FPGA IMPLEMENTATION OF JOINTLY OPERATING CHANNEL ESTIMATOR AND PARALLELIZED DECODER

A THESIS SUBMITTED TO
THE GRADUATE SCHOOL OF NATURAL AND APPLIED SCIENCES
OF
MIDDLE EAST TECHNICAL UNIVERSITY

BY

ÇAĞLAR KILCIOĞLU

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR
THE DEGREE OF MASTER OF SCIENCE
IN
ELECTRICAL AND ELECTRONICS ENGINEERING

SEPTEMBER 2009

Approval of the thesis:

FPGA IMPLEMENTATION OF JOINTLY OPERATING CHANNEL ESTIMATOR AND PARALLELIZED DECODER

submitted by **ÇAĞLAR KILCIOĞLU** in partial fulfillment of the requirements for the degree of **Master of Science in Electrical and Electronics Engineering Department, Middle East Technical University** by,

Prof. Dr. Canan Özgen
Dean, Graduate School of **Natural and Applied Sciences** _____

Prof. Dr. İsmet Erkmen
Head of Department, **Electrical and Electronics Engineering** _____

Assoc. Prof. Dr. Ali Özgür Yılmaz
Supervisor, **Electrical and Electronics Engineering Dept., METU** _____

Examining Committee Members:

Prof. Dr. Yalçın Tanık
Electrical and Electronics Engineering Dept., METU _____

Assoc. Prof. Dr. Ali Özgür Yılmaz
Electrical and Electronics Engineering Dept., METU _____

Assoc. Prof. Dr. Melek Diker Yücel
Electrical and Electronics Engineering Dept., METU _____

Assist. Prof. Dr. Çağatay Candan
Electrical and Electronics Engineering Dept., METU _____

Ayşe Öznur Gürtunca, M.S.
Lead Design Engineer, ASELSAN _____

Date: _____

I hereby declare that all information in this document has been obtained and presented in accordance with academic rules and ethical conduct. I also declare that, as required by these rules and conduct, I have fully cited and referenced all material and results that are not original to this work.

Name, Last Name: ÇAĞLAR KILCIOĞLU

Signature :

ABSTRACT

FPGA IMPLEMENTATION OF JOINTLY OPERATING CHANNEL ESTIMATOR AND PARALLELIZED DECODER

Kılcıoğlu, Çağlar

M.S., Department of Electrical and Electronics Engineering

Supervisor : Assoc. Prof. Dr. Ali Özgür Yılmaz

September 2009, 83 pages

In this thesis, implementation details of a joint channel estimator and parallelized decoder structure on an FPGA-based platform is considered. Turbo decoders are used for the decoding process in this structure. However, turbo decoders introduce large decoding latencies since they operate in an iterative manner. To overcome that problem, parallelization is applied to the turbo codes and the resulting *parallel decodable turbo code* (PDTC) structure is employed for coding. The performance of a PDTC decoder and parameters affecting its performance is given on an *additive white Gaussian noise* (AWGN) channel. These results are compared with the results of a parallel study which employs a different architecture in implementing the PDTC decoder. In the fading channel case, a pilot symbol assisted estimation method is employed for the channel estimation process. In this method, the channel coefficients are estimated by a 2-way LMS (least mean-squares) algorithm. The difficulties in the implementation of this joint structure in a fixed-point arithmetic and the solutions to overcome these difficulties are described in details. The proposed joint structure is tested with varying design parameters over a Rayleigh fading channel. The overall decoding latencies and allowed data rates are calculated after obtaining a reasonable performance from the design.

Keywords: iterative decoding, BCJR, parallel processing, channel estimation, FPGA

ÖZ

BİRLEŞİK İŞLEYEN KANAL KESTİRİCİ VE PARALELLEŞTİRİLMİŞ KOD ÇÖZÜCÜNÜN FPGA ÜZERİNDE GERÇEKLENMESİ

Kılcıoğlu, Çağlar
Yüksek Lisans, Elektrik ve Elektronik Mühendisliği Bölümü
Tez Yöneticisi : Doç. Dr. Ali Özgür Yılmaz

Eylül 2009, 83 sayfa

Bu tez çalışmasında birleştirilmiş kanal kestirici ve paralelleştirilmiş kod çözücü yapısnın yerinde programlanabilir geçit dizisi (FPGA) tabanlı bir değerlendirme platformu üzerinde gerçeklenmesi ele alınmıştır. Yapıdaki kod çözme işlemini gerçekleştirmek üzere turbo kod çözücüler kullanılmıştır. Fakat, turbo kod çözücüler oldukça yüksek gecikme sürelerine sahiptir. Bu problemi ortadan kaldırmak amacıyla paralelleştirme fikri turbo kodlara uygulanmış, ortaya çıkan paralel çözümlenebilir turbo kod (PDTC) yapısı sistemdeki kodlama ve kod çözümleme işlemleri için kullanılmıştır. Toplanır beyaz Gauss gürültüsü (AWGN) altında PDTC kod çözücünün başarımı ve bu performansa etki eden tasarım parametreleri incelenmiştir. Elde edilen sonuçlar bu çalışmaya paralel başka bir çalışmada farklı bir mimari kullanılarak gerçeklenen kod çözücünün verdiği sonuçlarla karşılaştırılmıştır. Kanal kestirimi için pilot sembol destekli kanal kestirim yöntemi kullanılmıştır. Bu yöntemde, kanal katsayıları çift yönlü LMS algoritması kullanılarak kestirilmektedir. Bu birleşik yapının sabit noktalı aritmetik altında gerçeklenmesi sırasında yaşanan zorluklardan ve bu zorlukları aşmak için uygulanan çözümlerden bahsedilmiştir. Önerilen birleşik yapı, çeşitli tasarım parametreleri ile Rayleigh sönümlü kanal altında sınanmıştır. Tasarımdan makul bir başarım elde

edildikten sonra gecikme süreleri ve veri hızları hesaplanmıştır.

Anahtar Kelimeler: yinelemeli çözüm, BCJR, paralel işleme, kanal kestirimi, FPGA

To my family...

ACKNOWLEDGMENTS

I would like to express my sincere gratitude and appreciation to Assoc. Prof. Dr. Ali Özgür Yılmaz for his guidance, encouragement, and support throughout my thesis work. I have benefited from his deep knowledge and discipline on research.

I would also like to convey thanks to jury members for their valuable comments on this thesis.

I am deeply grateful to my family for their love and support. Without them, this work could not have been completed.

I am deeply indebted to my friend Enes Erdin, whose help and stimulating suggestions helped me in all the time of research. We had great time while working together in the telecommunications laboratory while writing a paper to be submitted to IET.

I am thankful to my company ASELSAN Inc. for letting and supporting of my thesis study.

I also thank the Scientific and Technological Research Council of Turkey (TÜBİTAK) for providing the financial means throughout this study.

My special thanks go to my brothers Mustafa Yüksel, Mustafa Gökçe Baydoğan, and Çağlar Ata for their help, support, and cheerful presence through the course of this study. Thanks for giving me a shoulder to lean on whenever I need.

Finally, I am grateful to my fiance Özge Kırmızı for her love, continued motivating support and welcomed presence. Without her love, support, and continuous encouragement I would never have been able to complete the work presented in this thesis.

TABLE OF CONTENTS

LIST OF TABLES

TABLES

LIST OF FIGURES

FIGURES

CHAPTER 1

INTRODUCTION

In wireless communications, channel coding has an important role on enhancing the communication reliability and quality of service. That role was first indicated in Shannon's paper in 1948 [1]. In his paper, Shannon stated that reliable data transmission over a communication channel at any rate lower than the channel capacity was possible if appropriate error correction codes are used. As a result of wireless communication being pervaded, studies on the channel coding subject has increased. Turbo codes were first introduced by Berrou et. al. [2] in 1993. These codes are also called *parallel concatenated convolutional codes* (PCCC) due to their architecture. Among all the available codes of its time, turbo codes approached the Shannon limit the most. This great performance of turbo codes lies on the success of combined interleaving and soft-decision decoding [3].

Although turbo codes show very good performance at low SNR values, their large decoding latency due to their iterative decoding algorithms is a serious problem in high-speed communication systems, e.g., satellite communication systems. Construction of more efficient hardware structures is a solution to the decoding latency problem. We focus on that solution and utilize a parallel processing structure. Parallel processing is a method to reduce the decoding latency. In this thesis, we use the parallelization idea and investigate a parallelized structure for turbo codes. These code studies are called *parallel decodable turbo codes* (PDTC). The encoder side is parallelized to encode the substreams of data bits simultaneously. This parallel processing idea is also used at the decoder side. Owing to simultaneous decoding of substreams, decoding latency is decreased significantly.

Memory collision is one of the most important problems observed during the parallelization process. If two or more decoders/encoders try to access the same memory block at the

same clock instant, memory collision occurs and parallel decoders can not operate properly. The reason of having a memory collision is the permutation order of the interleaver. At the encoder side, specific collision-free interleavers can be constructed by using a matrix notation. In that manner, *row-column S-random interleaver*, which is introduced in [4], is of interest.

In digital mobile communications, one of the the main distortions introduced by the communication medium is the Doppler spread due to the movement of the receiver. This Doppler spread results in a time-varying fading channel when communication medium has many scatterers in it. The effect of fading on the transmission is a limiting factor in many communication systems and may mandate estimation of the *channel state information* (CSI) at the receiver side. Turbo codes have shown near-capacity performance over Rayleigh flat-fading channels with the perfect knowledge of the CSI [5]. Therefore, to achieve high-speed reliable communication, channel estimation corresponding to the estimation of the CSI is very often a necessary process. In our work, a joint structure of channel estimator and PDTC decoder is implemented to work in an iterative manner.

As a channel estimation algorithm, the *pilot symbol assisted estimation* method has been studied in this thesis. This method utilizes pilot symbols in the data sequence. At the receiver side, the channel estimation process is initiated with the help of the pilot symbols as their values are known. Adding pilot symbols into the transmission packets introduce some extra redundancy and bandwidth efficiency is reduced as a result. In general, the channel estimators use some sort of filtering to estimate the complex channel coefficients when pilot symbol assisted estimation is considered. In this thesis, we use a different approach which employs LMS filtering. The main reason of using LMS is to reduce the complexity and increase the operating frequency. In [6], a 2-*way LMS* approach has been introduced. In this method, the channel coefficient estimations are carried out in two directions different than the ordinary LMS algorithm, both in forward and in backward directions.

In our work, we studied two main subjects, PDTC decoders and pilot symbol assisted channel estimation using the 2-way LMS algorithm. The structures proposed in these algorithms are implemented on an FPGA board and tested with some different design parameters. Results are compared to make some reasonable choices. These results are particular to the implementation choices and the way of our implementations is not the only one. Our aim is just to show what can be done. Optimization should be performed in the different implementation

cases. The outline of the thesis is as follows.

