further information about the nature of the parity sets is given by the following theorem.

Theorem 7.2. *For every parity level l, where $0 \leq l \leq m-1$, $P_{l,0}$ and P_l are subspaces of S. The dimension of $P_{l,0}$ and P_l is $m-l-1$ and $m-l$, respectively.*

Proof. The zero vector is, for every level l, of even parity and therefore an element of every $P_{l,0}$. Scalar multiplication is closed in S since $cs \in S$ for all $c \in \{0,1\}, s \in S$. Let $s \in P_{l,j}$ and $\bar{s} \in P_{l,\bar{j}}$ where $j, \bar{j} \in \{0,1\}$. The values of the $l+1$ sub-parities $b_{l,i}$

Table 7.1: Parity set decomposition for the nonzero elements of $\mathbb{Z}_2[X]/\langle X^5 + X^2 + 1\rangle$ in 5-tuple representation. Parity sets on level l with $0 \leq l \leq 4$ are indicated with P_l, where even and odd parity is denoted with o and x, respectively. The parity set elements are aligned with lower level parity set elements in the aligned version which is shown in the right-hand side of the table.

Position	State	Unaligned	Aligned
		$P_0P_1P_2P_3P_4$	$P_0P_1P_2P_3P_4$
0	00001	x	x
1	00010	x	x
2	00100	x	x
3	01000	x	x
4	10000	x	x
5	00101	o o	o o x
6	01010	o o	o o x
7	10100	o o	o o x
8	01101	x	x
9	11010	x	x
10	10001	o o o	o o o o x
11	00111	x x	x
12	01110	x x	x
13	11100	x x	x
14	11101	o x	o x
15	11111	x x	x
16	11011	o o o	o o x
17	10011	x	x
18	00011	o x	o x
19	00110	o x	o x
20	01100	o x	o x
21	11000	o x	o x
22	10101	x x	x
23	01111	o o x	o o o x
24	11110	o o x	o o o x
25	11001	x	x
26	10111	o x	o x
27	01011	x	x
28	10110	x	x
29	01001	o x o	o x
30	10010	o x o	o x

for $0 \leq i \leq l$ will be j for s, and $\bar{j}$ for $\bar{s}$. Therefore, for the sum of s and $\bar{s}$, all the sub-parities will account for $j + \bar{j}$, and it follows $s + \bar{s} \in P_{l,j+\bar{j}}$. If s and $\bar{s}$ are drawn from the same parity set, the sum will be of even parity, otherwise it will be odd. $P_{l,0}$ and P_l are thus closed under addition and they both form a subspace of S.

Furthermore, $P_{l,0}$ and P_l are both vector spaces over $GF(2)$ and have according to Theorem 7.1, 2^{m-l-1} and 2^{m-l} elements, respectively. For this reason, the dimension of $P_{l,0}$ is $m - l - 1$ and the dimension of P_l is $m - l$. □

7.2.4 Blocks

The kernel of an element $s \in S^*$ is defined in this work as the largest sub-vector of s, which has a starting and ending coefficient of 1. For example, the kernel of the vector $s = [0, 0, 1, 0, 1, 0]^T$ is $[1, 0, 1]^T$. If an LFSR is initialised to a state vector s with its kernel of length c located in its least significant part, a total number of $m - c$ shifts will slide the kernel across the register to its most significant end. Provided that $m > 1$ and that the generated sequence is of maximum length, the next LFSR shift will push the most significant kernel bit into the feedback path of the register, which will modify the kernel. Should the kernel remain unaffected, the sequence would have a length of $m - c + 1$, which is impossible as the length would be smaller than the presumed maximum sequence length of $2^m - 1$.

The consecutive LFSR states that are generated by pure LFSR shifts, and which therefore share the same kernel of length c, are considered to form a block of size

$$b = m - c + 1. \tag{7.5}$$

As an example, the first five states in the sequence of Table 7.1 consolidate to a block of size $b = 5$. The zero vector forms an exception to the other vectors. It is considered to generate an even parity block of unspecified length that will be referred to as the zero-block.

The next theorem provides a useful relationship between blocks and parities.

Theorem 7.3. *Even or odd parity set membership is invariant within a block.*

Proof. Let $s \in S$ belong to a block of size b with $b > 1$, and let l denote a parity level with $0 \leq l \leq m - 1$. It is further assumed that $s \in P_{l,j}$ where $j \in \{0, 1\}$. According to (7.4), all the sub-parities of s for parity level l will have the same value j, such that $b_{l,i}(s) = j$ for $0 \leq i \leq l - 1$. Without loss of generality, it will be assumed that a

single kernel-conserving forward LFSR shift on s will lead to the successor $\bar{s}$ in the same block. The sub-parities of $\bar{s}$ will then be the cyclic shifted sub-parities of s, i.e. $b_{l,i}(\bar{s}) = b_{l,((i+1) \bmod (l+1))}(s)$, for $0 \leq i \leq l$. This means that the sub-parities of $\bar{s}$ are identical to those of s, and therefore $\bar{s} \in P_{l,j}$. □

The states of a sequence can thus be classified into blocks of even and odd parity for different levels of parity. In the following theorem, information is provided on the block structure for every level of parity.

Theorem 7.4. *For a maximum-length sequence and parity level l, where $0 \leq l \leq m-1$, the largest block is of odd parity and exhibits a size of $m-l$ with kernel $[1,\ldots,1]^T$ of length $l+1$. The next smaller block is of even parity and has size $m-l-1$ with kernel $[1,0,\ldots,0,1]^T$ of length $l+2$, where $l < m-1$. For every smaller block size b, where $1 \leq b \leq m-l-2$, there are $2^{m-l-b-2}$ blocks of even and $2^{m-l-b-2}$ blocks of odd parity. The total number of blocks on parity level l accounts for 2^{m-l-1}, of which $\lceil 2^{m-l-2} \rceil$ are of odd parity and $\lfloor 2^{m-l-2} \rfloor$ are of even parity.*

Proof. A block of even or odd parity can be characterised by its specific kernel according to Theorem 7.3. It is, therefore, sufficient to analyse the different kernels that occur in a maximum-length sequence. The longer a block is, the shorter is its kernel. The shortest kernel that satisfies the odd parity criterion is of length $c = l+1$ and consists entirely of ones $[1,\ldots,1]^T$. This is the shortest kernel that can set all the $l+1$ sub-parities $b_{l,i}$ to one; it determines the block of size $b = m-l$. With a kernel of size $c = l+1$ or shorter, every sub-parity is computed from at most a single kernel bit. To achieve even parity, all kernel bits would have to be set to zero. As the zero state is excluded from a maximum-length sequence, an even block of size $b = m-l$ or smaller does not exist.

There is only a single parity configuration for a kernel of size $c = l+2$, where $l < m-1$. The two outer bits of the kernel are ones and are the only two kernel bits that belong to a single sub-parity. This sets that particular sub-parity to even parity. The only overall parity class this kernel can qualify for is the even parity class. In this case, each of the remaining kernel bits has to equal zero, leading to the kernel $[1,0,\ldots,0,1]^T$ of length $l+2$, which characterises the block of size $m-l-1$.

For larger kernel sizes c, where $l+3 \leq c \leq m$, an outer kernel bit is always combined with at least one inner kernel bit to form a certain sub-parity. The outer kernel bits are fixed to ones and can thus be left out of the consideration as they do not contribute any degree of freedom to the parity, which can be controlled by

the inner kernel bits. For $c = l + 3$, each inner kernel bit contributes to one of the $l + 1$ sub-parities. There are two options, they all can be set to either zero or to one, depending on which of the two parity classes is targeted. With each increase of the kernel length, a new inner kernel bit contributes a degree of freedom to a sub-parity, doubling the choices for even and odd parity configurations. Therefore, both even and odd kernels number 2^{c-l-3}. Substituting c with the help of (7.5) leads to $2^{m-l-b-2}$.

The summation of the block frequencies for all block sizes for parity level l with $0 \leq l \leq m-3$, leads to the total number of $2 + 2\sum_{b=1}^{m-l-2} 2^{m-l-b-2} = 2 + 2(2^{m-l-2} - 1) = 2^{m-l-1}$, of which 2^{m-l-2} blocks are of even parity and 2^{m-l-2} blocks are of odd parity. On parity level $l = m-2$, there is only one block of even parity and one block of odd parity. Parity level $l = m-1$ has only one block of odd parity. □

7.2.5 M Transformation

There exists a useful relationship between the nonzero blocks of a certain parity level l, which is described by the next theorem.

Theorem 7.5. *Within the blocks of a parity level l of a maximum-length sequence, where $0 \leq l \leq m-1$, the $b-1$ bottom vectors of a block of size b with $b > 1$ of either even or odd parity, can be transformed into an even parity block of size $b-1$, with a linear transformation M of the form $M = T^x$ that is specific to the defining primitive polynomial $p(X)$. The transformation takes thereby the following shape*

$$M = \begin{bmatrix} 1 & \cdots & 0 & 0 & 1 \\ 1 & \cdots & 0 & 0 & p_{m-1} \\ \vdots & \ddots & \vdots & \vdots & \vdots \\ 0 & \cdots & 1 & 0 & p_3 \\ 0 & \cdots & 1 & 1 & p_2 \\ 0 & \cdots & 0 & 1 & \bar{p}_1 \end{bmatrix}.$$

Proof. At first, the existence of the linear transformation M is shown. The left-hand side of M is composed of the block with the kernel $[1, 1]^T$. Due to the fact that the generator polynomial $p(X)$ is primitive, it is guaranteed that this block will appear at some point in the sequence of the shift register states. The rightmost vector of M is simply the state vector that precedes the block. Since M is composed of consecutive state vectors, it describes a linear transformation of the form T^x as shown in Subsection 7.2.2.

To show the effect of the transformation M on the $b-1$ bottom vectors of a block of size b, an arbitrary bottom vector $s = [s_{m-1}, \ldots, s_0]^T$ of the block is considered. Since s is not a topmost block vector, it follows that $s_0 = 0$. With this information, the mapping of s can be expressed as

$$Ms = [s_{m-1} + s_0, s_{m-2} + s_{m-1}, \ldots, s_0 + s_1]^T.$$

Ms is thus the addition of s and its cyclically right-shifted version. The sub-parities for parity level l can therefore be computed as

$$b_{l,i}(Ms) = b_{l,i}(s) + b_{l,(i+1) \bmod (l+1)}(s),$$

for $0 \leq i \leq l \leq m-1$. Vector s belongs to either an even or odd parity block, which means that all the $b_{l,i}(s)$ have the same value and, therefore, $b_{l,i}(Ms) = 0$ for $0 \leq i \leq l \leq m-1$. It follows that $Ms \in P_{l,0}$. The kernel size of Ms increases by one in comparison to s, for its kernel is composed of the kernel of s and its right-shifted version. As a result, the bottommost $b-1$ vectors of a block of size b are mapped by M onto an even parity block of size $b-1$ within the same parity level. □

A block, whose lower vectors are mapped by the transformation M onto the vectors of another block, is considered to be linked to that block. Within a maximum-length sequence, the blocks of even and odd parity of a specific parity level are all linked up to chains by M. Each chain starts with a block of odd parity that links to blocks of even parity. The size of the chain blocks decreases with every link by one until a block of size one is reached. Odd parity blocks of size one do not link to any further blocks and form thus chains of length one.

The chain property is explained in more detail in what follows. The number of odd parity blocks of size b with $b > 1$ within parity level l, where $0 \leq l \leq m-2$, accounts for $\lceil 2^{m-l-3} \rceil$ as can be derived from Theorem 7.4. This number equals the number of blocks of even parity with size one on the same parity level. Furthermore, every block, whose size exceeds one, can be linked to a block of even parity and a size decreased by one in the same parity level according to Theorem 7.5. In this way all the even parity blocks become part of a chain. The only remaining blocks that have not been included in any chain yet, are odd parity blocks of size one, which are considered to form their own chain with only one element. This implies that all the chains start with an odd parity block and end with a block of size one. Chain elements other than the first one are all of even parity.

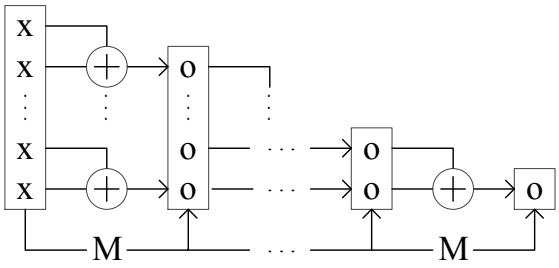

Figure 7.2: M transformation. It links up the blocks of every parity level to chains. A vector of odd parity is indicated with an x, whereas an even parity vector is indicated with an o. Subsequent block vectors can also be generated through the addition of consecutive block vectors.

```
                                1
                              1   1
                            1   0   1
                          1   1   1   1
                        1   0   0   0   1
                      1   1   0   0   1   1
                    1   0   1   0   1   0   1
                  1   1   1   1   1   1   1   1
                1   0   0   0   0   0   0   0   1
              1   1   0   0   0   0   0   0   1   1
            1   0   1   0   0   0   0   0   1   0   1
```

Figure 7.3: Pascal's triangle modulo two.

The chain property is an important result that will play a central role in subsequent sections. For the example sequence given in Table 7.1, $M = T^{17}$, since the state vector $[1, p_4, p_3, p_2, \bar{p}_1]^T$ is located at position 17 in the sequence. The bottommost vectors of the linked chain blocks have, in this case, a displacement of 17 positions in their occurrence in the sequence.

If the M transformation is applied to a non-topmost block vector s, the resulting vector is the sum of s and its right-shifted version,

$$Ms = s + T^{-1}s,$$

as can directly be derived from the definition of M. This relationship is illustrated in Figure 7.2, which shows a complete chain, starting form an odd parity block.

The effect of M on a non-topmost vector s of a block of size b can be described by Pascal's triangle modulo two which is shown in Figure 7.3. A row in the triangle can be obtained by the sum of the previous row with itself right shifted by one. This is the

reason why row r in the triangle with $r < b$ describes the effect of M^r on s as

$$M^r s = \sum_{i=0}^{r} T^{-i} \binom{r}{i} s.$$

If the longest chain on parity level zero is considered, the kernel of the starting block consists of a single one, as is the case with the first row of the triangle. The chain block kernels of the longest chain correspond therefore directly to the rows of the triangle.

Additionally, the following corollary can be established for the inverse transformation of M.

Corollary 7.6. *Within a parity vector space of a maximum-length sequence, a block of even parity of size $b-1$ is transformed by M^{-1} into the bottom $b-1$ vectors of a block of size b, which will be either of even or odd parity.*

Proof. According to Theorem 7.4, the number of even parity blocks of size $b-1$ equals the sum of the number of even and odd parity blocks of size b within the same parity level. Theorem 7.5 shows that the bottom vectors of every even or odd parity block of size b are mapped by M onto the vectors of an even parity block of size $b-1$. Therefore, the converse that the inverse transformation of M maps every even parity block of size $b-1$ to the bottom vectors of either an even or odd parity block of size b must hold. □

An even nonzero vector s is thus mapped by M^{-1} onto a vector t of the previous block. From the fact that the sum of t and its right-shifted version equals s, it is easily possible to calculate t bitwise from s, starting from the most or least significant bit, as illustrated in Figure 7.4.

On the basis of the chain relations between blocks, it is possible to formulate a reduction function that maps every vector to a specific vector of its chain on parity level zero, by shift operations and applications of M. The designated chain vector to which it is reduced could be, for instance, a vector of the starting block or the terminating vector of the chain. Since every chain ends with a block of size one, the number of size-one-blocks equals the number of chains. This means that by omitting the zero vector, the set of elements in S can easily be reduced to a subset of a quarter of its size.

A further useful property of the transformation M is given by the following theorem.

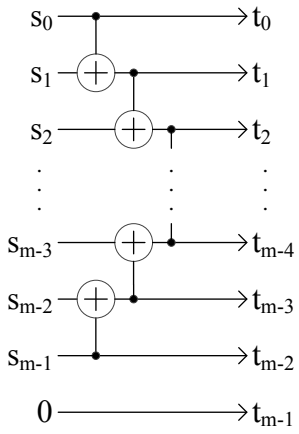

Figure 7.4: M^{-1} transformation. A nonzero even vector s is mapped by M^{-1} onto the corresponding block vector t of the previous block. The calculation of t can easily be accomplished in a serial way, starting from the most or least significant bit.

Theorem 7.7. *The transformation matrix M for a primitive polynomial of degree m fulfils the following condition*

$$\begin{aligned} &M^{2^t}[\underbrace{0,\ldots,0}_{r},s_{2^t-1},\ldots,s_0,0,\ldots,0]^T\\ =&[\underbrace{0,\ldots,0}_{r},s_{2^t-1},\ldots,s_0,s_{2^t-1},\ldots,s_0,0,\ldots,0]^T, \end{aligned}$$

where $r+2^{t+1}\leq m$.

Proof. Induction is used to prove the statement. Since the multiplication of M with a non-topmost block vector s leads to the sum of s and its right-shifted version, the case for $t=0$ holds. It is now assumed that the statement is true for t, leading to

$$\begin{aligned} &M^{2^{t+1}}[\underbrace{0,\ldots,0}_{r},s_{2^{t+1}-1},\ldots,s_0,0,\ldots,0]^T\\ =&M^{2^t}M^{2^t}([\underbrace{0,\ldots,0}_{r},s_{2^{t+1}-1},\ldots,s_{2^t},0,\ldots,0]^T+[\underbrace{0,\ldots,0}_{r+2^t},s_{2^t-1},\ldots,s_0,0,\ldots,0]^T)\\ =&M^{2^t}([\underbrace{0,\ldots,0}_{r},s_{2^{t+1}-1},\ldots,s_{2^t},0,\ldots,0]^T+[\underbrace{0,\ldots,0}_{r+2^t},s_{2^{t+1}-1},\ldots,s_{2^t},0,\ldots,0]^T\\ &+[\underbrace{0,\ldots,0}_{r+2^t},s_{2^t-1},\ldots,s_0,0,\ldots,0]^T+[\underbrace{0,\ldots,0}_{r+2^{t+1}},s_{2^t-1},\ldots,s_0,0,\ldots,0]^T)\\ =&([\underbrace{0,\ldots,0}_{r},s_{2^{t+1}-1},\ldots,s_{2^t},\underbrace{0,\ldots,0}_{2^t},s_{2^{t+1}-1},\ldots,s_{2^t},0,\ldots,0]^T \end{aligned}$$

$$+ [\underbrace{0,\ldots,0}_{r+2^t}, s_{2^t-1},\ldots,s_0,\underbrace{0,\ldots,0}_{2^t}, s_{2^t-1},\ldots,s_0,0,\ldots,0]^T)$$
$$=[\underbrace{0,\ldots,0}_{r}, s_{2^{t+1}-1},\ldots,s_0,s_{2^{t+1}-1},\ldots,s_0,0,\ldots,0]^T.$$

□

A direct consequence of the theorem is the following corollary.

Corollary 7.8. *For the transformation matrix M of degree m and $0 \leq t \leq \lfloor \log_2 m \rfloor$ it follows*

$$M^{2^t-1}[1,0,\ldots,0]^T = [\underbrace{1,\ldots,1}_{2^t},0,\ldots,0]^T.$$

Proof. With Theorem 7.7 it follows

$$M^{2^t-1}[1,0,\ldots,0]^T = M^{2^{t-1}}\cdots M^2 M[1,0,\ldots,0]^T = [\underbrace{1,\ldots,1}_{2^t},0,\ldots,0]^T. \quad (7.6)$$

□

7.2.6 Base Transformations

The parity subspaces $P_{l,0}$ and P_l of the vector space S for $0 \leq l \leq m-1$, can each be characterised by a distinct basis. Their dimensions coincide with the sizes of their largest blocks according to Theorem 7.2 and Theorem 7.4. Each of these blocks consists, furthermore, of a set of linearly independent vectors, for the vectors are generated by a single kernel shifted from one vector end to the other. It follows that the vectors of the unique even parity block of size $m-l-1$ form a basis for the vector space $P_{l,0}$. Similarly, the distinct basis for P_l is formed by the block of odd parity of size $m-l$. The distinct basis will be referred to as base blocks in what follows.

The blocks of a parity set have a fixed location within the sequence of LFSR states. It is possible to align the blocks of $P_{l,0}$ or P_l on parity level l with blocks of a parity set of a lower level by advancing the blocks of one parity set by a certain number of steps forward or backward in their occurrence in the sequence; in other words, $P_{l,0}$ or P_l can be mapped onto $P_{\bar{l},0}$ or $P_{\bar{l}}$ with a linear transformation T^x, where $0 \leq \bar{l} < l \leq m-1$. The vector space P_l will be of particular interest and is thus considered solely in what follows; the principle can, however, be applied in the same way to $P_{l,0}$. To compute the transformation T^x for an alignment of all the blocks of P_l with equally-sized blocks

of $P_{\bar{l}}$, where $0 \leq \bar{l} < l \leq m-1$, it is only necessary to determine the displacement x from the base block of P_l to an equally-sized block of $P_{\bar{l}}$ as will be explained in what follows.

