FPGA implementation of product accumulate codes

Koh, Tiong Aik

2009

Koh, T. A. (2009). FPGA implementation of product accumulate codes. Master's thesis, Nanyang Technological University, Singapore.

<https://hdl.handle.net/10356/47030>

<https://doi.org/10.32657/10356/47030>

Nanyang Technological University

Downloaded on 26 Apr 2023 19:30:09 SGT

FPGA IMPLEMENTATION OF PRODUCT ACCUMULATE CODES

KOH TIONG AIK

School of Electrical and Electronic Engineering

A thesis submitted to the Nanyang Technological University
in partial fulfilment of the requirement for the degree of
Master of Engineering

2009

Acknowledgement

I would like to express my heart-felt gratitude and thanks to my supervisor, Associate Professor Guan Yong Liang for his invaluable guidance and all the insightful discussions during my M. Eng. study.

Many thanks to Dr Ng Boon Chong (HACL) for his unremitting guidance, encouragement and assistance. Professor Tiffany Jing Li's help is very much appreciated, for her advice on possible ways of extending PA code.

Special thanks has to be given to my family and friends for their remarkable patience and constant support. It will not be possible for me to complete my study without them.

The application of PA code in an actual communications waveform could not have been possible without the great team we have of Adele Chong Peiyi and Zhong Kun, to discuss and design the best suited code structure.

Finally, I want to acknowledge Defence Science and Technology Agency - Directorate for Research and Development, DSO National Laboratories and Nanyang Technological University for this research opportunity, under contract DSOCL05258.

Contents

Acknowledgement	i
Contents	ii
Abstract	vi
List of Abbreviations	viii
List of Symbols	x
List of Tables	xii
List of Figures	xiii
1 Introduction	1
1.1 Motivation	1
1.2 Objectives	2
1.3 Major Contributions of the Thesis	3
1.4 Organization of the Thesis	4

Contents	iii
2 Literature Review and Background	5
2.1 Channel Coding in Digital Communication Systems	5
2.1.1 Error Correction Codes	7
2.2 Product Accumulate Code	12
2.3 PA Encoding	14
2.4 Message Passing Algorithm	15
2.5 PA Decoding	18
2.5.1 Inner Decoder	18
2.5.2 Outer Decoder	20
2.6 Choice of Decoding algorithm	24
3 Interleaver Design	27
3.1 Interleavers in PA Code	27
3.2 Coprime Interleaver and Relatively Prime Interleaver	28
3.3 Prime Factor Based Multi-Dimensional Helical Block Interleavers	30
3.4 Quadratic Permutation Polynomial Based Interleavers	34
3.5 Choice of Interleaver	39
4 EXIT Analysis	40
4.1 Extrinsic Information Transfer Chart	40
4.1.1 Finite Length EXIT Trajectories Analysis	43
4.2 EXIT Trajectories Analysis of Four Algebraic Interleavers	45
4.3 Divergence of EXIT Trajectory	52

Contents	iv
5 Encoder and Decoder Implementation	56
5.1 Encoder Design	56
5.1.1 Encoder Implementation	57
5.2 Cut-down Version of Encoder Implementation	62
5.3 Cut-down Encoder Interface and Timing Diagram	64
5.3.1 Cut-down Encoder Interface	65
5.3.2 Cut-down Encoder Timing Diagram	68
5.4 Decoder Design	71
5.4.1 2-Stage Pipelining	73
5.4.2 3-Stage Pipelining	78
5.5 Decoder Implementation	82
5.5.1 Inner Decoder Implementation	84
5.5.2 Outer Decoder Implementation	85
5.6 Encoder and Decoder Resource utilization	88
5.7 Decoder Interface and Timing Diagram	88
5.7.1 Decoder Interface	88
5.7.2 Decoder Timing Diagram	93
6 FPGA Test Results	96
6.1 Test Setup	96
6.1.1 Decoder Core	96
6.1.2 Decoder	97
6.1.3 BER Test Setup	98
6.2 BER Results	102
6.3 Benchmarking of Decoder Resource Utilization	104

Contents	v
7 Conclusion and Recommendations	106
7.1 Conclusion	106
7.2 Recommendations	107
Author's Publications	109
Bibliography	110

Abstract

Product accumulate (PA) codes are a class of low density parity check (LDPC)-like codes with a well defined structure, which allows flexible rate and length adaptation while maintaining good bit error rate (BER) performance. Compared to turbo codes, they provide similar performance but with significantly less decoding complexity and with a lower error floor. This thesis presents our study on the various design tradeoffs needed for the field programmable gate array (FPGA) implementation of PA codes and on the actual FPGA implementation of the PA type I encoder and decoder (codec).

The design goals target full reconfigurability on-the-fly, low resource utilization and high throughput while maintaining error correcting performance within acceptable limits. Judicious selection of the decoding algorithm and pipelining are crucial for high throughput. The final design adopts a 3-stage pipeline structure on the min-sum algorithm to enable simple implementation and low latency, to maximize instantiated resource utilization, and to increase throughput while keeping a minimal degradation of BER performance ($< 0.5dB$)

A second major challenge is the efficient design and implementation of interleavers and deinterleavers in the PA codec. We evaluate several classes of algebraic interleavers that allow easy reconfiguration and have low on-chip storage requirements.

We show that coprime interleavers that are good for software simulation may not be suitable for hardware realization. Instead, we propose to use quadratic permutation polynomial (QPP) based interleavers to strike a good balance of good BER performance, reconfigurability and ease of implementation in FPGA.

Extrinsic information transfer (EXIT) analysis is used to verify the BER performance differences of the different classes of interleavers. Divergence of the EXIT trajectory is shown as a main cause of the poor BER performance exhibited by the relatively prime and prime factor helical interleavers. The EXIT analysis verified that the QPP interleaver within a PA code does provide a good BER performance.

The codec implemented on a Xilinx Virtex II FPGA device (2v3000fg676-4) is capable of running at $125MHz$ with encoder throughput at $62.5Mbps$ and decoder throughput of approximately $4Mbps$ at 30 decoding iterations. It takes up 1737 slices (12.11%) and 18 block ram (18.75%) of the target device. When compared to the 'ideal' decoder, the practical decoder suffers a performance loss of only $0.4dB$ at BER 1×10^{-5} . To the best of our knowledge, this is the first reported implementation of PA codes in FPGA.

List of Abbreviations

APP	A-Posterior Probability
AWGN	Additive White Gaussian Noise
bps	Bits Per Second
BCH	Bose-Chaudhuri-Hocquenghem (code)
BER	Bit Error Rate
BPSK	Binary Phase Shift Keying
BRAM	Block Random Access Memory
codec	Encoder and Decoder
dB	Decibel
DPBRAM	Dual Port Block RAM
EXIT	Extrinsic Information Transfer
FEC	Forward Error Correction (code)
FER	Frame Error Rate
FIFO	First-In-First-Out (buffer)
FPGA	Field Programmable Gate Array
LDPC	Low Density Parity Check (code)
LFSR	Linear Feedback Shift Register
LLR	Log-Likelihood Ratio

List of Abbreviations

MCM	Multilevel Coded Modulation
MI	Mutual Information
MPA	Message Passing Algorithm
MS	Min-Sum (algorithm)
MSB	Most Significant Bit
PA	Product Accumulate (code)
PRBS	Pseudo Random Bit Sequence
QPP	Quadratic Permutation Polynomial (interleaver)
RA	Repeat Accumulate (code)
RM	Reed-Muller (code)
RS	Reed-Solomon (code)
SDR	Software Defined Radio
SNR	Signal to Noise Ratio
SP	Sum-Product (algorithm)
SPC	Single Parity Check (code)
TCM	Trellis Coded Modulation
TPC	Turbo Product Code
VHDL	VHSIC Hardware Description Language
VHSIC	Very High Speed Integrated Circuit

List of Symbols

E_b/N_o	Bit energy to noise density ratio
N	Block length of coded bits
K	Block length of uncoded bits
d	Hamming distance
r	Code Rate
Π_1	Interleaver 1 in outer code
Π_2	Interleaver 2 between outer and inner code
P	Number of SPC codewords in 1 SPC branch of PA code
T	Number of data bits in an SPC codeword
N_1	Block size of $\Pi_1 = P \times T$
N_2	Block size of $\Pi_2 = P \times (T + 2)$, equal to PA coded length N
$d_{i,j}$	j^{th} information data bit of i^{th} SPC codeword
p_i	Parity bit of i^{th} SPC codeword
(k)	k^{th} global turbo iteration
(l)	l^{th} local turbo iteration in outer decoder
$\boxplus$	Check operator of message passing algorithm
$L_{ch}(y_i)$	Channel LLR from i^{th} PA coded bit

List of Symbols

$L_{ef}(\cdot)$	LLR from forward pass in inner decoder
$L_{eb}(\cdot)$	LLR from backward pass in inner decoder
$L_e(\cdot)$	LLR from inner decoder to outer decoder
$L_{e1}(\cdot)$	LLR from upper SPC branch (non Π_1 interleaved)
$L_{e2}(\cdot)$	LLR from lower SPC branch (Π_1 interleaved)
$L_o(\cdot)$	LLR from outer decoder

List of Tables

3.1	QPP interleavers resource utilization	38
5.1	Resource utilization of full encoder against cut-down encoder	65
5.2	Interface of the cut-down encoder	67
5.3	Data stream for inner decoder at the k^{th} global iteration	86
5.4	PA CODEC Resource Utilization	88
5.5	Interface of PA decoder	89
6.1	PA CODEC vs LDPC-CC in Xilinx FPGA(2v3000fg676-4)	105

List of Figures

2.1	A typical digital communications system	6
2.2	BER performance of block codes	9
2.3	Code structure of PA-I code	13
2.4	Structure of a block of SPC codes	13
2.5	Outer encoding of a block of information bits	14
2.6	Inner encoding of a block of bits	15
2.7	Code graph of PA-I code	16
2.8	Message passing algorithm	16
2.9	Simplification of check nodes	17
2.10	Forward and backward message pass in inner decoder	20
2.11	Message passed to outer decoder	21
2.12	Message passed from inner decoder	22
2.13	Messages processed at check node	22
2.14	Message passed to inner decoder	23
2.15	Comparison of message passing algorithm	25
3.1	Different relatively prime interleaver vs coprime interleaver	29
3.2	Mix of relatively prime and coprime interleaver	30
3.3	Comparison of coprime interleavers and prime factor helical interleavers	32
3.4	Π_2 using coprime interleaver with Π_1 using different types of interleavers	33
3.5	Comparison of coprime interleaver and QPP interleaver	36
3.6	Comparison of QPP interleaver vs block interleaver for Π_1	37
3.7	Comparison of 4 algebraic interleavers	39

List of Figures	xiv
4.1 EXIT chart for a PA-I code (2048,1024, $E_b/N_o = 4dB$)	42
4.2 EXIT trajectory of a codeword using coprime interleaver (31, 11, 61, 13).	44
4.3 Divergence of EXIT trajectory of 1 code word using relatively prime interleaver	45
4.4 Scatter plot of EXIT trajectories's vertices using different interleavers. ($E_b/N_o = 5dB$, 1000 <i>codewords</i>)	47
4.5 Scatter plot of EXIT trajectories's vertices using different interleavers. ($E_b/N_o = 5dB$, 100 <i>codewords</i>)	48
4.6 Scatter plot of EXIT trajectories' vertices using different interleavers. ($E_b/N_o = 4dB$, 100 <i>codewords</i>)	49
4.7 Scatter plot of EXIT trajectories' vertices using different interleavers. ($E_b/N_o = 3dB$, 100 <i>codewords</i>)	50
4.8 Scatter plot of EXIT trajectories' vertices using different interleavers. ($E_b/N_o = 2dB$, 100 <i>codewords</i>)	51
4.9 Frame and Bit Error Rate using the 4 different interleavers	52
4.10 Divergence of EXIT trajectory of 1 code word using relatively prime interleaver, even with 400 iterations	54
4.11 Coprime vs relatively prime interleaver using 30 or 100 iterations . . .	55
5.1 Encoder implementation with 3 stage pipelining	59
5.2 Cut-down version of encoder	63
5.3 Interface diagram for encoder	66
5.4 Interface timing diagram for encoder	70
5.5 Schematic for a 2-stage pipeline	74
5.6 BER comparison of different pipelining schemes	77
5.7 Schematic for a 3-stage pipeline	78
5.8 BER improvement of 3-stage pipeline with 30 and 60 iterations . . .	81
5.9 Detailed decoder structure.	83
5.10 Interface diagram for decoder	92
5.11 Interface timing diagram for decoder input	94
5.12 Interface timing diagram for decoder output	95
6.1 Block diagram of the decoder core	97
6.2 Block diagram of the decoder	98
6.3 Block diagram of the BER test setup	101
6.4 BER comparison of hardware-in-the-loop vs simulation	103

Chapter 1

Introduction

1.1 Motivation

Product accumulate (PA) codes (Li et al. 2004) are a class of provably good codes which are linear time encodable and linear time decodable and have flexible code rates above $\frac{1}{2}$. For short code lengths (100 to 1000 bits), PA codes are claimed to have slightly better performance and lower complexity compared to Turbo and LDPC codes, and its code construction easier than other close-to-capacity codes such as concatenated tree (CT) codes. PA codes also have a regular code structure and the code word length can be easily scaled from hundreds of bits to hundreds of thousands of bits. Coupled with flexible code rate and the use of algebraic interleavers, PA codes are especially suited for software defined radios (SDR) platforms for applications in wireless and space communications, where reconfigurability of the waveform necessitates reconfigurability of the channel code.

The excellent performance of PA codes were demonstrated by computer simulations in (Li et al. 2004), but efficient hardware implementation of PA codec has not been reported. In this thesis, we will design and implement in FPGA a PA encoder and decoder (codec) to support throughput in the mega bits per second (*Mbps*) range, supporting full reconfigurability with minimal bit error rate (BER) performance degradation while keeping resource utilization low. Optimizations based

on hardware implementation considerations will be done and a hardware-in-the-loop BER test will also be conducted to verify the design.

1.2 Objectives

Following the motivations mentioned in chapter 1.1, the main objectives of this thesis are:

1. To develop an FPGA implementation of PA codes with investigation of design trade-offs for implementation considerations, including pipeline structure and interleaver choice. Specifically, the implementation will target the Xilinx Virtex II family of FPGA. Hardware-in-the-loop BER test will be conducted on a Nallatech XtremeDSP development kit with a Xilinx Virtex II 3Mgates FPGA device (2v3000fg676-4).
2. To target full reconfigurability on-the-fly, low resource utilization, low latency and high throughput in the FPGA codec design, while maintaining error correcting performance within acceptable limits. Specifically, the target is 2Mbps decoding throughput per decoder core with BER degradation of less than $0.5dB$ from the ideal decoder. Resource utilization will be measured against a comparable decoder published in the open literature, based on similar throughput and BER performance.
3. To consider other reconfigurable interleavers in addition to the coprime interleaver proposed in the original PA code papers; and to find suitable analysis framework to facilitate the final design choice among these interleavers.

The target of 2Mbps decoding throughput is estimated from a serial decoder which can complete 1 decoding iteration per clock cycle. Running at a system clock frequency of 60MHz (which is commonly achievable in the Xilinx Virtex II family of FPGA) with 30 iterations per codeword, a decoding throughput of 2Mbps is thus achievable.

1.3 Major Contributions of the Thesis

The major contributions of the thesis include:

1. For the encoder, a 4-stage pipeline structure was designed for high throughput and low resource utilization. Cut-down versions of the encoder and decoder, which removed some optional modules and interfaces to achieve even lower resource utilization, are also discussed in the thesis. For the decoder, a 3-stage pipeline structure was designed based on the min-sum (MS) algorithm to enable efficient decoding and low latency, to minimize resource utilization, and to increase throughput while keeping degradation of BER performance to below $0.5dB$.
2. A memory-based, serial architecture specifically designed for FPGA with on-the-fly full reconfigurability of the code was developed. As FPGA contains logic slices complemented by large amounts of on-chip memory, the proposed architecture takes advantage of the dual port nature of the on-chip memory, to balance the utilization between logic slices and memory blocks. The experience gained here will be useful for the implementation of other serially or parallelly concatenated codes, including turbo codes, repeat product codes, and convolutional product codes, too.
3. In addition to the original coprime interleaver (Li et al. 2004), the relatively prime interleaver (Xie et al. 2006), quadratic permutation polynomial (QPP) based interleaver (Sun and Takeshita 2005) and prime factor based multi-dimensional helical block interleavers were also investigated. The QPP based interleaver is shown to be the best choice in terms of implementation cost, reconfigurability and bit error rate performance.
4. Extrinsic information transfer (EXIT) analysis was used for the first time to prove that the divergence of the EXIT trajectory is a main cause of the poor

BER performance exhibited by the non-performing interleavers such as the relatively prime interleaver and prime factor helical interleaver. The EXIT analysis also confirms the convergence and superior BER performance of the QPP interleaver.

1.4 Organization of the Thesis

This thesis is organized in the following way:

Chapter 2 gives a literature review on error correction coding and an overview of the PA code structure and the message passing algorithm

Chapter 3 investigates the different types of interleavers and their BER performance

Chapter 4 uses EXIT analysis to verify the BER performance of the different interleavers and show the main cause of the performance differences of the different interleavers

Chapter 5 details the implementation of the encoder and decoder in FPGA, with detailed discussions on the pipeline structures

Chapter 6 verifies the performance of the implemented codec and compares it against another implemented codec reported in literature.

Chapter 7 concludes the thesis and provides suggestions for future work.

Chapter 2

Literature Review and Background

2.1 Channel Coding in Digital Communication Systems

Fig. 2.1 shows the block diagram of a typical digital communication system. The information source and sink includes any source coding scheme matched to the nature of the information. The encoder takes as input the information symbols from the source and adds redundant symbols to it. These coded symbols will then be modulated to a waveform, using a scheme suitable for the channel. After passing through the noisy channel, the received waveform will be demodulated and hard or soft estimates of the coded symbols will be generated. These will then be fed to the decoder, which will use these estimates to try to recover the information symbols. This is known as forward error correction (FEC) coding when the decoder can only work on the estimates it received and has no means of communicating to the encoder when it needs additional aid in decoding a particular codeword. This is the prevalent form of error correction coding used in most systems today. Note that samples of an additive noise process are added to the modulated symbols in the channel and these noise samples are assumed to be independent from the source symbols.

The field of channel coding started with Claude Shannon's 1948 landmark paper (Shannon 1948). For the next half century, its central objective was to find practi-

Figure 2.1: A typical digital communications system.

cal coding schemes that could approach channel capacity (hereafter called the ‘Shannon limit’) on well-understood channels such as the additive white Gaussian noise (AWGN) channel. This goal proved to be challenging, but not impossible.