In Chapter 2, convolutional codes and MAP (marginal a posteriori) decoders are reviewed. The encoder and decoder structures of turbo codes and the parallelization procedure on them to construct PDTC encoder and decoder structures are explained.

In Chapter 3, channel estimation methods using LMS and MMSE algorithms are described briefly. The joint channel estimator and decoder structure is given.

In Chapter 4, the details of implementation and optimizations applied during the realization of the proposed systems on an FPGA board is provided. The auxilary components in obtaining the resulting performances are described.

In Chapter 5, the FPGA-based testbed platform used to test the proposed systems is described. The software programs used during the simulation, implementation, debugging, and testing the systems are mentioned.

In Chapter 6, the results obtained after testing the systems are given and discussed. The resultant performances are compared in terms of various aspects.

CHAPTER 2

PARALLEL DECODABLE TURBO CODES

2.1 Introduction

In channel coding sender adds redundant data to the message to be transmitted so that the receiver can detect and correct the errors caused by the noisy medium. Coding techniques may differ based on the characteristics of transmission medium. Some of the effects of medium are additive noise and fading [7]. Different coding techniques can be listed in two main groups:

- **Block Codes :** They work on fixed-size blocks of bits or symbols of predetermined size. Some of the block codes are Reed Solomon (used in compact discs and computer hard drives), BCH, Golay, Hamming, and cyclic codes. These codes will not be considered hereafter since block codes are not in the scope of this thesis.

- **Convolutional Codes :** They differ from block codes in that they work on blocks of bits or symbols of arbitrary length. Details about their encoder and decoder structure are given in Sections 2.2 and 2.3, respectively.

Choosing the appropriate encoder/decoder structure was an important part of the system setup procedure in this study. Main considerations in determining the system components were low latency in time during encoding/decoding operations, low complexity allowing implementation on our FPGA board, and good error rate. After a careful trade off between these requirements, we decided to use a parallel decodable turbo code (PDTC) structure suggested by Orhan Gazi in his PhD. thesis work [8]. The reasons for this selection and how it matches the listed requirements will be explained with details in Chapter 6.

2.2 Convolutional Encoder

Convolutional encoders generate an output sequence according to a predefined state transition mechanism by accepting an input bit sequence of arbitrary length. They are usually illustrated as finite state machines. Two main elements of these encoders are:

- binary shift registers (flip-flops),
- binary adders (XOR operators).

In each time unit, convolutional encoders receive k input bits and produce n output bits. The ratio of these numbers gives the *code rate*, $R = k/n$, of that encoder. The *constraint length*, K, of a convolutional encoder is defined as the maximum number of bits in a single output stream that can be affected by any input bit [7]. If m_i denotes the number of shift registers in the i^{th} shift register block, the constraint length is calculated as,

$$K = \max(m_i + 1). \tag{2.1}$$

Since (2.1) is used in many publications, we have used it during this study. However, on some resources [9], constraint length is expressed as

$$K = n(\max(m_i + 1)), \tag{2.2}$$

since the all n output bits are simultaneously affected.

The term $\max(m_i)$ is called as the *memory order* and denoted by m. If we illustrate convolutional encoder by a finite state machine, there will be 2^m states[1].

In Figure 2.1, a convolutional encoder with two shift registers (denoted by D to emphasize "delay") in one block, $m_1 = 2$, is given. This encoder has 1 input bit, u, 2 output bits, c_1 and c_2, 4 states and its constraint length is 3. Trellis and state diagrams of this encoder are given in Figures 2.2 and 2.3, respectively.

In the trellis diagram given in Figure 2.2, it can be seen that encoding starts from state 0, S_0, and ends in the same state. This is achieved by adding some extra bit or bits to the input

[1] At any time instant, delay elements get a binary value, 0 or 1. By concatenating these binary values one determines which state the encoder is in.

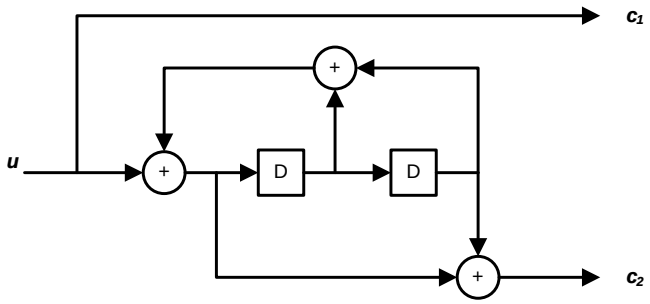

Figure 2.1: Systematic recursive convolutional encoder with $R = 1/2$ and $K = 3$.

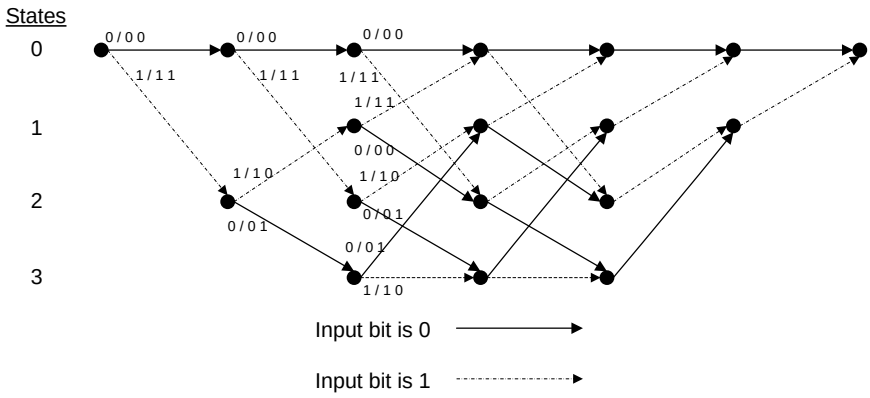

Figure 2.2: Trellis diagram for 4 input and 2 termination bits with the transitions shown in I/OO format.

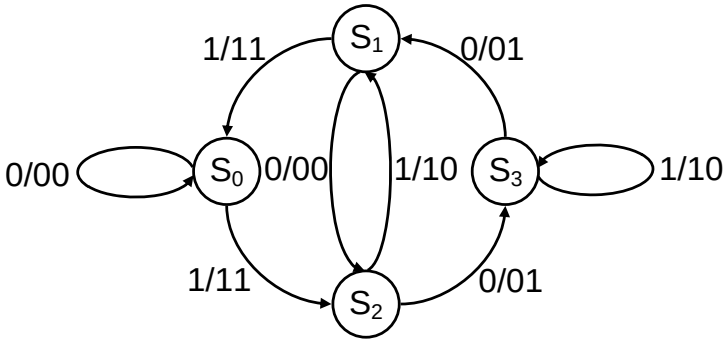

Figure 2.3: State diagram with the transitions shown in I/OO format.

after the input message ends and called *trellis termination*. We also guarantee the next packet to be encoded starting from S_0.

Convolutional encoders are divided into groups regarding their outputs. If the input bits are reproduced in the output codeword without any changes, this encoder is called *systematic*. The encoders are classified as recursive if the output bit (or bits) affects the following states with a feedback path in the structure. The convolutional encoder shown in Figure 2.1 is a recursive systematic ($c_1 = u$) code, whereas the one in Figure 2.4 is non-recursive and non-systematic.

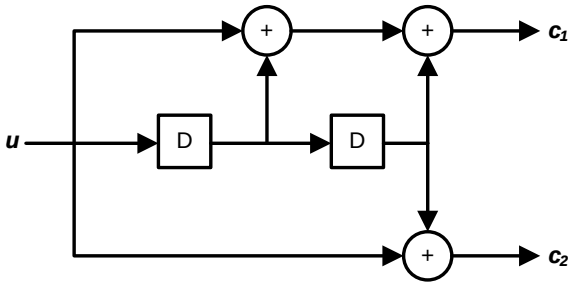

Figure 2.4: Non-systematic non-recursive convolutional encoder with $R = 1/2$ and $K = 3$.

Hereafter, the code given in Figure 2.1 will be used in the encoding/decoding procedures in this thesis and we will refer to this code as $(1, 5/7)$ code due to its generator polynomial written in octal form [9].

2.3 Marginal a Posteriori Decoding

Mainly, there are two types of decoding methods for convolutional codes:

- Maximum likelihood (ML) decoding methods,
- Marginal a posteriori (MAP) decoding methods.

ML decoding aims to find the most likely sequence, $\hat{\mathbf{v}}$, for a transmitted sequence $\mathbf{v}$. This method, basically, minimizes the word (or packet) error rate, $P(\hat{\mathbf{v}} \neq \mathbf{v} \mid \mathbf{r})$, where $\mathbf{r}$ denotes the received sequence. The Viterbi algorithm is the most popular ML algorithm.

The MAP method aims at minimizing the bit error rate (BER) by maximizing the marginal a posteriori probabilities. By applying MAP decoding, $P(\hat{u}_l \neq u_l \mid \mathbf{r})$ is minimized where u_l is the l^{th} bit of the transmitted sequence and $\hat{u}_l$ is the l^{th} bit of the decoder's decision. The BCJR algorithm is the best known example to MAP decoding [3].

Although these two methods exhibit comparable performance, MAP decoding has gained more importance lately due to its prevalence in iterative decoders. Bit/symbol likelihoods are required in such decoders which is directly produced by MAP algorithms.

The BCJR algorithm calculates the *a posteriori log-likelihood ratio* (*a posteriori L-value*) of an information bit. The reason of passing into the log-domain will be clarified later in this section.

The log-likelihood ratio (*LLR*, or simply *LL*) of an information bit u_l can be calculated as

$$LL(u_l) = \ln\left[\frac{p(u_l = +1|\mathbf{r})}{p(u_l = -1|\mathbf{r})}\right], \tag{2.3}$$

for a received sequence $\mathbf{r}$. Using this a posteriori L-value, a hard decision corresponding to u_l can be found by

$$\widehat{u_l} = \begin{cases} +1, & LL(u_l) > 0 \\ -1, & LL(u_l) < 0 \end{cases}. \tag{2.4}$$

In the remaining part of this section the BCJR decoding algorithm steps are to be explained without derivation. Detailed derivations can be found in [9].

The *forward metric*, denoted by α, at time l is defined as the probability of being at state s' at time l and having a received sequence $\mathbf{r}_{t<l}$ up to time l. Hence, the α metric is given as

$$\alpha_l = p(s_l = s', \mathbf{r}_{t<l}), \tag{2.5}$$

where s_l is the state at time l.