The base block of P_l will be denoted by B_l. It is now assumed that the transformation T^x transforms this block into an equally-sized block on parity level $\bar{l}$; this new block describes an isomorphic representation of P_l and will be denoted as the base block $B_{\bar{l}}$. If the topmost vector of a base block is included in a linear combination of the base vectors, then the resulting vector will be a topmost vector of a block, as only a kernel in a rightmost position can set the rightmost bit to one. Similarly, the bottommost base block vector needs to contribute to the linear combination to obtain a vector that constitutes a lower block boundary. This is one explanation why the aligned blocks will be equally-sized in each case.

Furthermore, there are two options for the base block $B_{\bar{l}}$. It can be either of even or odd parity. If it is of even parity, then any linear combination of the base vectors will result in an even parity vector. The blocks of even and odd parity of P_l would thus be purely aligned with even parity blocks on parity level $\bar{l}$.

If the base block $B_{\bar{l}}$ is of odd parity as is B_l, then each two aligned blocks will have the same parity. This is due to the fact that the parity of a vector obtained through a linear combination of odd vectors is odd if the number of vectors is odd, and even otherwise. Since T^x is an isomorphism, the number of base vectors involved in the linear combination of aligned vectors is the same and, therefore, also their parities.

The base block B_l is of size $m-l$. It can only be aligned with a single block of the same size on the previous parity level $\bar{l} = l-1$. This block is of even parity according to Theorem 7.4. The vector spaces of smaller parity levels $\bar{l}$ with $0 \leq \bar{l} \leq l-2$ contain $2^{l-\bar{l}-2}$ even and $2^{l-\bar{l}-2}$ odd parity blocks of size $m-l$. Hence, the number of possible alignments increases exponentially with the difference between the two parity levels l and $\bar{l}$.

An interesting alignment is achieved if each vector space P_l for $1 \leq l \leq m-1$, is aligned with its previous vector space $P_{\bar{l}}$, where $\bar{l} = l-1$. Table 7.1 shows on its right-hand side this kind of alignment of the parity vector spaces of the sample sequence. The number of blocks in a parity vector space is halved with every step into a higher parity level according to Theorem 7.4. Furthermore, two parity vector spaces on adjacent parity levels can only be aligned in a single way, where the higher level odd parity base block is aligned with the lower parity level even parity block of the same size, as described earlier. The lower level even parity block, is at the same

time, a base for $P_{l-1,0}$, which means that $P_{l-1,0}$ is aligned with P_l. In this constellation, an even parity block is aligned, in half of the cases, with an even parity block of the next higher level, and in the other cases with an odd parity block. Odd parity blocks, in contrast, do not have any corresponding block on the higher parity level. They are in this sense terminating blocks that lie on top of even parity blocks in the stack of parity levels.

Further insights into the nature of the block alignments is given by the following theorem.

Theorem 7.9. *An odd parity block of size b with $b > 1$ on parity level l, links through M to an odd parity block of size $b - 1$ on parity level $l + 1$, if the two parity vector spaces are aligned.*

Proof. Theorem 7.5 shows that an odd parity block of size b with $b > 1$ is linked to an even parity block of size $b - 1$ within the same parity level l. This smaller block can only be aligned with either an even or odd parity block of the next higher parity level $l + 1$ as described in this section. It is first assumed that the smaller block on parity level $l + 1$ is of even parity. If this is the case, the block would link back to a block of size b of either even or odd parity on the same level $l + 1$, according to Corollary 7.6. This block would be aligned with the initial odd parity block of size b on level l. Due to the fact that odd parity blocks are terminating blocks that do not align with any further even or odd parity blocks on the next higher level, a contradiction is established. This shows that the supposition is false, which means that the smaller block on parity level $l + 1$ must be of odd parity. □

An additional result is given by the following theorem.

Theorem 7.10. *If a complete alignment of all the parity vector spaces P_l for $0 \leq l \leq m-1$ is considered, then all the parity vector spaces on level $l = 2^t - 1$ for $0 \leq t \leq \lfloor \log_2 m \rfloor$ are already intrinsically aligned with each other.*

Proof. To align two adjacent parity vector spaces $P_{\bar{l}}$ and P_l with $0 \leq \bar{l} < l \leq m - 1$, it is necessary to determine the displacement between the base block of P_l and the biggest even parity block of $P_{\bar{l}}$. Within a single parity vector space, the base block is linked to the biggest even parity block through the transformation M, as can be derived from Theorem 7.4 and Theorem 7.5. This implies that in a complete alignment of all vector spaces, the base blocks of vector spaces on adjacent parity levels are also linked through M. In other words, the base block of every vector space needs

to be aligned with the equally-sized block of the longest chain on parity level zero, to achieve a complete alignment. The base block for parity level l can be identified through the kernel $[1, \ldots, 1]^T$ of size $l + 1$. For parity level zero, the base block is equivalent to the starting block of the chain. The bottommost vector of this base block equals $[1, 0, \ldots, 0]^T$. It follows, with the help of Corollary 7.8, that the base blocks for parity levels $l = 2^t - 1$ for $1 \leq t \leq \lfloor \log_2 m \rfloor$ appear at their corresponding positions in the chain, which means that their vector spaces are intrinsically aligned with the vector space on parity level zero. □

The sample sequence with its parity vector spaces in Table 7.1 serves as an illustration for Theorem 7.10. It can be seen that vector spaces on level one and three are already intrinsically aligned with the vector space on parity level zero.

The relations that have been developed in this section form the basis for Section 7.4 that will present the new approach for the computation of the discrete logarithm. The next section advances some of the results established in this section.

7.3 Additional Properties

In this section further properties for a maximum-length shift register sequence, generated by a primitive polynomial $p(X)$ of degree m, are derived complementing the ones established in Section 7.2.

7.3.1 Blocks

The next theorem gives useful information on how smaller blocks on higher parity levels can be generated from a block on parity level zero, where all the blocks share the same parity.

Theorem 7.11. *A block of size b on parity level zero is considered. If the bottom j vectors of this block are added together, where $j \leq b$, the bottommost vector of a block of the same parity class on parity level $j - 1$ of size $b - j + 1$ is obtained.*

Proof. Let $s = [s_{m-1}, s_{m-2}, \ldots, s_0]^T$ be the bottommost vector of the block on parity level zero. Due to the fact that s_0 to s_{b-2} are zero, it follows that the addition of the j

bottommost vectors of the block can be expressed as the vector

$$\bar{s} = \begin{bmatrix} s_{m-1} + s_0 + \cdots + s_{(j-2 \bmod (m))} \\ s_{m-2} + s_{m-1} + \cdots + s_{(j-3 \bmod (m))} \\ s_{m-3} + s_{m-2} + \cdots + s_{(j-4 \bmod (m))} \\ \vdots \\ s_0 + s_1 + \cdots + s_{(j-1 \bmod (m))} \end{bmatrix}.$$

It follows that the sub-parities $b_{j-1,i}(\bar{s}) = b_{0,0}(s)$ for $0 \leq i \leq j-1$. This means that the parity class of s on level zero is equivalent to the one of $\bar{s}$ on level $j-1$. Since s is a bottommost vector of a block of size b, it follows that $s_{m-1} = s_{b-1} = 1$; this implies that $\bar{s}_{m-1} = \bar{s}_{b-j} = 1$ and $\bar{s}_{b-j-1}$ to $\bar{s}_0$ are zero, so that $\bar{s}$ is the bottommost vector of a block of size $b - j + 1$. □

7.3.2 L Transformation

It has been shown in Subsection 7.2.5 that the elements of the maximum-length sequence can be combined to blocks, which can be further connected to 2^{m-2} chains on parity level zero. Thus, if every chain is reduced to one of its elements, for instance the last block of size one, the $2^m - 1$ sequence elements are reduced to about a quarter of their number. Each of the obtained elements can be characterised through the length of the chain on parity level zero to which it belongs. If a complete alignment of parity vector spaces is considered, as described in Subsection 7.2.6, the last block of a chain of length h reaches up to parity level $h-1$, where the odd parity terminating block of size one can be found. Therefore, the 2^{m-2} elements will be regarded in what follows either as the blocks of size one on parity level zero, characterised by the length of the chain to which they belong, or equivalently as the odd parity blocks of size one that are spread across the different parity levels.

The following conjecture gives information on how the 2^{m-2} elements can be transformed into each other.

Conjecture 7.12. *For a defining polynomial of degree m and parity level l with $1 \leq l \leq m-3$, there exist $\min(2^{l-1}, 2^{m-3-l})$ linear transformations of the form T^x, such that every odd parity block of size one on level l is mapped by one of the transformations onto an element of an even parity block on the same level.*

It is possible to transform all the 2^{m-2} odd parity blocks of size one into the single block on the highest parity level $m-1$ with the help of this conjecture. Starting from

an arbitrary odd parity block of size one on a certain parity level, the idea is to apply a transformation of the form T^x that will result in an odd parity block of size one on a higher parity level. This process will be repeated until the highest level has been reached. In regard to the transformations, several cases need to be distinguished.

First of all, the considered odd parity block of size one may be located on parity level zero. In this case, it is possible to transform the block to a block of the same parity and size but on a higher parity level in a simple way. First of all, an element of an even parity block is obtained by simply going one step forward or backward in the sequence of states. Primitive polynomials exhibit an odd number of terms, such that an addition of the polynomial to a vector changes its parity class on level zero. Furthermore, since the corresponding state vector of a size one block has the least- and most-significant bit set, moving one step forward or backward in the sequence implies an addition of the polynomial and, therefore, a change in parity class. The obtained even parity block element is either the first or the last vector of a block, depending on the direction of the step. This vector can now be reduced to the last element of the its chain, which will be an even parity block of size one. If the complete parity vector space alignment is taken into account, there will be a parity level, where the terminating block of size one is located. Since the last element of the considered chain is of even parity, the terminating block is on parity level one or higher; a transformation to this block completes the algorithm step, since the odd parity block of size one of a higher parity level has been reached.

If the odd parity block of size one that is to be transformed to a higher level is located on level l with $1 \leq l \leq \lceil \frac{m-2}{2} \rceil - 1$, a maximum number of 2^{l-1} transformations would need to be tested until an even parity block on the same level has been obtained according to Conjecture 7.12. This block can then be reduced to an odd parity block of size one on a higher level in the same way as has been described in the previous case.

The last case concerns parity levels l, where $\lceil \frac{m-2}{2} \rceil \leq l \leq m-3$. Since the number of odd parity blocks of size one accounts for 2^{m-l-3}, which is less or equal to 2^{l-1}, it is possible to select 2^{m-l-3} transformations that each map a block directly onto the block of the highest level. Therefore, a maximum number of 2^{m-l-3} transformations would need to be tested, until the highest level has been reached.

Conjecture 7.12 can be further refined for parity level two as follows.

Conjecture 7.13. *For a defining polynomial $p(X)$ of degree m, where $m \geq 6$, there exist two linear transformations of the form T^x that map each, one half of the odd parity blocks of size one on parity level two, onto elements of even parity blocks on the same parity*

level. The first transformation has an invariant first row of the form $[0, 0, 1, \ldots, 1, 1, 0]$ *for all primitive polynomials, so that the entire matrix can be specified as*

$$L_{2a} = \begin{bmatrix} 0 & 0 & 1 & \cdots & 1 & 1 & 0 \\ \overline{p_{m-1} + p_{m-2}} & 0 & p_{m-1} & \cdots & \overline{p}_{m-1} & \overline{p}_{m-1} & 1 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\ 1 & \overline{p}_1 & \overline{p}_1 & \cdots & p_1 & \overline{p_{m-1} + p_1} & \overline{p_2 + p_1} \\ 0 & 1 & 1 & \cdots & 1 & 0 & \overline{p_{m-1} + p_1} \end{bmatrix}.$$

With the knowledge of this first transformation, the second one can be obtained as $L_{2b} = L_{2a} + \operatorname{diag}(1, \ldots, 1)$.

The following theorem introduces the L transformation.

Theorem 7.14. *The linear transformation*

$$L = \begin{bmatrix} \overline{p}_{m-1} & 1 & 0 & \cdots & 0 & 1 \\ \overline{p}_{m-2} & 1 & 1 & \cdots & 0 & p_{m-1} \\ p_{m-3} & 1 & 1 & \cdots & 0 & p_{m-2} \\ p_{m-4} & 0 & 1 & \cdots & 0 & p_{m-3} \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ p_2 & 0 & 0 & \cdots & 1 & p_3 \\ p_1 & 0 & 0 & \cdots & 1 & \overline{p}_2 \\ 1 & 0 & 0 & \cdots & 1 & \overline{p}_1 \end{bmatrix}$$

maps odd parity blocks of size one on parity level one onto elements of even parity blocks on the same parity level for primitive polynomials $p(X)$ *of degree* $m \geq 4$.

Proof. Let s be a vector of length m that forms an odd parity block of size one on parity level one, i.e. $s \in P_{1,1}$ and $s_0 = s_{m-1} = 1$. Since $p(X)$ is a primitive polynomial of degree m, it exhibits an odd number of terms. Furthermore, if m is assumed to be even, it follows that

$$b_{1,0}(Ls) = \sum_{i=0}^{m-1} p_i + \sum_{i=0}^{\frac{m}{2}-2} s_{2i+1} = \sum_{i=0}^{m-1} p_i + b_{1,0}(s) + s_{m-1} = 0 + 1 + 1 = 0 \quad \text{and}$$

$$b_{1,1}(Ls) = \sum_{i=0}^{m-1} p_i + \sum_{i=1}^{\frac{m}{2}-1} s_{2i} = \sum_{i=0}^{m-1} p_i + b_{1,1}(s) + s_0 = 0 + 1 + 1 = 0.$$

Table 7.2: L transformation. It is indicated to how many different chains on parity level zero of a certain length, the blocks of odd parity on parity level one are mapped to by L according to Conjecture 7.15 for the polynomial degree m with $5 \leq m \leq 12$.

m	Chain Length									
	3	4	5	6	7	8	9	10	11	12
5	0	0	1	0	0	0	0	0	0	0
6	1	0	0	1	0	0	0	0	0	0
7	1	2	0	0	1	0	0	0	0	0
8	2	3	2	0	0	1	0	0	0	0
9	4	6	3	2	0	0	1	0	0	0
10	8	12	6	3	2	0	0	1	0	0
11	16	24	12	6	3	2	0	0	1	0
12	32	48	24	12	6	3	2	0	0	1

For the case that m is odd, it can be shown that

$$b_{1,0}(Ls) = p_0 + \sum_{i=0}^{m-1} p_i + \sum_{i=1}^{\frac{m-1}{2}-1} s_{2i} = p_0 + \sum_{i=0}^{m-1} p_i + b_{1,0}(s) = 1 + 0 + 1 = 0 \quad \text{and}$$

$$b_{1,1}(Ls) = \sum_{i=1}^{m-1} p_i + \sum_{i=0}^{\frac{m-1}{2}-1} s_{2i+1} = \sum_{i=1}^{m-1} p_i + b_{1,1}(s) = 1 + 1 = 0.$$

If both cases for m are considered together, it follows that $Ls \in P_{1,0}$. □

A further important property of the L transformation is given by the following conjecture.

Conjecture 7.15. *The linear transformation L as defined in Theorem 7.14, maps all 2^{m-4} odd parity blocks of size one on parity level one onto half the number of chains on parity level zero, i.e. 2^{m-5} different chains, for $m \geq 5$. These 2^{m-5} chains include the longest chain, $\lfloor \frac{1+2^{m-6}}{2} \rfloor$ chains of length 3 for $m \geq 6$, and $2^{m-h-2} - \lfloor 2^{m-h-4} \rfloor$ chains of length h, where $4 \leq h \leq m-3$, as illustrated by Table 7.2.*

The sequence given in Table 7.1 is used as an example with $m = 5$ to illustrate Theorem 7.14 and Conjecture 7.15. There are two blocks of size one on parity level one of the sequence, which are located at positions 14 and 26. The L transformation can be indicated for this example as $L = T^{10}$. It can be verified that an application of L leads to the vectors at positions 24 and 5, which have even parity on level one.

Furthermore, the vectors are part of one and the same chain on level zero as predicted by Conjecture 7.15, which means that the number of starting vectors has been reduced to half the number of chains.

The reduction of odd parity blocks of size one on level one that is obtained through the L transformation as outlined in Conjecture 7.15, can be further increased if the degree m of the determining polynomial is such that $m \geq 10$ with the following conjecture.

Conjecture 7.16. *The elements that are obtained by mapping the odd parity blocks of size one on level one through L for a primitive polynomial of degree m are considered. It is assumed that the resulting elements are each reduced to an element of their corresponding chain on level zero. The chain element to which the elements are reduced, is assumed to be equal for chains of the same length. According to Conjecture 7.15, there are* 2^{m-5} *different elements that are considered, grouped by the length of the chain to which they belong.*

For h *with* $7 \leq h \leq m-3$ *and* $\bar{h}$ *with* $4 \leq \bar{h} \leq h-3$, *there exist exactly* $\lceil 3 \cdot 2^{h-\bar{h}-4} \rceil$ *transformations of the form* T^x *that map each the elements that belong to chains of length* h *onto elements that belong to chains of length* $\bar{h}$.

At this point a small example using a primitive polynomial of degree $m = 12$ is provided in Table 7.3 to illustrate the element reductions that can be achieved efficiently with the presented results. The starting point are the 2^{m-2} odd parity blocks of size one to which all the nonzero field elements can easily be reduced using the chain properties as described in Subsection 7.2.5. Each of those blocks corresponds to a certain parity level and the number of blocks per level, according to Theorem 7.4, is indicated in Table 7.3. Blocks on level zero can be transformed to blocks on higher levels by going one step forward or backward in the sequence of states as described earlier in the subsection, which can be modelled by T or T^{-1}. For blocks on level two, L_{2a} and L_{2b} as introduced in Conjecture 7.13 can be used to map the blocks to blocks of higher levels. The odd parity blocks of a certain level l are contained in $P_{l,1}$ as introduced in Subsection 7.2.3. Furthermore, P_l can be aligned with $P_{\bar{l}}$, for $\bar{l} \leq l-2$, in such a way that blocks of the same parity align as described in Subsection 7.2.6. This means that for every level l, where $3 \leq l \leq m-1$, a transformation can be determined that will map the blocks on level l onto the blocks on level one. The initial number of $2^m - 1$ nonzero field elements has thus been reduced to the 2^{m-4} odd parity blocks of size one on level one that belong to $P_{1,1}$. A further reduction to half of the number of blocks is achieved if the L-transformation is applied as described in Conjecture 7.15.

Table 7.3: Illustration of the achievable reduction of the nonzero elements of a finite field using the example of a determining polynomial of degree 12. For every parity level, the number of odd parity blocks of size one is indicated to which the $2^{12} - 1$ elements can be initially reduced using the chain property. The blocks on level zero can be transformed to higher level blocks with the help of T or alternatively T^{-1}. In a next step, blocks on level two can be mapped to higher level blocks with L_{2a} and L_{2b}. Blocks of level three or higher can then be mapped onto the blocks of level one by considering the alignment of $P_{1,1}$ with $P_{3,1}$ to $P_{11,1}$. The blocks on level one can then be mapped by L onto half the number of blocks on higher levels. According to Conjecture 7.16, the resulting blocks on level 5 to 11 can be mapped, for instance, to the blocks on level two with a single transformation for each level.

Level	0	1	2	3	4	5	6	7	8	9	10	11
Blocks	512	256	128	64	32	16	8	4	2	1	0	1
T/T^{-1}	0	256	128	64	32	16	8	4	2	1	0	1
L_{2a}/L_{2b}	0	256	0	64	32	16	8	4	2	1	0	1
$P_{1,1}$	0	256	0	0	0	0	0	0	0	0	0	0
L	0	0	32	48	24	12	6	3	2	0	0	1
$P_{2,1}$	0	0	32	48	24	12	0	0	0	0	0	0

The resulting blocks that are located on level two or higher, can be mapped to the blocks on level two to five according to Conjecture 7.16. If the number of blocks that are obtained through the L-transformation on levels two to five are summed up, $29 \cdot 2^{m-10}$ blocks are obtained for $m \geq 10$. Thus, the number of nonzero field elements has been reduced, with a linear number of transformations in the size of the polynomial degree, from $2^m - 1$ to $29 \cdot 2^{m-10}$ for $m \geq 10$.