In 1974, Massey suggested considering error correcting codes and modulation as a single entity, known as coded modulation. This approach provides a higher spectral efficiency and coding gain than the serial concatenation of error correcting codes and modulation, by the joint design of codes and signal constellations. Some methods of combining coding and modulation includes trellis-coded modulation (TCM) (Ungerboeck 1982) and multilevel coded modulation (MCM) (Imai and Hirakawa 1977). Ungerboeck realized that in the bandwidth-limited regime, the redundancy needed for coding should be obtained by expanding the signal constellation while keeping the bandwidth fixed, rather than by increasing the bandwidth while keeping a fixed signal constellation, as is done in the power-limited regime. From capacity calculations, he showed that doubling the signal constellation should suffice to cater for additional redundancy for error correction, e.g., using a 32-QAM rather than a 16-QAM constellation. Ungerboeck invented clever trellis codes for such expanded constellations, using minimum Euclidean distance rather than Hamming distance as the design criterion.

As with convolutional codes, trellis codes may be optimally decoded by a Viterbi decoder, whose decoding complexity is proportional to the number of states in the encoder. In a coded modulation system, the soft decision channel outputs are processed

directly by the decoder. In contrast, in a classical FEC scheme, the hard decision bits from the demodulator are fed to a binary decoder.

In the 1990's, practical error correcting coding schemes that yield capacity approaching performance were invented. Although random coding theory implies that arbitrary codes tend to perform close to the capacity as the code length goes to infinity, it was not until the discovery of turbo codes (Berrou et al. 1993) and the rediscovery of low density parity check (LDPC) codes (Gallager 1963, MacKay 1999) in recent years that performance very close (within $1dB$) to the capacity limit has been achieved. However, for practical use, much work is still required for efficient implementation in hardware.

Current research in this field is primarily concerned with the construction, analysis and evaluation of high-performance, easily-implementable error correction codes, particularly low-complexity codes based on graphs.

2.1.1 Error Correction Codes

All error correcting codes are based on the same basic principle: redundancy is added to information in order to correct any errors that may occur in the process of storage or transmission. When the error correction code appends the redundant symbols to the information symbols to create the coded symbols, it is called a systematic code. Non-systematic codes do not have such easy separation of the redundant symbols from the information symbols. A short description of the different classes of error correction codes is given below.

Algebraic Coding

Traditional block codes use algebraic decoding. Algebraic coding theory is primarily concerned with linear (N, K, d) block codes over the binary field $\mathbb{F}_2$ (Costello and Forney 2007). A binary linear (N, K, d) block code consists of 2^K binary N -tuples, called codewords, which have the group property: i.e., the componentwise mod-2 sum

of any two codewords is another codeword. The parameter d denotes the minimum Hamming distance between any two distinct codewords, i.e., the minimum number of coordinates in which any two codewords differ. The theory generalizes to linear (N, K, d) block codes over nonbinary fields $\mathbb{F}_q$.

The principal objective of algebraic coding theory is to maximize the minimum distance d for a given (N, K) . The motivation for this objective is to maximize error correction capability. Over a binary symmetric channel (BSC: a binary-input, binary-output channel with statistically independent binary errors), the optimum decoding rule is to decode to the codeword closest in Hamming distance to the received n -tuple. With this rule, a code with minimum distance d can correct all patterns of $\lfloor (d-1)/2 \rfloor$ or fewer channel errors, where $\lfloor x \rfloor$ denotes the largest integer smaller or equal to x , but cannot correct some patterns containing a greater number of errors.

The information is processed on a block-by-block basis, treating each block of information bits as independent from others. In other words, block coding is a memory-less operation, in the sense that codewords are independent from others. Examples are Hamming codes (Hamming 1950), Bose-Chaudhuri-Hocquenghem (BCH) codes (Bose and Ray-Chaudhuri 1960, Hocquenghem 1959), Golay codes (Golay 1949), and Reed-Muller (RM) code (Muller 1954, Reed 1954).

Fig. 2.2 shows the bit error rate (BER) performance, which is also known as the probability of bit error $P_b(E)$, of several block codes using algebraic decoders. It can be seen that their BER performances are far from the Shannon limit.

In the 1960s, research in channel coding was dominated by the development of algebraic block codes, particularly cyclic codes. The algebraic coding paradigm used the structure of finite-field algebra to design efficient encoding and error-correction procedures for linear block codes operating on a hard-decision channel. The emphasis was on constructing codes with a guaranteed minimum distance d and then using the algebraic structure of the codes to design bounded-distance error-correction algorithms whose complexity grows only as a small power of d .

Figure 2.2: BER versus E_b/N_o for (31, 26, 3) Hamming code, (23, 12, 7) Golay code, and (31, 16, 7) shortened RM code with optimum hard-decision decoding, compared to uncoded binary PAM (Costello and Forney 2007).

In particular, the goal was to develop flexible classes of easily implementable codes with better performance than RM codes, which were amongst the best codes then. Cyclic codes are codes that are invariant under cyclic ('end-around') shifts of n -tuple codewords. Cyclic codes have a nice algebraic theory and attractively simple encoding and decoding procedures based on cyclic shift register implementations. Reed-Solomon (RS) codes (Reed and Solomon 1960) are an important family of codes in this category. Note that Hamming, Golay, and shortened RM codes can also be put into cyclic form.

The ability of RS codes to correct both random and burst errors makes them particularly well suited for applications such as magnetic tape and disk storage, where imperfections in the storage media sometimes cause bursty errors. They are also useful as outer codes in concatenated coding schemes. For these reasons, RS codes are probably the most widely deployed codes in practice.

These block codes can be implemented efficiently in hardware, but most of them lack a convenient soft decoding algorithm. Further, it is not always possible to change

code lengths and/or code rates easily.

Probabilistic Coding

‘Probabilistic coding’ is a name for an alternative line of development that was more directly inspired by Shannon’s probabilistic approach to coding (i.e. random coding). Whereas algebraic coding theory aims to find specific codes that maximize the minimum distance d for a given (n, k) code, probabilistic coding is more concerned with finding classes of codes that optimize average performance as a function of coding and decoding complexity. Probabilistic decoders typically use soft-decision (reliability) information, both as inputs and at intermediate stages of the decoding process. Classical coding schemes that fall into this class include convolutional codes, product codes, concatenated codes, trellis-coded modulation, and trellis decoding of block codes.

The output of a convolutional encoder depends not only on the current input information, but also on previous inputs or outputs, either on a block-by-block or a bit-by-bit basis. Convolutional codes use trellis decoding or list decoding. Unlike block codes, it is convenient to change code lengths of convolutional codes, and they have efficient soft decoding algorithms, like soft output Viterbi algorithm (SOVA) (Hagenauer and Hoeher 1989) and a posterior probability (APP) algorithm (Bahl et al. 1974) (whose hard output algorithm is known as maximum a posterior or MAP). The technique of puncturing further allows convolutional codes to change code rates flexibly without incurring additional complexity.

For many years, the competition between the algebraic and probabilistic approaches was cast as a competition between block codes and convolutional codes. Convolutional coding was motivated from the start by the objective of optimizing the tradeoff of performance versus complexity, which on the binary-input AWGN channel necessarily implies soft decisions and quasi-optimal decoding. In practice, most channel coding systems have used convolutional codes for random errors. Modern capacity-approaching codes are the ultimate fruit of this line of development.

Turbo Codes (Iterative Decoding)

These are recently developed compound and concatenated codes which are built upon blocks and/or convolutional codes and which uses soft iterative decoding. It was first found by Forney in 1966 (Forney 1966) that by concatenating an inner code and an outer code, it is possible to construct codes whose probability of error decreases exponentially at rates less than capacity, while complexity increases only algebraically. The discovery of turbo codes by Berrou et al. (Berrou et al. 1993) and the re-discovery of low density parity check codes (Gallager 1963) by MacKay, Luby and Richardson et al. (Luby et al. 1998, MacKay 1999, Richardson and Urbanke 2001) are the milestones in the advancement of this field.

In truth, certain elements of the turbo and turbo-like codes had been appreciated for a long time. It had been known since the early works of Elias that linear codes are as good as general codes (eg. nonlinear). Information theorists also understood that maximizing the minimum distance was not the key to getting to capacity; rather, codes should be ‘random-like,’ in the sense that the distribution of distances from a typical codeword to all other codewords should resemble the distance distribution in a random code. Another element of the turbo and turbo-like codes is the use of soft decisions (reliability information) not only as input to a decoder, but also in the internal workings of an iterative decoder. In 1962 Gallager had already developed the modern APP decoder for decoding LDPC codes and had shown that retaining soft-decision (APP) information in iterative decoding was useful even on a hard-decision channel such as a BSC.

Prompted by the impressive near-capacity performance of turbo-like codes (Berrou and Glavieux 1996, Richardson et al. 2001), parallel concatenated convolutional codes (PCCC) (Benedetto and Montorsi 1996), serial concatenated convolutional codes (SCCC) (Benedetto et al. 1996), hybrid concatenated convolutional codes (HCCC) (Divsalar and Pollara 1997) and other concatenated codes were proposed and shown to provide similar coding gain. They were also shown to have lower error floors than

the original turbo codes. Other examples include turbo product codes (TPC) (Caire et al. 1994, Elias 1954, Hagenauer et al. 1996) and regular/irregular repeat accumulate (RA/IRA) codes (Divsalar et al. 1998, Jin et al. 2000). Product accumulate (PA) codes (Li et al. 2001, 2004), which is the focus and the primary motivation in this work also belongs to this category. These codes all have two major features in common:

1. A (random) interleaver is usually incorporated (explicitly or implicitly) in the code structure
2. Soft-in soft-out (SISO) iterative decoding techniques are exploited to approximate the performance of an optimal decoder.

2.2 Product Accumulate Code

A PA code is a serial concatenation of an outer product code, a random interleaver (Π_2), and a inner rate-1 recursive convolutional code of the form $1/(1+D)$ (also known as the accumulator) (Li et al. 2004), where D is the delay operator. Fig. 2.3 shows the exact structure of the PA type I (PA-I) code, whose outer code takes the form of two parallel branches of single parity check (SPC) codes concatenated via a random interleaver (Π_1). The PA code discussed henceforth will refer to this particular code structure. The input data are taken in blocks of $P \times T$ bits arranged in P rows and T columns, and the output produces codewords of $P \times (T + 2)$ bits each. The parameters of the PA codes are $\{N, K, r\} = \{P \times (T + 2), P \times T, T/(T + 2)\}$, where N is the coded bit block length, K is the information data bit block length and r is the code rate derived from K/N . Fig. 2.4 shows the structure of the $T/(T + 1)$ SPC code that is used in the PA outer code.

The idea of concatenating an outer code and an interleaver with a rate-1 recursive inner code, particularly of the form of $1/(1+D)$, to achieve coding gains (interleaving gain) without reducing the overall code rate is widely recognised (Divsalar and Pollara

1997), (Narayanan and Stuber 1999), (Peleg et al. 1999). For low-rate codes (rate-1/2 or less), convolutional codes and even very simple repetition codes (Jin et al. 2000) are good outer code candidates to provide satisfactory performance. However, the construction of very-high-rate codes based on this concept poses a problem. From Divsalar et al.'s results (Benedetto et al. 1996), (Divsalar et al. 1998), the outer code needs to have a minimum distance of at least three to obtain an interleaving gain. To obtain good high-rate convolutional codes through puncturing, and in particular to maintain a d_{min} of three after puncturing, the original convolutional codes must have fairly long constraint length, which makes decoding computationally complex. On the other hand, 2-D SPC codes possess many nice properties for a concatenated high-rate coding structure, such as high rate, simplicity, and the availability of an efficient soft-decoding algorithm.

Figure 2.3: Code structure of PA-I code.

Figure 2.4: A block of P rows of $T/(T + 1)$ SPC codes.

2.3 PA Encoding

PA codes, as mentioned in chapter 1.1, are linear time encodable. The main operator required is a modulo-2 adder, coupled with the two interleavers in the code structure. As shown in Fig. 2.5, the input $P \times T$ information data bits are packed into a P by T matrix. In the first branch, the first bit of the P parity bits is computed from the sum (modulo-2) of the bits in the first row of the matrix, and so on. The P by T matrix is also interleaved through Π_1 and passed through an SPC encoder block in the second branch, where another set of P parity bits are computed in the same way as the first branch.

The information data and parity bits are then packed in a systematic manner, ie. the two sets of parity bits are appended to the information data bits. Subsequently these output bits from the outer encoder are interleaved through Π_2 and passed into the accumulator. The i^{th} PA coded bit is the sum (modulo 2) of the first i bits of the input to the accumulator. For example as shown in Fig. 2.6, the 6th coded bit is the sum (modulo-2) of the first six bits of the input. Note that due to the inner encoder, the PA output coded bits are not systematic. They are effectively being differentially encoded by the inner encoder.

Figure 2.5: Outer encoding of a block of information bits using SPC codes.

Figure 2.6: Inner encoding of a block bits using a modulo 2 accumulator.

2.4 Message Passing Algorithm

PA codes can be interpreted as a special class of LDPC codes, albeit as two layers of bipartite graph of bit and check nodes connected through an interleaver (Π_2). This is seen in Fig. 2.7, where C_1 denotes check nodes for systematic bits, C_2 are check nodes for bits after passing through Π_1 , x are coded bits after the outer code and Π_2 , and y are the PA coded bits. The lines connecting the nodes are also known as the edges of the node and are bi-directional in that, messages/information can flow in both directions.

Using this graph interpretation, PA codes can be efficiently decoded using the message-passing algorithm (MPA) (Schlegel and Perez 2004) on both the outer and the inner code. Fig. 2.8 shows a simple illustration of the message passing algorithm where $\oplus$ denotes bit nodes and $\boxplus$ denotes check nodes, α and β are the messages passed from bit to check nodes and γ is the message passed from check to bit nodes. In a bipartite graph, messages always flow from bit nodes to check nodes and from check nodes to bits nodes. There are no message flow between any bit nodes and this applies similarly to check nodes.

The sum-product (SP) implementation maximally extracts the coding gain promised

Figure 2.7: Bipartite code graph of PA-I code.

Figure 2.8: Illustration of message passing algorithm where LLR (α, β) from 2 bit nodes goes into a check node which then computes the extrinsic LLR (γ) to another bit node.

by the MPA (Kschischang et al. 2001), whereas the min-sum (MS) implementation sacrifices the performance in exchange for a considerably lower complexity. From (Schlegel and Perez 2004), the SP algorithm on the check node uses the $\boxplus$ operation which is also known as the ‘tanh’ rule as seen in eqn. 2.1. This can be simplified to form the MS algorithm, which approximates the ‘tanh’ rule with a simple *signed min* operation, by ignoring the exponential and logarithmic operations (Li et al. 2004). For both algorithms each bit node, using the $\oplus$ operation, consists of a summation operation on the input messages to generate the output message.

$$\begin{aligned}
\gamma = \alpha \boxplus \beta &= 2 \tanh^{-1} \left(\tanh \frac{\alpha}{2} \cdot \tanh \frac{\beta}{2} \right) \\
&= \log \frac{1 + e^{\alpha + \beta}}{e^{\alpha} + e^{\beta}} \\
&= \text{sign}(\alpha) \cdot \text{sign}(\beta) \cdot \min(|\alpha|, |\beta|) + \log \frac{1 + e^{-|\alpha + \beta|}}{1 + e^{-|\alpha - \beta|}} \\
&\approx \text{sign}(\alpha) \cdot \text{sign}(\beta) \cdot \min(|\alpha|, |\beta|)
\end{aligned} \tag{2.1}$$

Note that the check function can be easily extended to cater for multiple bit nodes passing messages into a check node, as shown in Fig. 2.9. Thus any check node functions can always be simplified into a series of 2-input, 1-output check node operations.

Figure 2.9: Simplification of a check node with multiple bit nodes (passing messages in), to a series of 2-input, 1-output check nodes.

Consider T bits forming a parity check: $\sum_{j=1}^T \oplus x_j = 0$. The SP and MS algorithm for a multi-bit check node is as follows:

$$\begin{aligned}
L_e(x_i) &= 2 \tanh^{-1} \left(\prod_{j=1, j \neq i}^T \tanh \left(\frac{L_a(x_j)}{2} \right) \right) \\
&\approx \prod_{j=1, j \neq i}^T \text{sign}(L_a(x_j)) \min_{\substack{j=1, \dots, T \\ j \neq i}} \{|L_a(x_j)|\},
\end{aligned} \tag{2.2}$$

where L_e and L_a denote the extrinsic and the *a priori* log-likelihood ratios (LLR) respectively.

2.5 PA Decoding

The turbo principle can be used to iteratively decode a serial (or parallel) concatenated system, where soft extrinsic information in log-likelihood-ratio (LLR) form is exchanged between the inner and outer code. The extrinsic information from one sub-decoder can then be used as *a priori* information by the other sub-decoder.

In the case of PA code, the decoding of the outer SPC code is done using a message-passing algorithm similar to that of LDPC codes. The outer decoder uses $L_e^{(k)}(x_i)$, from the inner decoder, as *a priori* information and produces extrinsic information $L_o^{(k)}(x_i)$. The inner rate-1 convolutional code can be decoded using a two-state BCJR algorithm, which generates the extrinsic information for bit x_i in the k^{th} turbo iteration, denoted by $L_e^{(k)}(x_i)$. As shown in Fig. 2.7, the combination of the outer code, the interleaver, and inner code can be represented using one 2-layered bipartite graph which contains bit nodes (representing the actual bits) and check nodes (representing a constraint such that connecting bit nodes should add up (modulo-2) to zero). Thus, a more computationally efficient approach is to use the message-passing algorithm as described in chapter 2.4, directly on the graph of the PA code including the inner code, whose subgraph has no cycles.

Additionally, two different iteration loops are possible in the PA decoder, where iterative decoding by message passing between the outer and inner decoder is known as the k^{th} global iteration, and message passing between the 2 SPC branches in the outer decoder is known as the l^{th} local iteration (Li et al. 2004).

2.5.1 Inner Decoder

Fig. 2.10 and Fig. 2.11 illustrates how messages evolve within the inner rate-1 convolutional code. One important constraint to using the message passing algorithm is that the outgoing message along an edge (lines connecting nodes) should contain information from all other source nodes except the incoming message from this edge.