Similarly, the *backward metric*, denoted by β, at time l is defined as the probability of receiving a sequence $\mathbf{r}_{t>l}$ after time l given that the state at time l is s,

$$\beta_l = p(\mathbf{r}_{t>l}|s_l = s). \tag{2.6}$$

As the third metric definition, the *branch metric* at time l is the probability of having a state transition from state s' to s at time l. It is denoted by γ and defined as

$$\gamma_l = p(s_{l+1} = s, \mathbf{r}_l|s_l = s'). \tag{2.7}$$

As a result of a few manipulations based on the definitions of α and β, it can be seen that α values are updated by a forward recursion, whereas β values are updated by a backward recursion as given by

$$\alpha_{l+1}(s) = \sum_{s' \in \sigma_l} \gamma_l(s', s)\alpha_l(s'), \tag{2.8}$$

$$\beta_l(s') = \sum_{s \in \sigma_{l+1}} \gamma_l(s', s)\beta_{l+1}(s), \tag{2.9}$$

with initial conditions,

$$\alpha_0(s) = \begin{cases} 1, & s = 0 \\ 0, & s \neq 0 \end{cases}, \tag{2.10a}$$

$$\beta_N(s) = \begin{cases} 1, & s = 0 \\ 0, & s \neq 0 \end{cases}. \tag{2.10b}$$

In (2.10b), N stands for the length of the input sequence[2]. In (2.8) and (2.9), σ_l denotes the set of all possible states from which a transition is possible at time l and σ_{l+1} denotes the set of all possible states to which a transition is possible at time $l + 1$. After having the initial conditions, α and β values can be calculated for the whole packet with the knowledge of γ values.

In an AWGN channel, branch metrics can be written as [9]

$$\gamma_l(s', s) = \mathrm{e}^{u_l L_a(u_l)/2} \mathrm{e}^{(L_c/2)(r_l \cdot v_l)}, \tag{2.11}$$

where $L_a(u_l)$ is the a priori bit probability[3], L_c is the channel reliability factor which is equal to $4E_s/N_0$ [9], and v_l denotes the output vector consisting of data and parity observations for transition from state s' to s. The dot product $(r_l \cdot v_l)$ gives the correlation between the transmitted and received vectors. Scaling this distance with L_c means that the observations are more reliable when SNR is high and a priori values are trusted more when SNR is low.

In order to perform the calculations given in (2.8), (2.9) and (2.11) in an easier way, these operations are usually realized in the logarithmic domain. The log-domain metric values are

[2] It is assumed that termination bits are added at the end of the packet in the encoder side. So, the final state is known to be the zero-state

[3] It must be noted that the L_a values for the termination bits are always 0.

given as follows:

$$\gamma_l^*(s',s) = \ln\gamma_l(s',s) = u_l\frac{L_a(u_l)}{2} + \frac{L_c}{2}(r_l\cdot v_l), \tag{2.12}$$

$$\alpha_{l+1}^*(s) = \ln\alpha_{l+1}(s) = \ln\sum_{s'\in\sigma_l} \mathrm{e}^{\left[\gamma_l^*(s',s)+\alpha_l^*(s')\right]}, \tag{2.13}$$

$$\beta_l^*(s') = \ln\beta_l(s') = \ln\sum_{s\in\sigma_{l+1}} \mathrm{e}^{\left[\gamma_l^*(s',s)+\beta_{l+1}^*(s)\right]}. \tag{2.14}$$

It can easily be seen that both forward and backward metric calculations can be simplified more by defining a max^* operation

$$max^*(x,y) = \ln(\mathrm{e}^x + \mathrm{e}^y) = \max(x,y) + \ln(1+\mathrm{e}^{-|x-y|}), \tag{2.15}$$

where the logarithmic term is usually called the *correction term*.

By using the multiple argument form of the max^* operation, (2.13) and (2.14) can be simplified as

$$\alpha_{l+1}^*(s) = max^*_{s'\in\sigma_l}\left[\gamma_l^*(s',s)+\alpha_l^*(s')\right], \tag{2.16}$$

$$\beta_l^*(s') = max^*_{s\in\sigma_{l+1}}\left[\gamma_l^*(s',s)+\beta_{l+1}^*(s)\right] \tag{2.17}$$

with the initial conditions,

$$\alpha_0^*(s) = \begin{cases} 0, & s=0 \\ -\infty, & s\neq 0 \end{cases}, \tag{2.18a}$$

$$\beta_N^*(s) = \begin{cases} 0, & s=0 \\ -\infty, & s\neq 0 \end{cases}. \tag{2.18b}$$

Figures 2.5 and 2.6 illustrate the use of max^* operation in α and β computations, respectively.

By skipping the intermediate steps, the log-likelihood formula in (2.3) can be rewritten using the formulas described above as [9]

$$LL(u_l) = \ln\left\{\sum_{(s',s)\in\Sigma_l^+} \mathrm{e}^{\beta_{l+1}^*(s)+\gamma_l^*(s',s)+\alpha_l^*(s')}\right\} - \ln\left\{\sum_{(s',s)\in\Sigma_l^-} \mathrm{e}^{\beta_{l+1}^*(s)+\gamma_l^*(s',s)+\alpha_l^*(s')}\right\} \tag{2.19}$$

where Σ_l^+ and Σ_l^- are the sets of transitions with the information bit is 0 and 1, respectively.

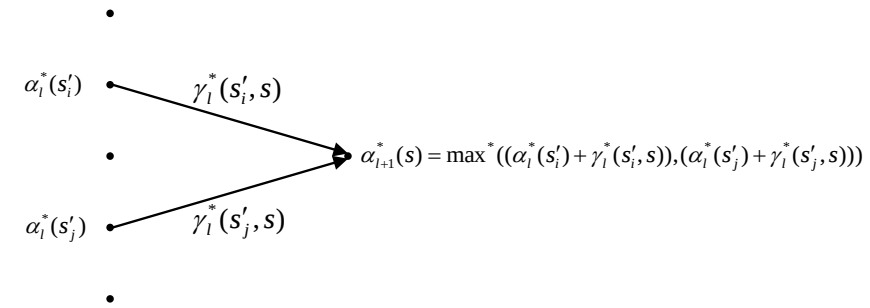

Figure 2.5: Forward recursion in calculation of $\alpha_{l+1}^*(s)$

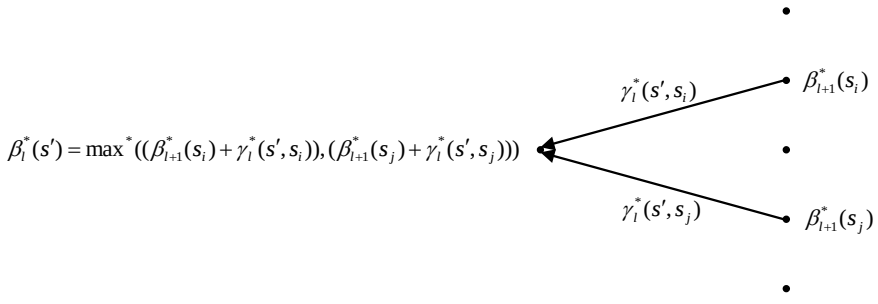

Figure 2.6: Backward recursion in calculation of $\beta_l^*(s')$

2.3.1 An Addition to the BCJR Algorithm

As described, the BCJR algorithm calculates the *LL* values of only the information (data) bits. However, for the channel estimation problem, to be described in Section 3.4, we will need the *LL* values also for the parity bits.

The state transitions depend on the information bits only. That's why, the metric value (α, β, and γ) calculations are not affected even if the parity bits are considered and these computations are carried out without any modification. This means that, no extra complexity is introduced for these parts. However, in the *LL* calculation step, given in (2.19), the transition sets are constructed by considering the information bits. For the computation of the *LL* values for the corresponding parity bits, the transition sets are needed for parity bits. The transition diagram given in Figure (2.7) can be used to construct these sets.

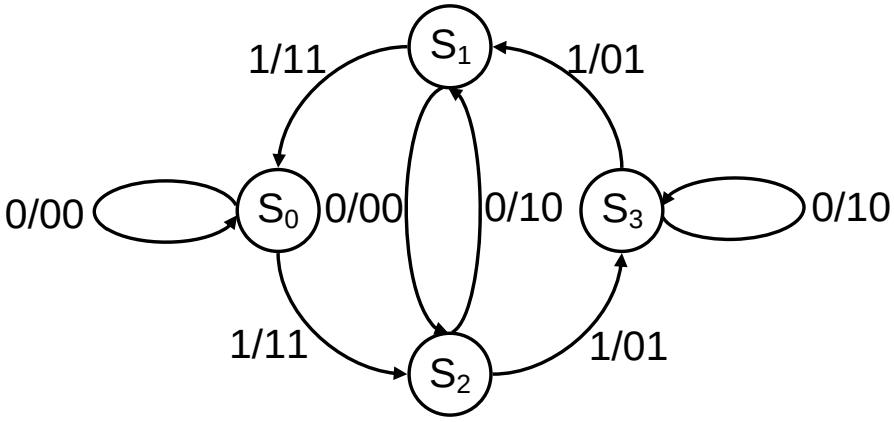

Figure 2.7: State diagram for the transitions of the parity bits.

After constructing these transition sets, (2.19) can be modified to calculate the *LL* values for the parity bits as

$$LL(p_l) = \ln\left\{\sum_{(s',s)\in\Sigma^+_{p,l}} e^{\beta^*_{l+1}(s)+\gamma^*_l(s',s)+\alpha^*_l(s')}\right\} - \ln\left\{\sum_{(s',s)\in\Sigma^-_{p,l}} e^{\beta^*_{l+1}(s)+\gamma^*_l(s',s)+\alpha^*_l(s')}\right\} \tag{2.20}$$

where $\Sigma^+_{p,l}$ and $\Sigma^-_{p,l}$ are the sets of transitions with the parity bit is 0 and 1, respectively.

2.4 Turbo Codes

Turbo codes were first introduced by Berrou et. al. in 1993 [2]. A turbo code encoder consists of two convolutional encoders and an interleaver, which is placed before the second encoder, as shown in Figure 2.8. The input bits, d (to denote "data" bits), are directly given to the first convolutional encoder and it produces first sequence of parity bits, p_1. The second constituent encoder gets the interleaved form of d as input and gives out the second parity bits, p_2. As a result of this parallel structure, turbo codes are also known as the *parallel concatenated convolutional codes* (PCCC). Constituent encoders are not necessarily same. On the decoder side, two decoders that can produce likelihoods as in BCJR are placed together with an interleaver and deinterleaver as shown in Figure 2.9. The received sequence, $\mathbf{r}$, in the decoder side can be separated into three subsequences:

- data observation sequence, $\mathbf{r}_d$

- first parity observation sequence, $\mathbf{r}_{p_1}$
- second parity observation sequence, $\mathbf{r}_{p_2}$

It must be noted that the interleaver used on the decoder side is equivalent to the one in the encoder side. Interleaving along with iterative decoding results in the close-to-capacity performance of turbo codes. However, using interleaver increases latency. Some interleaving algorithms are given in [8].