7.3.3 Q Transformation

The alignments of parity vector spaces P_0 and P_2 are considered. For these two vector spaces, there exist two alignments as described in Subsection 7.2.6. One alignment maps blocks of the same parity onto each other as can be seen in the left-hand side of the example given in Table 7.1, where the blocks of P_0 and P_2 that align have the same parity. The second possible alignment maps all the blocks of P_2 onto even parity blocks of P_0, which can be seen in the right-hand side of Table 7.1. The two alignments are determined by the locations of the odd and even parity blocks of size $m - 2$ in the sequence of states. The Q transformation is defined as the transformation that maps the odd parity block of size $m - 2$ onto the even parity block of size $m - 2$. Since P_2 has a quarter of the odd parity elements of P_0, it follows that Q maps at least a

quarter of the odd parity sequence elements onto even parity sequence elements. For the example in Table 7.1, $Q = T^{25}$.

Further information regarding the Q transformation is established in what follows.

Theorem 7.17. *For every parity level l, where $0 \leq l \leq m-3$, Q maps the only odd parity block of size $m-l-2$ onto the only even parity block of the same size on the same parity level.*

Proof. Let s be the bottommost vector of the only odd parity block of size $m-2$ on parity level zero, and $\bar{s}$ the corresponding vector of the even parity block. By definition of Q, it follows that $Qs = \bar{s}$. The bottommost vector of the only odd parity block of size $m-2-l$ on parity level l with $0 \leq l \leq m-3$, can be expressed using Theorem 7.11 as $\sum_{i=0}^{l} T^{-i}s$. If this vector is mapped by Q, it follows that

$$Q \sum_{i=0}^{l} T^{-i}s = \sum_{i=0}^{l} T^{-i}\bar{s}$$

which describes the bottommost vector of the only even parity block of size $m-2-l$ on parity level l. □

This result implies that the Q transformation allows to map a quarter of the odd parity blocks of parity level l with $0 \leq l \leq m-3$ onto odd parity blocks of parity level $l+2$ as explained in Figure 7.5. A quarter of the odd parity blocks on level l is mapped by Q onto equally-sized even parity blocks on the same level. These even parity blocks have their terminating block two levels higher up, if a complete block alignment is considered as outlined in Subsection 7.2.6. Thus, a quarter of the odd parity blocks can easily be upgraded to equally-sized blocks two levels higher up.

The only odd parity blocks of size one on the different parity levels, to which all the other vectors can easily be reduced to, are considered. Furthermore, it is assumed that all the vector spaces have been completely aligned. With the arguments provided in Figure 7.5, it follows that up to level $m-5$, a quarter of the blocks are mapped by Q onto blocks $\Delta = 2$ levels higher up. Furthermore, the only block on level $m-3$ is mapped by Q onto the only block on level $m-1$. In other words, for every level l, where $l \geq 2$, Q^{-1} maps all the corresponding blocks onto blocks $\Delta = 2$ levels further down as illustrated for $m = 10$ in Table 7.4.

Similarly to the Q transformation, it is possible to specify transformations that connect blocks that are further than $\Delta = 2$ levels apart. If a level difference of $\Delta = 3$ is considered, for instance, two transformations exist, E_0 and E_1, where the

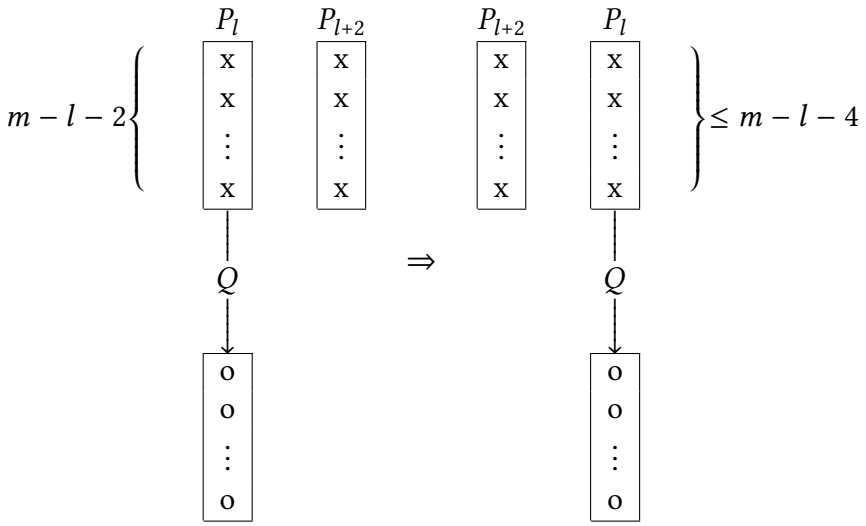

Figure 7.5: Q transformation. It maps the only odd parity block of size $m-l-2$ on parity level l, where $0 \leq l \leq m-3$, onto the only even parity block of the same size on the same parity level. A vector of odd parity for a specific level is indicated with an x, whereas an even parity vector is indicated with an o. If it is assumed that the base block of P_{l+2} is aligned with the only odd parity block of size $m-l-2$ of P_l, it follows that every odd parity block of P_{l+2}, is aligned with an equally-sized odd parity block of P_l. If all the blocks of P_{l+2} are advanced by Q, they will be in the appropriate alignment with the blocks of P_l if a complete block alignment of all parity vector spaces is considered, since the base block of P_{l+2} would need to be aligned with the only even parity block of size $m-l-2$ of P_l. This implies that Q maps a quarter of the odd parity blocks of P_l onto even parity blocks, which have their terminating block in a complete block alignment two levels further up in P_{l+2}.

Table 7.4: Q transformation. A defining polynomial of degree $m = 10$ is considered. For every parity level the number of odd parity blocks of size one is shown. Under the assumption that the parity vector spaces are completely aligned, Q^{-1} maps the blocks of a parity level l, $l \geq 2$, onto blocks two levels further down.

Level	0	1	2	3	4	5	6	7	8	9
Blocks	128	$\xleftarrow{Q^{-1}}$	32	$\xleftarrow{Q^{-1}}$	8	$\xleftarrow{Q^{-1}}$	2		0	
		64	$\xleftarrow{Q^{-1}}$	16	$\xleftarrow{Q^{-1}}$	4	$\xleftarrow{Q^{-1}}$	1	$\xleftarrow{Q^{-1}}$	1

Table 7.5: E transformation. A defining polynomial of degree $m = 10$ is considered. For every parity level the number of odd parity blocks of size one is shown. Under the assumption that the parity vector spaces are completely aligned, E^{-1} indicates how E_0^{-1} or E_1^{-1} map the blocks of a parity level l, $l \geq 3$, onto blocks three levels further down.

Level	0	1	2	3	4	5	6	7	8	9
Blocks	128	$\xleftarrow{E^{-1}}$		16	$\xleftarrow{E^{-1}}$		2	$\xleftarrow{E^{-1}}$		1
		64	$\xleftarrow{E^{-1}}$		8	$\xleftarrow{E^{-1}}$		1		
			32	$\xleftarrow{E^{-1}}$		4			0	

inverse transformations maps the odd parity blocks of size one of each level, onto corresponding blocks three levels further down, as depicted in Table 7.5. In general, the number of possible transformations increases exponentially with the difference in levels Δ, since the number of corresponding alignments increases in the same way as explained in Subsection 7.2.6.

If a transformation is applied which maps the blocks of each level l, $l \geq \Delta$, onto blocks Δ levels further down, a block clustering is obtained. The following theorem specifies the number of clusters that is obtained in this way.

Theorem 7.18. *For a defining primitive polynomial of degree m, a transformation is considered that maps odd parity blocks of size one of each level l onto corresponding blocks Δ levels further down, where $\Delta \leq l \leq m-1$ and $2 \leq \Delta \leq m-1$. The resulting number of clusters equals*

$$\sum_{i=0}^{\Delta-3} \lfloor 2^{m-i-3} \rfloor + \lceil 2^{m-\Delta-1} \rceil + \lfloor 2^{m-\Delta-2} \rfloor. \tag{7.7}$$

Proof. Higher parity blocks are mapped by the transformation onto blocks of lower parity, which means that it is sufficient to determine the number of blocks on the lowest levels. Since the difference in levels between mapped blocks accounts for Δ, the sum of the number of the blocks on the lowest Δ levels equals the number of resulting clusters. From level 0 to $m-3$, the number of odd parity blocks of size one on level l equals 2^{m-l-3}. There is no block of this type on level $m-2$, but one on level $m-1$. □

By using transformation Q to map higher parity blocks onto lower parity blocks, the number of clusters that is obtained equals $\lceil 2^{m-3} \rceil + \lfloor 2^{m-4} \rfloor$, since $\Delta = 2$. As an example the 192 clusters for $m = 10$ are considered which are shown in Figure 7.6.

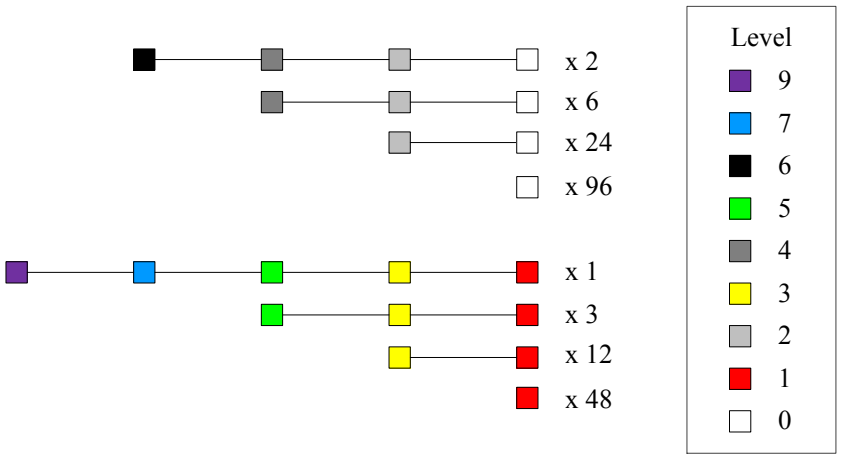

Figure 7.6: Clustering using the Q transformation for $m = 10$. Each coloured box represents an odd parity block of size one, where the colour signifies the corresponding parity level. A connection between two boxes indicates that the higher parity box can be transformed with Q into the lower parity box. For each cluster prototype the number of instances is shown. The total number of clusters is 192.

Table 7.6: Number of clusters obtained using the Q and E transformations for a defining polynomial of degree up to $m = 10$.

Degree	Number of Clusters
1	1
2	1
3	1
4	1
5	4
6	10
7	19
8	37
9	73
10	147
11	297

If Q and the two E transformations are considered together, for instance, the number of clusters reduces from 192 to 147, for the case of $m = 10$; the resulting clustering is shown in Figure 7.7. The number of clusters obtained by considering all three transformations for polynomial degrees up to $m = 10$ can be found in Table 7.6.

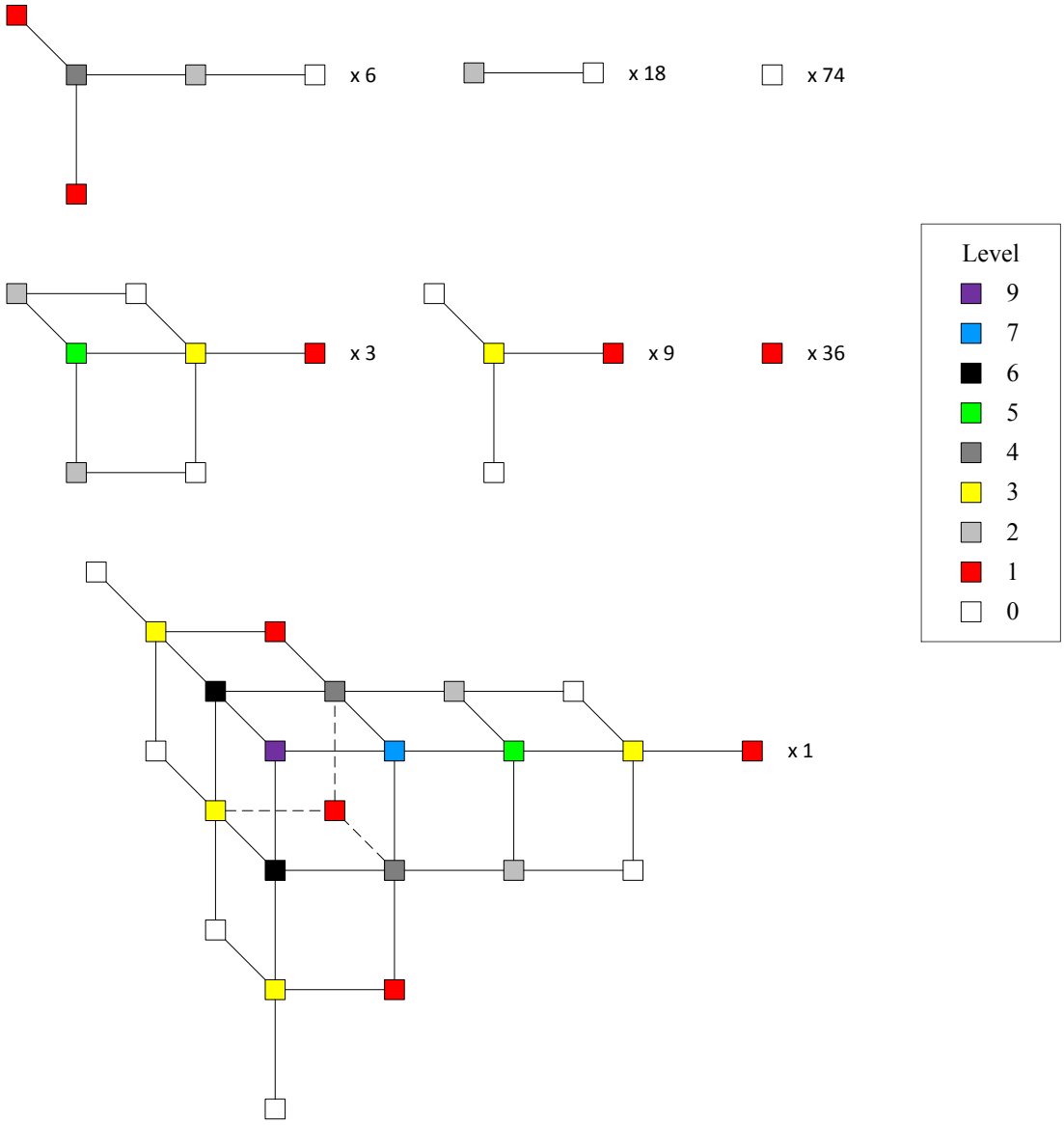

Figure 7.7: Clustering using the Q and E transformations for $m = 10$. Each coloured box represents an odd parity block of size one, where the colour signifies the corresponding parity level. A horizontal connection between two boxes indicates that the higher parity box can be transformed with Q into the lower parity box. The other two connection types indicate the relationship for the two E transformations. For each cluster prototype the number of instances is shown. The total number of clusters is 147.

7.3.4 Parity Subspaces

This subsection establishes additional relationships between the different parity vector spaces. The following theorem gives further information on the intrinsic alignment of the vector spaces.

Theorem 7.19. *A primitive polynomial of degree m is considered. The base block of parity level l, $0 \leq l \leq m-1$, is intrinsically aligned with a block on level $\bar{l}$, where $0 \leq \bar{l} \leq l$, only if $\frac{l+1}{\bar{l}+1}$ is an integer. If the integer is even, the parity of the block on level $\bar{l}$ is even, otherwise it is odd.*

Proof. Let s be the bottommost vector of the base block on level l, so that

$$s = [\underbrace{1, \ldots, 1}_{l+1}, 0, \ldots, 0]^T.$$

Base blocks are of odd parity and for the sub-parities for level l it follows that $b_{l,i}(s) = 1$ for all i with $0 \leq i \leq l$. Now it is assumed that $0 \leq \bar{l} \leq l$ and

$$\frac{l+1}{\bar{l}+1} = r.$$

If r is an integer,

$$b_{\bar{l},i}(s) = \sum_{j=0}^{r-1} b_{l,i+(\bar{l}+1)j}(s) \equiv r \pmod 2,$$

for all i with $0 \leq i \leq \bar{l}$. Therefore, if r is additionally even, all sub-parities on level $\bar{l}$ are also even and $s \in P_{\bar{l},0}$, otherwise all sub-parities are odd and $s \in P_{\bar{l},1}$.

For the case, where r is not an integer, the following two sub-parities for level $\bar{l}$

$$b_{\bar{l},0}(s) = \sum_{j=0}^{\lceil r \rceil - 1} s_{m-1-j(\bar{l}+1)} \equiv \lceil r \rceil \pmod 2 \quad \text{and}$$
$$b_{\bar{l},\bar{l}}(s) = \sum_{j=0}^{\lfloor r \rfloor - 1} s_{m-1-j(\bar{l}+1)-\bar{l}} \equiv \lfloor r \rfloor \pmod 2$$

are different, which implies that $s \notin P_{\bar{l}}$. □

This result implies, for instance, that the base block of a parity vector space P_l for a primitive polynomial of degree m, where $0 \leq l \leq m-1$, is intrinsically aligned with

an even parity block in P_0 if l is odd. Otherwise, the base block is aligned with an odd parity block in P_0.

Before the next theorem is proven, the following lemma is introduced.

Lemma 7.20. *A defining primitive polynomial of degree m is considered. For* $0 \leq r \leq \frac{m}{2}$, *it follows*

$$M^r[\underbrace{1,\ldots,1}_{r},0,\ldots,0]^T = (T^0 + T^{-r})M^{r-1}[1,0,\ldots,0]^T.$$

Proof. The relationship follows from

$$\begin{aligned} M^r[\underbrace{1,\ldots,1}_{r},0,\ldots,0]^T &= M^r(T^0 + T^{-1} + \cdots + T^{-(r-1)})[1,0,\ldots,0]^T \\ &= (T^0 + T^{-1} + \cdots + T^{-(r-1)})MM^{r-1}[1,0,\ldots,0]^T \\ &= (T^0 + T^{-1} + \cdots + T^{-(r-1)})(T^0 + T^{-1})M^{r-1}[1,0,\ldots,0]^T \\ &= (T^0 + T^{-r})M^{r-1}[1,0,\ldots,0]^T. \end{aligned}$$

□

Theorem 7.21. *For a primitive polynomial of degree m, all parity vector spaces* P_l *for* $0 \leq l \leq m-1$ *are considered. Let r be the negative displacement between the base block of* $P_{(2i-1)-1}$, $1 \leq i \leq \lfloor \frac{m-1}{2} \rfloor + 1$, *and the block in* P_0 *that would align with the base block in a complete alignment. It follows that the displacement between the base block of* $P_{(2i-1)2^j-1}$ *and the corresponding block in* P_0 *in a complete alignment equals* $r2^j$ *for* $0 \leq j \leq \lfloor \log_2 \frac{m}{2i-1} \rfloor$.

Proof. Induction is used to prove the statement. The bottommost vector of the base block of parity vector space P_l is

$$[\underbrace{1,\ldots,1}_{l+1},0,\ldots,0]^T.$$

In a complete alignment of the parity vector spaces, this vector of P_l needs to be aligned with vector $M^l[1,0,\ldots,0]^T$ of P_0 according to Subsection 7.2.5 and Subsection 7.2.6. The following statement is therefore to be proven

$$T^{r2^j}M^{(2i-1)2^j-1}[1,0,\ldots,0]^T = [\underbrace{1,\ldots,1}_{(2i-1)2^j},0,\ldots,0]^T$$

The case for $j = 0$ is fulfilled, since according to the premise

$$T^r M^{(2i-1)-1}[1, 0, \ldots, 0]^T = [\underbrace{1, \ldots, 1}_{2i-1}, 0, \ldots, 0]^T.$$

It is now assumed that the statement is true for j. With Lemma 7.20 it follows

$$\begin{aligned}
& T^{r2^{j+1}} M^{(2i-1)2^{j+1}-1}[1, 0, \ldots, 0]^T \\
&= T^{r2^j} M^{(2i-1)2^j} T^{r2^j} M^{(2i-1)2^j-1}[1, 0, \ldots, 0]^T \\
&= T^{r2^j} M^{(2i-1)2^j} [\underbrace{1, \ldots, 1}_{(2i-1)2^j}, 0, \ldots, 0]^T \\
&= T^{r2^j} (T^0 + T^{-(2i-1)2^j}) M^{(2i-1)2^j-1}[1, 0, \ldots, 0]^T \\
&= (T^0 + T^{-(2i-1)2^j}) T^{r2^j} M^{(2i-1)2^j-1}[1, 0, \ldots, 0]^T \\
&= (T^0 + T^{-(2i-1)2^j}) [\underbrace{1, \ldots, 1}_{(2i-1)2^j}, 0, \ldots, 0]^T \\
&= [\underbrace{1, \ldots, 1}_{(2i-1)2^{j+1}}, 0, \ldots, 0]^T.
\end{aligned}$$

□

It is now possible to specify the transformations between the parity vector spaces in a complete alignment for a defining polynomial of degree m as follows. Let R_l denote the transformation that establishes the alignment between the vectors of parity level l and the corresponding vectors on parity level zero, i.e. R_l maps the block of size $m - l$ of the longest chain on level zero onto the base block on level l; and even more precisely

$$R_l M^l [1, 0, \ldots, 0]^T = [\underbrace{1, \ldots, 1}_{l+1}, 0, \ldots, 0]^T.$$

Due to the fact that every nonnegative integer l can be expressed as $l = (2i + 1)2^j - 1$ with $1 \leq i$ and $0 \leq j$, Theorem 7.21 permits every R_l with odd l from a particular $R_{\bar{l}}$ with even $\bar{l}$ to be derived:

$$\begin{array}{cccc}
R_1 = R_0^{2^1} & R_3 = R_0^{2^2} & \cdots & R_{1\cdot 2^j-1} = R_0^{2^j} \\
R_5 = R_2^{2^1} & R_{11} = R_2^{2^2} & \cdots & R_{3\cdot 2^j-1} = R_2^{2^j} \\
R_9 = R_4^{2^1} & R_{19} = R_4^{2^2} & \cdots & R_{5\cdot 2^j-1} = R_4^{2^j} \\
R_{13} = R_6^{2^1} & R_{27} = R_6^{2^2} & \cdots & R_{7\cdot 2^j-1} = R_6^{2^j} \\
\vdots & \vdots & \ddots & \vdots \\
R_{(2i-1)2^1-1} = R_{(2i-2)}^{2^1} & R_{(2i-1)2^2-1} = R_{(2i-2)}^{2^2} & \cdots & R_{(2i-1)2^j-1} = R_{(2i-2)}^{2^j}.
\end{array}$$

Every parity vector space is trivially aligned with itself, so that $R_0 = T^0$; this implies that $R_{2^j-1} = 0$ for all $j \geq 0$, and confirms the result of Theorem 7.10. From the definition of the Q transformation in Subsection 7.3.3 it follows that $R_2 = Q^{-1}$.