A decoding iteration in the inner decoder starts when the LLR information L_o from the outer decoder (except the first iteration) and the channel information L_{ch} becomes available. The message obtained from the channel can be calculated as

$$L_{ch}(y_i) = \log \frac{Pr(r_i|y_i = 0)}{Pr(r_i|y_i = 1)} \quad (2.3)$$

where r_i is the received signal corresponding to the coded bit y_i

With these two messages, the forward and backward pass of the inner decoder can be processed, where these 2 passes are necessary to produce the exact *a posteriori* probabilities within the cycle-free graph of the inner code (Schlegel and Perez 2004). Note that these two passes are independent and can be computed concurrently as seen in Fig. 2.10 as follows

$$L_{ef}^{(k)}(y_i) = L_o^{(k-1)}(x_i) \boxplus (L_{ch}(y_{i-1}) + L_{ef}^{(k)}(y_{i-1})), \quad (2.4)$$

$$2 \leq i \leq N$$

$$L_{eb}^{(k)}(y_i) = L_o^{(k-1)}(x_{i+1}) \boxplus (L_{ch}(y_{i+1}) + L_{eb}^{(k)}(y_{i+1})), \quad (2.5)$$

$$1 \leq i \leq N - 1$$

where $L_{ef}(y_i)$ and $L_{eb}(y_i)$ denotes the (extrinsic) messages passed "forward" and "backward" to bit y_i from the sequence of bits/checks before and after the i^{th} position, respectively. The superscript (k) denotes the k^{th} global turbo iteration between the inner and outer decoders (as opposed to the local iterations in the decoding of the outer SPC codes) and the subscript i denotes the i^{th} bit/check node. N here is the size of the output PA codeword and is equal to the block size of interleaver Π_2 , $N_2 = N = P \times (T + 2)$.

$L_o^{(k-1)}(x_i)$ is the message received from the outer SPC code in the $(k - 1)^{th}$ turbo iteration (between inner and outer codes). $L_o^{(0)}(x_i) = 0$, for all i , since in the first turbo iteration the inner code gets no information from the outer code. The boundary

conditions are

$$L_{ef}^{(k)}(y_1) = L_o^{(k-1)}(x_1) \boxplus \infty = L_o^{(k-1)}(x_1), \quad (2.6)$$

$$k \geq 1$$

$$L_{eb}^{(k)}(y_N) = 0, \quad k \geq 1 \quad (2.7)$$

Figure 2.10: A forward and backward pass of the message passing algorithm within the inner decoder.

After the forward and backward pass, all the messages can then be processed and sent to the outer decoder as seen in Fig. 2.11. The extrinsic messages sent out to the outer decoder at bit x_i at the k^{th} turbo iteration $L_e^{(k)}(x_i)$ is computed as

$$L_e^{(k)}(x_i) = (L_{ch}(y_{i-1}) + L_{ef}^{(k)}(y_{i-1})) \boxplus (L_{ch}(y_i) + L_{eb}^{(k)}(y_i)) \quad (2.8)$$

$$1 \leq i \leq N$$

2.5.2 Outer Decoder

For the outer decoder, using MPA, the decoding algorithm can be separated into three main steps as follows.

Figure 2.11: Processing of LLR messages from forward and backward pass to generate the extrinsic LLR messages to pass to the outer decoder.

Outer Decoding: Step 1

At each edge of each bit node in the outer code, message from the inner decoder is summed with messages from other check nodes (from the previous iteration) to generate LLR messages to the corresponding check node as seen in Fig. 2.12. This summing is for the local turbo iteration at the outer code (Li et al. 2004), where best performance would be for each SPC branch to update sequentially as each would then have the most up-to-date information. From Fig. 2.7, it can be seen that the bit nodes corresponding to the input $P \times T$ data bits will be connected to two check nodes each from the upper and lower SPC branch respectively. For bit nodes corresponding to the parity bits, they will only be connected to check nodes from their respective SPC branch.

Note that the constraint “the outgoing message along an edge (lines connecting nodes) should contain information from all other source nodes except the incoming message from this edge” must be satisfied.

Outer Decoding: Step 2

Subsequently, as seen in Fig. 2.13, all the messages from the bit nodes will be processed at each check nodes bearing in mind that the incoming message from an edge is not

Figure 2.12: Step 1: Summing of LLR messages from inner decoder and other check nodes to generate the LLR messages to pass to a check node.

used to calculate the out going message on that edge. The SP or MS algorithm can be used here in the check nodes. Also, check nodes for the lower SPC branch works on bit nodes permuted by interleaver Π_1 , as seen in the code graph of Fig. 2.7, thus there is a need to do interleaving before the checking for the lower SPC branch.

Figure 2.13: Step 2: Processing of LLR messages from other bit nodes to generate the LLR messages back to a particular bit node.

Steps 1 and 2 can be expressed by the following equations:

$$\begin{aligned}
 L_{e1}^{(k,l)}(d_{i,j}) &= \left(\sum_{1 \leq t \leq T, t \neq j} \boxplus (L_e^{(k)}(d_{i,t}) + L_{e2}^{(k,l-1)}(d_{i,t})) \right) \boxplus L_e^{(k,l)}(p_i), \\
 L_{e1}^{(k,l)}(p_i) &= \left(\sum_{1 \leq t \leq T} \boxplus (L_e^{(k)}(d_{i,t}) + L_{e2}^{(k,l-1)}(d_{i,t})) \right),
 \end{aligned} \tag{2.9}$$

where data bits $d_{i,1}, d_{i,2}, \dots, d_{i,T}$ and parity bits p_i participates in the i^{th} SPC codeword

$(1 \leq i \leq P)$ where P and T are the input code parameters. Note that $L_e^{(k)}(x_i)$ from the inner decoder will be de-interleaved by Π_2 to get $L_e^{(k)}(d_{i,j})$ and $L_e^{(k)}(p_i)$ at the input to the outer decoder, where t is a dummy variable. $L_{e1}^{(k,l)}$ are LLR messages from the upper SPC branch of the k^{th} global iteration and l^{th} local iteration (as discussed in section 2.5) while $L_{e2}^{(k,l)}$, which is not shown here, is that for the lower SPC branch. The calculation of $L_{e2}^{(k,l)}$ is similar except that the data bits have been permuted by the interleaver Π_1 . Thus an identical SPC decoder (to that of the upper SPC branch) can be used, with the addition of an interleaver and de-interleaver at the input and output of the decoder respectively.

Outer Decoding: Step 3

Lastly, as seen in Fig. 2.14, all the messages from the check nodes of both the upper and lower SPC branch are summed at each individual bit nodes before sending back to the inner decoder. As bit nodes from the lower SPC branch have been permuted by interleaver Π_1 , there is a need to de-interleave the LLR messages from the lower SPC branch, to allow combining with LLR messages from the upper SPC branch, at each particular bit node.

Figure 2.14: Step 3: Summing of LLR messages from all the check nodes to generate the extrinsic LLR messages to pass to the inner decoder.

Step 3 can be expressed as follows:

$$\begin{aligned}
 L_o^{(k,l)}(d_{i,j}) &= L_{e1}^{(k,l)}(d_{i,j}) + L_{e2}^{(k,l)}(d_{i,j}), \\
 L_o^{(k)}(x_i) &= \Pi_2 \left(L_o^{(k,L)}(d_{i,j}), L_{e1}^{(k,L)}(p_i) L_{e2}^{(k,L)}(p_i) \right)
 \end{aligned} \tag{2.10}$$

Note that LLR information from the two SPC branches for the information data bits are summed together at the bit nodes corresponding to the $P \times T$ information data bits. These are then concatenated with the bit nodes for the parity bits and interleaved by Π_2 before sending back to the inner decoder for the next global iteration. As there can be multiple l local iteration in the outer decoder during the k^{th} global iteration, the extrinsic information sent back to the inner decoder $L_o^{(k)}(x_i)$ should use the latest information at the last local iteration L .

To generate the decoded information bits, all the messages going into each bit node (including messages from the inner decoder) is summed. Each bit node corresponding to an information data bit would then contain the LLR for that information bit. A hard decision can then be made on each LLR to get back the decoded bits.

2.6 Choice of Decoding algorithm

The SP algorithm maximally extracts the coding gain promised by MPA at the cost of higher complexity than the MS algorithm, which suffers from a BER performance loss. A modified MS algorithm is also proposed in literature (Eleftheriou et al. 2001), (Chen et al. 2005), which introduces a compensation term to (2.2) to bridge the performance gap between the MS and the SP algorithms. This compensation term is just a simple addition or subtraction of a constant, thus the main increase in complexity comes from the processing needed to decide the sign of the constant term. This decision requires two additions, two modulus operations, two compares and one AND operation. This compares favorably to the sum-product algorithm, which requires log and exponential operations. The correction term chosen is ± 0.5 with the information bit energy normalized to one.

Another possibility is to scale the output LLR of each bit and check node at the decoder with a constant term, to possibly compensate for the distortion to the extrinsic LLR due to the use of MS algorithm.

The performance of the different algorithms can be seen in Fig. 2.15 with code parameter of (1800, 1350, 0.75), using floating point simulation and 30 iterations of the decoder. MS compensated algorithm is able to close the gap to SP algorithm and the scaling of LLR also allows the MS algorithm to close the performance gap at BER 1×10^{-5} . Interestingly, some performance gain was also observed when the scaling of the output LLR is applied to the SP algorithm. Further analysis is recommended in chapter 7 to investigate this phenomenon.

Figure 2.15: Comparison of sum-product, min-sum, min-sum algorithm with a correction term of ± 0.5 , min-sum with scaled LLR of 0.9 and sum-product with scaled LLR of 0.9

Nonetheless, the increased complexity of the modified MS algorithm appears to outweigh the marginal performance improvement for turbo product codes (TPC) and SPC codes (Hu et al. 2004), thus the modified MS algorithm is not recommended for these codes. From (Li et al. 2004), the MS algorithm without compensation incurs only 0.2dB of performance loss compared to the SP algorithm at BER 1×10^{-5} for a code of parameter (8192, 4096, 0.5), similar to that seen here.

Scaling of the LLR also require multipliers which will add to the FPGA resource requirements. Certain scaling terms can be chosen to reduce the complexity, such as 0.75, which can be derived from $0.75 = 0.5 + 0.25$ where the scaling terms of 0.5 and 0.25 can be generated easily in FPGA with the proper clipping operations. Nonetheless resource requirements will still be increased. Considering the significant reduction in resource consumption, here we choose the MS algorithm with no LLR scaling in our FPGA implementation, to minimize resource utilization.

Chapter 3

Interleaver Design

3.1 Interleavers in PA Code

Each PA code involves two interleavers Π_1 and Π_2 , of lengths $N_1 = P \times T$ and $N_2 = P \times (T + 2)$ respectively, whose implementation directly affects the throughput efficiency. Since interleavers that are easy to simulate in software do not necessarily lend efficient implementation in hardware, we first evaluate several interleavers with a search target for the ones that are efficient to implement in FPGA, easy to reconfigure and providing good BER performance. We focus our search effort on algebraic interleavers, since they can be easily reconfigured and require storage of only a few parameters. Memory expensive interleavers, such as S-interleavers, are not considered here.

In chapter 4, we will use EXIT analysis to investigate the cause of the BER performance differences of these algebraic interleavers, under the constraint of finite codeword length.

3.2 Coprime Interleaver and Relatively Prime Interleaver

Because of their good performance and reconfigurability, the original paper that proposed PA codes (Li et al. 2004) suggested using coprime interleavers based on congruential sequences

$$\mathbf{F}(x+1) = (a \cdot \mathbf{F}(x) + b) \bmod N, \quad (3.1)$$

where x denotes the input sequential position from 0 to $N-1$, $\mathbf{F}(x)$ denotes the interleaved position, $\bmod$ is the modulus operator, and the parameters a and b satisfy certain constraints as specified in (Li et al. 2004). However, in the context of FPGA, the modulus function in (3.1) becomes expensive to implement when $a > 1$ and N is not a power of two. Hence, coprime interleavers are generally not suited for FPGA realization.

In the special case of $a = 1$, coprime interleavers reduce to relatively prime interleavers (Xie et al. 2006), (Crozier et al. 1999):

$$\mathbf{F}(x+1) = (\mathbf{F}(x) + b) \bmod N = (b \cdot (x+1) + \mathbf{F}(0)) \bmod N \quad (3.2)$$

where both the recursive form and the direct form is shown. In the recursive form, the modulus operation can be easily implemented with a compare-and-subtract operation, since $b < N$ and the result from the addition will not exceed $2N$. However, our simulations, plotted in Fig. 3.1 ($P = 225$ and $T = 6$), show that using relatively prime interleavers in a PA code incurs a performance loss of $> 1\text{dB}$ at BER 1×10^{-4} as well as a noticeable error floor. For this and subsequent plots in this chapter, simulation is done using floating point representation, min-sum algorithm, 30 iterations for the decoder and no scaling of LLR at the decoder (i.e. scaling term of unity), unless otherwise stated.

Recall that there are two interleavers in the PA code with size $N_1 = P \times T$ and $N_2 = P \times (T+2)$ corresponding to Π_1 and Π_2 respectively. Using these classes

of interleavers, these two interleavers can be expressed with parameters a_1, b_1, a_2, b_2 respectively. A variety of ratios of b to the interleaver size, b/N , has been tested, including the golden ratio $((\sqrt{5}-1)/2 \approx 0.618)$ and $1/\sqrt{2}$. Although the golden ratio does provide the best performance of the relatively prime interleavers, it still exhibits an error floor and thus performance is greatly degraded.

Figure 3.1: BER performance comparison of coprime interleaver ($a_1 = 31, b_1 = 11, a_2 = 61, b_2 = 13$) against relatively prime interleaver ($a_1 = 1, a_2 = 1$) with different coefficients of $b = 0.717N, 0.707N$ and $0.618N$ for $P = 225$ and $T = 6$.

Additional tests on the combinations of coprime interleavers ($a > 1$) and relatively prime interleavers ($a = 1$), reveal that Π_2 , the interleaver between the inner and the outer code, is particularly sensitive to the rigid structure of the relatively prime interleavers. We observe some 0.5dB performance degradation at $\text{BER } 1 \times 10^{-5}$ when using a coprime interleaver for Π_1 and a relatively prime interleaver for Π_2 , but hardly any degradation the other way around; see Fig. 3.2.

Figure 3.2: BER performance comparison of PA codes with coprime and relatively prime interleavers

3.3 Prime Factor Based Multi-Dimensional Helical Block Interleavers

A (two-dimensional) helical block interleaver, or, simply, helical interleaver, as defined in (Chi 1992), is a block interleaver whose row size R is relatively prime to the column size C :

$$\mathbf{F}(x) = \mathbf{r}(x) + (\mathbf{c}(x) \cdot R) \quad (3.3)$$

where

$$\mathbf{r}(x) = x \bmod R, \quad \mathbf{c}(x) = x \bmod C$$

and x denotes the input sequential position from 0 to $N - 1$.

This construction can be conveniently generalized to multiple dimensions by using the prime factors of the interleaver size. Consider an interleaver of size N , where

N is factorized into its prime factors $f_1^{n_1}, f_2^{n_2}, \dots, f_m^{n_m}$. Since the sizes of the different dimensions of a helical interleaver must be relatively prime to each other, the resulting helical interleaver has a maximum dimension of m with a size of $f_i^{n_i}$ in each dimension. It is also possible to reduce the dimensions by combining multiple $f_i^{n_i}$'s in one dimension. Take the example of $N = 1848 = 2^3 \cdot 3 \cdot 7 \cdot 11$. It is possible to have a 4-dimensional helical interleaver with size 11, 8, 7 and 3 respectively on each dimension. It is also possible to have 3-dimensional helical interleavers, such as $11 \times 21 \times 8$ or $11 \times 24 \times 7$.

For an m -dimensional interleaver with dimensional sizes $l_0, l_1, \dots, l_{m-1}$ respectively, the interleaved position of x follows the following simple rule:

$$\mathbf{F}(x) = a_0 + (a_1 l_0) + (a_2 l_0 l_1) + \dots + (a_{m-1} l_0 l_1 \dots l_{m-2}) \quad (3.4)$$

where $a_i = x \bmod l_i, i = 0, \dots, m-1$. Implementation of this interleaver requires m cyclic counters to calculate a_i 's, and $m-1$ adders and $m-1$ multipliers to calculate $\mathbf{F}(x)$. Here the counters can individually be either up or down counters.

While prime factor based helical interleavers are relatively efficient to implement, their performances in PA codes remain to be desired. From the simulation curves in Fig. 3.3 ($P = 231, T = 6$ for the prime factor helical interleaver, with dimensions (11, 9, 7, 2) for Π_1 and (11, 8, 7, 3) for Π_2 . $P = 225, T = 6$ for the coprime interleaver with parameters $a_1 = 31, b_1 = 11, a_2 = 61, b_2 = 13$), it can be seen that the performance degrades significantly for BER below 1×10^{-3} and thus the helical interleaver is not a good choice performance-wise.

We also examined different types of interleavers for Π_1 while keeping the coprime interleaver for Π_2 . For the helical block interleaver, the array step size is one, ie the normal helical interleaver. For helical block interleaver (golden ratio of P), the array step size is chosen such that it is the golden ratio of the array depth, where the array in this case is 225×6 , thus the step size is 139. The block interleaver (also known as transpose interleaver) reads in data row by row and reads out column by

Figure 3.3: BER performance comparison of coprime interleavers and prime factor helical interleavers

column. As seen in Fig. 3.4, all of them exhibit similar performances. This indicates that the interleaver design is more critical for the serial concatenation between the outer code and inner code and less so for the parallel concatenation within the outer code, in agreement with the results seen in Fig. 3.2. It is generally recognized that serial concatenation is more advantageous over parallel concatenation in terms of BER performance (Benedetto et al. 1996), and in the context of PA codes, the serial concatenation dominates the performance.

Figure 3.4: BER performance comparison with Π_2 using coprime interleaver and Π_1 using different types of interleaver including relatively prime, block, helical (golden ratio of P), helical interleavers and also coprime interleaver

3.4 Quadratic Permutation Polynomial Based Interleavers

Maximum contention free interleavers (Takeshita 2006) enable the block decoding to be performed as parallel decoding of several disjoint segments of the block, since the memory access for different segments are guaranteed contention free. Let N be the block word size, M be the number of memory blocks and also the number of parallel decoders and W be the window size, i.e. the number of soft information bits stored in a memory block.

$$N = M \times W \quad (3.5)$$

Under this constraint, a maximum contention free interleaver will ensure that there is no memory contention among the M parallel decoders, with each having access to any one of the memory blocks at each time.

A quadratic permutation polynomial (QPP) based interleaver, recently proposed in (Sun and Takeshita 2005) and satisfying the constraint in (3.5), scrambles index x to $\mathbf{F}(x)$ according to the following rule:

$$\mathbf{F}(x) = (ax + bx^2) \bmod N, \quad (3.6)$$

where the parameters a and b satisfy the constraints that b is divisible by the prime factors of N and a is not. Since the quadratic term in (3.6) makes the modulus operation difficult, it is desirable to express (3.6) in a recursive form (Cheng et al. 2005)

$$\begin{aligned} \mathbf{F}(x+1) &= [\mathbf{F}(x) + \mathbf{H}(x)] \bmod N, \\ \mathbf{H}(x) &= [a + b + 2bx] \bmod N, \\ \mathbf{H}(x+1) &= [\mathbf{H}(x) + 2b] \bmod N, \end{aligned} \quad (3.7)$$

which then completely eliminates the multiplication operation. If $2b$ is limited to

be smaller than N and since $\mathbf{H}(x)$ is smaller than N , the modulus operation in the calculation of $\mathbf{H}(x+1)$, and thus $\mathbf{F}(x+1)$, can be efficiently realized through a simple compare-and-subtract operation.