As seen in Figure 2.9, we apply subtraction on log-likelihood values after each BCJR decoder. By subtracting the input *LL* values from the computed ones, the *extrinsic information* is obtained, which is the true likelihood estimation of that decoding step. More details about the structure of turbo codes are given in [2].

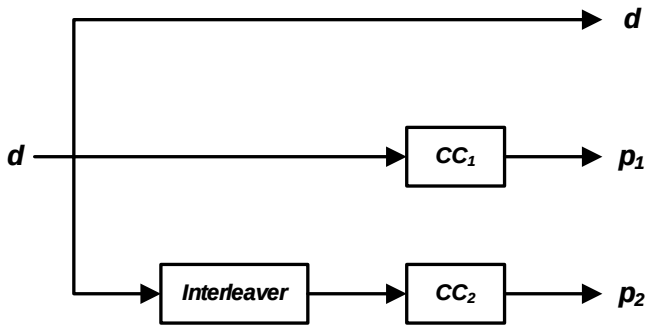

Figure 2.8: Turbo code encoder structure. CC_1 and CC_2 are two convolutional encoders and they operate in parallel.

2.5 PDTC Structure

Benedetto and Montorsi have investigated performance of turbo codes in their studies and they have found out that these codes show a great performance at low SNR [10]. However, high decoding latency comes out as the main drawback for these codes. Reducing the decoding latency is a major problem when speed is taken into consideration. For speeding up the decoding process the parallelization idea has been introduced in the decoder structure and the same idea is applied to the encoder side in [11, 12, 13].

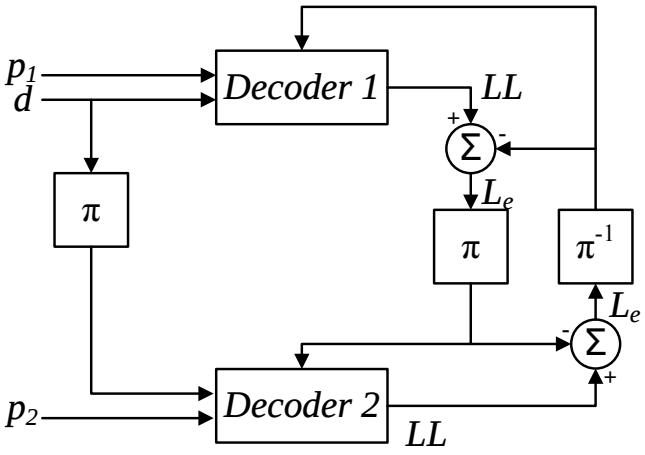

Figure 2.9: Turbo code decoder structure with the usage of extrinsic information.

The FPGA architecture gives us the chance for benefiting from the power of parallel processing. This is achieved by using multiple processors operating in parallel. However, that parallel structure limits the number of utilizable interleaver algorithms. Using an arbitrary interleaver structure may cause a memory collusion problem. For that reason, memory collision-free interleavers are used in PDTC structures.

2.5.1 Encoding Parallel Decodable Turbo Codes

Parallel Decodable Turbo Code (PDTC) encoder consists of convolutional encoders concatenated in parallel such that each one operate on parallelized information bits simultaneously. The PDTC encoder structure is illustrated in Figure 2.10. For the upper encoder cluster, data bits are first sent to a serial-to-parallel converter (S/P) to form N subsequences and each subsequence is encoded in parallel (and separately). These N parallel convolutional encoders are not necessarily the same. Here $Encoder_1 1$ shows the first encoder of the upper cluster and $Encoder_1 N$ denotes the last one. The output of the upper cluster is formed by outputs which come from N parallel encoders. So, a parallel-to-serial converter (P/S) is used for generating first parity sequence, p_1. In the lower cluster, an interleaver (π) is placed before serial-to-parallel converter and then the converter forms M parallel substreams. Encoding operation is accomplished in the same manner as the upper cluster, but with M parallel encoders.

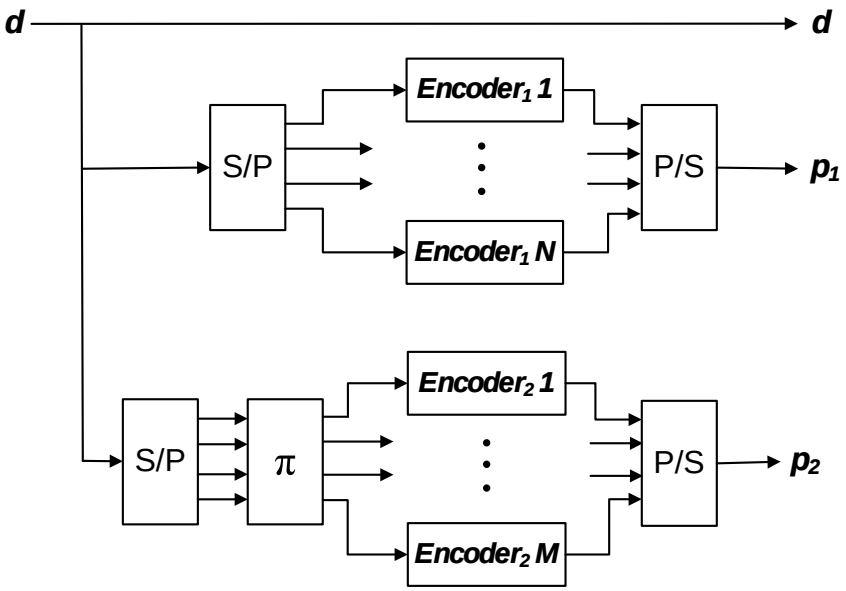

Figure 2.10: PDTC encoder with two clusters, one with N and the other with M constituent encoders.

2.5.2 Decoding Parallel Decodable Turbo Codes

To decrease the decoding latency, each MAP decoder in the TC decoder is replaced with a cluster of MAP decoders which are concatenated in parallel. With this method, each decoder in a cluster operates on D/N data (information) bits where D is the number of data bits in a packet and N is the number of parallel MAP decoders in that cluster. The clusters may contain different numbers of parallel decoders, say that first cluster has N parallel decoders and the second one has M as in the encoder side. However, $N = M$ is generally preferred. In our system we have used 4 parallel decoders (encoders) in each cluster in the decoder (encoder) side, i.e. $N = M = 4$. The decoder structure for a PDTC is given in Figure 2.11. The decoders in the first cluster ($Decoder_1 1, \ldots, Decoder_1 4$) operate on the data bits, d, and the first parity bits, p_1. After the first cluster finishes its job, the extrinsic information (L_e) is generated from the computed log-likelihood (LL) values. These L_e values are used as a priori information by the decoders of the second cluster ($Decoder_2 1, \ldots, Decoder_2 4$). These decoders operate on the interleaved form of data bits and the second parity bits, p_2. The LL and L_e values are calculated and so the first iteration is completed. If the desired number of iterations have not been reached, then the L_e values are deinterleaved and given to the first cluster for the

next iteration. After the final iteration has been finalized the computed LL values are used to estimate the transmitted bit sequence.

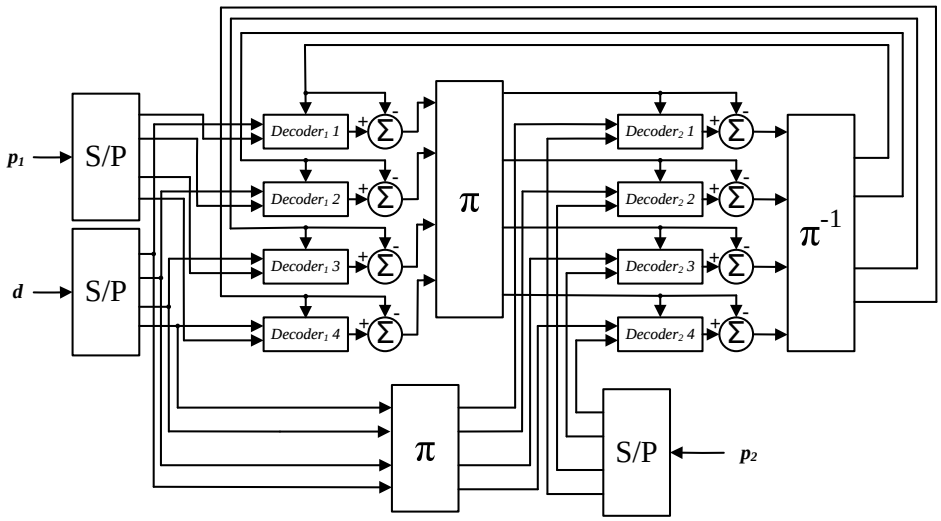

Figure 2.11: PDTC decoder with two clusters, each having 4 constituent decoders.

2.5.3 Memory Collision-Free Interleaver

Although parallelization reduces the decoding latency, it creates some extra problems and memory collision problem is one of them. The memory collision problem can be explained as follows. All the decoders in a cluster run in parallel and access the memory locations where the extrinsic information (L_e) generated by the other cluster's decoders are stored. During this access more than one decoders in that cluster may try to use the stored data at the same memory block at the same clock instant. However, it is impossible to implement this operation with the current memory architectures. Hence, memory collision occurs on that memory segment. An interleaver which causes memory collision is shown in Figure 2.12.

The memory collision should be avoided to implement a parallelized turbo decoder operating properly. This problem stems from the permutation order of the interleaver. So, it is important to use a well-designed interleaver structure such that each decoder (and also encoder) in a cluster should try to access different memory segments at each clock instant. For this reason, we have decided to use a *row-column S-random* (*RCS-random*) interleaver

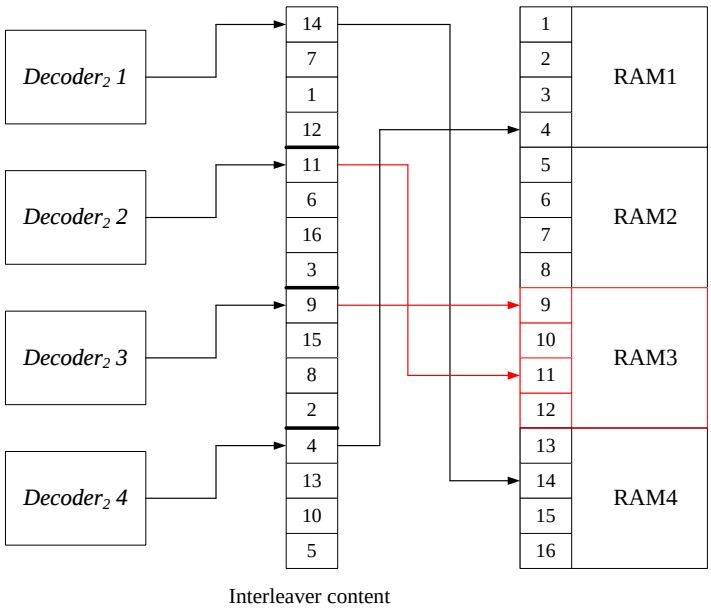

Figure 2.12: A PDTC decoder interleaver structure that does not prevent memory collision.

proposed in [4].