7.4 Computing Discrete Logarithms

In this section a new approach is presented to compute the discrete logarithm k of a polynomial $s(X)$ to the base X, such that $X^k \equiv s(X) \pmod{p(X)}$, where $p(X)$ is a primitive polynomial of degree m as described in Section 7.1. This approach is based on the newly established properties from Section 7.2. The discrete logarithm k corresponds in this scenario to the position of the corresponding state vector s of $s(X)$ in the sequence of shift register states, since it has been defined in Subsection 7.2.1 that the state at sequence position zero equals $s[0] = [0, \ldots, 0, 1]^T$.

The main operating principle of the proposed algorithm comprises in the application of repeated linear transformations of the form T^x on the initial vector s, until a certain designated vector $\bar{s}$ is reached. It is known by how many steps each transformation in the algorithm advances a vector in the sequence of states. Moreover, as the position of the designated vector $\bar{s}$ in the sequence is also known, it is possible to work out the starting position of s, once s has been transformed into $\bar{s}$. In what follows, $\bar{s}$ is chosen to be $\bar{s} = [0, \ldots, 0, 1]^T$, which means that if k steps are necessary to reach the vector from the starting position of s, the discrete logarithm will simply correspond to the value $(-k)$.

The transformations that are applied to s to reach $\bar{s}$, revolve around two key functions that are executed repeatedly: reduction and mapping. In the reduction step, s is reduced to a predefined element of its chain on parity level zero as described in Subsection 7.2.5. This element is chosen to be the first element of the chain in what follows, which ensures that $\bar{s}$ can be reached as it is the topmost vector of the biggest odd parity block and, therefore, the first element of the longest chain.

In the mapping step, a linear transformation T^x is applied to the reduced vector. The set of reduced vectors has been divided into partitions according to the length of the chain to which they belong. Each partition features its own specific linear transformation that is used for the mapping. To minimise the runtime of the algorithm, it is necessary to find an optimal set of linear transformations.

The algorithm has been split into an initialisation and a computation phase which are explained in more detail below. Vectors all have a length of m if not specified

otherwise. To determine the leading and trailing zeros of a vector, functions $lz()$ and $tz()$ have been introduced.

7.4.1 Initialisation

During the initialisation phase, a set of algorithm parameters is computed that is specific to the underlying defining polynomial $p(X)$. The pseudocode for this phase is shown in Algorithm 7.1.

Algorithm 7.1 Initialisation pseudocode.

```
1:  procedure INIT(p, m, f)
2:      T = [p|I_{m-1} / 0]
3:      for i = 0 to m − 1 do
4:          pl[i] = 0
5:      end for
6:      k = 0
7:      s = [1, 0, ..., 0]^T
8:      repeat
9:          s = Ts
10:         k = k + 1
11:         if (s = [0, ..., 0, 1, 1]^T) pm = k − 1
12:         if (s = [0, ..., 0, 1, ..., 1]^T) then
13:             pl[m − 1 − lz(s)] = pl[m − 1 − lz(s)] + k
14:         end if
15:         if (s = [0, ..., 0, 1, 0, ..., 0, 1]^T) then
16:             pl[m − 1 − lz(s)] = pl[m − 1 − lz(s)] − k
17:         end if
18:     until s = [1, 0, ..., 0]^T
19:     globalise T, m, f, pl, pm
20: end procedure
```

In a first step, the matrix T is set up according to (7.2), to model the multiplication of a polynomial by X or equivalently a single shift operation of the LFSR.

Subsequently, the sequence of LFSR states is traversed to locate the positions of particular states within the sequence that will define corresponding transformations on the basis of T. These include the matrix M as defined by Theorem 7.5, which is modelled as $M = T^{pm}$. Furthermore, one variant of the reduction function of the main code requires the set of base transformations that aligns all the adjacent parity vector spaces. To translate a vector on parity level l into its corresponding counterpart on the aligned parity level $l + 1$, the transformation $T^{pl[l+1]}$ is used. These transformations

can be derived from the relative positioning of the base blocks of the two vector spaces as described in Subsection 7.2.6. The base blocks are identified through their kernels according to Theorem 7.4.

Parameter f that is supplied to the initialisation procedure is a vector of size $m-2$ with components ranging from 1 to 2^m-2 to describe the transformations that are used for the mapping function.

7.4.2 Main Computation

The algorithm consists essentially of the reduction and mapping steps that are executed in a loop as shown in the pseudocode in Algorithm 7.2 and explained in what follows in more detail. The variable k keeps track of the number of shifts that are

Algorithm 7.2 Main function pseudocode.

```
1: function LOG(s)
2:     k = 0
3:     loop
4:         {s, k} = REDUCE(s, k)
5:         if (lz(s) = m − 1) return −k
6:         {s, k} = MAP(s, k)
7:     end loop
8: end function
```

applied to the starting vector s during the reduction and mapping steps. Once the terminating vector $\bar{s}$ has been reached, which is identified by a chain length of $m-1$, the algorithm returns the discrete logarithm. Since the reduction function reduces s to the first vector of the first block of its chain, the length of the chain is simply determined by counting the number of leading zeros of the reduced vector and adding one to the result. Alternatively, s can directly be compared to $\bar{s}$, instead of checking for the appropriate chain length.

Reduction

The reduction function is based on the chain property of maximum-length shift register sequences as introduced in Subsection 7.2.5. In what follows it will by tailored to reduce an input vector $s \in S^*$ to the first vector of the starting block of the chain on parity level zero to which s belongs. Two different implementation variants are presented for this purpose.

The first implementation relies on the fact that each chain element can be regarded as the bottommost element of a stack, on top of which blocks from aligned parity vector spaces of higher parity levels are located as derived in Subsection 7.2.6. It can be further deduced from Theorem 7.9 that the stack grows with the distance from the starting block of the chain. The first block of a chain is always of odd parity and therefore at the same time the terminating block for the first stack. This means that the first stack ends already on parity level zero. If the chain has a second element, for instance, its stack would reach up to parity level one. In this way, the height of a stack is an indicator for the position within the chain. Therefore, a method to locate the starting block of the chain, is to determine the height of the stack to which s belongs by identifying the parity level of the terminating block of the stack. For this purpose it is necessary to know the displacements of the different parity vector spaces against each other, to climb up the stack. Once the height of the stack is determined, the offset to the first vector of the starting block can be computed and s transformed accordingly as demonstrated with the pseudocode in Algorithm 7.3.

Algorithm 7.3 Reduction function variant 0 pseudocode.

```
1:  function REDUCE_0(s, k)
2:      l = 0
3:      t = s
4:      while t ∈ P_{l,0} do
5:          l = l + 1
6:          t = T^{pl[l]} t
7:      end while
8:      k = k − l ∗ (pm + 1) − tz(s)
9:      s = T^{−l∗(pm+1)} T^{−tz(s)} s
10:     return {s, k}
11: end function
```

The second variant of the reduction function is based on the fact that only the first block of a chain is of odd parity. Through repeated applications of $M^{-1} = T^{-pm}$, it is possible to traverse the chain backwards and test each chain element for odd parity until the starting block has been reached. In a last step, it is necessary to shift the vector to the topmost position within the starting block. The pseudocode for this approach is shown in Algorithm 7.4. This second approach is more efficient than the first one, since it dispenses with the need for climbing up the stack of elements. Furthermore, it only needs to check for even parity on parity level zero, instead of on different parity levels.

Algorithm 7.4 Reduction function variant 1 pseudocode.

```
1: function REDUCE_1(s, k)
2:     while s ∈ P_{0,0} do
3:         s = T^{-pm} s
4:         k = k − pm
5:     end while
6:     k = k − tz(s)
7:     s = T^{-tz(s)} s
8:     return {s, k}
9: end function
```

In both variants of the reduction function, a linear search is performed. The first variant searches for the terminating block of the stack on successive parity levels, whereas the second variant traverses the chain element by element until the starting block is found. It is possible to speed up the process in both cases by performing a binary search instead, for example.

For the first reduction variant, it is possible to use the result of Theorem 7.10 for further simplification. Since all the vector spaces on levels $l = 2^t - 1$ for $1 \leq t \leq \lfloor \log_2 m \rfloor$ are intrinsically aligned with the vector space on level zero, the starting vector s can be directly used to test the parity for those specific parity levels, without the need of applying alignment transformations. This is particularly advantageous in a hardware implementation, where the parity of s can be tested for different levels in parallel.

In comparison to the presented two reduction functions, a reduction to the last vector of the chain, for instance, can easily be realised by determining the size of the block and the position within the block to which s belongs. With this information the offset to the last element can directly be computed, since the block sizes within the chain are always decreasing by one as outlined in Subsection 7.2.5. However, this information alone would not permit a deduction of the total chain length, which is required by the mapping function in the next step. For this purpose one of the first two functions would need to be employed in combination as they provide a mechanism for determining the chain length.

It is, for the algorithm, essentially irrelevant to which vector of the chain is reduced, since different chain vectors can be transformed into each other through a linear transformation of the form T^x, which can be taken care of by the mapping step without any extra cost. Therefore, the second reduction function variant seems to be, at present, the most favourable.

Mapping

The mapping function applies a simple linear transformation of the form T^x to the input vector. This transformation has been made dependent on the length of the chain to which the vector belongs. In the presented mapping function, each input vector is assumed to be reduced to the first vector of its chain. It is therefore possible to determine, easily, the length of the underlying chain, by counting the number of leading zeros of the input vector and adding one to the result. The pseudocode for the mapping function is shown in Algorithm 7.5, where $f[i]$ denotes the number of shifts that is applied to a vector that belongs to a chain of length $i + 1$.

Algorithm 7.5 Mapping function pseudocode.

```
1: function MAP(s, k)
2:     l = lz(s)
3:     s = T^{f[l]} s
4:     k = k + f[l]
5:     return {s, k}
6: end function
```

In the shift register sequence, there exists exactly one chain of length $h = m$ on parity level zero which triggers the algorithm to terminate. It is therefore desirable for the mapping function to map the reduced elements of every chain, in as few steps as possible, onto the longest chain. Interestingly, a chain of length $h = m - 1$ does not exist, for there is no odd parity block of size $m - 1$. For every shorter chain length h with $1 \leq h \leq m - 2$, there are 2^{m-h-2} chain instances present, as this is the number of odd parity blocks of size h according to Theorem 7.4. Since there is only one chain of length $h = m - 2$, it is possible to map its reduced element, in a single step, directly onto the longest chain, ideally onto its reduced element. This transformation can be computed from the displacement of the two elements. For all other chain lengths, there is more than one chain present, whose reduced elements cannot be mapped in general in a single step onto the longest chain. The mappings depend on each other and it is necessary to find the best set of transformations that minimises the runtime for a given optimisation goal, such as the worst-case runtime.

7.4.3 Implementation Details

In an implementation of the algorithm, two obvious and significant improvements can be made to the pseudocode presented in the previous subsections. Firstly, the

transformation $T^{-tz(s)}$ is used to slide the kernel of s to the least significant bit positions of the vector. This operation can easily be realised as a shift operation in software or hardware and does not require a matrix-vector multiplication.

Secondly, the transformation T is used with a finite number of other exponents in the order of the polynomial degree. To be precise, if the second reduction function variant is considered, $m - 2$ transformations of the form $T^{fl[i]}$ for $0 \leq i \leq m - 3$, together with the transformation T^{-pm}, are required. These specific transformations can be precomputed during the initialisation phase then to be used for the main computation.

7.4.4 Example

To illustrate the operation of the algorithm with an example, the primitive polynomial $p(X) = X^6 + X + 1$ is considered as the defining polynomial. The underlying reduction function is assumed to reduce each nonzero state vector of the generated field to the first element of its corresponding chain. These chain elements are associated with nodes as listed in Table 7.7, where they are divided according to their chain length.

The algorithm takes an arbitrary nonzero vector as input and aims to transform it in as few steps as possible into the terminating vector, i.e. node 15, that corresponds to the longest chain, to deduce the position of the initial state vector within the maximum-length sequence of states. For this purpose the starting vector is reduced at first to one of the vectors listed in Table 7.7 with the help of the reduction function. If the terminating vector has not been reached, the mapping function is used to transform the reduced vector to another nonzero vector, which can then again be reduced to one of the vectors in Table 7.7. The application of the mapping and reduction function is repeated until the terminating vector is reached.

The mapping function defines essentially how the vectors in Table 7.7, and thus the nodes, are mapped onto each other. It is desired to use a mapping function that minimises the number of steps that need to be taken to reach the terminating node from any starting node, if the worst-case runtime is considered for instance. An analysis of all possible mapping function configurations has revealed that for the underlying generator polynomial, a maximum number of two steps is required in the best case to reach the terminating node from any other node. There are several mapping function configurations that are optimal from this point of view. One such configuration is $f = [2, 13, 16, 49]$, where each array element indicates by how many steps a reduced vector, whose chain length corresponds to the element index increased

Table 7.7: Reduced nonzero field elements of $\mathbb{Z}_2[X]/\langle X^6 + X + 1\rangle$ in 5-tuple representation with their corresponding chain lengths. Each element is associated with a node.

Node	State	Chain Length
0	100011	1
1	111011	1
2	101001	1
3	100101	1
4	101111	1
5	110111	1
6	111101	1
7	110001	1
8	010011	2
9	011001	2
10	010101	2
11	011111	2
12	001011	3
13	001101	3
14	000111	4
15	000001	6

by one, needs to be advanced in the maximum-length sequence. The graph for this specific configuration is shown in Figure 7.8. It can be seen that every node reaches the terminating node in no more than two steps.

In the considered sample configuration, each node is mapped to a node corresponding to either the same or a longer chain length. This is in general, however, not always the case. Some nodes may be mapped onto nodes with lower chain length.

7.5 Evaluation

The question investigated in this section concerns the runtime behaviour of the presented algorithm. Different performance measures can be applied, such as the worst-case or average-case execution time. The worst-case runtime is of particular interest and becomes the focus of attention in what follows.

The algorithm has a number of degrees of freedom that have an impact on the performance. Firstly, the determining primitive polynomial plays a role in the behaviour

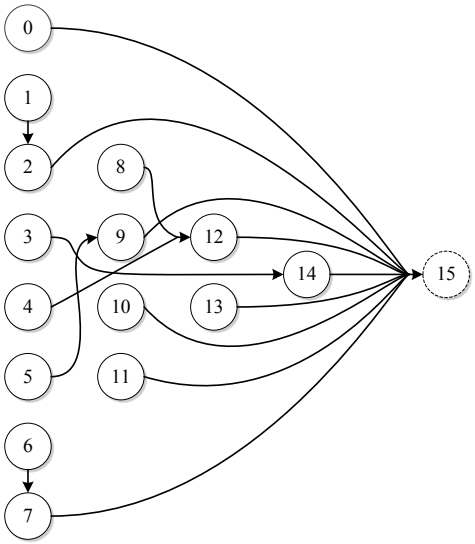

Figure 7.8: Sample mapping configuration $f = [2, 13, 16, 49]$ for $p(X) = X^6 + X + 1$. The terminal node is indicated through a dashed circle.

of the algorithm. Different polynomials of the same degree achieve different execution times; it is therefore desirable to evaluate over all possible primitive polynomials of a certain degree.

Secondly, the algorithm is greatly influenced by the configuration of the mapping function; it is necessary to take, for each polynomial, all the potential configurations into account to determine the best worst-case runtime. For a polynomial of degree m, the mapping function requires $m - 2$ parameters. One of those parameters can easily be derived, as it describes the mapping of the reduced vector that belongs to the single chain of length $h = m - 2$, onto a vector that belongs to the longest chain. It suffices therefore to set the parameter to the difference in sequence position between the terminating vector and the reduced vector that belongs to the single chain of length $h = m - 2$. The remaining parameters can take values in the range between 1 and $2^m - 2$. This leads to a maximum search space of $(2^m - 2)^{m-3}$ parameter combinations.

However, it is not necessary to evaluate every single combination to find the combination that minimises the worst-case runtime. A single mapping parameter is responsible for the mapping of a certain partition of nodes. If a mapping value implies a closed loop within that partition, it can be discarded, as otherwise not all nodes will be able to reach the terminating node. The remaining mapping values will map every source node onto a node outside the partition in a finite number of steps. In the example in Figure 7.8, nodes $[0 - 7]$, which constitute one partition, are mapped in a finite number of steps by mapping value 2 onto the following nodes outside the

Table 7.8: Best achievable worst-case number of mappings steps. For each polynomial degree the number of primitive polynomials and the worst-case numbers of mapping steps are indicated.

Degree	Primitive Polynomials	Mapping Steps
1	1	1*0
2	1	1*0
3	2	2*1
4	2	2*1
5	6	6*1
6	6	6*2
7	18	6*2, 12*3
8	16	16*3
9	48	48*4
10	60	60*5
11	176	176*6
12	144	144*8
13	630	1*10[a]

[a] Only the first of the 630 primitive polynomials of degree 13 has been fully evaluated, i.e. $p(X) = X^{13} + X^4 + X^3 + X + 1$.

partition [15, 15, 15, 14, 12, 9, 15, 15]. Two mappings are considered to be equivalent, if they map the same source nodes onto the same nodes outside the partition with the same number of steps. It is thereby irrelevant if the intermediate nodes differ in the two mappings. Furthermore, a mapping is considered to be better than another, if it requires fewer steps than the otherwise equivalent mapping. If a single parameter is considered, then its range of $2^m - 2$ possible mapping values can be reduced, for instance, by eliminating all those mappings that are already covered by an equivalent or better mapping.

A search with a bounding technique over polynomials of the first degrees has been conducted. The results for fully evaluated polynomials are shown in Table 7.8. More details and results on polynomials of higher degree can be found in Appendix A. The best achievable worst-case running time has been recorded as the number of mapping steps that the algorithm needs to undertake. For all primitive polynomials up to degree 12 and the first polynomials of degree 13 and 14, the number of mapping steps for the worst case does not exceed the degree m of the underlying polynomial.

Interestingly, for the considered polynomials, the number of steps seems to be invariant for polynomials of the same degree, except for the case of $m = 7$, where two different numbers of mapping steps are obtained. If the reduction step is performed with a binary search, its worst-case computation time equals $\lceil \log_2 m \rceil$. The algorithm starts and stops with a reduction step, so that the overall time complexity of the algorithm for evaluated polynomials up to degree 14 is bound by $(m + 1)\lceil \log_2 m \rceil$.

Primitive polynomials of higher degree could only been analysed to some extent due to the exponential parameter search space. With the available computing power, only a small fraction of the entire parameter space was searched with the help of a genetic algorithm. Results on the best sets of parameters that were obtained for polynomials up to degree 32 together with the corresponding number of mapping steps are provided in Appendix A. However, it is highly likely that those values can be improved on due to the small fraction of search space covered.

7.6 Conclusion

The finite field $GF(2^m)$, represented as the polynomial ring over $GF(2)$ modulo a primitive polynomial of degree m over $GF(2)$ has been considered. A new set of properties has been established for the elements of the field.