For the design of the deinterleavers, quadratic inverses for the quadratic permutation polynomials can be derived (Ryu and Takeshita 2006) if certain constraints are met.

$$\mathbf{F}^{-1}(x) = (cx + dx^2) \bmod N, \quad (3.8)$$

Thus it would be possible to build deinterleavers that are structurally identical to interleavers, but with different coefficients. Note that not all quadratic permutation polynomials will admit a quadratic inverse, though experience has shown that most do (Ryu and Takeshita 2006). The necessary and sufficient conditions for the existence of quadratic inverse for the Quadratic Permutation Polynomials and the algorithm for deriving the coefficients c and d is shown in (Ryu and Takeshita 2006).

To get around this possibility of no feasible quadratic inverses, we will exploit the dual-port feature of the on-chip dual port block rams (DPBRAM), and implement both the interleaver and the de-interleaver using the same interleaving pattern in (3.7).

DPBRAM allows the system to write in and read out data from either port. In our implementation, interleaving and de-interleaving are accomplished by writing data into one port of the DPBRAM and reading out data from the other port. Thus by having a sequential address on one port and the interleaved address on the other port, interleaving and de-interleaving can be accomplished, without the need for the explicit de-interleave sequence, as follows:

Interleave: Read addr $[\mathbf{F}(x)] \Leftarrow$ Write addr $[x]$

De-interleave: Read addr $[x] \Leftarrow$ Write addr $[\mathbf{F}(x)]$.

Fig. 3.5 shows that, using the MS algorithm, floating point simulation, 30 iterations of the decoder, QPP interleavers ($P = 225$ and $T = 6$) perform similarly to

coprime interleavers without visible rising of the error floor.

Figure 3.5: BER performance comparison of coprime interleaver and QPP interleaver

Fig. 3.6 shows the BER performance when Π_1 is the block interleaver and Π_2 is the QPP interleaver. Similar to that observed earlier for coprime interleaver, the effect of Π_1 on the BER performance is limited. QPP interleaving for Π_1 is slightly more complex than block interleaving in the generation of the interleaving address, however, QPP has the advantage of allowing parallel processing at the decoder which will result in a correspondingly higher data rate, though at the expense of more hardware resources.

Table 3.1 shows the resource utilization of QPP interleaver for two values of $N = P \times (T + 2)$. As Π_2 is the bigger of the two interleavers in PA code, its size will be used as the benchmark here. As can be seen, QPP interleaver does not use any multipliers or block rams to store the interleaving sequence. Instead only adders and

Figure 3.6: Comparison of using QPP interleaver vs block interleaver for Π_1 , with Π_2 using QPP interleaver

comparators will be used, thus needing only logic slices to implement. As the size gets bigger, the bitwidth of the adders and comparators increases, thus the increased slice utilization.

Table 3.1: QPP interleavers Resource Utilization, for an interleaver size of 2048 and 16384 bits

Xilinx FPGA(2v3000fg676-4)	QPP	
Max Interleaver size ($P \times (T + 2)$)	2048	16384
Slices (14,336)	84 (0.59%)	109 (0.76%)
Slice Flip Flops (28,672)	46 (0.16%)	58 (0.20%)
4 input LUTs (28,672)	152 (0.53%)	191 (0.67%)
MULT18X18s multipliers (96)	0 (0.00%)	0 (0.00%)
18Kb BRAM (96)	0 (0.00%)	0 (0.00%)
Max Frequency (MHz)	125	125

3.5 Choice of Interleaver

Fig. 3.7 shows the performance of coprime interleavers ($a_1 = 31, b_1 = 11, a_2 = 61, b_2 = 13$) against that of QPP interleavers ($a_1 = 17, b_1 = 30, a_2 = 11, b_2 = 180$), relatively prime interleavers ($a_1 = 1, b_1 = 833, a_2 = 1, b_2 = 111$) and prime factor based helical interleavers with dimensions $(11, 9, 7, 2)$ for Π_1 and $(11, 8, 7, 3)$ for Π_2 . From a BER performance point of view, coprime interleavers would have the best performance. However, their complexity in implementation in an FPGA does not justify their use.

Hence QPP interleavers present a best overall choice in BER performance, reconfigurability and compact implementation; and will be used in the FPGA implementation. In chapter 4, EXIT analysis will be used to validate the performance differences of the different interleavers and to investigate the cause of these performance differences.

Figure 3.7: BER performance comparison of coprime interleavers, QPP interleavers, relatively prime interleavers and prime factor based helical interleavers

Chapter 4

EXIT Analysis

4.1 Extrinsic Information Transfer Chart

EXIT analysis is commonly employed to predict the behavior of iterative systems. This is particularly useful for turbo codes since it uses an iterative decoder. The key idea here is to visualize the evolution of the mutual information (MI) exchanged between the component decoders, from iteration to iteration. An EXIT chart thus consists of a pair of curves which represent the mutual information transfer functions of each of the component decoders in the turbo decoding, assuming two sub-decoders in either serial or parallel concatenation. Each curve is essentially a plot of the average extrinsic mutual information I_e against the *a priori* mutual information I_a for the component decoder of interest. The mutual information here, is a measure of the degree of dependency between the log-likelihood variables $L_a(x)$ or $L_e(x)$ and the corresponding transmitted bit x . The mutual information takes on values between 0 for no knowledge, and 1 for perfect knowledge of the transmitted bits, dependent on the reliability of the likelihood variables. The terms I_a and I_e are related to P_a and P_e , the probability density functions (pdfs) of $L_a(x)$ and $L_e(x)$ respectively, the signal-to-noise ratio E_b/N_0 , and the code graph structure. Naturally, if the component decoders are identical, the two curves will be mirror images.

To simplify analysis, two common assumptions (ten Brink 2001) are made,

1. The independent assumption: For large interleavers the *a priori* values remain fairly uncorrelated from the respective channel observations over many iterations.
2. The Gaussian assumption: The probability density functions of the extrinsic output values (*a priori* values for the next decoder respectively) approach Gaussian-like distributions with increasing number of iterations.

Under these two assumptions, the *a priori* information, can be assumed to have a Gaussian distribution. As such, in the case of PA decoding as described in section 2.5, the two inputs $L_o(x_i)$ and $L_{ch}(y_i)$ into the inner decoder and the single input $L_e(x_i)$ into the outer decoder are assumed to have Gaussian distributions. This thus allows the sub-decoders to be modeled independently. Consequently, the inputs to each sub-decoder can be injected at any arbitrary mutual information level and by adjusting the variance of the Gaussian distribution, this thus allows the full EXIT chart (covering the full range of input MI from $0 \rightarrow 1$) to be plotted over any E_b/N_0 .

The MI of the *a priori* distribution I_a can be calculated (ten Brink 2001) as

$$I_a = \frac{1}{2} \cdot \sum_{x=-1,1} \int_{-\infty}^{\infty} p_a(\xi|X=x) \times \log_2 \frac{2 \cdot p_a(\xi|X=x)}{p_a(\xi|X=-1) + p_a(\xi|X=1)} d\xi, \quad (4.1)$$

$$0 \leq I_a \leq 1$$

where the binary random variable X denotes the transmitted bits with realizations $x \in \{\pm 1\}$. Satisfying the 2 assumptions above where a Gaussian distribution is assumed for $p_a(\xi|X=x)$, I_a can then be calculated as (ten Brink 2001)

$$I_a(\sigma_a) = 1 - \int_{-\infty}^{\infty} \frac{e^{-((\xi-\sigma_a^2/2)^2/2\sigma_a^2)}}{\sqrt{2\pi}\sigma_a} \cdot \log_2[1 + e^{-\xi}] d\xi \quad (4.2)$$

where σ_a^2 denotes the variance of the Gaussian distribution.

Similarly, the MI at the output of each sub-decoder I_e is calculated by

$$I_e = \frac{1}{2} \cdot \sum_{x=-1,1} \int_{-\infty}^{\infty} p_e(\xi|X=x) \times \log_2 \frac{2 \cdot p_e(\xi|X=x)}{p_e(\xi|X=-1) + p_e(\xi|X=1)} d\xi, \quad (4.3)$$

$$0 \leq I_e \leq 1$$

For the output extrinsic information, no probability distribution is assumed. Rather, the required pdfs can be estimated by generating histograms $p_e(\xi|X=x)$, of the output message of each sub-decoder through Monte Carlo simulation.

Fig. 4.1 shows such an EXIT chart for a rate 0.5 PA-I code (2048, 1024) at $E_b/N_o = 4dB$. Note that at this E_b/N_o level, the gap in the EXIT chart is wide open and most codewords will be able to converge quickly, thus the BER would be very low ($< 1 \times 10^{-7}$).

Figure 4.1: EXIT chart for a PA-I code (2048,1024, $E_b/N_o = 4dB$) using min sum algorithm, floating point simulation and random interleaver

4.1.1 Finite Length EXIT Trajectories Analysis

However, the independence assumption in EXIT charts is valid only for very large codes (eg. hundreds of thousands of bits). When code length is relatively small (eg. 2000 bits) this assumption may deviate noticeably from the real situation. Therefore a more detailed form of EXIT analysis for finite length PA codes under different interleavers is needed, where a full simulation of the decoder is performed. This is necessary as no assumption on the distribution of the *a priori* information is made. Only the channel information is modeled as having a Gaussian distribution.

In an iterative decoder, the output of one sub-decoder feeds into the input of the other sub-decoder; thus only the output MI needs to be calculated for the EXIT analysis. In a staircase plot, the MI trajectory will thus show the exchange of extrinsic information between the sub-decoders using the MI metric, ie an EXIT trajectory. Fig. 4.2 shows a typical MI trajectory for a PA code (1800, 1350, 0.75) using coprime interleavers with coefficients $a_1 = 31$, $b_1 = 11$, $a_2 = 61$ and $b_2 = 13$, where the subscript denotes interleavers Π_1 and Π_2 respectively. The '×' marker shows the input vs output MI of the inner decoder at each iteration step whereas the 'o' marker shows that of the outer decoder.

The end of the 1st iteration (corresponding to the 'o' marker here), is defined as the 1st output of the inner decoder being used as input to the outer decoder and plotting against the 1st output of the outer decoder. This is followed by the start of the 2nd iteration ('×' marker), where the 1st output of the outer decoder is then used as input to the inner decoder and plotting against the 2nd output of the inner decoder. The end of the 2nd iteration is denoted by the 2nd 'o' marker, which again uses the 2nd output of the inner decoder as input to the outer decoder and plotting against the 2nd output of the outer decoder. This process continues until the 30th iteration, and the EXIT trajectory is thus plotted. Note that there is no start of the 1st iteration ('×' marker) being plotted here as, at the start of the decoding, the outer decoder provides no information to the inner decoder (i.e. $L_o^{(0)}(x_i) = 0$, for all i). As

such, the inner decoder starts only with the channel information L_{ch} , as discussed in section 2.5.1.

Notice that the decoder was able to converge (MI = 1 for both sub-decoders) by the 13th iteration, although a total of 30 iterations were done.

Figure 4.2: EXIT trajectory of a codeword using coprime interleaver (31, 11, 61, 13). ($E_b/N_o = 2dB$)

For such iterative systems, convergence is not guaranteed, some codewords will not be able to converge properly and the output MI will always be stuck at a low value even with many iterations, especially so in the low E_b/N_o regime. Further more, another type of behavior is also possible, and that is the divergence of EXIT trajectories. Fig. 4.3 shows just such a divergence occurring when using a relatively prime interleaver at an E_b/N_o of 3dB. Even though the output MI for both outer and inner decoder has reached > 0.975 (by $\approx 11^{th}$ iteration); the trajectory suddenly diverges and the output MI drops to < 0.1 and < 0.6 , for outer and inner decoder respectively, within a few iterations. Notice how the EXIT trajectory then seems to

converge toward output MI of 1 again but the iteration limit of 30 was reached before it could reach 1.

With conventional EXIT charts where the trajectories are in a sense averaged out, such divergence of a few codewords cannot be captured. Furthermore such divergence can mean that a codeword will be decoded with a significant burst of error bits and thus can actually be the dominant cause of poor BER performance at the high E_b/N_o regime.

Figure 4.3: EXIT trajectory of 1 code word, using relatively prime interleaver. Divergence with 30 iterations. ($E_b/N_o = 3dB$)

4.2 EXIT Trajectories Analysis of Four Algebraic Interleavers

To investigate deeper into the BER performance difference of the four interleavers (coprime interleaver, QPP interleaver, relatively prime interleaver and prime factor

based helical interleaver) and explore if divergence is a possible cause, we will plot the EXIT trajectories for 100 codewords showing only the markers, for each of the four interleaver. A simulation for 1000 codewords had been done and the EXIT trajectories plot shows similar behavior to that for 100 codewords. However for a plot of 1000 codewords, the overwhelming number of markers necessitates a prohibitively bigger graph to discern the markers, as shown in Fig. 4.4, where most markers are indiscernible. Thus plots of 100 codewords will instead be used for illustration purposes here. Note that Fig. 4.4 is not plotted over the maximum MI range from 0 to 1 for both the input and output MI. Instead, it is zoomed in to a suitable range that is able to show the entire trajectory for nearly all the codewords. Also, as the *a priori* information is not assumed to take any distribution, only the channel LLR L_{ch} is Gaussian, thus it is not possible to plot the EXIT chart to cover the entire range of input MI from $0 \rightarrow 1$, as in the more common EXIT chart (ten Brink 2001).

This is also done similarly for the subsequent plots from E_b/N_o of 5, 4, 3 and $2dB$. A PA code of parameter $(N, K, r) = (1800, 1350, 0.75)$ (except prime factor helical interleaver which used $(1848, 1386, 0.75)$), min sum algorithm, 30 iterations for the decoder and floating point simulation has been used here.

$E_b/N_o = 5dB$

At $E_b/N_o = 5dB$, as shown in Fig. 4.5, most codewords are able to converge within a few iterations and the EXIT trajectories marks out the open EXIT tunnel similar to that in Fig. 4.1. Specifically, the '+' markers is concentrated along the curve of the inner decoder while the 'o' marker follows that of the outer decoder curve in Fig. 4.1. Although this was plotted with 100 codewords, the key features are similar to that in Fig. 4.4, which plotted for 1000 codewords. Due to the high E_b/N_o of $5dB$, the MI of the output extrinsic information of the inner decoder after the 1st iteration is already at a high level of around 0.9. As such, the plot is zoomed in on the upper range of the MI.

Figure 4.4: Scatter plot of EXIT trajectories's vertices using different interleavers. ($E_b/N_o = 5dB$, 1000 codewords)

Notice that for relatively prime interleavers, divergence seems to have occurred for one of the codeword. This can be seen by the presence of the '+' marker on the upper left quadrant of the plot for relatively prime interleaver, which implies that the EXIT trajectory had diverged, similarly to that of Fig. 4.3. Thus divergence here can be defined as an EXIT trajectory which falls out of the EXIT tunnel as that defined in the more common EXIT charts, with the MI dropping drastically within a few iterations. This can be more obviously seen in Fig. 4.6, which is plotted at a lower $E_b/N_o = 4dB$. Comparatively, the performance of the other three interleavers are similar.

Figure 4.5: Scatter plot of EXIT trajectories's vertices using different interleavers. ($E_b/N_o = 5dB, 100$ codewords)

$E_b/N_o = 4dB$

At $E_b/N_o = 4dB$, as shown in Fig. 4.6, the gap in the EXIT tunnel has closed a fair bit for all four interleavers. However for relatively prime interleaver and prime factor based helical interleaver, divergence of EXIT trajectory is seen for several codewords, with relatively prime interleaver having more diverging EXIT trajectories. The trajectory of the few diverging EXIT trajectories can be seen here to fall obviously out from the normal EXIT tunnel taken by most of the other EXIT trajectories. For coprime and QPP interleaver, the EXIT trajectories shows no divergence and thus the BER can be seen to outperform relatively prime and prime factor based helical

interleaver at $E_b/N_o = 4dB$, as shown in Fig. 4.9.

Figure 4.6: Scatter plot of EXIT trajectories' vertices using different interleavers. ($E_b/N_o = 4dB$, 100 codewords)

$E_b/N_o = 3dB$

At $E_b/N_o = 3dB$, as shown in Fig. 4.7, the gap in the EXIT tunnel is not visible for all four interleavers. Some codewords were not able to converge properly and thus are stuck at a low output MI. However, many codewords are still able to converge to output MI of 1, especially for coprime interleaver and QPP interleaver. For relatively prime interleaver and prime factor based helical interleaver, we again see that some codewords had EXIT trajectories which diverge, with relatively prime interleaver again having more diverging EXIT trajectories. Although these trajectory plots are

not able to properly show any EXIT tunnel gap, especially near the SNR threshold, the main phenomena of concern here are the diverging trajectories, which will not be separable or discernible using the more common EXIT chart presentation which averages across many codewords.

Figure 4.7: Scatter plot of EXIT trajectories' vertices using different interleavers. ($E_b/N_o = 3dB$, 100 codewords)

$$E_b/N_o = 2dB$$

At $E_b/N_o = 2dB$, as shown in Fig. 4.8, only coprime interleaver had a few codewords converge to MI of 1; whereas QPP interleaver had 1 codeword which squeezed through the EXIT tunnel gap but did not quite reach output MI of 1 within the iteration limit, though it did reach MI of ≈ 0.95 .

Relatively prime interleaver and prime factor based helical interleaver had no codewords converging and it seems that the EXIT tunnel gap is closed for these two interleavers at $E_b/N_o = 2dB$, thus no codewords were able to squeeze through. Notice that for these two poor performing interleavers, there does not seem to be any obvious diverging trajectories. In fact, it can be observed that obvious divergence occurs mainly for MI at higher ranges (> 0.8).

Figure 4.8: Scatter plot of EXIT trajectories' vertices using different interleavers. ($E_b/N_o = 2dB$, 100 codewords)

4.3 Divergence of EXIT Trajectory

From these EXIT trajectories plot, we can see that coprime interleaver has the best performance, in terms of the number of codewords converging or diverging, followed very closely by QPP interleaver. Prime factor based helical interleaver had less diverging EXIT trajectories than relatively prime interleaver thus is third in performance with relatively prime interleaver having the worst performance. This performance difference is also reflected in the frame error rate and bit error rate plots in Fig. 4.9. From this figure, one can also see that for relatively prime interleaver and prime factor based helical interleaver, there is an obvious error floor where the slope of the BER and FER curve is less steep than that of coprime interleaver and QPP interleaver.

Figure 4.9: Frame and Bit Error Rate using the 4 different interleavers, with MS algorithm, 30 iterations and floating point simulation

Divergence can happen to a codeword repeatedly, as seen in Fig. 4.10 where 400

iterations were done on a codeword using relatively prime interleaver at E_b/N_o of $3dB$. The iteration direction arrow and the '+' and 'o' markers have not been shown here but we can see that divergence occurred quite a few times as the decoder tries unsuccessfully to converge. Thus when using interleavers with such divergent behavior, decoding with more iterations may not be able to improve its BER performance. This effect can be seen in Fig. 4.11 where coprime and relatively prime interleavers are compared under 30 and 100 iterations. As can be seen, both BER performance improve by about $0.1dB$ and the error floor due to the use of relatively prime interleaver is not improved.