The magic behind RCS-random interleavers is the matrix structure. The operation of a RCS-random interleaver is as follows. First, the data sequence is put into a two-dimensional matrix. Then, the rows of the matrix are interleaved by distinct S-random interleavers. After that, the interleaved matrix is interleaved once more, but this time column-wise by different S-random interleavers. Finally, the elements of the matrix are encoded row-wise, i.e., each row is encoded by constituent encoders of the same encoder cluster. Since the number of rows does not change after the interleaving process, equal number of constituent codes are utilized by this method, that is the case of $N = M^4$. On the decoder side, the operations are carried out in the same manner for both the generated L_e values and received sequence. In that matrix structure, the memory collision occurs if two or more rows have bits at same column with variables stored in the same memory block. However, that event is prevented by the first *row interleaving* operation. However, if we use less number of blocks than the number of rows of the matrix, collision avoidance becomes impossible since at least two decoder variables will share the same memory block at the same clock instant.

[4] The upper encoder cluster operates on the uninterleaved form of the matrix.

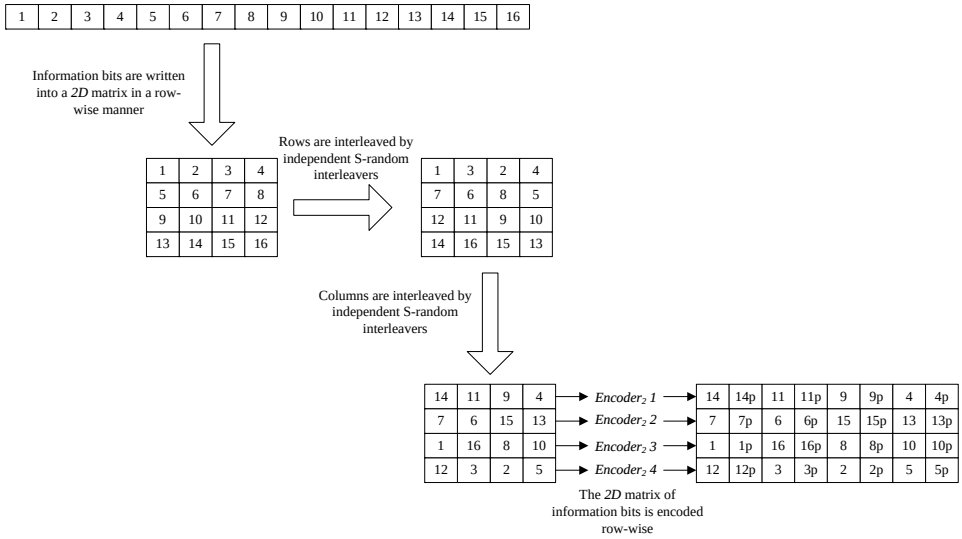

Figure 2.13: RCS-random interleaver generation and encoding of the information bits in parallel.

For a better understanding, we present an example that illustrates the use of RCS-random interleaver in a PDTC structure with $N = M = 4$. In this example encoding and decoding for the uninterleaved data will be skipped for simplicity. In Figure 2.13, the interleaving operation and the encoding operation according to the generated interleaver is shown. On the decoder side, the second cluster's decoders use the L_e values generated by the first cluster's decoders. The access sequence of those decoders in the second cluster is shown in Figure 2.14.

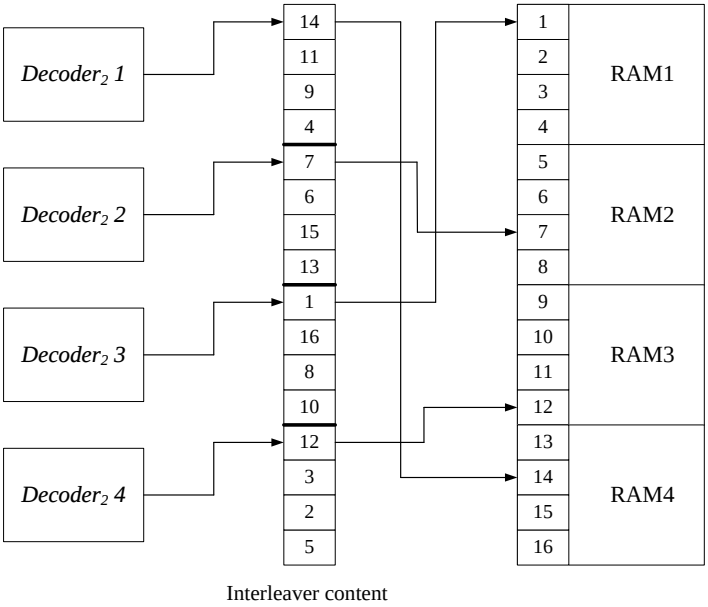

Figure 2.14: Constituent decoders of the second cluster get the addresses of L_e values and read the data in the given address to operate on.

CHAPTER 3

JOINT CHANNEL ESTIMATION AND DECODING

3.1 Introduction

In digital mobile communications, one of the the main distortions introduced by the communication medium is the Doppler spread due to the movement of the receiver. This Doppler spread results in a time-varying fading channel when communication medium has many scatterers. It is removed by estimating the *channel state information* (CSI). To achieve high-speed reliable communication, channel estimation corresponding to the estimation of the CSI is very often a necessary process.

There exist some different channel estimation algorithms in the literature. These different estimation techniques can be listed in two main groups:

- **Blind Estimation :** Blind Estimation techniques use the statistical properties of the transmitted signals without any knowledge about the transmitted symbols. These methods do not require a training sequence. Since no training sequence is used in the packets, blind estimation provides more efficient usage of bandwidth. Some blind estimation methods are listed in [14] and [15]. They will not mentioned anymore since blind estimation methods are not in the scope of this thesis.

- **Pilot Symbol Assisted Estimation :** These methods utilize pilot symbols placed in the data sequence. The channel estimation process is initiated with the help of these pilot symbols as their bit values are known in the receiver side. Adding pilot symbols into the data sequence introduces some redundancy and bandwidth efficiency somewhat decreases.

In this chapter of the thesis, we will work on an channel estimation method using the pilot symbol assisted estimation technique. In the design, we will use the turbo decoder described in Chapter 2 together with the channel estimator to form an iterative structure. The joint estimation and decoding structure is first proposed in [16].

3.2 Channel Model

As the channel model, we consider a discrete-time time-varying fading channel with AWGN so that the received signal at time instant k can be written as

$$r_k = f_k a_k + \eta_k \tag{3.1}$$

where f_k is the sample from a time-varying correlated fading process at time instant k, a_k is the transmitted symbol, and η_k is the sample from circularly symmetric complex Gaussian random variable with mean 0 and variance σ_η^2. For the fading coefficients f_k, it is assumed that they are independent of the transmitted symbols and the noise.

The Rayleigh fading process used in the model is generated by using the Jakes' model given in [17]. In Jakes' model, it is assumed that the real and imaginary parts of the complex channel coefficients (f_k values) are independent with the autocorrelation function of

$$R_f[k] = J_0(2\pi f_d k T_s) \tag{3.2}$$

where $J_0(.)$ is the zeroth order Bessel function of the first kind, f_d is the relative Doppler frequency between the transmitter and the receiver, and T_s is the symbol period. To simplify the future calculations, we assume normalized flat fading, $E\{|f_k|^2\} = 1$, and unit energy transmitted symbols, $E\{|a_k|^2\} = 1$.

3.3 Transmitter Model

At the transmitter side, we use pilot symbol assisted modulation (PSAM) as proposed in [16]. The transmitter model is given in Figure 3.1.

In this model, a data sequence $\{d_k\}$ is first encoded by a turbo encoder. The encoded bit sequence $\{c_k\}$ is passed through a channel interleaver. The resulting interleaved sequence

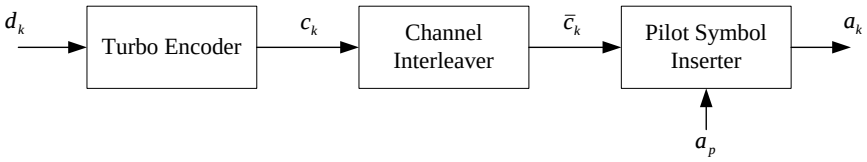

Figure 3.1: Joint channel estimation and decoding transmitter

$\{\bar{c}_k\}$ is then split into groups of $(M-1)$ bits and the known pilot symbols $\{a_p\}$ are placed in the center of each group periodically. The final sequence $\{a_k\}$ is transmitted over the time-varying Rayleigh fading channel. In this work, all the sequences are in polar form, i.e., $\{d_k\}, \{c_k\}, \{a_k\} \epsilon \{-1, 1\}$. The parameter M is called the *pilot symbol spacing* and it is assumed to be odd [16]. These pilot symbols may take on different values.

In this model, a channel interleaver is required for a better performance because turbo encoding may not be sufficient in the existence of a fading channel induced errors, i.e., burst errors. For that reason, a channel interleaver is used to scramble the symbols. The effect of channel interleaver on the performance gain is given in [5].

3.4 Receiver Model

The iterative channel estimation structure proposed in [16] is used with a feedback path from the turbo decoder at the receiver side. The receiver structure is given in Figure 3.2.

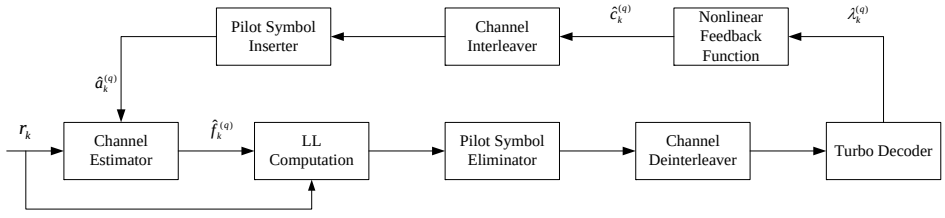

Figure 3.2: Joint channel estimation and decoding receiver

After the reception of a sequence, the sequence is given to the channel estimator. At the

first run of channel estimation, a filtering operation is performed to obtain the initial estimates of fading coefficients $\{f_k^{(q)}\}$ where superscript $^{(q)}$ denotes the iteration number of the channel estimation and $q = 1$ for this first iteration. Using these initial estimates $\{\hat{f}_k^{(1)}\}$ together with the received sequence, the log-likelihood ratios (*LL*) are calculated and fed to a *Pilot Eliminator* module to remove the *LL* of the pilots. After removing the pilots, the remaining *LL*s are passed through a channel deinterleaver and the resultant *LL* sequence is given to a *soft-in soft-out* (SISO) decoder. In our system, turbo decoders are used for the decoding process.