Based on some of these properties, a novel approach for the solution of the discrete logarithm in the multiplicative group of the finite field has been proposed. This has led to an algorithm with linearithmetic time requirements in the degree of the defining polynomial, for at least all primitive polynomials of degree up to 12 and the first primitive polynomials of degree 13 and 14. The algorithm requires a set of parameters (mapping configuration) in the order of the polynomial degree causing the space requirements to be linear with respect to the polynomial degree. Due to the fact that the parameter search space grows exponentially ($O(2^{m^2})$), determining the optimal set of parameters for a specific polynomial is currently impractical and the reason why partial results are provided for considered single polynomials of degree 14 to 32.

A question that directly follows and remains to be answered is the asymptotic runtime behaviour of the algorithm. It is also interesting to speculate whether it can be proven that loop-free mapping configurations exist for all primitive polynomials of degree $m \geq 13$ as is the case for $m \leq 12$. Moreover, an efficient method to determine an optimal mapping configuration that minimises the worst-case runtime for a given generator polynomial, would also be desirable.

The mapping function has been exclusively optimised against the worst-case execution time. As an interesting alternative optimisation goal, the average-case execution time could be targeted.

The algorithm can be modified and extended in many ways. For instance, it might be possible to use a different partitioning of the reduced set of elements for the mapping function that leads to an improvement in execution time. It is also conceivable that more than one terminating vector could be used to further reduce the running time.

Part IV

Conclusion

Chapter 8

Conclusion

Cyclic codes have gained wide popularity as error-detecting codes due to their inherent algebraic properties that permit easy implementation and effective detection of errors. This thesis supports the position of cyclic codes as a powerful class of error-control code whose properties extend beyond simple error detection. The thesis demonstrates contributions in the generation of efficient, programmable, parallel cyclic code circuits, and in the potential of cyclic codes for efficient error correction. These contributions are described in more detail, together with highlighting possible areas for future exploration within this concluding chapter.

8.1 Programmable CRC

A cyclic code is characterised through its generator polynomial which influences the specific error detection and correction capabilities of the code, depending on the length of the data that is to be protected, as outlined in Chapter 2. In addition, different applications may run on the same system with completely different cyclic code requirements, as in the case of SpiNNaker which is described in Chapter 4. It was shown how cyclic code circuits can overcome the limitations of a single generator polynomial by allowing the circuit to be flexibly programmed with any polynomial within the design constraints. In Chapter 5 a new method for computing the transition and control matrix of a parallel cyclic code circuit was presented. This method allows the efficient realisation of programmable parallel circuits that operate at high speeds, reconfigure rapidly to new polynomials, require few implementation resources, and are energy-efficient when compared with alternative schemes.

8.1.1 Future Work in Programmable CRC

With an efficient programmable cyclic code circuit that can reconfigure rapidly to new generator polynomials, it is feasible to change the polynomial after each processed word of a data stream. The calculation of the next polynomial can be made dependent on factors including the current input data word and the current state of the circuit, i.e. the calculated redundancy and the polynomial. It would be interesting to investigate if a polynomial adjustment algorithm can be devised that has advantages over fixed cyclic code generator polynomials, from both error detection and correction points of view.

8.2 Error Correction

The correction of a single-bit error on the basis of a cyclic code requires the computation of the discrete logarithm in finite cyclic groups, represented as the polynomial ring over the binary field modulo the cyclic code generator polynomial, as outlined in Chapter 2. No efficient algorithm is known for the evaluation of the discrete logarithm in these groups and, moreover, it is also widely believed that no such algorithm can be devised as described in Chapter 3. Nonetheless, this work focused on the exploration of new algorithms in the quest for an efficient calculation of discrete logarithms in relevant groups.

A new approach was developed for calculating discrete logarithms in Chapter 6. For groups that have an order equal to a Mersenne number with an exponent of a power of two, a deterministic generic algorithm was devised based on size differences of cyclotomic cosets. The algorithm requires only constant space and exhibits a worst-case asymptotic running time of the square root of the group order. It was shown that the average- and worst-case running times of the algorithm can be improved for certain cases where the discrete logarithm values occur with unequal probabilities. Furthermore, properties were developed or highlighted for relevant sequences that are considered by the algorithm.

For finite fields with binary characteristic, represented as the polynomial ring over the binary field modulo a primitive polynomial, new properties were developed in Chapter 7. On the basis of a subset of these properties, a novel approach was proposed for computing discrete logarithms in the cyclic multiplicative groups of these fields. It resulted in a deterministic algorithm with linear space and linearithmic time requirements in the degree of the defining polynomial, for at least all polynomials

up to degree 12 and the first polynomials of degree 13 and 14. The algorithm requires a set of parameters in the order of the polynomial degree, where the parameter search space grows exponentially in the polynomial degree. For this reason partial results on the running time for single polynomials of higher degrees up to 32 were provided.

8.2.1 Future Work in Error Correction

The research conducted on discrete logarithm algorithms generated a number of open questions that present potential future research opportunities:

- Under the assumption of the existence of an efficient algorithm for the computation of the discrete logarithm in finite cyclic groups represented in the ring of polynomials modulo a polynomial, the efficient correction of single-bit errors based on cyclic code is feasible. It needs to be investigated further to determine to what extent this assumption would also enable the efficient correction of multi-bit errors.
- For the proposed algorithm for discrete logarithms for group orders that equal Mersenne numbers with an exponent of a power of two, it may be possible to use the developed sequence properties to improve the algorithm. It may also be the case that new properties can be found that will enable a speed-up of the algorithm. In particular, it needs to be investigated if the proposed algorithm reduces the initial discrete logarithm problem into smaller subgroups, similar to the Silver-Pohlig-Hellman algorithm, as this permits alternative algorithms to be employed in those subgroups.
- The proposed generic algorithm for discrete logarithms was tailored to group orders of special Mersenne numbers. It remains an open question as to whether the algorithm can be generalised to all group orders and what the resulting execution overheads would be.
- An algorithm, efficient in time and space, was proposed for the computation of discrete logarithms in the multiplicative groups of small finite fields represented in the polynomial ring over the binary field modulo a primitive polynomial. It was shown that the algorithm is applicable at least to all defining polynomials up to degree 12, and the first polynomials of degree 13 and 14. For single polynomials of degree 15 to 32, it is known that a mapping configuration exists that allows the algorithm to terminate under all conditions, however it is unclear if one exists that also results in an efficient worst-case execution time. It would be interesting to investigate if, for all polynomials with a degree

exceeding 12, loop-free mapping configurations exist, and also what would be the best asymptotic worst-case runtime behaviour of the algorithm. A related open question concerns the best achievable average asymptotic runtime of the algorithm.

- The determination of the overall best mapping configuration for a specific polynomial and optimisation goal was achieved through a brute force attack which is impractical for polynomials of higher degree due to the exponential search space. It may be the case that an efficient method can be devised that allows the computation of the mapping configuration for at least the best worst- or average-case; such a method may also assist in the analysis of the asymptotic runtime behaviours. Alternatively, it may be possible to develop good heuristics to reduce the search space and employ evolutionary algorithms to find a close approximation for the optimal set of values.
- A number of conjectures were established that need analysis to determine if they can be proven. Proofs for the conjectures concerning the L transformation in particular could lead to further insights into the efficient computation of discrete logarithms in the relevant groups.
- The proposed algorithm to compute discrete logarithms in multiplicative groups of small finite fields, represented in the polynomial ring over the binary field modulo a primitive polynomial, might also be easily applicable to defining polynomials that are not primitive, but irreducible. If the defining polynomial is reducible, then it induces only a cyclic group, for which the algorithm might also be easily employed. Moreover, it should be investigated if the algorithm can be adapted to finite fields with a characteristic other than two.
- It was conjectured that, for a defining primitive polynomial of degree m and a parity level l with $1 \leq l \leq \lfloor \frac{m-2}{2} \rfloor$, 2^{l-1} linear transformations of the form T^x exist such that each odd parity block of size one on level l is mapped by one of the transformations onto an even parity block element on the same level. It would be interesting to investigate whether, for every level l, a set of these 2^{l-1} transformations exists, such that the transformations can easily be computed from each other. In particular, it is an open question if the displacement r between the two transformations L_{2a} and L_{2b} for level two can easily be determined, such that $L_{2a} = T^r L_{2b}$.
- It is currently unknown if a simple correlation exists between the displacements of equally-sized blocks on a certain parity level. If the displacements can easily

be computed, alignments of different parity vector spaces can simply be obtained and therefore also transformations such as E_0 and E_1.

8.3 Summary

The work presented in this thesis provides successful solutions for the addressed research objectives:

Efficient Programmable CRC Circuits

A novel method was proposed for the efficient realisation of programmable parallel cyclic code circuits. The resulting circuits can rapidly be configured with a generator polynomial, exhibit fast operating speeds, have low resource requirements, and are energy-efficient at the same time when compared to alternative solutions.

Algorithms for Computing Discrete Logarithms

Two new approaches were developed for computing discrete logarithms to facilitate the correction of single-bit errors based on cyclic codes.

The first approach is generic in nature leading to a deterministic algorithm for group orders that equal a Mersenne number with an exponent of a power of two; this algorithm has constant space requirements and runs in the worst case in the order of the square root of the group order. It was shown how the algorithm can be improved if the discrete logarithm values occur with unequal probabilities and that certain properties hold for the associated sequences.

The second approach for the computation of discrete logarithms is based on a subset of newly developed properties for finite fields of binary characteristic represented as the polynomial ring over the binary field modulo a primitive polynomial. For evaluated small fields, a deterministic efficient algorithm with linear space and linearithmic time requirements in the degree of the defining polynomial was devised.

Appendix A

Mapping Configurations

In what follows, the best mapping configurations that have been found for primitive polynomials up to degree 32 for the proposed discrete logarithm algorithm in Chapter 7 are reported. Polynomials of degree 1 and 2 are not listed as they exhibit only one chain and, therefore, require no mapping steps. All primitive polynomials from degree 3 to 12 and the first polynomials of degree 13 and 14 are listed. For all higher degree polynomials, up to degree 31, only the first polynomial is reported in each case. The Ethernet polynomial serves as a representative for polynomials of degree 32, due to its significance in the Ethernet technology.

Table entries are all in decimal notation, except for the mapping configurations, whose values are indicated in hexadecimal notation. The first value of a mapping configuration is always to be applied to chains of length one, the second, if it exists, to chains of length two, and so forth. For the mapping, it is assumed that elements have been reduced to the last element of their corresponding chain, however, the configurations can easily be converted to other scenarios. There are, in general, several configurations that lead to one and the same number of steps for each polynomial. For the number of steps for all polynomials of degrees up to and including 13, the configuration with the lowest values is reported, where the first of the values is considered as the most-significant one.

The *steps* value indicates the worst-case number of mapping steps that is necessary to map a group element to the longest chain for the indicated configuration. For all polynomials up to degree 12 and the first polynomial of degree 13, the best achievable configuration is reported. The steps for all other polynomials are the best ones that have been obtained through an evolutionary search algorithm; they are indicated with a less-than-or-equal sign to emphasise that better configurations may exist.

Degree	Polynomial	Steps	Configuration
3	3	1	001
3	5	1	006
4	3	1	002 00D
4	9	1	001 002
5	5	1	010 005 019
5	9	1	00F 007 006
5	15	1	00A 002 01C
5	23	1	019 003 00F
5	27	1	006 002 010
5	29	1	015 005 003
6	3	2	002 00C 010 031
6	27	2	001 013 011 01D
6	33	2	008 021 016 00E
6	39	2	001 03C 006 014
6	45	2	001 005 00B 022
6	51	2	006 00A 006 02B
7	3	3	001 015 044 006 055
7	9	3	001 005 00E 00C 056
7	15	3	001 005 004 013 04D
7	17	3	001 00B 008 00E 029
7	29	2	06F 02E 029 05E 03E
7	39	2	061 004 057 05E 07A
7	43	3	001 008 00D 008 021
7	57	2	008 037 04C 00D 041
7	63	3	001 00C 02E 004 02C
7	65	3	001 019 02D 04B 02A
7	75	2	032 013 010 021 008
7	83	2	01B 007 070 004 077
7	85	3	001 006 00B 004 05E
7	101	2	00F 008 035 005 005
7	111	3	001 00B 006 01B 04D
7	113	3	001 009 071 01A 032
7	119	3	001 002 02F 007 032
7	125	3	001 007 01D 005 053
8	29	3	00A 054 063 0F2 0B0 06B

Degree	Polynomial	Steps	Configuration
8	43	3	009 04E 047 0F7 06A 098
8	45	3	008 024 093 091 0A7 0AC
8	77	3	00A 024 0BF 00A 011 0DA
8	95	3	00C 009 039 05D 035 03F
8	99	3	00B 06B 05C 028 02C 01F
8	101	3	004 0F3 014 09C 077 025
8	105	3	006 09B 04C 036 048 053
8	113	3	00B 011 0DF 0A5 01D 094
8	135	3	00D 006 08C 060 01E 05C
8	141	3	005 01E 018 008 035 0E0
8	169	3	003 097 08F 008 03A 067
8	195	3	001 05B 00A 017 043 0A3
8	207	3	00E 055 04D 016 01A 078
8	231	3	01A 02F 092 04A 027 087
8	245	3	005 04E 0C6 0A2 0C3 0C0
9	17	4	004 00B 0A3 0C3 070 010 1E1
9	27	4	00C 039 126 0AB 019 006 10C
9	33	4	005 02B 019 13F 048 051 01E
9	45	4	005 044 02B 033 010 04B 158
9	51	4	004 045 013 056 095 051 0E8
9	89	4	005 03D 1DA 07B 039 004 04E
9	95	4	004 04C 013 0C9 02C 020 1B2
9	105	4	005 1DF 022 064 194 058 1B1
9	111	4	006 04B 108 16B 178 114 011
9	119	4	007 118 0A7 195 0B5 036 05E
9	125	4	001 0B8 0D2 03B 01D 098 037
9	135	4	001 040 09C 00D 093 024 1F8
9	149	4	004 075 065 181 18A 027 1DE
9	163	4	004 003 0D2 015 154 073 138
9	165	4	004 01B 173 02F 071 078 021
9	175	4	006 160 05E 0F5 178 029 0E5
9	183	4	00A 0AD 011 009 05B 023 0ED
9	189	4	001 11A 03B 040 0D3 104 0E4
9	207	4	001 11A 02F 070 083 005 0C0
9	209	4	004 148 0C1 0B9 0B5 07A 0A7
9	219	4	004 0BA 0AA 1AE 0FF 03A 130

Degree	Polynomial	Steps	Configuration
9	245	4	006 0B1 177 157 05C 03C 11B
9	249	4	001 036 168 01C 03A 050 1C8
9	275	4	004 1BE 19C 058 003 0BD 17F
9	277	4	001 1DE 0E5 00E 0BB 0F6 0C7
9	287	4	006 1EA 173 169 089 00F 048
9	291	4	004 09C 0EF 013 042 024 080
9	305	4	005 059 024 107 16A 093 117
9	315	4	00E 00B 0EF 0BC 0B0 031 130
9	335	4	005 029 0D2 07B 00C 006 05D
9	347	4	005 0A5 022 171 1A6 096 0D3
9	353	4	007 047 015 012 071 03C 0F3
9	363	4	003 055 0E1 1CA 161 06D 12C
9	365	4	003 12F 13D 0C7 068 16F 0CF
9	371	4	006 037 05C 11B 035 06A 0CF
9	383	4	001 1FD 0B8 10E 066 009 01F
9	389	4	001 13C 138 085 078 09C 007
9	399	4	001 087 0A8 043 0A4 04D 0BF
9	437	4	009 0EC 088 14A 00C 011 112
9	441	4	005 05D 1EC 0A0 106 00D 1A1
9	455	4	001 031 010 1BC 126 013 140
9	459	4	004 05C 18D 13D 060 00A 1A2
9	461	4	001 0BF 04A 00E 0C5 1F9 13F
9	469	4	001 0E4 12B 019 1D8 0AF 11A
9	473	4	006 010 13E 14B 053 01E 1EE
9	483	4	006 047 09D 08F 17D 0DE 1B7
9	489	4	004 149 0C2 05E 01E 015 04D
9	507	4	001 01E 064 095 005 062 1E0
10	9	5	001 35C 05D 082 2A5 0D3 015 0DD
10	27	5	006 02E 202 131 307 2F7 032 3AB
10	39	5	006 1C2 03B 0BE 327 280 2A1 33A
10	45	5	00B 07F 3A0 226 040 050 059 37F
10	101	5	005 243 027 171 1E0 07C 060 0F5
10	111	5	001 1ED 110 055 13E 15A 2BA 3D4
10	129	5	008 044 1E3 25B 25F 245 2B6 322
10	139	5	006 021 028 195 012 1CC 014 3DD
10	197	5	012 078 2EF 05E 09C 0AE 027 165

Degree	Polynomial	Steps	Configuration
10	215	5	004 05B 262 062 045 29B 017 307
10	231	5	006 1AF 0B8 026 007 0D1 02F 204
10	243	5	001 0DB 22A 0B1 0CF 17F 0E6 247
10	255	5	004 0F5 199 19E 0DA 221 015 0BF
10	269	5	004 0EB 007 09F 099 058 053 247
10	281	5	004 0BE 011 28F 1A6 143 11B 29A
10	291	5	001 1E2 06B 0A2 2C7 0D3 2B9 21C
10	305	5	006 0F4 0DA 178 0CE 1BD 032 30A
10	317	5	008 00D 3B0 021 14A 224 0B4 3F1
10	323	5	00F 176 121 012 0EC 2AB 03A 072
10	343	5	011 189 11F 019 0A5 2AD 047 04D
10	363	5	007 0C3 063 104 050 06C 0B2 180
10	389	5	006 246 05F 091 366 2B0 019 1B8
10	399	5	005 070 3AB 0E1 068 064 014 38E
10	407	5	004 02E 1C0 136 264 261 04C 2D4
10	417	5	006 216 092 196 076 085 015 080
10	455	5	008 16C 27D 0AF 011 140 333 254
10	485	5	005 2EE 0D1 1F0 130 2A5 0C6 00E
10	503	5	00C 06E 39C 031 047 267 011 1B5
10	507	5	005 0C8 103 0D8 35C 061 017 1D8
10	531	5	009 00C 012 1E6 0CE 10E 006 087
10	533	5	016 071 117 0CB 18E 07A 179 38D
10	549	5	001 065 0C0 367 365 013 146 1E3
10	567	5	008 18D 15A 39C 017 0FE 014 233
10	579	5	007 086 127 020 2D7 142 346 378
10	591	5	005 1F6 016 251 3F5 3C8 05D 313
10	603	5	007 0A8 16F 2C5 08A 157 131 25C
10	633	5	004 246 1D5 03E 05A 012 049 1B8
10	639	5	005 055 038 04B 04F 37A 197 150
10	649	5	00E 016 25E 10C 07F 0D7 0C9 022
10	693	5	009 258 0B1 058 0D1 184 34D 27F
10	705	5	00A 1BE 101 37F 3CD 10F 017 054
10	723	5	00E 25B 1C3 102 293 1AC 201 1A3
10	735	5	001 28E 19B 118 0DC 253 3CD 173
10	765	5	001 2D8 19C 0FB 0A0 02E 01F 227
10	791	5	011 0AE 383 104 068 00B 014 350

Degree	Polynomial	Steps	Configuration
10	797	5	008 253 332 137 25E 26E 0CC 1AB
10	801	5	00B 291 31A 022 031 1B9 109 0C5
10	825	5	008 02A 064 00D 19C 033 01C 1FB
10	839	5	007 095 0BF 06C 137 277 0B5 0AF
10	845	5	004 0DA 039 291 011 256 0AF 12B
10	853	5	019 04C 0B6 1BC 18A 01A 02C 3B2
10	857	5	001 306 320 20E 007 02A 038 0F8
10	867	5	007 232 246 34A 25D 06F 24B 1CC
10	893	5	00D 2C9 062 1BA 097 014 034 24A
10	909	5	006 045 081 021 24C 09F 0EC 071
10	915	5	00B 04B 179 0E6 073 02C 07E 0EC
10	945	5	008 195 1D8 13E 1C6 209 145 02B
10	987	5	008 049 265 182 06A 066 032 28C
10	1011	5	001 14B 315 0FA 067 0DC 077 2AF
10	1017	5	00A 2B3 09D 32A 21D 27A 078 340
11	5	6	406 5BF 367 2CF 6ED 051 554 066 232
11	23	6	0B3 1C4 386 19E 7E2 1F2 070 2C0 4D7
11	43	6	119 042 360 497 10C 6BA 0F3 022 4E7
11	45	6	066 1E4 330 505 563 00B 610 087 4DE
11	71	6	0D8 059 1D1 304 5FD 7DA 099 179 14E
11	99	6	148 1A8 09B 283 3EA 045 095 12C 06C
11	101	6	031 073 126 0E7 1C6 0B0 12F 131 78B
11	113	6	12C 1A0 0AE 43F 779 032 083 093 0EF
11	123	6	03D 04E 534 47F 267 347 049 28C 406
11	141	6	025 049 144 0A2 316 330 03F 228 0E1
11	149	6	280 142 0F3 3A2 22D 15D 0DC 0B2 3FC
11	159	6	043 11A 49E 594 5DB 3CA 35D 2AB 6B4
11	169	6	08B 248 1ED 491 442 3E6 2AA 03D 5B6
11	177	6	033 18D 13A 1D7 38A 166 3CF 402 3F6
11	207	6	105 349 6AC 0E8 1FD 310 16B 2E7 01B
11	209	6	046 3B0 20C 322 41C 006 059 3FD 409
11	225	6	158 0EE 182 119 753 1F3 1ED 092 710
11	231	6	382 1DF 1BC 725 2A1 08F 368 080 390
11	235	6	4A6 0E2 25F 074 0DB 3D2 3C9 05F 06E
11	245	6	082 14F 00D 397 18D 586 362 247 62E
11	269	6	123 0C6 125 713 319 0BC 31B 1AB 29B