From the BER and FER plots, it can be seen that QPP interleaver has similar BER and FER performance compared to coprime interleaver, and from the EXIT trajectories analysis, QPP interleaver has minimal number of diverging codewords as compared to relatively prime interleaver and prime factor based interleaver. As such, it verifies the conclusion from section 3.5 that QPP interleaver presents the best overall choice in BER performance, reconfigurability and compact implementation, and thus will be used in the FPGA implementation

Figure 4.10: EXIT trajectory of 1 code word, using relatively prime interleaver. Divergence even with 400 iterations. ($E_b/N_o = 3\text{dB}$)

Figure 4.11: Coprime vs relatively prime interleaver using 30 or 100 iterations, with MS algorithm and floating point simulation

Chapter 5

Encoder and Decoder Implementation

5.1 Encoder Design

The encoder of a PA code consists of two main modules namely: the SPC encoder for the outer code and the accumulator for the inner code. Note that the accumulator works on the binary field and thus is identical to that of a binary differential encoder. The PA encoder is not iterative, unlike the decoder, thus its throughput can be much higher relative to that of the decoder. In this design, a pipelined encoder will be used and as the throughput is not the main constraint here, a bit-serial encoder has been chosen.

Before encoding can start, the encoder first takes in a set of parameters which will be used to configure the **entire** encoder. Note that this means that although the encoder is pipelined, it does not support codewords of different parameters within the pipeline. Any parameter change can only happen when the encoder is empty. An *Encoder Empty* flag is available to facilitate these changes, where flag asserted high ('1') means the encoder is empty. This set of parameters should be derived either offline or by another processor controlling the encoder and are dependent on the main PA code parameters of: P , T , a_1 , b_1 , a_2 and b_2 , where the last four are coefficients for

the two interleavers in PA code. These parameters are needed as the encoder operates serially and will depend on various counters, within the different finite state machines (FSM) in the encoder, to support the different code structures. These parameters will be described in sections 5.3 and 5.7 which details the interfaces and timing diagrams of the encoder and decoder respectively.

This approach allows flexibility in reconfiguring the encoder without needing resources in the FPGA to compute the entire set of parameters needed by the encoder to generate the required code structure. As the parameters do not change often, this saves resources which would otherwise be idle for most of the time. A similar approach is done for the decoder.

5.1.1 Encoder Implementation

In this section, the implementation of the PA encoder will be detailed, according to the encode process discussed in chapter 2.3.

Input Information Bits

The implemented encoder is as shown in Fig. 5.1. The input to the encoder comes from either a 32-bit wide data input port or the built in pseudo-random bit sequence (PRBS) generator, *PRBS21*, which generates an M-sequence using memory length of 21 bits, for a sequence period of $2^{21} - 1$ bits. The PRBS generator is useful for measuring BER performance in a loopback test as this sequence is commonly supported by most BER testers (e.g. Rohde & Schwarz SMIQ03B signal generator). The generator is built using linear feedback shift registers (LFSR) with feedback taps defined by a set of coefficients. In this implementation, the feedback tap has been hard-coded to be at position 19 and 21 of the shift registers. The input interface assumes the data comes from a first-in-first-out (FIFO) memory device with read latency of two system clock cycles. A multiplexer is then used to choose between the PRBS or the input data and the information bits will then be stored in the *Encoder*

In FIFO which has a width of 32 bits and is 512 words deep. Note that the depth of this FIFO can be easily changed to accommodate bigger codeword lengths.

Parallel to Serial Converter

The main part of the encoder actually begins after this point at the *Parallel to Serial* module. For this encoder, it can be viewed as a 4-stage pipeline made up of the *Encoder In FIFO*, *Serial Data DPBRAM*, *SPC Encoded DPBRAM* and *Encoder Out FIFO* modules. Note that it is possible to remove some of the internal input and output interface modules such as the *PRBS21*, *Encoder In FIFO*, *Encoder Out FIFO*, etc. so as to reduce resource utilization. The full encoder design in Fig. 5.1 aims to enable easy interfacing by using FIFOs as the main interface, which may not always be the preferred way. Also, if these internal interface modules are removed, the external interfaces to the PA encoder will have to take care of more of the interfacing requirements such as buffering of data. Such a cut-down encoder will be shown in section 5.2.

As data are being written into the *Encoder In FIFO*, the *Parallel to Serial* module will use the PA parameters to determine when the required number of $P \times T$ bits has been written in and proceed to serialize the required number of 32-bits input words and storing it into the *Serial Data DPBRAM* module with the help of the *PingPong Flow Control* module. It is assumed that for $P \times T$ not a multiple of 32 bits, zero padding will be done on the most significant bits (MSB) of the last 32-bit word.

The DPBRAM has two ports which can be independently configured to be read, write or read and write. In this case, port A is write only for the *Parallel to Serial* module and port B is read only for the *SPC encoder* module. A 16K word (1 bit per word) DPBRAM is used here and is divided into 2 halves of 8K words each. Data bits will be written into one half by the *Parallel to Serial* module while the other half will be read by the *SPC encoder* module for processing, thus the maximum codeword length is 8K bits. Note that at this point, the data are still uncoded, thus the block length is $P \times T$ bits.

Figure 5.1: Encoder implementation with 3-stage pipelining: parallel to serial, SPC encoder (outer code) and differential encoder (inner Code)

When both read and write for the two ports are done, the read and write can swap over to the corresponding other half of the *Serial Data DPBRAM* as controlled by the *PingPong Flow Control* module. This is done by reserving the most significant bit (MSB) of the 14-bit address into the DPBRAM to denote either the odd ('1') or even ('0') half of the DPBRAM. Thus the *Parallel to Serial* module generates an address of 13 bits and appends it to an MSB flag maintained by itself, which it toggles after every block of data, to form the address written to the DPBRAM. A similar flag is maintained by the *SPC encoder* module.

PingPong Flow Control

To facilitate this flow control mechanism, the *PingPong Flow Control* module maintains two flags, for the even and odd half respectively, to denote whether that half of the memory is empty ('0') or full ('1'). Upon reset, the module will deassert both flags. Both the input and output modules being coordinated by the *PingPong Flow Control* module have been set to default start at the even half of the DPBRAM. When the first block of data bits come in, the MSB will be '0' and data will be written into the even half of the DPBRAM. When writing of this block of bits is completed, the *Parallel to Serial* module will generate a *ENCODER_DATA_EVEN_BLOCK_FILLED_STB* strobe, which the *PingPong Flow Control* module will use to assert the *ENCODER_DATA_EVEN_BLOCK_FULL* flag to denote that it is now full. The MSB flag at the *Parallel to Serial* module will then toggle to '1' with the *ENCODER_DATA_ODD_BLOCK_FULL* flag still at '0', thus the *Parallel to Serial* module can then proceed to write into the odd half of DPBRAM.

SPC Encoder (Outer Code)

The *SPC encoder* module upon seeing that the *ENCODER_DATA_EVEN_BLOCK_FULL* flag is asserted, will proceed to read out the bits in the even half for processing. Note that another MSB flag is maintained by the *SPC encoder* module

to facilitate its reading of the *Serial Data DPBRAM*. After all the bits have been read out and processing is completed, the *SPC encoder* module will generate a *SER_DATA_EVEN_BLOCK_EMPTIED_STB* strobe which the *PingPong Flow Control* module will use to de-assert the *ENCODER_DATA_EVEN_BLOCK_FULL* flag to denote that it is now empty. The *SPC encoder* module will also toggle its MSB flag after every block of data. This pingpong style of flow control is then done for all subsequent blocks of data by toggling between the even and odd half of the DPBRAM.

One SPC encoder has been instantiated here which will do the SPC encoding for the upper and lower branch of the SPC codes, as seen in Fig. 2.3, sequentially. Thus this creates a bottleneck for the encoder as throughput will be half that of an encoder which instantiates two SPC encoders for the outer code. Nonetheless, the throughput is still much higher compared to the decoder and thus minimizing resource utilization takes priority here. As this is a serial encoder, throughput here is just slightly below half of the clock rate. For a clock rate of 100MHz, the throughput is around 49Mbps, where throughput here is defined as the rate for PA coded bits¹. Some clock cycles are wasted during transition between the upper and lower branch of the SPC encoding and also between codewords, thus the throughput is slightly below half of the clock rate. Note that the number of wasted clock cycles also depends on the code parameters, thus this figure changes very slightly for different code structures.

QPP Interleaver

For the upper and lower branches of the SPC codes, the input for both comes from the $P \times T$ information bits, with the only difference being that the lower branch includes an additional interleaver Π_1 . To implement the QPP interleaving, an address generation module, *Interleaver 1 Address Generator*, is required to generate the interleaved address $\mathbf{F}_1(x)$ to read from the *Serial Data DPBRAM* module as described in section 3.4. For the upper branch, the address generation is from a simple up counter that counts serially from $0 \rightarrow N_1 - 1$, where $N_1 = P \times T$.

¹It is also possible to measure the throughput in terms of the rate for data bits, but for ease of comparison and to make it independent of code rate, here we measure it against coded bits.

The output from the SPC encoder will then be written serially to the *SPC Encoded DPBRAM* module with the $P \times T$ information data bits first, followed by the P parity bits from the upper SPC branch and lastly, the P parity bits from the lower SPC branch. Note that at this point, the SPC encoder will thus generate a block of bits of length $P \times (T + 2)$. Another *PingPong Flow Control* module is used for flow control to the *Differential Encoder* module (inner code).

Differential Encoder (Inner Code)

Between the outer and inner code, there is an interleaver, Π_2 . To implement this interleaving, the *Differential Encoder* module will use the *Interleaver 2 Address Generator* module to generate the interleaved address $\mathbf{F}_2(x)$ for reading out from *SPC Encoded DPBRAM*. A simple differential encoder will then proceed to encode the bits as they are being read out. These PA coded output bits will be immediately written out to the *Encoder Out FIFO* module.

Note that before the *Differential Encoder* module begins to process a block of bits, it will check that the *Encoder Out FIFO* module has sufficient space to accommodate this block of PA coded bits. An *Encoder_Out_FIFO_Level* signal from the *Encoder Out FIFO* module is used for this check.

5.2 Cut-down Version of Encoder Implementation

Fig. 5.2 shows a cut-down version of the PA encoder with minimal interfacing modules. For input, the external interface will directly write data into the *Serial Data DPBRAM* using the interfacing signals from the *PingPong Flow Control* module. Also, the major difference here is that data will have to be serially written in as compared to the full encoder where 32-bit words are written into the *Encoder In FIFO*.

Figure 5.2: Cut-down version of the encoder for minimal resource utilization

At the output, the external interface will have to be responsible for buffering of the PA coded bits. When the *Differential Encoder* module is ready to output data, each bit will be generated serially with a corresponding *write enable* signal. This enables easy interface to a FIFO. The *Differential Encoder* module takes in an *Encoder_Out_FIFO_Level* signal to check that the output external interface is ready to take in a new block of PA coded bits. For the cut-down PA encoder, it can be viewed as a 2-stage pipeline made up of the *Serial Data DPBRAM* and *SPC Encoded DPBRAM* modules.

Note that as external interfaces are still needed, this design does not mean that the overall system resource requirements will be reduced significantly. Nonetheless, if the external input interface is already a serial interface, there will be savings in the removal of the *Parallel to Serial* and *Encoder In FIFO* modules. The trade-off is increased interfacing requirements on the external interfaces and tighter coupling of the internal and external interfaces.

Table 5.1 shows the resource utilization comparison between the full encoder and cut-down encoder. Due to some interface constraints within the encoder module, the maximum block size $P \times (T + 2)$ has been limited to 8191. In the 1st column, the numbers bracketed shows the total number of that particular primitive in the target FPGA. Note that as the implementation is in VHDL (with some Xilinx IPcore) and can be easily ported to different Xilinx FPGAs with widely differing amount of resources, the resource utilization as a percentage of the total resource of this particular FPGA is just for reference purpose.

5.3 Cut-down Encoder Interface and Timing Diagram

In view of the relatively higher complexity of the interface for the cut-down encoder against that of the full encoder, here we will present in detail the interface and timing diagram for the cut-down encoder.

Table 5.1: Comparison of resource utilization of full encoder against cut-down encoder, for a maximum PA coded block size of 8191 bits

Xilinx FPGA(2v3000fg676-4)	Full Encoder	Cut-down Encoder
Max Block size ($P \times (T + 2)$)	8191	8191
Slices (14,336)	578 (4.03%)	322 (2.25%)
Slice Flip Flops (28,672)	419 (1.46%)	268 (0.93%)
4 input LUTs (28,672)	978 (3.41%)	507 (1.77%)
18Kb BRAM (96)	5 (5.21%)	3 (3.13%)
Max Frequency (MHz)	125	125

5.3.1 Cut-down Encoder Interface

Table 5.2 shows the interface ports for the cut-down encoder with the signal name (bitwidth in brackets), direction of the port and a brief description of each signal. A symbol for this encoder can be seen in Fig. 5.3, as generated by the synthesizer (Xilinx ISE version 9.2). Note that the bitwidth of most of these signals can be changed at synthesis time with the use of *generic mapping* in VHDL. These bitwidth would then limit the range of the signal eg. $INTLVR2_PARAM_2B(12:0)$ denotes that the parameter $2B$ would have a possible value from 0 to 8191.

A synchronous design has been used here, thus all signals must be with respect to the $SYSCLK$ signal. A $RESET$ signal is available which, when asserted, will immediately abort all encoding and reset the encoder pipeline.

This is then followed by the set of parameters required to define the code structure based on the six primary parameters of: P , T , a_1 , a_1 , a_2 and b_2 . For the QPP interleavers, as it is maximum contention free, it is possible to do parallel processing whereby different sub-blocks of a codeword can be read by multiple interleavers through a cross-bar switch to multiple DPBRAMs. Thus for QPP interleaver, it is possible to define different starting index for the parallel processing. However, in this implementation, no parallelism is done for the encoder in order to minimize resource utilization. Thus the index x starts from 0, as seen by the values for F_1 and F_2 . The same is applied to H_1 and H_2 , with the starting index $x = 0$, thus $H_1 = a_1 + b_1$ and $H_2 = a_2 + b_2$.

The next four ports from *ENCODER_DATA_EVEN_BLOCK_FILLED_STB* to *ENCODER_DATA_ODD_BLOCK_FULL* are used to interface with the *PingPong Flow Control* module for writing data into the cut-down encoder. For the strobe signals, they need only be asserted high ('1') for 1 *SYSCLK* period.

Next are the signals used for actually writing and reading data and coded bits from the encoder, using the write enables for the input and output of the encoder respectively. Lastly is the *Encoder Empty* flag used to facilitate changing of the code parameters.

Figure 5.3: Interface diagram for cut-down encoder

Table 5.2: Interface of the cut-down encoder

Interface Signals	Direction	Signal Description
SYSCLK	In	System clock signal
RESET	In	Reset signal, '1' = <i>reset</i>
ENCODER_TLEN_SUB1(3:0)	In	PA parameter: $T - 1$
INTLVR1_PARAM_2B(12:0)	In	PA parameter: $2 \times b_1$
INTLVR1_PARAM_N(12:0)	In	PA parameter: $N_1 = P \times T$
INTLVR1_START_ADDR_FX(12:0)	In	PA parameter: $F_1 = 0$
INTLVR1_START_ADDR_GX(12:0)	In	PA parameter: $H_1 = a_1 + b_1$
INTLVR2_PARAM_2B(12:0)	In	PA parameter: $2 \times b_2$
INTLVR2_PARAM_N(12:0)	In	PA parameter: $N_2 = P \times (T + 2)$
INTLVR2_START_ADDR_FX(12:0)	In	PA parameter: $F_2 = 0$
INTLVR2_START_ADDR_GX(12:0)	In	PA parameter: $H_2 = a_2 + b_2$
ENCODER_DATA_EVEN_BLOCK_FILLED_STB	In	Strobe when even DPBRAM filled
ENCODER_DATA_ODD_BLOCK_FILLED_STB	In	Strobe when odd DPBRAM filled
ENCODER_DATA_EVEN_BLOCK_FULL	Out	Flag assert when even DPBRAM full
ENCODER_DATA_ODD_BLOCK_FULL	Out	Flag assert when odd DPBRAM full
ENCODER_DATA_IN	In	Binary serial input data bits
ENCODER_ADDR_IN(12:0)	In	Address for input bits
ENCODER_DATA_WR_EN	In	Write enable for input data bits
Continued on next page		

Table 5.2 – continued from previous page

Interface Signals	Direction	Signal Description
ENCODER_FIFO_OUT_RFD	In	Output FIFO Ready for data from encoder
ENCODER_PA_CODED_OUT	Out	PA Coded bits
ENCODER_PA_CODED_FIFO_WR_EN	Out	Write enable for output PA coded bits
ENCODER_EMPTY	Out	‘1’ = Encoder empty, can change parameters

5.3.2 Cut-down Encoder Timing Diagram

Fig. 5.4 shows the timing diagram for the cut-down encoder where five blocks of data, of two different code parameters, are written in for encoding. After the 1st block of data is written in, the *ENCODER_DATA_EVEN_BLOCK_FILLED_STB* signal is strobed. Consequently the *ENCODER_DATA_EVEN_BLOCK_FULL* flag will be asserted by the *PingPong Flow Control* module. After the requisite number of clock cycles for processing, the encoder will check that the *ENCODER_FIFO_OUT_RFD* flag is asserted and write out the 1st block of PA coded bits.

At this point, the *Encoder Empty* flag will be asserted and the parameters are changed for the next three blocks of data which are of a smaller codeword length. Note the pipelined structure of the encoder, as the 3rd and 4th data blocks are written in before the 2nd PA coded block had been written out.

After the 4th PA coded block has been written out, *Encoder_EMPTY* flag will be asserted again and a parameter change back to that used for the 1st block is done. The 5th data block is then written in. Note that when writing in data to the encoder, as long as the *BLOCK_FILLED_STB* for either the even or odd block is not strobed, processing will not proceed. Thus it is possible to have gaps between the bits being

Chapter 5: Encoder and Decoder Implementation

69

written in, as seen here. For the cut-down encoder, due to the 2-stage pipelining and the use of the PingPong flow control mechanism, there can be a maximum of four codewords within the encoder pipeline, with 2 codewords stored in each DPBRAM.