To get a better performance from the feedback mechanism, the decoder needs to calculate the probabilities of the parity symbols together with the data symbols. The parity *LL* calculation issue, handled in Section 2.3.1, steps in at that point. After the decoding process, the computed *LL*s $\{\lambda_k{}^{(q)}\}$ of the coded sequence $\{c_k\}$ are passed through a *nonlinear feedback function* to obtain the estimates of the code symbols $\{\hat{c}_k^{(q)}\}$. This feedback function can produce hard-decision or soft-decision estimates. After the estimation of coded symbols, these estimates are first interleaved and then pilots are added in the same manner explained in the transmitter side. After all, the resulting sequence estimation $\{\hat{a}_k^{(q)}\}$ is given back to the channel estimator. The estimator runs again, but this time it uses the whole $\{\hat{a}_k^{(q)}\}$ set, not only the pilot values $\{a_p\}$. After the estimation is completed, the operations explained for the first estimation and feedback mechanism are repeated iteratively.

In the receiver structure proposed in [16], turbo decoder runs for one iteration after each iteration of the estimation process. In our system, we prefer to operate the turbo decoder for some iteration numbers to get more reliable *LL*s for the code symbols and then feed these $\{\lambda_k{}^{(q)}\}$ values back to the estimator to complete the estimation iteration.

3.5 Channel Estimator

The receiver model given in Figure 3.2 requires a channel estimator for proper operation. Different algorithms are available to estimate the channel. In this part, we will describe the estimation method which employs the LMS filtering. An estimation method which uses a *minimum mean-squared error* (MMSE) filtering technique instead of LMS is given in [16].

In the channel estimation method which we have used in our system, the estimation

process is carried out with a well-known LMS algorithm [15]. Using an LMS filter instead of MMSE allows less computations. That's why channel estimator structures with LMS has a lower complexity when compared to the ones with MMSE.

This technique derives the first estimates by using the pilot symbols only. Since the values of only pilot symbols are available at the receiver side, applying an LMS algorithm is not applicable. So, the channel is estimated at the pilot locations and these estimated values are assumed to be constant around the pilot symbols. The structure of a received packet is given in Figure 3.3 with a pilot symbol spacing of M.

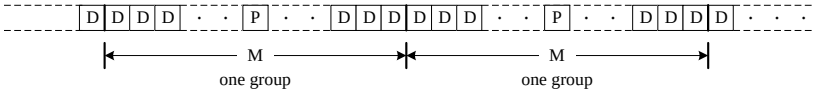

Figure 3.3: Packet structure of a received packet where P denotes the pilot symbols.

In the following iterations, an LMS filter is employed instead of MMSE filter while estimating the complex fading coefficients. The channel estimation in the $q-th$ iteration is computed by using the estimates of the transmitted sequence in the $(q-1)-st$ iteration $\hat{a}_k^{(q-1)}$,

$$\hat{f}_{k+1}^{(q)} = \hat{f}_k^{(q)} + \beta e_k^{(q)} \hat{a}_k^{(q-1)} \tag{3.3}$$

where β stand for the *step size* of the LMS algorithm and $e_k^{(q)}$ is called the *error term*, defined as

$$e_k^{(q)} = r_k - \hat{f}_k^{(q)} \hat{a}_k^{(q-1)}. \tag{3.4}$$

To improve the error performance of an LMS filtering method, a different approach can be used, called the 2-*way LMS* [6]. In this approach the estimation process is carried out in two ways, in forward and backward directions. In both ways the LMS algorithm is applied.

For the forward channel estimation case, the computations are the same as the ones previously described in the regular LMS algorithm. If we call the forward estimates of complex fading channel coefficients for the $q-th$ iteration as $\{\hat{h}_k^{(q)}\}$,

$$\hat{h}_{k+1}^{(q)} = \hat{h}_k^{(q)} + \beta e_{f,k}^{(q)} \hat{a}_k^{(q-1)} \tag{3.5}$$

where $e_{f,k}^{(q)}$ stand for the error term in the *forward* direction,

$$e_{f,k}^{(q)} = r_k - \hat{h}_k^{(q)} \hat{a}_k^{(q-1)}. \tag{3.6}$$

In the backward direction, the same computations as in the forward case are applied with small differences. The backward estimations of complex fading channel coefficients $\{\hat{g}_k^{(q)}\}$ are computed as follows.

$$\hat{g}_{k-1}^{(q)} = \hat{g}_k^{(q)} + \beta e_{b,k}^{(q)} \hat{a}_k^{(q-1)}, \tag{3.7}$$

$$e_{b,k}^{(q)} = r_k - \hat{g}_k^{(q)} \hat{a}_k^{(q-1)}. \tag{3.8}$$

The estimates of complex fading channel coefficients are $\{\tilde{f}_k^{(q)}\}$ the average of the estimates computed in forward and backward directions,

$$\tilde{f}_k^{(q)} = \frac{\left(\hat{h}_k^{(q)} + \hat{g}_k^{(q)}\right)}{2} \tag{3.9}$$

It must be noted that, the 2-way recursive computations are initiated by using the estimation results of the previous iteration.

$$\hat{h}_0^{(q)} = \tilde{f}_0^{(q-1)}, \tag{3.10}$$

$$\hat{g}_{p-1}^{(q)} = \tilde{f}_{p-1}^{(q-1)}, \tag{3.11}$$

where p is the length of the packet including the pilot symbols.

This 2-way LMS filtering method can also be used for determining the carrier frequency offset to overcome the synchronization problem.

3.6 Nonlinear Feedback Function

The nonlinear feedback function, shown in Figure 3.2, generates the estimates of the coded sequence $\{\hat{c}_k\}$ using the *LL*s computed by the turbo decoder $\{\lambda_k\}$. This function can be implemented in two different ways as mentioned in Section 3.4,

1. **Soft-decision feedback function :** In the soft-decision feedback case, the function uses the trigonometric tanh(.) operation for estimation of coded sequence [16]. For the $q-th$ iteration the function operation is as follows:

$$\hat{c}_k^{(q)} = \tanh\left(\frac{\lambda_k{}^{(q)}}{2}\right). \tag{3.12}$$

2. **Hard-decision feedback function :** The hard-decision case uses the limiting values of the given soft-decision feedback function in (3.12),

$$\hat{c}_k^{(q)} = \begin{cases} +1, & \lambda_k{}^{(q)} > 0 \\ -1, & \lambda_k{}^{(q)} \leq 0 \end{cases} . \tag{3.13}$$

CHAPTER 4

FPGA IMPLEMENTATION

As mentioned before, our aim is to implement the structures described in Chapters 2 and 3 on an FPGA-based system. During the implementation, we have faced with some problems due to the fixed-point architecture of the FPGAs. On the other hand, we were able to make some optimizations by using the parallel processing capability of the FPGA structure. In this chapter, we mainly concentrate on the implementation aspects of parallelization of Turbo Decoders and estimating the fading channel characteristics by LMS algorithm on an FPGA-based system. It must be noted that the implementation steps listed in this chapter are determined during the simulations. Resulting performances of the designs on the ML-402 platform are discussed in Chapter 6.

4.1 PDTC Decoder Implementation on the FPGA

The decoding algorithm for a PDTC decoder using the MAP decoding algorithm is given in Sections 2.3 and 2.5.2 with details. However, implementing a soft-in soft-out (SISO) decoder on an FPGA inherently faces some problems since it has limited resources which do not let one easily use floating-point arithmetic or large fixed-point arithmetic. Throughout this section, we will describe our solutions to the problems and optimizations we have applied.

4.1.1 The Center to Top Algorithm

When the metric calculations in (2.13) and (2.14) are considered, it can be seen that the two operations are independent of each other. This gives the ability to calculate α and β metrics simultaneously assuming that all of the received values are available for branch metric

calculations. This assumption is valid for the iterative decoding schemes since decoding process can begin after receiving the whole packet. By this algorithm, the decoding time can be halved. Consider a decoder running on 40 information bits. At time 0 the metric values are initialized, that is $\alpha^*(0)$ and $\beta^*(39)$ values are generated as defined in (2.18a) and (2.18b). After that, α^* and β^* values are calculated without computing any LL value up to time 20 as shown in Figure 4.2.

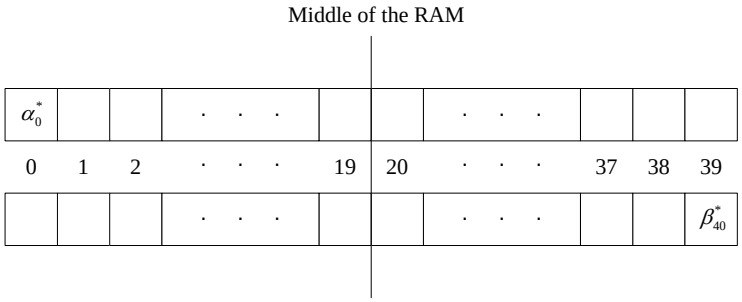

Figure 4.1: α and β values are initialized at time $t = 0$.

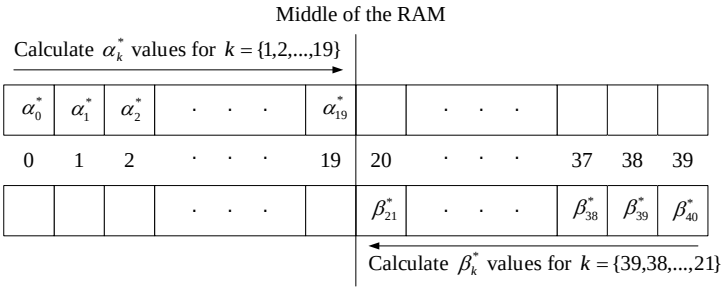

Figure 4.2: α and β values are calculated up to time $t = 20$ with forward and backward recursive calculations.

At time 20, both of α_{20}^*, β_{21}^* and α_{19}^*, β_{20}^* values are available together with the branch metrics for the time instants, γ_{20}^* and γ_{19}^*. So, $LL(u_{20})$ and $LL(u_{19})$ are computed and given out as shown in Figure 4.3.