Degree	Polynomial	Steps	Configuration
11	275	6	125 0C3 012 0C1 0AD 31E 043 4FB 644
11	293	6	15C 16A 10E 2D5 563 11A 31F 020 489
11	297	6	1D6 12E 725 3E4 361 408 06B 0B1 1E6
11	315	6	237 04F 50C 110 143 689 49D 07A 72F
11	317	6	1F2 11B 023 237 176 42F 4E2 171 6E3
11	325	6	2DF 3EB 2D2 221 4C3 140 6A9 020 63D
11	329	6	4E9 114 0D7 090 3D4 426 254 090 619
11	337	6	015 2C3 43B 07A 3BD 3CF 263 13B 249
11	347	6	081 037 4C1 35B 228 249 03F 581 4DB
11	371	6	2E1 345 043 29F 38A 566 7C9 158 104
11	373	6	1E6 687 258 064 229 2D8 2B9 268 070
11	383	6	093 060 08A 3F0 6B1 6A8 06D 024 79E
11	387	6	348 2CA 466 06C 11F 016 0D5 029 3C9
11	399	6	10E 0A4 04E 371 035 15E 2AF 25B 2D4
11	427	6	07C 26E 60A 5A9 248 218 69E 160 4C4
11	429	6	54E 55B 248 52D 149 1DE 195 080 449
11	441	6	02A 0A0 10A 272 2BD 459 0AD 0C8 238
11	455	6	16B 18A 112 073 513 71F 77F 138 674
11	473	6	1B6 237 2DF 177 12D 3A6 364 2D1 5C7
11	485	6	058 32D 04A 230 7C7 1E1 209 0D4 0A6
11	503	6	26F 262 027 48B 6DB 014 19D 009 27F
11	513	6	3D2 228 092 740 2D2 349 6C6 03E 5CD
11	519	6	033 008 16C 039 0B6 428 180 47C 7F6
11	531	6	085 0B1 156 770 059 08B 121 080 049
11	533	6	441 044 3AA 008 2A7 536 151 02B 03C
11	553	6	079 11D 0F3 77F 6C9 493 318 14F 1C2
11	585	6	1DC 0A1 542 675 579 2FD 123 0AD 376
11	609	6	030 62F 131 268 058 710 12F 0F7 074
11	621	6	06E 2B6 2D8 059 101 30B 003 0C6 198
11	633	6	028 0A5 0D3 30F 038 13D 044 19F 759
11	639	6	0BB 518 201 4B5 072 610 044 141 2E6
11	645	6	080 03B 46A 2D4 333 6B9 333 027 7C3
11	657	6	08F 17E 072 015 094 21B 0CE 0AE 403
11	669	6	32D 341 6C4 03B 64C 123 369 2D5 4BD
11	679	6	1C3 186 218 3FA 475 3BE 038 35D 37F
11	683	6	036 24F 37F 129 08A 1D6 26D 13E 7BD

Degree	Polynomial	Steps	Configuration
11	691	6	007 0D0 1DF 67A 1E2 036 170 02A 267
11	693	6	063 032 113 06C 38B 3F7 122 176 7CC
11	725	6	015 39C 2BC 642 329 38C 30E 1BF 033
11	735	6	05E 085 125 493 0CA 0A3 6FB 087 1B7
11	745	6	100 00F 126 09B 156 0A7 1C6 019 78F
11	751	6	112 022 332 0A7 31B 068 06F 3DF 30C
11	753	6	080 1EC 331 01B 24A 328 08A 347 1D1
11	763	6	10B 05A 0E6 455 48F 08C 27A 125 7A4
11	771	6	0E2 4F0 01D 040 038 620 7AC 452 069
11	777	6	1A1 0EA 0E2 1E1 04A 6F9 4EA 0CD 564
11	785	6	12A 0E0 228 724 3B6 4FD 07F 01A 71E
11	819	6	02B 290 0B2 1BA 58A 089 1CC 187 56E
11	831	6	062 33A 213 387 756 1FA 79E 1CE 206
11	833	6	09D 116 391 1BC 076 028 030 3CE 321
11	843	6	021 049 057 506 5B6 448 097 536 105
11	857	6	277 1AF 6FA 56C 013 216 206 218 3B6
11	863	6	031 3F9 2BA 6DA 101 217 0AE 1B6 405
11	869	6	0DF 34E 48A 1E2 0E9 1B1 281 659 667
11	879	6	026 5B3 55B 549 60E 5D9 509 220 1D9
11	893	6	06F 05C 179 4CA 076 3BD 3DB 5F3 734
11	903	6	120 143 173 4F5 20A 523 0A0 018 154
11	907	6	0B8 060 16F 69B 32B 627 1AC 284 320
11	915	6	06B 078 045 46B 4EF 1EE 44B 01F 213
11	917	6	0AB 063 465 65F 1D7 68B 6FF 01B 342
11	943	6	03D 082 147 258 102 540 089 08C 77C
11	951	6	218 0AB 261 147 2A6 304 145 096 243
11	957	6	04C 0E0 029 7F4 07B 462 3C1 2E4 71E
11	969	6	1D4 136 2B2 01A 469 123 3D3 1AE 11C
11	987	6	011 248 201 31B 172 213 037 2C3 280
11	989	6	0EB 36D 3C7 00B 22B 19D 155 0CA 0E1
11	999	6	00C 28D 128 4D9 76A 12C 592 02D 48C
11	1005	6	036 545 22B 545 41C 4B1 4CC 19C 0CB
11	1035	6	133 06C 5A1 36B 11B 68F 1D1 040 789
11	1037	6	1D1 068 0C4 1F5 680 2D9 685 0D9 796
11	1049	6	4B4 07F 09F 57C 103 0F8 2E2 0A8 436
11	1055	6	04E 2EF 157 128 100 0E2 0A8 14A 1C7

Degree	Polynomial	Steps	Configuration
11	1111	6	280 036 139 0EA 6EB 112 016 0A8 38A
11	1121	6	00C 06E 2BF 23C 349 311 005 077 793
11	1131	6	04B 116 1C7 6BD 1EB 05D 422 034 55F
11	1139	6	054 20D 128 348 613 63D 0E0 065 3B6
11	1157	6	1F5 048 4B3 08F 172 070 2F7 07A 7B6
11	1161	6	016 034 11D 544 258 38C 48C 013 1BB
11	1175	6	07C 0FE 02D 689 144 13D 57E 090 543
11	1179	6	492 021 178 03A 0C0 29C 099 032 027
11	1181	6	0D9 13A 543 126 4D1 7EF 1A2 0B3 5EC
11	1203	6	08E 027 0DD 5D4 0D7 0D0 19D 08F 115
11	1215	6	01A 0BA 0DF 15C 170 028 0B0 519 338
11	1223	6	4B8 32E 1F2 16B 69E 671 047 339 1CA
11	1229	6	054 099 0C6 546 14F 77B 553 0E2 291
11	1235	6	18B 028 00E 190 724 1E6 04C 05C 6EA
11	1237	6	149 0CF 157 196 13C 2AE 082 14A 598
11	1251	6	115 02E 58C 5D8 427 0AA 15A 036 449
11	1257	6	277 070 710 7EC 21E 5DD 2F9 134 6FB
11	1271	6	222 081 7CF 028 178 320 17B 1DD 3C6
11	1283	6	201 520 12B 334 0B5 2BE 437 084 076
11	1295	6	173 122 0D3 0ED 3D1 678 648 216 3F8
11	1309	6	026 31F 307 165 249 190 21A 0FB 4DF
11	1319	6	024 261 06C 20D 1BE 1B9 76A 03E 59D
11	1325	6	06E 104 7B3 446 7E7 4C4 1D9 02B 6FA
11	1345	6	104 176 32C 192 7EF 230 0EC 08C 318
11	1351	6	135 209 659 18F 7B6 4DF 17C 092 5F5
11	1365	6	034 3A1 417 284 19D 279 08B 270 042
11	1369	6	012 4C3 6F8 188 7F8 4A1 0EE 078 33B
11	1379	6	057 224 173 32A 728 12D 0CA 070 2A0
11	1391	6	188 2DF 0C1 13F 23C 0CA 1F7 09D 51F
11	1393	6	356 06D 4DA 036 6F3 411 43C 6AA 791
11	1427	6	115 026 1F2 65C 436 0CF 575 0E7 7D8
11	1439	6	147 06B 0B2 3C8 325 039 33A 0C5 793
11	1449	6	175 33C 2A1 0B8 77F 4FB 203 02D 324
11	1467	6	029 191 1BF 6A7 5A6 557 6F6 14C 46C
11	1469	6	067 03E 121 011 35F 0F5 03F 200 57F
11	1481	6	104 04B 0BB 585 64C 5E4 601 3A4 0D0

Degree	Polynomial	Steps	Configuration
11	1495	6	1D1 0A3 392 401 48F 322 3D2 036 316
11	1499	6	1F5 16D 314 13C 6E8 718 10A 1B1 393
11	1505	6	0AA 35C 0E1 6B8 18B 786 76A 018 3F9
11	1511	6	22E 25D 4DD 026 184 7E5 049 351 5A1
11	1525	6	07E 1AD 124 730 745 4ED 651 022 05B
11	1541	6	069 014 689 2F4 69A 40C 185 45D 009
11	1565	6	0CE 153 15C 010 21E 146 372 27E 6AB
11	1569	6	108 074 04B 0E1 7C8 18C 498 45D 6B1
11	1575	6	17A 21E 237 16F 78F 13A 08E 096 0AF
11	1579	6	229 1E7 053 0B7 41D 078 055 00D 20A
11	1587	6	174 056 153 1D9 161 012 5DD 1F3 635
11	1593	6	070 39B 059 7A9 012 33B 07D 033 18B
11	1607	6	13C 0AE 096 15D 33F 692 3F1 238 750
11	1611	6	03E 054 4DE 0A9 5FF 3ED 09B 20D 262
11	1621	6	0E0 223 423 48A 60F 2DF 327 307 480
11	1631	6	0FC 0E1 25E 107 06B 407 33E 123 3B9
11	1649	6	0DF 0B0 1CE 095 577 3D5 016 25F 46F
11	1659	6	106 11E 1D6 1AA 518 08F 1DF 1A6 25E
11	1661	6	360 1D4 264 0B1 047 25F 3EF 35F 373
11	1665	6	166 189 6B0 13C 30D 0A6 142 3E1 328
11	1683	6	0E9 222 1ED 0CB 1DB 06C 0C1 62B 2BC
11	1695	6	043 2DA 609 1DE 3B2 2B6 11F 462 106
11	1699	6	1E2 389 249 079 13C 727 183 074 475
11	1723	6	0CB 139 0B3 044 784 225 36F 119 4E9
11	1743	6	04A 13A 01B 275 0C7 0BB 096 07D 47E
11	1757	6	0FC 242 026 141 269 258 58D 187 5BC
11	1779	6	447 1A2 529 345 0E2 3AD 0B5 05D 439
11	1785	6	4E1 00E 0FA 36F 035 13D 50B 314 580
11	1803	6	5D9 10F 0A0 6B0 77F 5F6 1B2 0B1 407
11	1817	6	26B 2D3 459 315 0D7 175 254 678 52B
11	1841	6	150 01A 153 27C 664 058 7E4 07D 7E4
11	1847	6	067 10C 0F1 0FC 4E4 5B0 284 038 381
11	1885	6	126 327 262 5A8 467 618 0F6 19F 083
11	1899	6	16F 03A 1B0 23E 144 3AA 079 087 2E0
11	1901	6	038 1D8 230 2B3 17D 085 2F6 38B 626
11	1909	6	1D0 30B 223 5B4 2FE 482 435 420 4F3

Degree	Polynomial	Steps	Configuration
11	1923	6	02D 059 2BA 495 138 7E1 107 024 638
11	1937	6	017 3E3 2AA 26C 6EF 65E 5BD 3D7 14B
11	1943	6	089 07D 64D 5A2 380 03F 210 0F4 6F9
11	1947	6	291 2C2 0F2 552 47B 62E 4BF 4A8 06C
11	1959	6	0A7 037 2D6 14C 611 72E 094 03B 446
11	1965	6	0D5 330 221 2F0 6FD 3D5 422 573 3FA
11	1973	6	07F 1B6 116 5A9 04F 150 678 543 648
11	1997	6	05F 205 032 204 0D7 1BD 229 12D 5F9
11	2003	6	1E7 0A6 314 0AC 00D 17C 361 2E6 4C7
11	2021	6	17B 3A5 434 041 591 59E 137 487 519
11	2025	6	256 1F3 7D9 116 4C1 489 1E7 03D 061
12	83	8	001 D04 65C 610 5DD FDF 291 212 2FF B36
12	105	8	001 63F 5D3 0B8 04C 81F 428 60F 104 7A3
12	123	8	00A 0EE 8BC 878 047 0D4 2E5 6EE 290 3C3
12	125	8	006 167 85A 2CF DC9 083 A04 011 FD4 351
12	153	8	004 146 BFF 715 3AB B70 564 CAB 1D0 E3A
12	209	8	005 2BA 6D4 0C5 7F9 7BD 15D 2B4 0B9 5BA
12	235	8	004 7EE 620 154 092 B29 12F 69D 526 A29
12	263	8	012 1E8 45A 588 FA1 2D7 C26 CF0 24A E16
12	287	8	007 10F 3F5 A4C 1DC 50C 0D3 0F8 266 C72
12	291	8	00E 526 1C0 868 5EC 83A EA9 C34 401 AD8
12	315	8	00E 4C5 0C3 097 5E5 93F B13 781 2CD 42E
12	335	8	004 1A3 564 347 78B 832 250 483 64D B63
12	343	8	001 395 6AA 047 B0A 729 3BA 995 14D 579
12	353	8	009 5B9 068 300 A4D 7CA 177 811 02E A45
12	363	8	001 3E9 3AF 1F8 1BE 6B1 739 01B 0E2 C15
12	389	8	001 25C 518 339 486 4F1 2EA 8E9 0C7 8AB
12	435	8	009 5CD 33F 6BB 2A4 228 2DF 68E 038 347
12	473	8	007 1AC 7F2 17D B5F 8F5 FF1 143 2CC 1D3
12	479	8	004 0F1 3D9 1A0 AF2 EA8 DCA CB8 906 2F6
12	525	8	00B 8EA 465 B1B A3D B4C 98D 2B3 01A 322
12	567	8	007 966 C05 E2E 4F2 312 ED8 28C C4C 698
12	573	8	00B 78E 032 F89 6E5 867 076 2CC 00D 870
12	615	8	015 12F 0D8 25F 379 B80 3A5 019 4EA 30A
12	627	8	024 3EF FA6 3D2 476 925 4F3 1AC 21B 616
12	639	8	001 405 1F6 39D 1AF 45C BA0 D5C 41D E30

Degree	Polynomial	Steps	Configuration
12	697	8	001 DC9 6A6 B94 471 328 772 423 738 235
12	705	8	001 7A2 80E 055 0C1 A53 BD7 9F0 005 85C
12	715	8	001 078 0A8 24F 78E 13D 650 20A 100 42D
12	783	8	006 6FC 075 422 9B7 67A DDC 34A 130 478
12	797	8	00D 4F3 3AC 490 826 B27 C80 45F 12F 543
12	801	8	001 E38 010 D33 DCA 6EC CD3 499 DAF 1C5
12	825	8	009 198 3AD 331 108 A29 F52 61E 487 24C
12	831	8	00A 1BC 289 498 E0B 14F 022 04B 0C0 AF8
12	845	8	013 187 56A 30F EE5 C5F 5C0 6F9 102 E77
12	881	8	00D DC1 075 102 401 116 F5E 878 7AB E2C
12	921	8	009 24B 3B9 2B2 CAE 210 C38 714 2D0 DB3
12	931	8	005 277 492 4EF 0A2 985 26C A8B 092 952
12	937	8	001 126 849 35B A7E 360 88D BDC 07B DCA
12	1031	8	001 449 25D 453 183 54E FCB 3D2 2A0 FF5
12	1073	8	006 327 06A 1E6 803 21E 377 1C1 3A8 754
12	1079	8	007 2F2 485 015 217 23E 335 278 0DB D0C
12	1103	8	004 1D1 07A DFC 865 40A 048 375 A70 E2D
12	1117	8	012 060 308 FE0 36A 914 64B BFB 1AA 425
12	1127	8	006 16B 02B 0B3 FDD 7E9 1A1 291 67E E93
12	1141	8	013 1C0 E42 381 7D5 864 181 420 48D 5BE
12	1191	8	007 766 DEB A59 0BE 74E B23 1F8 215 6B2
12	1197	8	00E 2B2 165 FA4 0A7 CA3 C26 149 13A D4C
12	1235	8	00A 76B 817 181 1B0 67C 799 2BA 21E 893
12	1295	8	001 17B 19B 64C 3FA 4C8 798 134 8C3 B2E
12	1309	8	008 25B 43E 31E 8FF 0BB 0A1 1AC 025 85E
12	1357	8	001 B1A 053 636 764 1D6 CCB BA3 FAB 4E4
12	1427	8	008 09C 8AA BE7 3F2 1DB 3A3 BAB FE9 68F
12	1477	8	011 6B0 087 0DB 645 13A 678 0D8 2F1 A41
12	1495	8	001 E0B 08F 452 671 368 925 27B 49D 045
12	1501	8	01D 354 54A 022 16B 2BF AFB 515 47C B0F
12	1515	8	006 1AA 06A 5FC 18A 2B4 4E7 715 55F A4F
12	1545	8	004 321 318 6DE 2B1 F52 AE9 B1B 070 CDD
12	1607	8	010 193 61A 32A A56 444 749 16F 018 E6B
12	1621	8	006 4B1 099 995 867 EEF E6A 456 054 B1B
12	1625	8	00D DF8 CA0 C70 6C0 86E FC2 11D 02D 188
12	1701	8	005 394 1D6 0E1 168 436 040 0AC 8E3 2B3

Degree	Polynomial	Steps	Configuration
12	1725	8	005 12E 269 25D 56B C6F 208 085 0B2 ED0
12	1813	8	006 2EA 0B4 583 936 047 CAD EE3 499 7A1
12	1817	8	01C 0D5 12F 04F 249 864 1A5 4D9 0FB ABC
12	1859	8	005 347 722 61C 5EC 82E BF8 013 134 4F1
12	1861	8	007 1E2 871 058 34B 1FC 0B4 1F8 135 BDA
12	1909	8	00B 3DD 1E7 55C B45 D4E 501 41C 549 4F0
12	1929	8	00A 739 E95 405 4EA 038 91F AFE 02C 78F
12	1965	8	005 1B5 07A 086 115 E29 C37 4BE 3F8 12F
12	1971	8	001 3D4 83B 1DB 371 8BE 8AC 5F4 01F F11
12	1983	8	008 B7C 9FA 7A4 8C9 43E 326 21F 042 3D8
12	1985	8	006 350 01C 1E9 4EF 9CC 378 2ED 137 CAE
12	2135	8	004 7DB 197 14D 5C1 5C0 7F2 194 297 823
12	2141	8	006 47F 0A4 1A9 32E 842 3FD BA0 5A6 B0E
12	2193	8	009 1B9 3DF 4C6 03F 98A 051 C8B 63F 527
12	2199	8	047 475 5BC 175 5C5 6D3 457 1D9 297 67A
12	2233	8	007 1BE 15A EAB D0D 58C 845 898 3A2 6AD
12	2287	8	006 237 2D7 0E3 50F 240 045 37A 27D 87A
12	2331	8	008 3F5 7B4 6E8 622 499 1D9 217 39A C09
12	2357	8	005 2F9 029 413 607 286 B66 F0E 01F 970
12	2369	8	001 78E 133 180 1B3 020 9AD 57D 114 4C9
12	2405	8	00B 4A8 9DD 4E9 044 C9A 9A3 323 023 76C
12	2427	8	012 25F 0D7 04B 55A 424 005 AF2 117 A00
12	2443	8	001 732 A4C 712 A38 D1C 166 506 028 859
12	2481	8	016 11D B63 547 155 06D E4F 287 0A7 CB8
12	2493	8	004 76E 020 680 4EA 3C2 753 670 23A 0EE
12	2505	8	012 113 533 227 183 3B0 00F 5BD 163 9E9
12	2511	8	00B 424 05F 1F6 40E D24 72F 4A8 261 704
12	2535	8	007 195 3FB 582 4E1 C7E CE6 3A7 497 E69
12	2587	8	006 20F 6E4 2A3 868 8CB 871 F31 0A9 721
12	2603	8	026 A2A 210 820 239 362 F12 630 06B 5D4
12	2611	8	009 093 06B C1D 63E 0D5 D89 3AF 0C1 7A6
12	2665	8	003 42C 65A 859 C3A D99 A8C 509 A93 BD2
12	2699	8	00A 3A3 934 497 CA1 9D7 0ED 066 0AD A2B
12	2769	8	004 1F1 22B 3E3 7EA 00D FF6 1D2 21B 3EA
12	2785	8	004 238 1D0 45A 5EE 6C3 025 153 5C8 5D6
12	2805	8	005 25B 12F 2FA 537 8C7 72C 0D9 294 5B0