5.4 Decoder Design

As PA codes should be iteratively decoded with a considerable number of iterations to maximize the coding gain, the design of the decoder is crucial to realize a high throughput, low latency and efficient decoder with minimal resource utilization while maintaining BER performance within acceptable limits. The choice of min-sum algorithm for the check nodes, as discussed in section 2.6 and QPP interleaver for Π_1 and Π_2 in section 3.5, plays a major part in ensuring that these goals can be met. Nonetheless, more can still be done for the decoder hardware design. Parallelism and pipelining are the two most common techniques in hardware design to further increase throughput and decrease latency, though at the cost of increased complexity, resource utilization and possible degradation in BER performance. These will be discussed presently.

From section 2.4, a PA code can be interpreted as having two layers of bipartite graph of variable and check nodes connected through interleaver Π_2 , with interleaver Π_1 for the parallel concatenation in the outer code. Thus full parallelism here entails simultaneous processing of all the bit/check nodes at each bit/check node layer, as seen in the code graph in Fig 2.7, according to the decoding algorithm of section 2.5. Full pipelining, on the other hand, would be the simultaneous processing of all the bit/check node layers. Note that pipelining and parallelism, as defined here, can be done independently of each other. Though with full pipelining/parallelism, some BER performance degradation may be incurred, for a similar number of decoding iteration, as nodes do not work with the latest information at each iteration as described by the decoding algorithm. As expected, maximum throughput and minimum latency would be achieved if both full parallelism and full pipelining are utilized.

QPP interleaver has the property of maximum contention free for the memory access. This essentially allows partial parallel processing of the code graph by partitioning the bit/check nodes into $M \times W$ blocks as discussed in section 3.4, where $W = 1$ results in a fully parallel design. Certainly, the more parallelism there are,

the lower the latency and the higher the throughput. Nonetheless, the amount of required resource increases significantly with more parallelism.

In this design, no parallel processing of the bit and check nodes at each sub-decoder will be done. Instead, each bit/check node will be processed serially within that bit/check node layer. This is mainly because the target throughput requirement is not very high, in the order of a few megabits per second (Mbps), in order to tradeoff for minimum resource utilization. Also, a simple way of increasing throughput would be to instantiate multiple decoders as desired in the FPGA and process several PA codewords in parallel by using a de-multiplexer and multiplexer at the input and output of the decoders respectively. Note however that this would not be able to reduce latency, unlike a full parallel decoding of each node layer.

Other than parallelism, the other critical design consideration is the pipelining of the decoder to meet high throughput and low latency requirements. As the PA code utilizes a serial concatenation, the decoding algorithm requires that the inner and outer decoders process in sequence, such that each will have the latest *a priori* information possible at each iteration, in order to maximize coding gain for a particular number of iterations. However, as both sub-decoders will be instantiated within the FPGA, such a scheme would mean that as one sub-decoder is processing, the other sub-decoder will have to be in idle mode. This obviously would not be an efficient manner of using the instantiated resources. The total latency required to process one PA codeword for such a non-pipelined scheme is:

$$T = T_{load} + Iter \times (T_{inner} + T_{outer}) + T_{unload} \quad (5.1)$$

where T_{load} and T_{unload} are the times required to load L_{ch} into the decoder and to unload the decoded bits out of the decoder, respectively. T_{inner} and T_{outer} are the time required for the inner and outer decoders to process for one iteration and $Iter$ is the number of iterations done. Throughput would then be the reciprocal of total latency as codewords are processed sequentially.

Thus pipelining for the PA decoder would explore how to ensure that most of the instantiated resources are doing useful processing work, most of the time. Generally speaking, that is to enable both the inner and outer decoders to process concurrently, without waiting for the latest input *a priori* information possible in each iteration. A direct benefit would be that latency would be reduced against that of no pipelining, for a similar number of iterations. This is because latency can then be calculated as:

$$T = MAX(T_{load}, T_{unload}) + Iter \times (MAX(T_{inner}, T_{outer})) \quad (5.2)$$

where MAX is the maximum function. The unloading and loading can also be pipelined to minimize latency, by having separate input and output buffers to do the loading and unloading concurrently. Similarly, as throughput is reciprocal of total latency, it will thus also be increased. Understandably, some BER performance drop is anticipated as sub-decoders are not working with the latest input *a priori* information possible in each iteration.

5.4.1 2-Stage Pipelining

In this section, a 2-stage pipelining of the PA decoder is investigated, with the aim of maximizing throughput and utilization of instantiated resources while minimizing latency and degradation of BER performance.

The natural and simplest pipelining arranges the decoder into two stages, which are separated by Π_2 and which instantiates respectively the inner sub-decoder and the outer sub-decoder. In a serial decoder, it allows one bit be processed per clock at each stage and, because of the decoupling effect of Π_2 , the two stages may process different bits at any unit time. An immediate consequence is that the data processed by the outer decoder and by the inner decoder are not strictly sequenced. For example, the outer decoder may be processing a bit using reliability information coming from either the current iteration or the previous iteration of the inner decoding. Likewise,

the inner decoder may sometimes process reliability information that is drawn from an older iteration of the outer decoder.

The schematic for such a 2-stage pipeline structure can be seen in Fig. 5.5, where the *Register* modules are the output of each pipeline stage.

Figure 5.5: Schematic for a 2-stage pipeline structure

To recap, the decoding algorithm at the k^{th} global iteration for a non-pipelined decoder is:

Inner Decoder

$$\begin{aligned} L_{ef}^{(k)}(y_i) &= L_o^{(k-1)}(x_i) \boxplus (L_{ch}(y_{i-1}) + L_{ef}^{(k)}(y_{i-1})), \\ L_{eb}^{(k)}(y_i) &= L_o^{(k-1)}(x_{i+1}) \boxplus (L_{ch}(y_{i+1}) + L_{eb}^{(k)}(y_{i+1})), \\ L_e^{(k)}(x_i) &= (L_{ch}(y_{i-1}) + L_{ef}^{(k)}(y_{i-1})) \boxplus (L_{ch}(y_i) + L_{eb}^{(k)}(y_i)) \end{aligned}$$

Outer Decoder

$$\begin{aligned} L_{e1}^{(k,l)}(d_{i,j}) &= \left(\sum_{1 \leq t \leq T, t \neq j} \boxplus (L_e^{(k)}(d_{i,t}) + L_{e2}^{(k,l-1)}(d_{i,t})) \right) \boxplus L_e^{(k,l)}(p_i), \\ L_{e1}^{(k,l)}(p_i) &= \left(\sum_{1 \leq t \leq T} \boxplus (L_e^{(k)}(d_{i,t}) + L_{e2}^{(k,l-1)}(d_{i,t})) \right), \\ L_o^{(k,l)}(d_{i,j}) &= L_{e1}^{(k,l)}(d_{i,j}) + L_{e2}^{(k,l)}(d_{i,j}), \\ L_o^{(k)}(x_i) &= \Pi_2 \left(L_o^{(k,L)}(d_{i,j}), L_{e1}^{(k,L)}(p_i), L_{e2}^{(k,L)}(p_i) \right) \end{aligned} \tag{5.3}$$

as described in section 2.5.1 and 2.5.2. Note that $L_{ef}^{(k)}(y_i)$ and $L_{eb}^{(k)}(y_i)$ can be processed in parallel, after which their updated information will be used to process for $L_e^{(k)}(x_i)$, within the same k^{th} iteration.

The decoding algorithm for a 2-stage pipelined decoder at the k^{th} global iteration can be described by:

Inner Decoder

$$\begin{aligned} L_{ef}^{(k)}(y_i) &= L_o^{(k-1)}(x_i) \boxplus (L_{ch}(y_{i-1}) + L_{ef}^{(k-1)}(y_{i-1})), \\ L_{eb}^{(k)}(y_i) &= L_o^{(k-1)}(x_{i+1}) \boxplus (L_{ch}(y_{i+1}) + L_{eb}^{(k-1)}(y_{i+1})), \\ L_e^{(k)}(x_i) &= (L_{ch}(y_{i-1}) + L_{ef}^{(k)}(y_{i-1})) \boxplus (L_{ch}(y_i) + L_{eb}^{(k)}(y_i)) \end{aligned}$$

Outer Decoder

$$\begin{aligned} L_{e1}^{(k,l)}(d_{i,j}) &= \left(\sum_{1 \leq t \leq T, t \neq j} \boxplus (L_e^{(k-1)}(d_{i,t}) + L_{e2}^{(k-1,l-1)}(d_{i,t})) \right) \boxplus L_e^k(p_i), \\ L_{e1}^{(k,l)}(p_i) &= \left(\sum_{1 \leq t \leq T} \boxplus (L_e^{(k-1)}(d_{i,t}) + L_{e2}^{(k-1,l-1)}(d_{i,t})) \right), \\ L_o^{(k,l)}(d_{i,j}) &= L_{e1}^{(k,l)}(d_{i,j}) + L_{e2}^{(k,l)}(d_{i,j}), \\ L_o^{(k)}(x_i) &= \Pi_2 \left(L_o^{(k,L)}(d_{i,j}), L_{e1}^{(k,L)}(p_i), L_{e2}^{(k,L)}(p_i) \right) \end{aligned} \tag{5.4}$$

where the difference from the non-pipelined decoder is in bold. $L_{ef}^{(k-1)}$ and $L_{eb}^{(k-1)}$ denotes the use of information from the previous iteration in the calculation of the forward and backward pass of the inner decoder. For the outer decoder, it does not use the latest $L_e^{(k)}$ from the inner decoder for the decoding but instead uses $L_e^{(k-1)}$ so as to allow it to run concurrently with the inner decoder.

To clarify, pipelining within the decoder can be divided into macro-pipelining and micro-pipelining, where macro-pipelining refers to pipelined processing **between** the bit/check node layers or sub-decoders while micro-pipelining is for the processing **within** each node layer or sub-decoder. Note that due to the synchronous nature of implementing in FPGA, mathematical operations are often pipelined using registers to hold the intermediate results, to ensure that the design is able to run at a high clock speed. As such, the computation of $L_{ef}^{(k)}(y_i)$ and $L_{eb}^{(k)}(y_i)$ cannot be completed within one clock cycle, which will severely impact throughput and latency due to the recursive nature of the computation. Micro-pipelining is then used here to remove

this recursive dependency. This will be described in detail in section 5.5.1.

Thus the use of $L_{ef}^{(k-1)}$ and $L_{eb}^{(k-1)}$ in the inner decoder is a micro-pipelining whereas the use of $L_e^{(k-1)}$ in the outer decoder is a macro-pipelining. The difference in macro- and micro-pipelining will be detailed in section 5.5, which will show the memory devices associated with each type of pipelining.

The 2-stage decoder pipelining described in this section refers to the macro-pipelining stages, similarly for the 3-stage decoder pipelining described in the next section. Thus strictly speaking, for a 2-stage decoder with the inner decoder and outer decoder running concurrently, there is no requirement for $L_{ef}^{(k-1)}$ and $L_{eb}^{(k-1)}$; only need is for outer decoder to use $L_e^{(k-1)}$. Nonetheless, such a micro-pipelining within the inner decoder is desirable to improve throughput and latency.

The BER performance of a 2-stage pipelined decoder, as described above, can be seen in Fig. 5.6. At a BER of 1×10^{-5} , there is only a very slight degradation of $\approx 0.1dB$ when compared to the non-pipelined decoder.

Figure 5.6: BER comparison of different pipelining schemes: no pipelining, 2-stage pipelining and 3-stage pipelining with QPP interleaver, 30 iterations, MS algorithm and floating point simulation

5.4.2 3-Stage Pipelining

With 2-stage pipelining, the outer decoder is assumed to be processed by a single stage of macro-pipelining. However, a closer inspection of the outer decoder will reveal the need for another stage of macro-pipelining at Step 3 of section 2.5.2. This is because of the two parallel branches of SPC codes in the outer code. When two SPC decoders are instantiated in parallel here, they will generate two streams of information where the lower stream is interleaved by Π_1 . There is then a need to deinterleave the lower stream before it can be combined with the upper stream as described in Step 3.

To perform de-interleaving, the whole block of LLR information for that codeword has to be written into a DPBRAM using the interleaving sequence $\mathbf{F}_1(x)$, and then be read out serially. This however creates a delay of one codeword block in the pipeline and Step 3 of the outer decoding cannot be processing all the time. A macro-pipeline stage will have to be introduced at this point to enable the decoder pipeline to be running at full utilization.

The schematic for such a 3-stage pipeline structure can be seen in Fig. 5.7, where the *Register* modules are the output of each pipeline stage.

Figure 5.7: Schematic for a 3-stage pipeline structure

Thus at the inner decoder, the input from the outer decoder $L_o^{(k-1)}$ will be delayed

by another iteration to $L_o^{(k-2)}$ as described by:

Inner Decoder

$$\begin{aligned} L_{ef}^{(k)}(y_i) &= L_o^{(k-2)}(x_i) \boxplus (L_{ch}(y_{i-1}) + L_{ef}^{(k-1)}(y_{i-1})), \\ L_{eb}^{(k)}(y_i) &= L_o^{(k-2)}(x_{i+1}) \boxplus (L_{ch}(y_{i+1}) + L_{eb}^{(k-1)}(y_{i+1})), \\ L_e^{(k)}(x_i) &= (L_{ch}(y_{i-1}) + L_{ef}^{(k)}(y_{i-1})) \boxplus (L_{ch}(y_i) + L_{eb}^{(k)}(y_i)) \end{aligned}$$

Outer Decoder

$$\begin{aligned} L_{e1}^{(k,l)}(d_{i,j}) &= \left(\sum_{1 \leq t \leq T, t \neq j} \boxplus (L_e^{(k-1)}(d_{i,t}) + L_{e2}^{(k-1,l-1)}(d_{i,t})) \right) \boxplus L_e^k(p_i), \\ L_{e1}^{(k,l)}(p_i) &= \left(\sum_{1 \leq t \leq T} \boxplus (L_e^{(k-1)}(d_{i,t}) + L_{e2}^{(k-1,l-1)}(d_{i,t})) \right), \\ L_o^{(k,l)}(d_{i,j}) &= L_{e1}^{(k-1,l)}(d_{i,j}) + L_{e2}^{(k-1,l)}(d_{i,j}), \\ L_o^{(k)}(x_i) &= \Pi_2 \left(L_o^{(k,L)}(d_{i,j}), L_{e1}^{(k,L)}(p_i), L_{e2}^{(k,L)}(p_i) \right) \end{aligned} \tag{5.5}$$

where the difference from the 2-stage pipelined decoder is in **bold**.

The astute reader may wonder about the dependency of $L_e^{(k)}(x_i)$ on $L_{ef}^{(k)}(y_{i-1})$ and $L_{eb}^{(k)}(y_i)$, and whether a stage of micro-pipelining should be introduced for $L_e^{(k)}(x_i)$ to use information from the forward and backward passes of the previous iteration, since the latter seems to allow the forward, the backward, and the outward passes to be performed concurrently. This is actually not necessary, since the micro-pipelining to $L_{ef}^{(k)}(y_i)$ and $L_{eb}^{(k)}(y_i)$ removes the recursive dependency within L_{ef} and L_{eb} themselves. In other words, it is possible to calculate $L_{ef}^{(k)}(y_i)$ or $L_{eb}^{(k)}(y_i)$ at any index y_i without dependency on y_{i-1} or y_{i+1} as these would already have been calculated in the previous global iteration $k - 1$. Consequently, the calculation of $L_{ef}^{(k)}(y_i)$ and $L_{eb}^{(k)}(y_i)$ should be coordinated such that the result can be immediately used by $L_e^{(k)}(x_i)$ without any further micro-pipelining. This will be shown in section 5.5.1. Certainly, registration of the intermediate results within the FPGA is still necessary though this will not have any impact on the recency of the information being used in the computation and thus is not considered as pipelining of the decoder.

The BER performance of this 3-stage pipelined decoder, can be seen in Fig. 5.6.

At a BER of 1×10^{-5} , there is a degradation of $\approx 0.25dB$ when compared to the non-pipelined decoder.

Although the 3-stage pipelined decoder suffers a bigger BER degradation than that of the 2-stage pipeline, the 2-stage pipeline does not allow the outer decoder to be fully pipelined, thus creating a bottleneck which will reduce the throughput and latency. As such, the 3-stage pipelined structure will be adopted for this implementation, which will be further detailed in section 5.5.

To investigate the performance loss caused by the 3-stage pipelined design, simulations using the pipelined and non-pipelined structure under 30 or 60 decoding iterations has been done. The result as seen in Fig. 5.8 shows that the performance loss resulted from the out-of-sequence processing can be well regained by increasing the number of decoding iterations. Certainly, doubling of the number of iterations seems like a high price to pay to regain the $0.25dB$ BER performance. Nonetheless, it is possible to use early termination of decoding to improve average throughput to offset these additional decoding iterations, the implementation of which is discussed in section 5.5.2.

Briefly, in this implementation early termination is decided when the parity check constraints of the interleaved and non-interleaved SPC code branches in the outer code are satisfied for a predetermined number of iterations. This will likely indicate that convergence has been achieved, thus further iterations are not necessary for hard decision outputs. From simulation, it was observed that at BER of 1×10^{-5} , i.e. high E_b/N_o regime, most codewords would have converged much earlier than the maximum 30 iterations allowed, with only a minimal number of codewords needing the full 30 iterations to converge. Certainly, these “troublesome” codewords need to be allowed to decode for the full 30 iterations, else they can become the dominant error events and can cause error flooring. However in most cases, the extra iterations goes to waste for the other codewords, thus average throughput can be greatly improved if early termination is employed.

Note that other forms of early termination are possible, such as the use of cyclic

redundancy checks (CRC) at the output of the decoder for error detection and subsequent signaling to the decoder for early termination.

Figure 5.8: BER improvement of 3-stage pipeline with 30 and 60 iterations, with MS algorithm and floating point simulation

5.5 Decoder Implementation

The complete structure of the proposed decoder is illustrated in Fig. 5.9, where the memory device at the output of the three stages of macro-pipelining are shaded and labeled $DPBRAM L_e$, $DPBRAM L_o$ *Non-interleaved/Interleaved* and $DPBRAM L_o$, respectively.

$DPBRAM L_{ef}$ and $DPBRAM L_{eb}$ corresponds to the micro-pipelining done at the inner decoder. $DPBRAM LLR$ *Non-interleaved/Interleaved* is a micro-pipelining at the SPC checker which allows for parallel SPC parity check of both the upper and lower SPC branch.

All the notations here follow those in (Li et al. 2004). Below we will highlight a few implementation details.

Figure 5.9: Detailed decoder structure.

5.5.1 Inner Decoder Implementation

The inner decoder processes information in three passes, the forward pass, the backward pass, and the outward pass to the outer decoder, through three *Check* modules. The data streams exchanged between these modules are illustrated in Table 5.3 where each row corresponds to one clock tick. It should be understood that the outputs from the *Add* module and the *Check* module are all registered, so there is one clock delay between the input and output of each module; but we did not depict that in the table for the sake of clarity. Note that these registrations are not the micro/macro pipelining. Also, the input to the *Check Lef/Leb* module comes from the combination of the *DPBRAM L_o* output and the respective *Add L_{ef}/L_{eb}* output.