That process, starting from the center of the frame, continues to the end and simultaneously to the beginning of the frame. That's why this algorithm is named as "center to top" [12]. The rest of the process after the calculation of the center log-likelihood values is shown

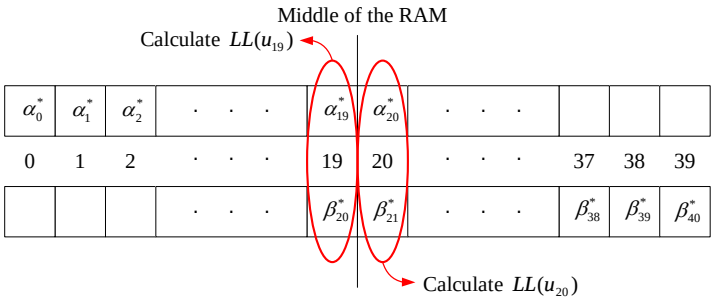

Figure 4.3: At time $t = 20$, $LL(u_{19})$ and $LL(u_{20})$ are calculated.

in Figure 4.4.

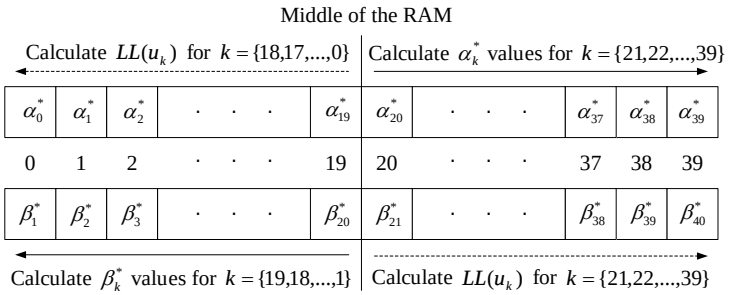

Figure 4.4: LL calculation continues in parallel with the α/β calculation to the end of the bit sequence, that is $t = 39$.

It must be noted that α and β metric values do not have to be written to memory after the midpoint, since LL values are calculated simultaneously. So, not only the decoding time but also the memory usage is halved by this algorithm. The decoding of a 40-bit sequence (20 information bits and 20 parity bits) just takes 40 clock cycles and this process uses a memory block of length 40[1].

4.1.2 Observation Quantization

In the conventional mathematical model, a +1 or −1 is assumed to be transmitted for BPSK, an appropriate noise is added and calculations are carried on with these assignments. An

[1] The width of the used memory depends on the representation of the metric values in the fixed-point architecture

AWGN channel for BPSK modulation can be modeled as

$$y_k = h_k x_k + n_k, \tag{4.1}$$

for any time instant k where y_k is the received symbol, h_k is the channel gain ($\sqrt{E_s}$ in an AWGN channel with E_s being the signal energy), x_k is the transmitted bit ($x_k = \mp 1$) and n_k is a circularly symmetric complex Gaussian random variable with mean 0 and variance N_0.

The conditional probability of a received symbol y_k can be expressed as

$$f(y_k|h_k, x) = \frac{1}{\pi N_0} \mathrm{e}^{-\frac{|y_k - h_k x|^2}{N_0}}, \tag{4.2}$$

$$\ln(f(y_k|h_k, x)) = -\ln(\pi N_0) - \frac{|y_k|^2}{N_0} - \frac{|h_k|^2 |x_k|^2}{N_0} + \frac{2}{N_0} \Re\{y_k h_k^* x^*\}, \tag{4.3}$$

$$= C + \frac{2}{N_0} \Re\{y_k h_k^* x^*\}, \tag{4.4}$$

where C is a constant and has no effect on the MAP calculations. Hence, the function can be redefined as

$$\ln(f(y_k|h_k, x)) \doteq \frac{2}{N_0} \Re\{y_k h_k^* x^*\}, \tag{4.5}$$

where $\doteq$ denotes equality with a constant.

As we use fixed-point arithmetic, the metric values in the BCJR algorithm are represented by a fixed number of bits, K. However, the decoder is not guaranteed to work properly with this representation unless the channel observations (input of the decoder) are carefully quantized. For that reason, we need to quantize observations by a quantization factor, q, such that the represented observations lay in a set S smaller than the set of numbers represented by K bits. After that, the quantized observation probability for $x = 1$ is used in decoding with

$$Q_k = Q(\ln(f(y_k|h_k, x = 1))) = \left\lfloor \frac{2/N_0 \Re(y_k h_k^*)}{q} \right\rfloor. \tag{4.6}$$

If we apply the AWGN channel model given in (4.1) on (4.6) for a BPSK modulation, we get

$$Q_k = \left\lfloor \frac{2\sqrt{E_s}/N_0 \Re\{y_k\}}{q} \right\rfloor, \tag{4.7}$$

$$= \left\lfloor \frac{2\sqrt{E_s}/N_0 \Re\{(\sqrt{E_s} + n_k)\}}{q} \right\rfloor, \tag{4.8}$$

$$= \left\lfloor \frac{2E_s}{N_0 q} + \frac{2\sqrt{E_s}}{N_0 q} n_I \right\rfloor, \tag{4.9}$$

where n_I is the real part of the complex Gaussian noise with mean 0 and variance $N_0/2$.

Recalling that a finite number of bits are used in representing numbers, the question is how to choose q. If q is chosen to be very small, Q_k's will be large and the formulas such as (2.19) will not function properly due to overflow. If q is chosen to be very large, then the difference in noise values of the observations will not be properly passed to the decoder and then soft decoding will suffer. We resolve the problem above by the compromise that the packet is normalized with respect to its absolute maximum symbol value, $ObsMax$. If we represent that value with a predefined value, $NormMax$ (absolute maximum value after the quantization is performed) then we get a set $S = \{-NormMax, -NormMax + 1, \ldots, NormMax - 1, NormMax\}$ for decoder's input sequence. This information can be combined with a well known property of the Gaussian distribution that, in a normally distributed set with mean μ and variance σ^2, observing a number p such that $|p| > \mu + 3\sigma$ has a probability of about $1/1000$. To be able to apply that property, we need to identify the mean and variance of the random variable $A = \frac{2E_s}{N_0 q} + \frac{2\sqrt{E_s}}{N_0 q} n_I$.

$$E\{A\} = \frac{2E_s}{N_0 q} \tag{4.10}$$

$$\begin{aligned} \sigma_A &= \frac{2\sqrt{E_s}}{N_0 q}\sigma_{n_I} = \frac{2\sqrt{E_s}}{N_0 q}\frac{\sqrt{N_0}}{\sqrt{2}} \\ &= \sqrt{\frac{2Es}{N_0 q}}\frac{1}{\sqrt{q}} \\ &= \frac{\sqrt{E\{A\}}}{\sqrt{q}} \end{aligned} \tag{4.11}$$

After the quantization of the packet, it is known that symbols greater than $+NormMax$ or smaller than $-NormMax$ can occur in the packet with a small probability. If we neglect the small probability of $1/1000$, we can define $NormMax$ as

$$NormMax = E\{A\} + 3\sigma_A \tag{4.12}$$

$$= E\{A\} + 3\frac{\sqrt{E\{A\}}}{\sqrt{q}} \tag{4.13}$$

By replacing (4.10) in (4.13), we get

$$NormMax = \frac{2E_s}{N_0 q} + 3\sqrt{\frac{2E_s}{N_0 q}}\frac{1}{\sqrt{q}}. \tag{4.14}$$

By solving this equation, q can be calculated as

$$q = \frac{\frac{2E_s}{N_0} + 3\sqrt{\frac{2E_s}{N_0}}}{NormMax}. \tag{4.15}$$

As it is obvious in (4.15), q is a function of the SNR (E_s/N_0) for a selected $NormMax$ value. Instead of calculating the q value for each packet, a look-up table (LUT) can be used. A relatively large LUT that stores the q values in 8 bits, 3 for integer part and 5 for the decimal part gives a precision of $1/2^5$ and yields a satisfactory performance. This q value will be used to represent any floating-point number in our used representation if needed.

4.1.3 max^* Approximation

The max^* expression given in (2.15) contains two terms: the maximization term ($\max(x,y)$) and the correction term ($f_c(|x-y|) = \ln(1+e^{-|x-y|})$). This correction term poses a trouble when it is needed to be expressed in fixed-point arithmetic. It is not possible to easily realize the *naturallogarithm* and *exponential* functions fully in such a system. That's why, some approximations have been applied to realize the correction term $f_c(|x-y|)$ in the max^* operation. Four different approximations have been listed in [18] with the resulting BER performances. These methods can be listed as follows.

1. **Max-log-MAP approximation:** With this method, the correction term is neglected ($f_c(|x-y|) \approx 0$) for all (x,y) pairs and the max^* operation is approximated as the ordinary max operation,
$$max^*(x,y) \approx \max(x,y). \tag{4.16}$$

2. **Constant log-MAP approximation:** This method takes the correction term as a constant value C or as zero. To give out the result, the difference of the two operands ($|x-y|$) is compared to a predetermined threshold value T and if the difference is higher than that threshold the correction term gets the value 0. However, if the difference is found to be smaller than the threshold then the correction term is expressed with a constant value. This algorithm can be formulated as
$$f_c(|x-y|) = \ln(1+e^{-|x-y|}) \approx \begin{cases} 0 & \text{if } |x-y| > T \\ C & \text{if } |x-y| \leq T \end{cases}. \tag{4.17}$$

3. **Linear log-MAP approximation:** In this method, the correction term is approximated by a piece-wise linear function.
$$f_c(|x-y|) \approx \begin{cases} 0 & \text{if } |x-y| > T \\ a(|x-y|-T) & \text{if } |x-y| \leq T \end{cases}. \tag{4.18}$$

The minimum mean-squared error (MMSE) algorithm gives a reasonable solution to (4.18) as $a = -0.24904$ and $T = 2.5068$ [18]. It must be noted that, the T value must be quantized with the q value defined in (4.15) since it is an expression in unquantized floating-point domain. The new T value will be $T' = T/q$.

4. **Lookup table (LUT) approximation:** In this method, a table is used to store the $f_c(|x-y|)$ terms. The performance of this method strongly depends on the depth and width of the table. The width of the table determines the precision of the correction term projected to the max^* operation. On the other hand, to span a set of $|x-y|$ values within more precise intervals, larger tables must be used. The most important issue of the LUT approximation method is that the table must store the values in the quantized form. To generate a LUT, we can write an equation as the following.

$$LUT(i) = \left\lfloor \frac{\ln(1 + e^{-iq})}{q} \right\rfloor \tag{4.19}$$

where $i = |x - y|$ and q is the quantization factor calculated during the observation quantization process. When compared with the other approximation methods, using a LUT requires much more resources to be implemented but approximates the floating-point max^* operation in the best way (when a table with appropriate depth and width is used).