Degree	Polynomial	Steps	Configuration
12	2827	8	018 720 161 D6C 8AE F28 7E8 299 102 8DE
12	2835	8	001 4B2 0F3 85C 1DA 450 0D9 195 038 3F6
12	2847	8	007 13A 94F 275 16A 062 111 479 0FA BBF
12	2903	8	009 5AB 0D0 089 C26 EB7 AAC 6A2 61C A53
12	2961	8	004 246 647 674 704 1D9 1B1 198 250 BD1
12	2983	8	00A 45E 5D7 4E5 AF6 932 FA6 B26 FCA 2D7
12	3007	8	00B 45A 1F7 236 2CB 27A 9EE 015 0C0 6E4
12	3009	8	011 16A 99C 5D0 DB8 85F 04F 206 00B C3C
12	3027	8	00B 6FB 031 AE7 1C8 1AE 960 4DE 259 5FF
12	3077	8	004 276 022 26C C76 22D 349 2BF 5DC 00A
12	3089	8	004 526 2B5 285 05E C1D 2CE EEE 0E2 1E9
12	3095	8	00A 1EB 331 B04 867 87C ACB 2F6 247 E13
12	3111	8	010 373 BDE 9D3 720 DE9 3E2 CA6 0B0 7E1
12	3149	8	032 B28 8B1 22D 71D BC5 6AA 070 184 194
12	3207	8	00F 7E0 0D7 129 317 849 AC0 8E5 150 81E
12	3231	8	00C 230 113 F2F 292 CAD 053 55B 06F DCE
12	3237	8	008 6B1 BDC D63 AE1 467 6F9 1C7 04B 94D
12	3259	8	00C 2D6 6EB CE5 AAE 55B 363 4CF 028 D28
12	3269	8	00A CB6 0DB EDB D92 BEF 2AE E4F 5AD 16C
12	3273	8	005 309 360 96E 3A5 08C 4EB B9A 1EA CF5
12	3279	8	005 A34 13C 46A 5EB 31E 5A3 3E4 C6F 5CA
12	3315	8	008 485 60E CD1 642 770 A6A 1B6 0C4 196
12	3335	8	00A 322 247 415 798 D9A DD0 00C 330 1EC
12	3363	8	022 102 4CC 1B2 F8F 422 C8A 348 271 985
12	3395	8	007 1F4 5B0 64E EE0 E71 577 66D 080 7DC
12	3409	8	001 E69 62D 14C 1CD A62 F6A 342 044 A86
12	3419	8	00A 142 61D C0E 1FE 20E 9DA 230 1F9 5AC
12	3445	8	004 044 49C 0D9 FC5 166 48C 259 08F FBA
12	3461	8	001 1B1 127 598 964 EF1 025 645 23D 2F3
12	3465	8	00A 619 37A 484 7DB 36D 3B4 D3C 1B0 967
12	3605	8	005 4D9 1BA 008 5AE 8DA 85A 77C 1F7 4D1
12	3609	8	004 2E1 1D0 751 1BC FF6 182 82B 10F B87
12	3631	8	009 060 3A3 534 405 591 1F0 434 5A8 6F2
12	3653	8	005 412 56A 240 71C DB8 AFF 12B 3CA 1D2
12	3665	8	005 155 1B0 6C6 335 EE0 0A7 883 145 49C
12	3687	8	00A 55D E55 321 1D3 0E3 23B 856 250 A35

Degree	Polynomial	Steps	Configuration
12	3699	8	024 703 20B 074 6F8 C12 628 028 A92 8FB
12	3727	8	009 59C 48B 58B 526 04A 7FD 1C9 0C3 90D
12	3811	8	004 316 276 9CF 054 203 FB7 16D 316 785
12	3857	8	005 5B1 1B1 A43 804 99A 7AF 21C 179 38D
12	3867	8	001 BBD 0A4 77D 30F B5A AE8 102 041 440
12	3879	8	00E 3EB 437 1BA 2EC F5F 040 8CA 22B 231
12	3953	8	00A 2F5 0D6 3C6 665 8E0 73C A2B 6F9 D09
12	3993	8	003 18E 12B 5F0 03D 9A4 317 281 211 507
12	4027	8	006 066 43D 289 1F4 107 453 DF3 20A 91B
12	4029	8	007 3D7 504 C84 FEE B48 0EB DDD 74A C27
12	4041	8	001 0C7 23A 3FE 282 BA3 45F 2A2 1F6 1CF

Degree	Polynomial	Steps	Configuration
13	27	10	00000006 00001173 00001569 000002E8 0000086B 0000010E 000012B5 00001F0B 00000083 00000277 00000E8B
13	39	≤10	00000016 0000068A 000019B4 00000381 00000354 00001DA2 00000A5F 000001EA 00000019 00000303 00001974
13	53	≤10	00000014 0000012F 000008D5 000000E3 00000500 00000A8B 000006D1 00000D3B 00000153 00000998 00001766
13	83	≤10	00000011 000001F6 000001D8 0000117A 000011D4 00000E0B 00001DB9 00000D9D 0000061A 00001761 00001E08
14	43	≤13	00003DC0 00000129 0000034D 000006CD 00001BF5 0000054F 0000322A 00000638 00000046 00002BCE 000001C7 0000017E
14	57	≤14	00000255 0000140B 00000173 00000F5D 000000B7 000002E7 000034D3 00000EBD 000016A9 00000854 00002725 00003850
15	3	≤18	0000000D 000001C1 000000B5 00005165 0000732A 00005237 0000134B 000072D4 00001D91 00001667 00000C9E 0000374F 00007F2D

Degree	Polynomial	Steps	Configuration
16	45	≤28	00001FAD 00007ECF 00006661 000051CC 0000697B 00000DB2 0000DBE0 00000041 0000F07F 0000F19A 0000C609 000003F3 00000221 0000B0C1
17	9	≤61	0001036D 00007CDA 00011FE2 00002A5D 0001E6DE 00002043 00008A24 00003BD3 00012548 0000039B 0000A8FD 0001CB73 00011F81 0000361F 00006CEB
18	39	≤81	0000843B 00021AD7 000344FF 00023FC1 00016B9A 00018474 000280BB 000390A5 00016DD2 0001556A 0001EE56 0002E38F 0003C840 000240D6 000304AA 00037AE1
19	39	≤113	0005299B 000495D6 00052D38 000163AA 00002CD3 0003723A 0003070D 0001D428 0002BCBD 0003EE87 0007F799 0004436A 00066828 00057A57 00024D87 0001E6D4 0004E3C0
20	9	≤166	000AE538 0008BBE4 000C8E91 00049F0B 000E72D3 0009578C 0000A351 000253CC 0004B6A2 0009A742 000285B3 00017391 000D7A0C 00092A1D 000698D7 00040102 0004CE8D 0009B470
21	5	≤291	000576E7 00140E67 001062A4 0016D0F5 00098BCF 00145193 000E5B31 001B0C7E 001C7B95 0009118E 0005C7A6 001756B3 000B717D 000F0EE2 00044B20 0007831A 00054F8D 0015141D 001F896F
22	3	≤400	00268980 0030ECE3 003681F6 00375861 0000F6B0 000A3D3C 00154596 00234095 0004C035 0024E5E7 002DB34D 0009D8C7 003EF417 0027B38A 003A70F7 0019BF85 000133B8 000C7603 00365798 003B73E8

Degree	Polynomial	Steps	Configuration
23	33	≤569	000A6465 00486ADB 000E4806 0013B97D 000786C8 006CAB7E 000CA432 0023BE8D 007D686C 0037206A 00782F41 0076AC67 00612496 00369C23 00070158 007BDF35 0035DE3F 0048A4E4 004BBDED 0067D3EF 005658C1
24	27	≤905	004774A4 001AC2CA 00F84F68 0091A8FB 000F814A 002F9BCF 00F81D2F 0060CC1F 006418DF 001DC95B 003A096B 0044D19C 003A04BD 00DEF33E 006FD6E3 00D442A4 0018ECD8 00B57C7D 008897F5 00F9E3FF 003D1195 00382D5D
25	9	≤1387	00567FDE 018FE8DD 00033387 013319BC 01471F55 01436837 0055C83D 00F9A114 015F452D 01640357 00B9CC52 0167F429 00593170 016B82EE 00B3983B 0122C78C 0111CE0A 012DCDF9 01ADD7A7 0132B534 01918353 0087762F 003E79CB
26	71	≤2013	006962A1 0289CA9F 01909C6F 01933ADC 00B7B022 02741D2C 024D9E38 034E8B2A 0303E017 005A6D76 007BEEE8 035E64F1 003CE575 01D6C37A 02E499D0 009DA383 028BB31D 0137A24D 0127BE35 030FAC8B 036BB17F 00F3500E 0193A155 018E1FDA
27	39	≤5797	03C9F6CD 05468C8E 01617EB8 06DD3ACE 00306CBA 01D2881B 01C5C7B8 07742302 01925731 00F48EC1 061FDBFB 0472F9D2 05D8F3E4 07FE5676 01159A93 059A860C 00ABB609 04351335 03F6947D 041B1E64 07F1D344 07AFBCE9 01FEDFA3 0079F2F9 01732922

Degree	Polynomial	Steps	Configuration
28	9	≤8056	03D1B1B6 0581C8A8 01C6B20F 0AB12C82 0190091A 01327485 080B80DA 04933FDD 0FAC6F2D 09928229 05A3CABF 0C9CD60C 0685578B 022D57D2 00A1C913 049A91BA 0B5EC368 0BA2C228 0F16BC50 0E8A8E35 0F095DE9 02F01D6E 004572A9 0DE3BB1F 04FA6941 04F03280
29	5	≤9569	08735A86 0AADC14D 0ECAEA07 110A2F37 087B7A96 11F7FE1D 12B547A5 1AD81694 1CF20334 09A9CF23 062AA3A4 0C34124F 10C71018 0ABBB01F 0D9CF109 0E26ABF0 1464EB09 146D8DCA 1840FA7C 0CA804F4 1751221A 08A69531 1E4AB34C 0E1FD109 02F7B23E 130FAF35 00FB4F3A
30	83	≤10563	3A60662A 2B31E568 293E5022 316F0BF8 084A9BD0 103089A5 163CB93D 340DCADA 0DA71080 35F52834 22A5ACDA 100B0F1E 32EBA53C 24E26D52 10791B3C 3049EF69 1D2B6168 096B107E 21231947 3C569F92 3DD1E40F 12DDBF91 3B5C3F4A 1092BFA9 0E9C70FC 06D9DE07 119571BF 000BC8D2
31	9	≤21120	6CBBED58 4EEE9371 64511DEC 7362449A 4A04EFE4 0CE28EA9 141BF300 753A2E5F 3F3C890B 217E13D4 28EA7DC3 0B67A71B 01F1628D 335D676F 63109B3F 6D9054D7 15656497 29DF15BF 030A394F 2CE35F05 3CE70663 4F2420DA 5F213707 400B55B0 18773B43 67F6DBE2 68253F43 06A5CA14 000BFFDE

Degree	Polynomial	Steps	Configuration
32	79764919 (Ethernet)	≤28404	29F631DB 69B4AB1E 67B69DF4 263A31FE 4AFB7882 6E433C07 7F66BBF3 32AEBA18 7663EABE 4F3E5EA1 77AF3DA3 74274A6D 32C01345 621BA771 1C5CBB85 3278ACCE 3819536E 34FCCC0A 61C0187E 74976060 5BF7155F 7EB1D250 6B954F8D 1D891372 448D2221 70DF6506 716AA6D2 1DF650D1 4BA13234 D21F45B0

Bibliography

[AD93] Leonard M. Adleman and Jonathan DeMarrais. "A subexponential algorithm for discrete logarithms over all finite fields". In: *Mathematics of Computation* 61.203 (Sept. 1993), pp. 1–15. ISSN: 0025-5718. DOI: 10.1090/S0025-5718-1993-1225541-3.

[AD94] Leonard M. Adleman and Jonathan DeMarrais. "A Subexponential Algorithm for Discrete Logarithms over All Finite Fields". In: *Advances in Cryptology - CRYPTO '93*. Ed. by Douglas R. Stinson. Vol. 773. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, July 1994, pp. 147–158. ISBN: 978-3-540-57766-9. DOI: 10.1007/3-540-48329-2_13.

[Adl79] Leonard M. Adleman. "A subexponential algorithm for the discrete logarithm problem with applications to cryptography". In: *20th Annual Symposium on Foundations of Computer Science*. Los Alamitos, CA, USA: IEEE Computer Society, Oct. 1979, pp. 55–60. DOI: 10.1109/SFCS.1979.2.

[Adl94] Leonard M. Adleman. "The function field sieve". In: *First International Symposium on Algorithmic Number Theory*. Vol. 877. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer-Verlag, 1994, pp. 108–121. ISBN: 3-540-58691-1. DOI: 10.1007/3-540-58691-1_48.

[AH99] Leonard M. Adleman and Ming-Deh A. Huang. "Function Field Sieve Method for Discrete Logarithms over Finite Fields". In: *Information and Computation* 151.1-2 (May 1999), pp. 5–16. ISSN: 08905401. DOI: 10.1006/inco.1998.2761.

[AS90] G. Albertengo and R. Sisto. "Parallel CRC generation". In: *IEEE Micro* 10.5 (Oct. 1990), pp. 63–71. ISSN: 0272-1732. DOI: 10.1109/40.60527.

[Ber65] Elwyn Berlekamp. "On decoding binary Bose-Chadhuri-Hocquenghem codes". In: *IEEE Transactions on Information Theory* 11.4 (Oct. 1965), pp. 577–579. ISSN: 0018-9448. DOI: 10.1109/TIT.1965.1053810.

[BF02] J. Bainbridge and S. Furber. "Chain: a delay-insensitive chip area interconnect". In: *IEEE Micro* 22.5 (Sept. 2002), pp. 16–23. ISSN: 0272-1732. DOI: 10.1109/MM.2002.1044296.

[BM77] Garrett Birkhoff and Saunders Mac Lane. *A Survey of Modern Algebra*. 4th ed. Macmillan Publishing Co., Inc., 1977. ISBN: 0-02-310070-2.

[Bra+96] Michael Braun, Jörg Friedrich, Thomas Grün, and Josef Lembert. "Parallel CRC Computation in FPGAs". In: *FPL '96: Proceedings of the 6th International Workshop on Field-Programmable Logic, Smart Applications, New Paradigms and Compilers*. Ed. by Reiner Hartenstein and Manfred Glesner. Vol. 1142. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 1996, pp. 156–165. ISBN: 978-3-540-61730-3. DOI: 10.1007/3-540-61730-2_16.

[BRC60] R.C. Bose and D.K. Ray-Chaudhuri. "On a class of error correcting binary group codes". In: *Information and Control* 3.1 (Mar. 1960), pp. 68–79. ISSN: 0019-9958. DOI: 10.1016/S0019-9958(60)90287-4.

[BS96] Eric Bach and Jeffrey Outlaw Shallit. *Algorithmic Number Theory, Volume I: Efficient Algorithms*. Foundations of Computing. Cambridge, Massachusetts, USA: The MIT Press, 1996. ISBN: 0-262-02405-5.

[Buc01] Johannes Buchmann. *Computing discrete logarithms in the multiplicative group of a finite field*. Tech. rep. CRYPTREC, Dec. 2001.

[Chi64] Robert T. Chien. "Cyclic Decoding Procedures for Bose-Chaudhuri-Hocquenghem Codes". In: *IEEE Transactions on Information Theory* 10.4 (Oct. 1964), pp. 357–363. ISSN: 0018-9448. DOI: 10.1109/TIT.1964.1053699.

[CMP03] Richard E. Crandall, Ernst W. Mayer, and Jason S. Papadopoulos. "The twenty-fourth Fermat number is composite". In: *Mathematics of Computation* 72.243 (Dec. 2003), pp. 1555–1572. ISSN: 0025-5718. DOI: 10.1090/S0025-5718-02-01479-5.

[Cop84] Don Coppersmith. "Fast evaluation of logarithms in fields of characteristic two". In: *IEEE Transactions on Information Theory* 30.4 (July 1984), pp. 587–594. ISSN: 0018-9448. DOI: 10.1109/TIT.1984.1056941.

[CP06] C. Cheng and K.K. Parhi. "High-Speed Parallel CRC Implementation Based on Unfolding, Pipelining, and Retiming". In: *IEEE Transactions on Circuits and Systems II: Express Briefs* 53.10 (Oct. 2006), pp. 1017–1021. ISSN: 1057-7130. DOI: 10.1109/TCSII.2006.882213.

[CPR03] G. Campobello, G. Patane, and M. Russo. "Parallel CRC realization". In: *IEEE Transactions on Computers* 52.10 (Oct. 2003), pp. 1312–1319. ISSN: 0018-9340. DOI: 10.1109/TC.2003.1234528.

[CS09] Junho Cho and Wonyong Sung. "Efficient Software-Based Encoding and Decoding of BCH Codes". In: *IEEE Transactions on Computers* 58.7 (2009), pp. 878–889. ISSN: 0018-9340. DOI: 10.1109/TC.2009.27.

[CW94] Douglas W. Clark and Lih-Jyh Weng. "Maximal and near-maximal shift register sequences: efficient event counters and easy discrete logarithms". In: *IEEE Transactions on Computers* 43.5 (May 1994), pp. 560–568. ISSN: 0018-9340. DOI: 10.1109/12.280803.

[DA01] Peter Dayan and L. F. Abbott. *Theoretical neuroscience: computational and mathematical modeling of neural systems*. Computational neuroscience. The MIT Press, 2001. ISBN: 0-262-04199-5.

[Der01] J.H. Derby. "High-speed CRC computation using state-space transformations". In: *GLOBECOM'01. IEEE Global Telecommunications Conference (Cat. No.01CH37270)*. IEEE, 2001, pp. 166–170. ISBN: 0-7803-7206-9. DOI: 10.1109/GLOCOM.2001.965100.

[DH76] W. Diffie and M. Hellman. "New directions in cryptography". In: *IEEE Transactions on Information Theory* 22.6 (Nov. 1976), pp. 644–654. ISSN: 0018-9448. DOI: 10.1109/TIT.1976.1055638.

[Elg85] T. Elgamal. "A public key cryptosystem and a signature scheme based on discrete logarithms". In: *IEEE Transactions on Information Theory* 31.4 (July 1985), pp. 469–472. ISSN: 0018-9448. DOI: 10.1109/TIT.1985.1057074.

[FB09] S. Furber and A. Brown. "Biologically-Inspired Massively-Parallel Architectures - Computing Beyond a Million Processors". In: *ACSD'09: Proceedings of the 9th International Conference on Application of Concurrency to System Design*. 2009, pp. 3–12. ISBN: 978-0-7695-3697-2. DOI: 10.1109/ACSD.2009.17.

[FT07] Steve Furber and Steve Temple. "Neural systems engineering". In: *Journal of the Royal Society, Interface / the Royal Society* 4.13 (Apr. 2007), pp. 193–206. ISSN: 1742-5689. DOI: 10.1098/rsif.2006.0177.

[Fur+12] Steve B. Furber, David R. Lester, Luis A. Plana, Jim D. Garside, Eustace Painkras, Steve Temple, and Andrew D. Brown. "Overview of the SpiNNaker System Architecture". In: *IEEE Transactions on Computers* (2012). Preprint. ISSN: 0018-9340. DOI: 10.1109/TC.2012.142.