In (Li et al. 2004), it was suggested that both the forward pass and the backward pass can be made parallel by using the information from the previous iteration rather than waiting for the current iteration. In our FPGA implementation, previous global iteration information is utilized not for parallelizing but rather to enable micro-pipelining within the inner decoder. For example, by the time the extrinsic information $L_{ef}(1)$ gets produced at the output of the *Check L_{ef}* module and stored in *DPBRAM L_{ef}*, it will be too late to feed it as an input to the *Add L_{ef}* module for calculating $L_{ef}(2)$ within the next clock. This is due to the latency involved in the *Check L_{ef}* output and in the writing and reading processes of *DPBRAM L_{ef}*. Hence, in our implementation, extrinsic information $L_{ef}(1)$ used in the calculation of $L_{ef}(2)$ actually comes from the previous iteration.

From the data flow, we see that the L_o input to both *Check L_{ef}/L_{eb}* modules are the same, thus only one *DPBRAM L_o* is needed. For *DPBRAM L_{ch}*, one port will be used to write in L_{ch} during the loading phase, before decoding starts. Subsequently, both ports will be used for reading during decoding, thus one *DPBRAM L_{ch}* suffices.

For the inner decoder, one Π_2 address generator is shared among all the three *Check* modules for interleaving and de-interleaving. This is possible because, as explained in section 3.4, only the interleaving pattern is needed to achieve both interleaving and de-interleaving. In practice, the generated interleaving address has to be

delayed for a few clock cycles before being used by the various *Check* modules respectively, depending on the number of registrations done along the data path. Thus the block of bits being written into *DPBRAM* L_e will already be Π_2 deinterleaved into the $P \times T$ systematic information data bits, 1st set of P parity bits belonging to the upper SPC branch and 2nd set of P parity bits belonging to the lower SPC branch.

For the *Check* modules, they are a 2-input, 1-output $\boxplus$ operation as shown in (2.1) where the min-sum algorithm has been implemented.

5.5.2 Outer Decoder Implementation

The outer decoder consists of three components, the non- Π_1 -interleaved upper SPC branch, the Π_1 -interleaved lower SPC branch, and the *Add* module which combines information from the two SPC branches to feed back to the inner decoder. An additional *Add* module is required to combine the reliability information from both the inner and the outer decoders, and to make the final hard decoding decision. These hard decided decoded bits can then be used by the *SPC Checker* module to examine whether the decoded bits satisfy the parity check constraints in both SPC branch, and if so, to flag for early termination of decoding. Because of the hard decision, the *DPBRAM LLR Interleaved/Non-Interleaved* modules in the *SPC Checker* module are configured as 1-bit wide. The *SPC Checker* module can be removed if early termination is not necessary.

Note that *DPBRAM* L_e has been triplicated, such that the duplicates will enable the two SPC branches to run concurrently, and the third will enable Step 3 of the outer decoding (which prepared the final decoder output by combining all the messages from the outer and inner decoders) to run efficiently. Similarly, *DPBRAM* L_o *Non-interleaved/Interleaved* has been duplicated to enable the parallel processing of both SPC branches. *DPBRAM LLR Interleaved/Non-Interleaved* actually contains the same LLR information (L_e) from the inner decoder, and the interleaving for Π_1 is only performed when the *SPC Checker* modules reads out data.

Table 5.3: Data stream for inner decoder at the k^{th} global iteration

Check $L_e(x_i)$	Check $L_e(x_i)$			Check $L_{ef}(y_i)$ Input	Check $L_{ef}(y_i)$ Output	Check $L_{eb}(y_{i-1})$ Input	Check $L_{eb}(y_{i-1})$ Output
	Add $L_{ef}(y_i)$	Output	Add $L_{eb}(y_{i-1})$				
Output							
$L_e(1)$	∞^*		$(L_{ch}(1) + L_{eb}(1))$	$L_o(1)$	$L_{ef}(1)$	$L_o(1)$	$L_{eb}(0)^\dagger$
$L_e(2)$	$(L_{ch}(1) + L_{ef}(1))$		$(L_{ch}(2) + L_{eb}(2))$	$L_o(2)$	$L_{ef}(2)$	$L_o(2)$	$L_{eb}(1)$
$L_e(3)$	$(L_{ch}(2) + L_{ef}(2))$		$(L_{ch}(3) + L_{eb}(3))$	$L_o(3)$	$L_{ef}(3)$	$L_o(3)$	$L_{eb}(2)$
$\cdot$	$\cdot$		$\cdot$	$\cdot$	$\cdot$	$\cdot$	$\cdot$
$\cdot$	$\cdot$		$\cdot$	$\cdot$	$\cdot$	$\cdot$	$\cdot$
$L_e(x_i)$	$(L_{ch}(y_{i-1}) + L_{ef}(y_{i-1}))$		$(L_{ch}(y_i) + L_{eb}(y_i))$	$L_o(x_i)$	$L_{ef}(y_i)$	$L_o(x_i)$	$L_{eb}(y_{i-1})$
$\cdot$	$\cdot$		$\cdot$	$\cdot$	$\cdot$	$\cdot$	$\cdot$
$\cdot$	$\cdot$		$\cdot$	$\cdot$	$\cdot$	$\cdot$	$\cdot$
$L_e(N-1)$	$(L_{ch}(N-2) + L_{ef}(N-2))$		$(L_{ch}(N-1) + L_{eb}(N-1))$	$L_o(N-1)$	$L_{ef}(N-1)$	$L_o(N-1)$	$L_{eb}(N-2)$
$L_e(N)$	$(L_{ch}(N-1) + L_{ef}(N-1))$		$(L_{ch}(N) + 0^*)$	$L_o(N)$	$L_{ef}(N)^\dagger$	$L_o(N)$	$L_{eb}(N-1)$

* boundary condition † not used

These duplications are necessary as these DPBRAMs have been configured with port A for writing in information and port B for reading it out. Thus when a second computation also requests the information in DPBRAM but cannot get in perfect synchronization with the current read out, a duplicate DPBRAM must be used.

One such example is when the data for $DPBRAM L_e$ in the upper SPC branch is read out serially, and at the same time, the data for $DPBRAM L_e$ in the lower SPC branch is also read out using the interleaving address $F_1(x)$. Since these two sequences of read addresses do not in general align with each other, sharing a single $DPBRAM L_e$ is not possible. Further, recall that the upper SPC branch operates only on the systematic bits and the 1st set of parity bits, whereas the lower SPC branch concerns only the systematic bits and the 2nd set of parity bits. Both SPC branch must be set to be idle when the unwanted information is being read out (which is rather tricky to implement), should they share a single $DPBRAM L_e$.

The introduction of a duplicate DPBRAM ensures that both SPC branches in the outer decoder can be updated simultaneously within $P \times (T + 1)$ clock cycles. Since the inner decoder takes $P \times (T + 2)$ clock cycles, $(T + 2)/(T + 1)$ local iterations may be performed within the outer decoder for each global iteration. Compared to the ideal serial update, the parallel processing of both SPC branches trades a slight performance degradation for time efficiency.

The third $DPBRAM L_e$ feeds into the *Add* module and subsequently the *SPC Checker* module, which uses the information of all the $P \times (T + 2)$ bits and is therefore inconvenient to share the work with the other two $DPBRAM L_e$ modules each containing information for only subsets of parity bits. It should also be noted that $DPBRAM L_o$ *Non-interleaved* for the upper SPC branch and $DPBRAM L_o$ *Interleaved* for the lower SPC branch should set an initial value of zero to the parity bits irrelevant to the respective branch. Such a zero-initialization simplifies the implementation, as both *Add* modules can identically read out all the $P \times (T + 2)$ bits from the two $DPBRAM L_o$ *Non-interleaved/Interleaved* modules without any special alignment of data streams.

5.6 Encoder and Decoder Resource utilization

The resource utilization for the full version encoder and the decoder is analyzed in Table 5.4. Note that as the decoder is required to store soft information bits of at least 5-bits each, its maximum block size has been limited to 2047 bits here for minimum resource utilization. Certainly, these limits can be increased, with a corresponding increase in resource utilization. The combined encoder and decoder (CODEC) takes up only 12% of the logic slices and 18% of the BRAM of the target device, Xilinx 2v3000FG676-4.

Table 5.4: PA CODEC Resource Utilization

Xilinx FPGA(2v3000fg676-4)	Encoder	Decoder
Max Block size ($P \times (T + 2)$)	8191	2047
Slices (14,336)	578 (4.03%)	1159 (8.08%)
Slice Flip Flops (28,672)	419 (1.46%)	1136 (3.96%)
4 input LUTs (28,672)	978 (3.41%)	1645 (5.74%)
18Kb BRAM (96)	5 (5.21%)	13 (13.54%)
Max Frequency (MHz)	125	125

5.7 Decoder Interface and Timing Diagram

5.7.1 Decoder Interface

Table 5.5 shows the interface ports for the cut-down decoder with the signal name, direction of the port and a brief description of each signal. A symbol for this decoder can be seen in Fig. 5.10, as generated by the synthesizer (Xilinx ISE version 9.2).

The *SYSCLK* and *RESET* signals bear a similar function to those in the encoder. A set of parameters for the PA code are stored in signals from *DECODER_TLEN* to *LO_ADDR_DELAY_ADD2*. They are used to configure the code structure based on the six primary parameters of: P , T , a_1 , b_1 , a_2 and b_2 .

CHKLEN is the number of SPC codewords consecutively meeting the parity constraints in both SPC branches that will assert the *LLR_DECODE_SUCCNFAIL* flag.

If *MAXNMATCHED_ITER* is asserted high, the decoder will be forced to iterate up to the number specified by *MAX_LE_ITER*; otherwise, the decoder may terminate earlier after having processed a minimum number of iterations specified by *MIN_LE_ITER* and meeting the check constraints of *CHKLEN*.

Signals *DECODER_PACODED* to *DECODER_NEWBLK_STARTED_STB* are for the input of channel LLR to the decoder whereas, signals *LLR_CHK_OUT_RTS* to *LLR_CHK_OUT_WR_EN* are for the output of the decoded bits.

Table 5.5: Interface of PA decoder

Interface Signals	Direction	Signal Description
SYSCLK	In	System clock signal
RESET	In	Reset signal, '1' = <i>reset</i>
DECODER_TLEN(3:0)	In	PA parameter: T
DECODER_TLEN_SUB1(3:0)	In	PA parameter: $T - 1$
PTLEN_SUB4(10:0)	In	PA parameter: $P \times T - 4$
PTLEN_SUB1(10:0)	In	PA parameter: $P \times T - 1$
PTLEN(10:0)	In	PA parameter: $P \times T$
PT1LEN_SUB1(10:0)	In	PA parameter: $P \times (T + 1) - 1$
PT1LEN(10:0)	In	PA parameter: $P \times (T + 1)$
PT2LEN_SUB1(10:0)	In	PA parameter: $P \times (T + 2) - 1$
PT2LEN_SUB2(10:0)	In	PA parameter: $P \times (T + 2) - 2$
INTLVR2.PARAM_2B(10:0)	In	PA parameter: $2 \times b_2$
INTLVR2.PARAM_N(10:0)	In	PA parameter: $N_2 = P \times (T + 2)$
INTLVR2.START_ADDR_FX(10:0)	In	PA parameter: $F_2 = 0$
INTLVR2.START_ADDR_GX(10:0)	In	PA parameter: $H_2 = a_2 + b_2$
INTLVR1.PARAM_2B(10:0))	In	PA parameter: $2 \times b_1$
Continued on next page		

Table 5.5 – continued from previous page

Interface Signals	Direction	Signal Description
INTLVR1.PARAM_N(10:0)	In	PA parameter: $N_1 = P \times T$
INTLVR1.START_ADDR_FX(10:0)	In	PA parameter: $F_1 = 0$
INTLVR1.START_ADDR_GX(10:0)	In	PA parameter: $H_1 = a_1 + b_1$
LO_ADDR_DELAY(4:0)	In	PA parameter: $7 + T$
LO_ADDR_DELAY_ADD2(4:0)	In	PA parameter: $7 + T + 2$
CHKLEN(11:0)	In	Number of SPC codewords checked in outer code. Used for LLR_DECODE_SUCCNFAIL
MAXNMATCHED_ITER	In	'1' = decode max iteration, '0' = decode till consecutive <i>CHKLEN</i> number of SPC codewords satisfied parity constraint in outer code
MIN_LE_ITER(5:0)	In	Minimum number of iterations
MAX_LE_ITER(5:0)	In	Maximum number of iterations
DECODER_PACODED(6:0)	In	Decoder input soft bits, channel LLR
DECODER_PACODED_FIFO_RD_EN	Out	Read enable to FIFO holding channel LLR
DECODER_PACODED_FIFO_ND	In	'1' = new codeword available for decoding
DECODER_NEWBLK_STARTED_STB	Out	Strobe to signal decoding started, parameters can be changed for new codeword
LLR_CHK_OUT_RTS	Out	Ready to send decoded bits
LLR_CHK_OUT_RFD	In	Ready for FIFO to receive decoded bits
Continued on next page		

Table 5.5 – continued from previous page

Interface Signals	Direction	Signal Description
LLR_CHK_OUT	Out	Hard decisioned decoded bits
LLR_DECODE_SUCCNFAIL	Out	‘1’ = successful decoding, ‘0’ = failure, some SPC codewords in outer code does not satisfy parity constraint
LLR_DECODED_ITER(5:0)	Out	Number of iterations done
LLR_CHK_OUT_WR_EN	Out	Write enable for decoded bits

Figure 5.10: Interface diagram for decoder

5.7.2 Decoder Timing Diagram

Fig. 5.11 shows the timing diagram for the input to the PA decoder. To minimize latency, unloading the decoded bits from the previous codeword and loading the channel LLR of the current codeword will be processed in overlapped time.

At the input to the decoder, *DECODER_PACODED_FIFO_ND* will be asserted high when a new codeword is available. When the decoder is free, it will then proceed to latch in the set of PA parameters for this new codeword; then read the channel LLR, *DECODER_PACODED*, from the input FIFO using the *DECODER_PACODED_FIFO_RD_EN* signal. A two clock read latency to the FIFO is assumed here. After the codeword has been read in, it will strobe *DECODER_NEWBLK_STARTED_STB* to signify that decoding has started and a new set of parameters can be presented at the decoder input port for the next codeword. The decoder will then wait for two iterations to allow the LLR messages to propagate fully to the outer decoder. Next, it proceeds to the decoding process, and waits till *MIN_LE_ITER* of iterations is reached before starting check of SPC codewords. The minimum number of iterations for *MIN_LE_ITER* is set to two.

Fig. 5.12 shows a zoomed out view of the complete loading, decoding and unloading process for one codeword. Due to the good channel condition, i.e. high E_b/N_o , the checking constraints are satisfied by the 7th iteration, at which point the *LLR_DECODE_SUCCNFAIL* flag will be asserted. Note that this flag is asserted irrespective of *MAXNMATCHED_ITER*.

At the decoder output, when processing of a codeword has met the termination conditions, *LLR_CHK_OUT_RTS* will be asserted high. When *LLR_CHK_OUT_RFD* from the external interface is asserted, the hard decoded bits *LLR_CHK_OUT* will be sent out together with the write enable signal *LLR_CHK_OUT_WR_EN*. The success or failure condition based on consecutive *CHKLEN* SPC codewords meeting parity constraints is reflected by the *LLR_DECODE_SUCCNFAIL* flag. The number of iterations done is also shown by *LLR_DECODED_ITER* which will be updated when unloading starts.

Figure 5.11: Interface timing diagram for decoder input

Figure 5.12: Interface timing diagram for decoder output

Chapter 6

FPGA Test Results

6.1 Test Setup

To test the BER performance of the implemented PA code, a hardware-in-the-loop test with AWGN noise from a Xilinx AWGN IPcore has been conducted. The test is conducted by implementing the design into an XtremeDSP development kit from Nallatech. Baseband binary phase shift keying (BPSK) modulation is assumed with perfect synchronization. Below, the block diagrams of the test setup will be reviewed, showing the hierarchy of modules building up to the test setup itself.

6.1.1 Decoder Core

Fig. 6.1 shows the block diagram of the *Decoder Core* module which is made up of the inner and outer decoder as detailed in Fig. 5.9. This decoder core is the main processing engine of the decoder and with a suitable external interface, is actually the cut-down version of the decoder, similar to the cut-down version of the encoder in section 5.2. In this case, the optional *SPC Checker* module for early termination is outside this level of the hierarchy.

Figure 6.1: Block diagram of the decoder core consisting of the inner and outer decoder

6.1.2 Decoder

Going up a level, Fig. 6.2 shows the PA decoder with the *SPC Checker* module which consists of the *DPBRAM LLR Interleaved/Non-Interleaved* modules and the *LLR Checker* module. Additionally, a *Decoder Controller* module is necessary to coordinate the various modules within the decoder, by triggering these modules during the loading, decoding and unloading phases of each codeword.

After loading and starting a codeword decoding, the *Decoder Controller* module will wait till the specified minimum number of iterations has been processed; then it will take the *LLR_DECODE_SUCCNFAIL* flag together with the *MAXN-MATCHED_ITER* signal and the current iteration number, to decide whether decoding should terminate. Once decoding is terminated, it will trigger the *SPC Checker* module to output the decoded bits, initialize the *Decoder Core* module for the new

codeword, latch in the new set of PA parameters and trigger the L_{ch} module to read in a new codeword for decoding. The interface and timing diagram to this level of the hierarchy corresponds to that detailed in section 5.7. This level of the hierarchy will allow the minimum amount of external interface required to utilize the PA decoding function.

Figure 6.2: Block diagram of the decoder consisting of the decoder core, the decoder controller and the SPC checker module (LLR BRAM Non-interleaved/Interleaved and LLR Checker)

6.1.3 BER Test Setup

Finally, at the topmost level is the actual hardware-in-the-loop test setup shown in Fig. 6.3. Here, data source comes from the PRBS generator inside the full version of the encoder. The binary PA coded bits generated will then be mapped as follows:

$$\begin{aligned}
 '0' &\Rightarrow +1 \Rightarrow "0000011111111111" \\
 '1' &\Rightarrow -1 \Rightarrow "1111100000000001"
 \end{aligned} \tag{6.1}$$

where bit ‘0’ is mapped under baseband BPSK to a value of +1 corresponding to the coded bit energy E_c of one, and is represented in fixed point, 16-bit signed 2’s complement format of 4Q11 (1 sign bit, 4 integer bit and 11 fraction bits) by “0000011111111111”. This is similarly done for bit ‘1’. The actual energy for each information bit is by $E_b = E_c/r$, where r is the code rate.