[Fur12] Steve B. Furber. "To build a brain". In: *IEEE Spectrum* 49.8 (Aug. 2012), pp. 45–49. ISSN: 0018-9235. DOI: 10.1109/MSPEC.2012.6247562.

[Gal02] Joseph A. Gallian. *Contemporary Abstract Algebra*. 5th ed. Houghton Mifflin, 2002. ISBN: 0-618-12214-1.

[Gal12] Steven D. Galbraith. *Mathematics of public key cryptography*. Cambridge University Press, 2012. ISBN: 1-107-01392-5.

[GF11] Martin Grymel and Steve B. Furber. "A Novel Programmable Parallel CRC Circuit". In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 19.10 (2011), pp. 1898–1902. ISSN: 1063-8210. DOI: 10.1109/TVLSI.2010.2058872.

[GG05] Solomon W. Golomb and Guang Gong. *Signal Design for Good Correlation: For Wireless Communication, Cryptography, and Radar*. Cambridge, U.K.: Cambridge University Press, 2005. ISBN: 978-0-521-82104-9.

[Gol82] Solomon W. Golomb. *Shift Register Sequences*. Revised ed. Laguna Hills, CA, USA: Aegean Park Press, 1982. ISBN: 0-89412-048-4.

[Gor93] Daniel M. Gordon. "Discrete Logarithms in $GF(p)$ using the Number Field Sieve". In: *SIAM Journal on Discrete Mathematics* 6.1 (Jan. 1993), p. 124. ISSN: 0895-4801. DOI: 10.1137/0406010.

[Gra+04] R. Granger, A. Holt, D. Page, N. Smart, and F. Vercauteren. "Function Field Sieve in Characteristic Three". In: *Algorithmic Number Theory*. Ed. by Duncan Buell. Vol. 3076. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2004, pp. 223–234. ISBN: 978-3-540-22156-2. DOI: 10.1007/978-3-540-24847-7_16.

[Hoc59] Alexis Hocquenghem. "Codes correcteurs d'erreurs". French. In: *Chiffres* 2 (Sept. 1959), pp. 147–156.

[JL02] Antoine Joux and Reynald Lercier. "The Function Field Sieve Is Quite Special". In: *Algorithmic Number Theory*. Ed. by Claus Fieker and David Kohel. Vol. 2369. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2002, pp. 343–356. DOI: 10.1007/3-540-45455-1_34.

[JL03] Antoine Joux and Reynald Lercier. "Improvements to the general number field sieve for discrete logarithms in prime fields. A comparison with the gaussian integer method". In: *Mathematics of Computation* 72.242 (Nov. 2003), pp. 953–968. ISSN: 0025-5718. DOI: 10.1090/S0025-5718-02-01482-5.

[JL06] Antoine Joux and Reynald Lercier. "The Function Field Sieve in the Medium Prime Case". In: *Advances in Cryptology - EUROCRYPT 2006*. Ed. by Serge Vaudenay. Vol. 4004. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2006, pp. 254–270. ISBN: 978-3-540-34546-6. DOI: 10.1007/11761679_16.

[Jou+06] Antoine Joux, Reynald Lercier, Nigel Smart, and Frederik Vercauteren. "The Number Field Sieve in the Medium Prime Case". In: *Advances in Cryptology - CRYPTO 2006*. Ed. by Cynthia Dwork. Vol. 4117. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2006, pp. 326–344. ISBN: 978-3-540-37432-9. DOI: 10.1007/11818175_19.

[Kan+12] Eric R. Kandel, James H. Schwartz, Thomas M. Jessell, Steven A. Siegelbaum, and A.J. Hudspeth. *Principles of neural science*. 5th ed. New York: McGraw-Hill Medical, 2012. ISBN: 0-07-139011-1.

[Kas64] Tadao Kasami. "A decoding procedure for multiple-error-correcting cyclic codes". In: *IEEE Transactions on Information Theory* 10.2 (Apr. 1964), pp. 134–138. ISSN: 0018-9448. DOI: 10.1109/TIT.1964.1053649.

[KC04] Philip Koopman and Tridib Chakravarty. “Cyclic Redundancy Code (CRC) Polynomial Selection For Embedded Networks”. In: *DSN ’04: Proceedings of the 2004 International Conference on Dependable Systems and Networks*. Washington, DC, USA: IEEE Computer Society, 2004, pp. 145–154. ISBN: 0-7695-2052-9. DOI: 10.1109/DSN.2004.1311885.

[Kel] Wilfrid Keller. *Prime factors $k \cdot 2^n + 1$ of Fermat numbers F_m and complete factoring status*. URL: http://www.prothsearch.net/fermat.html.

[KLS02] Michal Křížek, Florian Luca, and Lawrence Somer. *17 Lectures on Fermat Numbers: From Number Theory to Geometry*. Springer, Feb. 2002. ISBN: 0-387-95332-9.

[Knu97] Donald E. Knuth. *The Art of Computer Programming, Volume II: Seminumerical Algorithms*. 3rd ed. Reading, Massachusetts: Addison Wesley, 1997. ISBN: 0-201-89684-2.

[Koo02] Philip Koopman. “32-bit cyclic redundancy codes for Internet applications”. In: *DSN ’02: Proceedings of the 2002 International Conference on Dependable Systems and Networks*. Washington, DC, USA: IEEE Computer Society, 2002, pp. 459–468. ISBN: 0-7695-1597-5. DOI: 10.1109/DSN.2002.1028931.

[KRM08] Christopher Kennedy and Arash Reyhani-Masoleh. “High-speed parallel CRC circuits”. In: *Proceedings of the 42nd Asilomar Conference on Signals, Systems and Computers*. Oct. 2008, pp. 1823–1829. ISBN: 978-1-4244-2940-0. DOI: 10.1109/ACSSC.2008.5074742.

[KRM09] Christopher Kennedy and Arash Reyhani-Masoleh. “High-speed CRC computations using improved state-space transformations”. In: *2009 IEEE International Conference on Electro/Information Technology*. IEEE, June 2009, pp. 9–14. ISBN: 978-1-4244-3354-4. DOI: 10.1109/EIT.2009.5189575.

[LC83] Shu Lin and Daniel J. Costello. *Error Control Coding*. Prentice Hall, 1983. ISBN: 0-13-283796-X.

[Len+93] A. Lenstra, H. Lenstra, M. Manasse, and J. Pollard. “The number field sieve”. In: *The development of the number field sieve*. Ed. by Arjen Lenstra and Hendrik Lenstra. Vol. 1554. Lecture Notes in Mathematics. Springer

Berlin / Heidelberg, 1993, pp. 11–42. ISBN: 978-3-540-57013-4. DOI: 10.1007/BFb0091537.

[Len91] H. W. Lenstra. "Finding isomorphisms between finite fields". In: *Mathematics of Computation* 56.193 (Jan. 1991), pp. 329–347. ISSN: 0025-5718. DOI: 10.1090/S0025-5718-1991-1052099-2.

[LN86] Rudolf Lidl and Harald Niederreiter. *Introduction to finite fields and their applications*. Revised ed. New York, NY, USA: Cambridge University Press, 1986. ISBN: 0-521-46094-8.

[Lov92] Renet Lovorn. "Rigorous, subexponential algorithms for discrete logarithms over finite fields". PhD thesis. University of Georgia, June 1992.

[LP98] Renet Lovorn Bender and Carl Pomerance. "Rigorous discrete logarithm computations in finite fields via smooth polynomials". In: *Computational Perspectives on Number Theory: Proceedings of a Conference in Honor of A.O.L. Atkin*. Ed. by Duncan A. Buell and Jeremy T. Teitelbaum. Vol. 7. AMS/IP Studies in Advanced Mathematics. Providence, Rhode Island, USA: American Mathematical Society, 1998, pp. 221–232. ISBN: 0-8218-0880-X.

[Mao+01] Bu-Qing Mao, Farid Hamzei-Sichani, Dmitriy Aronov, Robert C Froemke, and Rafael Yuste. "Dynamics of Spontaneous Activity in Neocortical Slices". In: *Neuron* 32.5 (Dec. 2001), pp. 883–898. ISSN: 08966273. DOI: 10.1016/S0896-6273(01)00518-9.

[Mas69] James Massey. "Shift-register synthesis and BCH decoding". In: *IEEE Transactions on Information Theory* 15.1 (Jan. 1969), pp. 122–127. ISSN: 0018-9448. DOI: 10.1109/TIT.1969.1054260.

[McC90] Kevin S. McCurley. "The discrete logarithm problem". In: *Cryptology and Computational Number Theory (Proceedings of Symposia in Applied Mathematics)*. Ed. by Carl Pomerance. Vol. 42. Providence, Rhode Island, USA, 1990. ISBN: 0-8218-4310-9.

[Meg61] J. Meggitt. "Error correcting codes and their implementation for data transmission systems". In: *IEEE Transactions on Information Theory* 7.4 (Oct. 1961), pp. 234–244. ISSN: 0018-9448. DOI: 10.1109/TIT.1961.1057659.

[Mos96] Michele Mosca. "Discrete logarithms in finite fields". MA thesis. Wolfson College, University of Oxford, 1996.

[MVO96] Alfred J. Menezes, Scott A. Vanstone, and Paul C. Van Oorschot. *Handbook of Applied Cryptography*. 1st ed. Boca Raton, FL, USA: CRC Press, Inc., 1996. ISBN: 0-8493-8523-7.

[Nec94] V. I. Nechaev. "Complexity of a deterministic algorithm for the discrete logarithm". In: *Mathematical Notes* 55.2 (Feb. 1994), pp. 165–172. ISSN: 0001-4346. DOI: 10.1007/BF02113297.

[Ngu09] Gam D. Nguyen. "Fast CRCs". In: *IEEE Transactions on Computers* 58.10 (2009), pp. 1321–1331. ISSN: 0018-9340. DOI: 10.1109/TC.2009.83.

[Odl00] Andrew M. Odlyzko. "Discrete Logarithms: The Past and the Future". In: *Designs, Codes and Cryptography* 19.2-3 (Mar. 2000), pp. 129–145. ISSN: 0925-1022. DOI: 10.1023/A:1008350005447.

[Odl85] Andrew M. Odlyzko. "Discrete logarithms in finite fields and their cryptographic significance". In: *Advances in Cryptology: Proc. of Eurocrypt 84, Lecture Notes in Computer Science*. Ed. by Thomas Beth, Norbert Cot, and Ingemar Ingemarsson. Vol. 209. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 1985, pp. 224–314. ISBN: 978-3-540-16076-2. DOI: 10.1007/3-540-39757-4_20.

[PB61] W.W. Peterson and D.T. Brown. "Cyclic Codes for Error Detection". In: *Proceedings of the IRE*. Vol. 49. 1. Jan. 1961, pp. 228–235. DOI: 10.1109/JRPROC.1961.287814.

[Pet60] W.W. Peterson. "Encoding and error-correction procedures for the Bose-Chaudhuri codes". In: *IEEE Transactions on Information Theory* 6.4 (Sept. 1960), pp. 459–470. ISSN: 0018-9448. DOI: 10.1109/TIT.1960.1057586.

[PH78] S. Pohlig and M. Hellman. "An improved algorithm for computing logarithms over $GF(p)$ and its cryptographic significance (Corresp.)" In: *IEEE Transactions on Information Theory* 24.1 (Jan. 1978), pp. 106–110. ISSN: 0018-9448. DOI: 10.1109/TIT.1978.1055817.

[Pla+07] Luis A. Plana, Steve B. Furber, Steve Temple, Mukaram Khan, Yebin Shi, Jian Wu, and Shufan Yang. "A GALS Infrastructure for a Massively Parallel Multiprocessor". In: *IEEE Design & Test of Computers* 24.5 (Sept. 2007), pp. 454–463. ISSN: 0740-7475. DOI: 10.1109/MDT.2007.149.

[Pla+11] Luis A. Plana, David Clark, Simon Davidson, Steve Furber, Jim Garside, Eustace Painkras, Jeffrey Pepper, Steve Temple, and John Bainbridge. "SpiNNaker: Design and Implementation of a GALS Multicore System-on-Chip". In: *ACM Journal on Emerging Technologies in Computing Systems* 7.4 (Dec. 2011), pp. 1–18. ISSN: 1550-4832. DOI: 10.1145/2043643.2043647.

[Pol78] J. M. Pollard. "Monte Carlo Methods for Index Computation (mod p)". In: *Mathematics of Computation* 32.143 (July 1978), p. 918. ISSN: 0025-5718. DOI: 10.2307/2006496.

[Pom87] Carl Pomerance. "Fast, rigorous factorization and discrete logarithm algorithms". In: *Discrete Algorithms and Complexity*. Ed. by D. S. Johnson, T. Nishizeki, A. Nozaki, and H. S. Wilf. Vol. 15. Orlando, Florida: Academic Press Inc, Apr. 1987, pp. 119–143. ISBN: 0-12-386870-X.

[Pra57] Eugene Prange. *Cyclic Error-Correcting Codes in two symbols*. Tech. rep. No. AFCRC-TN-57-103, ASTIA Document No. AD133749. Bedford, Massachusetts, USA: Electronics Research Directorate, Air Force Cambridge Research Center, Air Research and Development Command, United States Air Force, Sept. 1957.

[PW72] W.W. Peterson and E.J. Weldon. *Error-Correcting Codes*. 2nd ed. The MIT Press, 1972. ISBN: 0-262-16-039-0.

[PZ92] T.-B. Pei and C. Zukowski. "High-speed parallel CRC circuits in VLSI". In: *IEEE Transactions on Communications* 40.4 (Apr. 1992), pp. 653–657. ISSN: 0090-6778. DOI: 10.1109/26.141415.

[RF89] T. R. N. Rao and Eiji Fujiwara. *Error-Control Coding for Computer Systems*. Prentice Hall, 1989, p. 524. ISBN: 0-13-284068-5.

[RG88] Tenkasi V. Ramabadran and Sunil S. Gaitonde. "A tutorial on CRC computations". In: *IEEE Micro* 8.4 (Aug. 1988), pp. 62–75. ISSN: 0272-1732. DOI: 10.1109/40.7773.

[Rie94] Hans Riesel. *Prime Numbers and Computer Methods for Factorization*. 2nd ed. Birkhäuser Boston, 1994. ISBN: 0-8176-3743-5.

[RM64] L. Rudolph and M. Mitchell. "Implementation of decoders for cyclic codes (Corresp.)" In: *IEEE Transactions on Information Theory* 10.3 (July 1964), pp. 259–260. ISSN: 0018-9448. DOI: 10.1109/TIT.1964.1053672.

[Ros93] Kenneth H Rosen. *Elementary Number Theory and Its Applications*. 3rd ed. Addison-Wesley Publishing Company, 1993. ISBN: 0-201-57889-1.

[Sch+96] Oliver Schirokauer, Damian Weber, Thomas Denny, and Henri Cohen. "Discrete logarithms: The effectiveness of the index calculus method". In: *Algorithmic Number Theory*. Ed. by Henri Cohen. Vol. 1122. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 1996, pp. 337–361. ISBN: 978-3-540-61581-1. DOI: 10.1007/3-540-61581-4_66.

[Sch02] Oliver Schirokauer. "The Special Function Field Sieve". In: *SIAM Journal on Discrete Mathematics* 16.1 (2002), p. 81. ISSN: 0895-4801. DOI: 10.1137/S0895480100372668.

[Sch05] Oliver Schirokauer. "Virtual logarithms". In: *Journal of Algorithms* 57.2 (Nov. 2005), pp. 140–147. ISSN: 0196-6774. DOI: 10.1016/j.jalgor.2004.11.004.

[Sch08] Oliver Schirokauer. "The impact of the number field sieve on the discrete logarithm problem in finite fields". In: *Algorithmic Number Theory*. Ed. by Joseph P. Buhler and Peter Stevenhagen. Vol. 44. MSRI Publications. Cambridge University Press, 2008, pp. 397–420. ISBN: 0-521-80854-5.

[Sch10] Oliver Schirokauer. "The number field sieve for integers of low weight". In: *Mathematics of Computation* 79.269 (Jan. 2010), pp. 583–583. ISSN: 0025-5718. DOI: 10.1090/S0025-5718-09-02198-X.

[Sch91] C.P. Schnorr. "Efficient signature generation by smart cards". In: *Journal of Cryptology* 4.3 (1991), pp. 161–174. ISSN: 0933-2790. DOI: 10.1007/BF00196725.

[Sch93] O. Schirokauer. "Discrete Logarithms and Local Units". In: *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences* 345.1676 (Nov. 1993), pp. 409–423. ISSN: 1364-503X. DOI: 10.1098/rsta.1993.0139.

[Sha48a] Claude E. Shannon. "A Mathematical Theory of Communication (Part I)". In: *Bell System Technical Journal* 27.3 (July 1948), pp. 379–423.

[Sha48b] Claude E. Shannon. "A Mathematical Theory of Communication (Part II)". In: *Bell System Technical Journal* 27.4 (Oct. 1948), pp. 623–656.

[Sha71] Daniel Shanks. "Class Number, a Theory of Factorization, and Genera". In: *Proceedings of Symposia in Pure Mathematics*. Ed. by Donald J. Lewis. Vol. 20. Providence, Rhode Island, USA: American Mathematical Society, 1971, pp. 415–440.

[Shi+01] M.-D. Shieh, M.-H. Sheu, C.-H. Chen, and H.-F. Lo. "A systematic approach for parallel CRC computations". In: *Journal of Information Science and Engineering* 17.3 (May 2001), pp. 445–461.

[Sho97] Victor Shoup. "Lower Bounds for Discrete Logarithms and Related Problems". In: *Advances in Cryptology EuroCrypt 97*. Ed. by Walter Fumy. Vol. 1233. Lecture Notes in Computer Science. Springer-Verlag, 1997, pp. 256–266. ISBN: 978-3-540-62975-7. DOI: 10.1007/3-540-69053-0_18.

[Spr01] M. Sprachmann. "Automatic generation of parallel CRC circuits". In: *IEEE Design & Test of Computers* 18.3 (May 2001), pp. 108–114. ISSN: 0740-7475. DOI: 10.1109/54.922807.

[SPW09] Bianca Schroeder, Eduardo Pinheiro, and Wolf-Dietrich Weber. "DRAM errors in the wild". In: *SIGMETRICS'09: Proceedings of the eleventh international joint conference on Measurement and modeling of computer systems*. New York, New York, USA: ACM Press, 2009, pp. 193–204. ISBN: 978-1-60558-511-6. DOI: 10.1145/1555349.1555372.

[Sti02] Douglas Robert Stinson. *Cryptography : theory and practice*. 2nd ed. Boca Raton, FL, USA: Chapman & Hall/CRC, 2002. ISBN: 1-58488-206-9.

[Tes00] Edlyn Teske. "On random walks for Pollard's rho method". In: *Mathematics of Computation* 70.234 (Feb. 2000), pp. 809–826. ISSN: 0025-5718. DOI: 10.1090/S0025-5718-00-01213-8.

[TFM96] Simon Thorpe, Denis Fize, and Catherine Marlot. "Speed of processing in the human visual system." In: *Nature* 381.6582 (June 1996), pp. 520–522. ISSN: 0028-0836. DOI: 10.1038/381520a0.

[Toa+09] Ciaran Toal, Kieran McLaughlin, Sakir Sezer, and Xin Yang. "Design and Implementation of a Field Programmable CRC Circuit Architecture". In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 17.8 (Aug. 2009), pp. 1142–1147. ISSN: 1063-8210. DOI: 10.1109/TVLSI.2008.2008741.

[TW11] Andrew Tanenbaum and David J. Wetherall. *Computer Networks*. 5th ed. Boston, Massachusetts: Pearson Education, 2011. ISBN: 0-13-255317-1.

[Zie+96] J. F. Ziegler et al. "IBM experiments in soft fails in computer electronics (1978-1994)". In: *IBM Journal of Research and Development* 40.1 (Jan. 1996), pp. 3–18. ISSN: 0018-8646. DOI: 10.1147/rd.401.0003.

[Zie74] Neal Zierler. "A conversion algorithm for logarithms on $GF(2^n)$". In: *Journal of Pure and Applied Algebra* 4.3 (1974), pp. 353–356. ISSN: 0022-4049. DOI: 10.1016/0022-4049(74)90016-4.

[Zie96] J. F. Ziegler. "Terrestrial cosmic rays". In: *IBM Journal of Research and Development* 40.1 (Jan. 1996), pp. 19–39. ISSN: 0018-8646. DOI: 10.1147/rd.401.0019.

[ZL79] J. F. Ziegler and W. A. Lanford. "Effect of cosmic rays on computer memories." In: *Science (New York, N.Y.)* 206.4420 (Nov. 1979), pp. 776–788. ISSN: 0036-8075. DOI: 10.1126/science.206.4420.776.