Our data format is consistent to the output format of the Xilinx AWGN IPcore version 1.2, which similarly outputs a 16-bit 4Q11 fixed point number. This core permits users to specify the signal-to-noise ratio (SNR) for the noise output assuming a sample energy E_s of one. Note that $E_s = E_c \cdot m$ where m is the modulation order and $m = 1$ for BPSK. Thus the SNR value into the AWGN core need only take into account the code rate r during calculation of the actual E_b/N_o . For example, consider a baseband BPSK operated at an E_b/N_o of 3dB and a code rate of 0.75 (which amounts to a code rate penalty of $10\log 0.75 = -1.25dB$). The proper SNR we should specify for the AWGN core is, $SNR = E_s/N_o = (E_c \cdot m)/N_o = (E_b \cdot r \cdot m)/N_o = 3 - 1.25 = 1.75dB$.

The two 16-bit numbers from the PA encoder and the AWGN core will be summed together to form a 17-bit 5Q11 number. From Hu et al. (2004), it has been shown that using five bits to represent the channel LLR is sufficient, thus there is a need to do clipping and flooring before sending it to the decoder. In this case, the *Clip and Floor* module will clip and floor the 5Q11 number to a 1Q3 number (1 sign, 1 integer, 3 fraction bits). The purpose of this one “integer” bit is to avoid the unnecessary clipping of the noisy samples at low E_b/N_o , where the instantaneous energy of the noisy sample may frequently exceed one.

As the encoder outputs the PA coded bits serially, there is a need to buffer the noisy samples at the FIFO just before the *PA Decoder* module for at least one codeword block of size N . The FIFO outputs a FIFO level signal which is used by the *PA Decoder* module to trigger reading in of a new codeword block for decoding.

At the output of the *PA Decoder* module, the decoded bits will be streamed to the *BER Tester* module, a Rohde & Schwarz SMIQ03B signal generator, together with

a clock signal. This is necessary as the *BER Tester* module only takes in a source synchronous signal. Within the *BER Tester* module, it provides a PRBS reference data stream using coefficients identical to that used in the *PRBS Generator* of the *PA Encoder* module. Note that for such PRBS BER testing, the tester needs only to know the length and coefficients of the PRBS sequence to setup the LFSR. Subsequently, when the first 21 bits of the data bits are received, it will be used as the seed for the shift registers in the LFSR to begin generating the PRBS for all the subsequent bits to do the bit comparison and BER calculation. As such, the assumption here is that the 1st 21 bits should be error free. At BER range below 1×10^{-2} , this should be easily satisfiable.

In case there are errors in the seed bits, the BER calculated should then be around the region of 0.5 as the reference PRBS stream would not be correct. The *BER Tester* module will actually detect when the BER is too high (≥ 0.1) and reseed the LFSR accordingly to restart the BER calculation.

Figure 6.3: Block diagram of the BER test setup consisting of the PA encoder, decoder, AWGN IPcore and BER tester

6.2 BER Results

Fig. 6.4 demonstrates the performance of the implemented FPGA decoder based on the MS algorithm, 3-stage macro-pipelining, QPP interleavers and fixed-point representation. With 5 bits used for the channel LLR representation and 6 bits for extrinsic LLR in the decoder, there is only a negligible degradation ($< 0.1dB$) in BER performance, compared to floating point simulation. In comparison, the “performance-oriented” decoder discussed in (Li et al. 2004) uses the SP algorithm, no pipelining, coprime interleavers and floating point representation, yet performs only about $0.4dB$ better at BER 1×10^{-5} . Both decoders were fixed at 30 iterations.

The simulated results and the experimental results of the 3-stage macro-pipelined FPGA decoder using QPP interleaver, MS algorithm, fixed-point representation, 30 iterations, are also compared. The BER curves agree well except for the low BER region where the implemented decoder performs slightly better. This is attributed to the fact that the simulation assumes that each stage uses all the information from the previous iteration, whereas the implemented decoder actually uses some current iteration information.

Figure 6.4: BER performance comparison of hardware-in-the-loop vs simulation

6.3 Benchmarking of Decoder Resource Utilization

In terms of throughput and resource utilization, the PA decoder takes up much more resources than the encoder at a similar throughput ($slices \geq 30\times$, $BRAM \geq 39\times$), thus the focus of the benchmarking will be on the PA decoder.

Table 6.1 shows the resource utilization of the PA decoder implemented in an Xilinx Virtex II FPGA (2v3000fg676-4) compared against that of an LDPC-CC implementation (Bates and Block 2005) in an Altera Stratix FPGA (EP1S80). For a fair comparison, both are set to a code rate of 0.5 with BER 1×10^{-5} at E_b/N_o of 3.1dB (by adjusting the number of decoding iterations), and two Altera Logic Elements (LE) are assumed equivalent to one Xilinx Slice. $P = 500$ and $T = 2$ are used for PA code. To get the same data rate of 40Mbps, six PA decoders need to be instantiated while 10 LDPC-CC processors are needed. Here the *SPC Checker* module is removed and thus the PA decoder will always complete 18 decoding iterations with each PA decoder using 11 BRAM. Compared to LDPC-CC, slice utilization is lower for PA codes but BRAM utilization is higher. Thus in a Xilinx FPGA, the utilization ratio of Slices to BRAM is more balanced for PA codes than for LDPC-CC. Hence it is possible to pack more PA decoders into a Xilinx FPGA chip to fully harness the available resources.

Table 6.1: PA CODEC vs LDPC-CC in Xilinx FPGA(2v3000fg676-4)

	PA	LDPC-CC
Code Rate	0.5	0.5
$\frac{Eb}{No}$	3.1dB	3.1dB
BER	1×10^{-5}	1×10^{-5}
Clock Rate	120MHz	40, 120MHz
Iteration	18	10
No. of Processors	6	10
Throughput	40Mbps	40Mbps
Slices (14,336)	6,954 (48.51%)	11,940 (83.29%)
BRAM (96)	66 (68.75%)	20 (20.83%)

Chapter 7

Conclusion and Recommendations

7.1 Conclusion

In this thesis, we have successfully designed and implemented a PA code, in VHDL targeting FPGA devices, that achieves a high throughput with an efficient implementation.

We have investigated several practical design trade-offs for implementation considerations. The originally suggested coprime interleaver was found to be not suitable for implementation in FPGA, thus several different interleavers were evaluated to select the ones that are efficient to implement in FPGA, easy to reconfigure and provide good BER performance. QPP interleaver was found to be the best choice in terms of implementation cost, reconfigurability and BER performance. Relatively prime and prime factor based multi-dimensional helical interleavers were also investigated but their BER performance were found to be inferior to that of the QPP interleaver. EXIT analysis was done to show that divergence of the EXIT trajectories was the main cause of the poor BER performance of these interleavers.

The min-sum algorithm was also chosen for the decoder over the sum-product algorithm due to simpler implementation at acceptable BER performance loss.

An efficient 3-stage macro-pipelined FPGA implementation of PA decoder and a 4-stage macro-pipelined encoder has been developed. The CODEC is fully re-

configurable, economic in resource consumption, and highly efficient in utilizing the resources instantiated. Implemented in a Xilinx Virtex II FPGA device running at 125MHz, the encoder offers a throughput of 62.5Mbps and the decoder offers a throughput of 4Mbps at 30 decoding iterations. A cut-down version of the encoder with only 2-stage macro-pipelining is also discussed where some interfacing modules were removed at the cost of added complexity to the external interface. Another cut-down version of the decoder that eliminates the *SPC Checker* module is also available. This module facilitates early termination of the decoder for potentially higher average throughput, which may not be necessary in some applications.

Hardware-in-the-loop BER test has been performed on a Nallatech Xtreme DSP development kit to validate the design. Performance measurement reveals a gap of only 0.4dB at BER 1×10^{-5} from the “ideal” PA decoder discussed in (Li et al. 2004) that targets performance only.

7.2 Recommendations

Some suggestions for future work are given below:

1. As mentioned in section 3.4, QPP interleaver has the maximum contention free property which allows parallelism in memory access. This can be done together with the pipelining already implemented to increase throughput at the decoder; and is especially beneficial for codeword block size greater than 2K bits. Currently, at 8-bit quantization, a DPBRAM module using 1 Xilinx BRAM primitive can only hold a codeword of maximum length 2K bits. Codeword lengths larger than 2K would require more BRAM primitives to be concatenated together to form a larger DPBRAM. For example, a codeword of length 4K bits requires the concatenation of two BRAM primitives to form a 4K DPBRAM module. With parallelism of two, the throughput could be doubled with this doubling of BRAM primitives by instantiating two DPBRAM to store one half

of the codeword each. Certainly, slice utilization need to be increased in order to support this doubling of throughput.

2. From the graph perspective, a PA code can be viewed as a 2-layer LDPC code. Currently, the outer and inner code are separately treated as two regular LDPC codes whose data nodes and parity nodes assume their respective uniform degrees. Irregular LDPC codes have been shown to provide better BER performance (Richardson et al. 2001), thus some irregularity in the PA code graph could improve BER performance. To maintain the rate-1 inner code, the irregularity should probably be introduced in the outer code. Interleavers suited for FPGA implementation while providing good BER performance have to be carefully chosen. As such, any irregularity should try to make use of the QPP interleaver, which had been proved to provide good BER performance for the regular PA code, or to select only interleavers which can be efficiently implemented in FPGA.
3. In section 2.6, it was observed that scaling of the output LLR at the bit and check node of the decoder is able to improve BER performance even for the SP algorithm running in a floating point simulation. Note that message passing algorithm is non-optimal as compared to maximum-likelihood decoding, especially for PA codes which have loops (not cycle-free) in the factor graph of the outer code. Further analysis is recommended to aid in extracting even more performance out of the message passing algorithm. Also, some possible causes for this phenomenon may be the finite word length of the codeword and a saturation value of 30 at the bit and check nodes in the decoder (with the information bit energy normalized to 1).

Author's Publications

1. T.A. Koh and Y.L. Guan and B.C. Ng and Tiffany J. Li, "*Reconfigurable FPGA implementation of product accumulate codes*", IEEE Workshop on Signal Processing Systems (SiPS'07), Shanghai, China, pp. 249-254.
2. T.A. Koh and Y.L. Guan and B.C. Ng and Tiffany J. Li, "*Reconfigurable FPGA implementation of product accumulate codes*", Milcom 2007, Florida, USA.
3. T.A. Koh and Y.L. Guan and B.C. Ng and Tiffany J. Li, "*FPGA Implementation of A Reconfigurable Serial Pipelined Product Accumulate Code*", Submitted to IEEE Transactions on Circuits and Systems I.

Bibliography

- Bahl, L. R., Cocke, J., Jelinek, F., Raviv, J., Mar. 1974. Optimal decoding of linear codes for minimizing symbol error rate. *IEEE Trans. Inf. Theory* 20, 284–287.
- Bates, S., Block, G., May 2005. A memory-based architecture for FPGA implementations of low-density parity-check convolutional decoders. In: *Proc. IEEE International Symposium on Circuits and Systems*, vol. 1. pp. 336–339.
- Benedetto, S., Divsalar, D., Montorsi, G., Pollara, F., Aug. 1996. Serial concatenation of interleaved codes: Performance analysis, design and iterative decoding. TDA Progress Report 42-126, Jet Propulsion Laboratory.
- Benedetto, S., Montorsi, G., Mar. 1996. Unveiling turbo codes: some results on parallel concatenated coding schemes. *IEEE Trans. Inf. Theory* 42, 409–428.
- Berrou, C., Glavieux, A., Oct. 1996. Near optimum error correcting coding and decoding: turbo-codes. *IEEE Trans. Commun.* 44, 1261–1271.
- Berrou, C., Glavieux, A., Thitimasjshima, P., May 1993. Near shannon limit error-correcting coding and decoding: turbo codes (1). In: *Proc. IEEE Intl. Conf. Commun.* Geneva, Switzerland, pp. 1064–1070.
- Bose, B. C., Ray-Chaudhuri, D. K., Mar. 1960. On a class of error correcting binary group codes. *Information and Control* 3, 68–79.
- Caire, G., Taricco, C., Battail, G., 1994. Weight distribution and performance of the iterated product of single-parity-check codes. In: *Proc. IEEE GLOBECOM*. pp. 206–211.

- Chen, J., Dholakia, A., Eleftheriou, E., Fossorier, M. P. C., Hu, X.-Y., Aug. 2005. Reduced-complexity decoding of LDPC codes. *IEEE Trans. Commun.* 53 (8), 1288–1299.
- Cheng, M., Nakashima, M., Hamkins, J., Moision, B., Barsoum, M., Apr. 2005. A decoder architecture for high-speed free-space laser communications. In: *Proc. of SPIE*, vol. 5712. pp. 174–185.
- Chi, D. T., Oct. 1992. A new helical interleaver. In: *Proc. MILCOM '92*, vol. 2. pp. 799–804.
- Costello, D. J., Forney, G. D., Jun 2007. Channel coding: The road to channel capacity. *Proceedings of the IEEE* 95 (6), 1150–1177.
- Crozier, S., Lodge, J., Guinand, P., Hunt, A., Jun. 1999. Performance of turbo-codes with relative prime and golden interleaving strategies. In: *Proc. 6th Int. Mobile Satellite Conf. Ottawa*, pp. 268–274.
- Divsalar, D., Jin, H., McEliece, R. J., Sep. 1998. Coding theorems for 'turbo-like' codes. In: *Proc. Allerton Conf. Commun. and Control*. pp. 201–210.
- Divsalar, D., Pollara, F., Sep. 1997. Serial and hybrid concatenation codes with applications. In: *Proc. Intl. Symp. On Turbo Codes and Related Topics*. Brest, France, pp. 80–87.
- Eleftheriou, E., Mittelholzer, T., Dholakia, A., Jan. 2001. Reduced-complexity decoding algorithm for low-density parity-check codes. *IEE Electronics Letters* 37 (2), 102–104.
- Elias, P., Sep. 1954. Error-free coding. *IRE Trans. Inform. Theory* 4, 29–37.
- Forney, G. D., 1966. *Concatenated Codes*. MIT Press, Cambridge, MA.
- Gallager, R. G., 1963. *Low-density parity-check codes*. MIT Press, Cambridge, MA.

- Golay, M. J. E., Jun 1949. Notes on digital coding. *Proceedings of IRE* 37, 657.
- Hagenauer, J., Hoeher, P., Apr. 1989. Viterbi algorithm with soft-decision outputs and its applications. *IEEE Trans. Inf. Theory* 13, 260–269.
- Hagenauer, J., Offer, E., Papke, L., Mar. 1996. Iterative decoding of binary block and convolutional codes. *IEEE Trans. Inf. Theory* 42, 429–445.
- Hamming, R. W., 1950. Error detecting and correction codes. *Bell Systems Technical Journal* 29, 147–160.
- Hocquenghem, A., 1959. Codes correcteurs d'erreurs. *Chiffres* 2, 147–160.
- Hu, R., Li (Tiffany), J., Kurtas, E., 2004. Quantization and quantization sensitivity of soft-output product codes for fast-speed applications. In: *Proc. Sarnoff Symposium on advances in wired and wireless communications*. pp. 249–254.
- Imai, H., Hirakawa, S., May 1977. A new multilevel coding method using error-correcting codes. *IEEE Trans. Inf. Theory* 23 (3), 371–377.
- Jin, H., Khandekar, A., McEliece, R. J., Sep. 2000. Irregular repeat-accumulate codes. In: *Proc. 2nd Int. Symp. Turbo Codes and Related Topics*. Brest, France, pp. 1–8.
- Kschischang, F. R., Frey, B. J., Loeliger, H.-A., Feb. 2001. Factor graphs and the sum-product algorithm. *IEEE Trans. Inf. Theory* 47 (2), 498–519.
- Li, J., Narayanan, K. R., Georgiades, C. N., Jun. 2001. A class of linear-complexity, soft-decodable, high-rate, 'good' codes: construction, properties and performance. In: *Proc. Intl. Symp. Inform. Theory*. Washington DC, p. 122.
- Li, J., Narayanan, K. R., Georgiades, C. N., Jan. 2004. Product accumulate codes: A class of codes with near-capacity performance and low decoding complexity. *IEEE Trans. Inf. Theory* 50 (1), 31–46.

- Luby, M. G., Mitzenmacher, M., Shokrollahi, M. A., Spielman, D. A., 1998. Analysis of low density codes and improved designs using irregular graphs. In: Proc. ACM Symposium. pp. 249–258.
- MacKay, D. J. C., Mar. 1999. Good error-correcting codes based on very sparse matrices. *IEEE Trans. Inf. Theory* 45 (2), 399–431.
- Muller, D. E., Sep 1954. Application of boolean algebra to switching circuit design and to error detection. *IRE Trans. Electron. Comput* 3, 612.
- Narayanan, K. R., Stuber, G. L., Jul. 1999. A serial concatenation approach to iterative demodulation and decoding. *IEEE Trans. Commun.* 47, 956–961.
- Peleg, M., Sasson, I., Shamai (Shitz), S., Elia, A., Nov. 1999. On interleaved, differentially encoded convolutional codes. *IEEE Trans. Inf. Theory* 45, 2572–2582.
- Reed, I. S., Sep 1954. A class of multiple-error correcting codes and the decoding scheme. *IRE Trans. Inform. Theory* 4, 3849.
- Reed, I. S., Solomon, G., 1960. Polynomial codes over certain finite fields. *SIAM Journal of Applied Mathematics* 8, 300–304.
- Richardson, T. J., Shokrollahi, M. A., Urbanke, R., Feb. 2001. Design of capacity approaching irregular low-density parity-check codes. *IEEE Trans. Inf. Theory* 47, 619–637.
- Richardson, T. J., Urbanke, R., Feb. 2001. The capacity of low-density parity check codes under message-passing decoding. *IEEE Trans. Inf. Theory* 47, 599–618.
- Ryu, J., Takeshita, O. Y., Mar. 2006. On quadratic inverses for quadratic permutation polynomials over integer rings. *IEEE Trans. Inf. Theory* 52 (3), 1254–1260.
- Schlegel, C. B., Perez, L. C., 2004. *Trellis and Turbo Coding*, 1st Edition. John Wiley & Sons, Inc.

- Shannon, C. E., 1948. A mathematical theory of communication. *Bell Systems Technical Journal* 27, 379–423.
- Sun, J., Takeshita, O. Y., Jan. 2005. Interleavers for turbo codes using permutation polynomials over integer rings. *IEEE Trans. Inf. Theory* 51 (1), 101–119.
- Takeshita, O. Y., May 2006. On maximum contention-free interleavers and permutation polynomials over integer rings. Correspondence to the *IEEE Trans. Inf. Theory*, The Ohio State University.
- ten Brink, S., Oct. 2001. Convergence behavior of iteratively decoded parallel concatenated codes. *IEEE Trans. Commun.* 49 (10), 1727–1737.
- Ungerboeck, G., Jan. 1982. Channel coding with multilevel/phase signals. *IEEE Trans. Inf. Theory* 28 (1), 55–67.
- Xie, K., Wang, W., Li (Tiffany), J., Apr. 2006. On the analysis and design of good algebraic interleavers. In: *Proc. 4th Int. Symp. Turbo Codes and Related Topics (Turbo)*. Munich, Germany.