As it is obvious in the explanations above, to get a better numeric approximation a more complex method is needed to be used. This complexity brings some extra calculations together and that results in more resource consumption, slower clock speed, and higher decoding latency. With all these in the hand, the approximation method must be decided after a carefully carried on analysis. As declared at the beginning of this chapter, decoding latency and resource consumption are two important requirements of our design as well as the error correction performance. Among the 4 described methods, only max-log-MAP approximation neglects the correction term. So, the q value is not used since no real value needs to be represented in our own fixed-point domain. This means that, no reading from q-storing LUT is needed and no latency is introduced due to reading of the q value. That brings an extra speed for max^* operation. After all, we have decided to use the max-log-MAP approximation method by accepting a performance degradation up to 0.2 dB as seen in Figure 4.5. This figure is obtained after a simulation over 2000 packets with each containing 160 data bits. As a

result of using this approximation method in the decoding process, the MAP decoder is named the *max-log-MAP decoder* and we have carried on the studies with that decoder architecture.

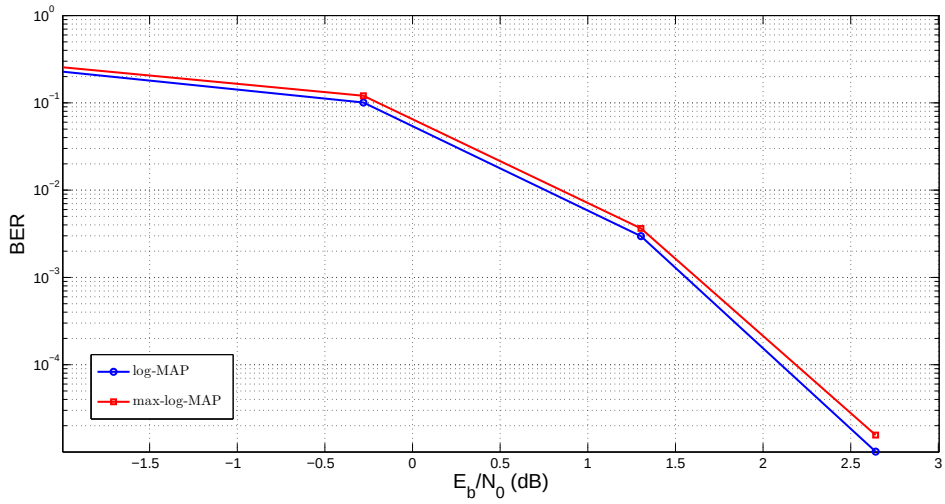

Figure 4.5: The performance difference between a log-MAP decoder (using the ordinary max^* operation with infinite precision) and a max-log-MAP decoder.

Studies in [19] and [20] have shown that max-log-MAP decoders work without any need on SNR estimation. In other words, that decoder does not need an exact SNR estimation to operate properly. So, not only the max^* operators but also the constituent BCJR decoders do not need the q-storing LUT. When this LUT is removed, the decoder consumes less resources and operates on faster clock speeds.

4.1.4 Fixed-Point Summation and Subtraction

Using a restricted set ($[-(2^{K-1}-1), 2^{K-1}-1]$ where K is the metric size) to represent metric values forces us to introduce new summation and subtraction operations with the closure property in the given set. The operation *clipsum*, denoted by $\oplus$, replaces with the regular summation. Under the assumption of $plus\_inf = 2^{K-1}-1$ and $minus\_inf = -plus\_inf$,

$$a \oplus b = \begin{cases} plus\_inf, & a \geq plus\_inf \text{ or } b \geq plus\_inf \\ minus\_inf, & a \leq minus\_inf \text{ or } b \leq minus\_inf \\ plus\_inf, & a + b \geq plus\_inf \\ minus\_inf, & a + b \leq minus\_inf \\ a + b, & else. \end{cases} \tag{4.20}$$

Similarly, a new subtraction operation *clipsubtract* ($\ominus$) is introduced as

$$a \ominus b = a \oplus (-b). \tag{4.21}$$

4.1.5 Node (α, β) Metric Normalization

In (2.8) and (2.9) it has been shown that α and β values are updated in a recursive manner. As the computations go further, these metric values may overflow ($> plus\_inf$) or underflow ($< minus\_inf$). To solve this problem, α and β values are normalized at each trellis step. After each forward recursion, maximum of the newly generated forward metric values is subtracted from these values and α metrics are updated with these normalized values. The same is applied to the β metrics. After the normalization process, we get a maximum value of 0 for α and β metrics at each time instant and prevent underflow and overflow cases. Once any of the metric values reach $plus\_inf$ or $minus\_inf$, further calculations will not be able to diverge from that value due to the *clipsum* operation given in (4.20). So, normalization acts an important role in implementation of the BCJR decoder in fixed-point arithmetic. Also, some normalization procedure may be used even in floating-point case to speed up the system. Another approach to node metric normalization can be found in [21].

4.1.6 Memory Complexity

Before the decoding process, the observations have to be stored in different memory blocks in order to use them in a parallel decoder structure. For that reason, a memory structure is defined as follows. If there are N decoders operating in parallel, then there must be N independent memory blocks for data bit observations (d in Figure 2.11). Accordingly, N memory blocks for parity observations and N memory blocks for interleaved parity observations (p_1 and p_2 in Figure 2.11, respectively). In addition to these, N memory blocks are also defined for interleaver (memory collision-free) tables.

Log-likelihood values are stored in RAMs, too. Each decoder needs an a priori probability (L_a) and generates log-likelihood ratio (LL) and *extrinsic information* (L_e), where in our design L_e's are calculated within the MAP decoder[2]. These L_e and L_a notations are eligible

[2] The extrinsic information is generated inside the decoder to decrease the system complexity at the expense of maximum clock speed.

for the decoders running in the first cluster. In the second cluster, decoders use L_e values as L_a and generates the L_e values which will be used as L_a in the next iteration. The word "cluster" is used just for picturing the system and corresponds to the blocks in half of an iteration. In fact, decoders only change their state to switch the input and output log-likelihood ratios (L_a and L_e). Since LL values are final results, they are updated (overwritten) after each *cluster* run. That structure brings out a memory usage of $3N$ memory blocks for log-likelihood ratio storage in the PDTC decoder.

Summing up all yields a usage of $7N$ number of memory blocks for a PDTC decoder with N constituent decoders in each cluster.

4.2 Channel Estimation Implementation on the FPGA

The channel estimation algorithm for the joint estimation and decoding structure is given in Chapter 3 with details, and shown in Figures 3.1 and 3.2. In this section, we will describe the implementation steps of the proposed structure on an FPGA-based platform.

4.2.1 Pilot Symbol Insertion and Elimination

The simulations given in [16] have been carried out for two different pilot symbol spacing values, $M = 11$ and $M = 21$. In our system, we use a reasonable M value by considering these previous studies, $M = 17$. There are two main factors that makes this choice reasonable: the Doppler frequency range of the simulated channel (to be discussed in Chapter 6), and the value $M - 1$ must be a divisor of the packet length (512).

As shown in Figure 3.1, the interleaved form of the encoded bits is given to the *Pilot Symbol Inserter* module for the pilot insertion process. To accomplish this, these bits are first split into groups of 16 bits. After that, the pilot symbols are placed into the center of each group. Although these pilot symbols may be assigned different values, we use $\{a_p\} = 1$ in our structure. The input and output sequences of the pilot inserter module are given in Figure 4.6.

To remove these pilot symbols, a *Pilot Symbol Eliminator* module is assigned at the receiver side as shown in Figure 3.2. The operation applied in that module is exactly the inverse of the insertion process given in Figure 4.6.

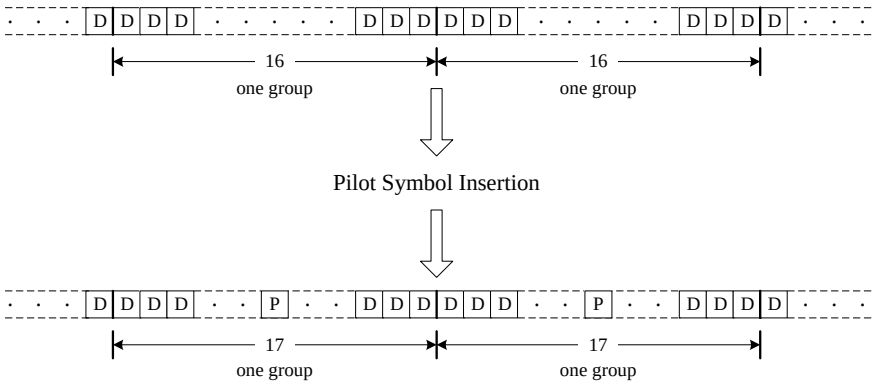

Figure 4.6: Input and output packet structures of the pilot symbol inserter module.

Since $M = 17$ is considered in the design, $512/16 = 32$ pilot symbols are inserted into the packet. After that insertion, the packet length becomes 544, i.e. , some redundancy is introduced due to pilot symbol assisted channel estimation algorithm.

4.2.2 Channel Estimator

The two channel estimation algorithms for the pilot symbol assisted estimation method were described in Section 3.5 in detail. In this section, we give out the details about the implementation of the *2-way LMS* estimation algorithm.

Similar to the PDTC case, the main problem appears as the realization of a SISO algorithm in fixed-point arithmetic, this time a filtering algorithm. To handle that problem, the received values are first represented in fixed-point. Thus, a quantization process is applied to the received sequence to represent these values by a predetermined number of bits, i.e., L. Different than the PDTC case, the channel is a Rayleigh fading channel. So, a limiting value (like *NormMax* in the PDTC structure) is not easy to define.

The received sequences are first quantized by MATLAB on PC and then, quantized forms of the sequences are given to the evaluation platform through the UART (to be described in Chapter 5). Inside the channel estimator, these quantized forms are used instead of the real values. After that quantization process, we get the L-bit representation sequences of the in-phase and quadrature components of the received values. These two sequences are handled

through two parallel and identical processes. Since they are identical, we give details about just the operations on the in-phase component. It must be noted that the same is carried out on the quadrature component.

For the first iteration of the channel estimator, a zero forcing equalization is applied. The reason of using the zero forcing algorithm is to reduce the complexity and thus the resource consumption. The first iteration of the estimation process is carried out as the following. The channel is estimated at the pilot locations using a zero forcing algorithm. Since the modulation is BPSK and the pilot symbol values are always "1", the received values at the pilot locations are regarded as the fading coefficient at those time instants. After that, the group of each pilot (the nearest $M-1$ locations to that pilot, shown in Figure 4.6) is assumed to have the same fading coefficient as the pilot. This first channel estimation iteration is shown in Figure 4.7.

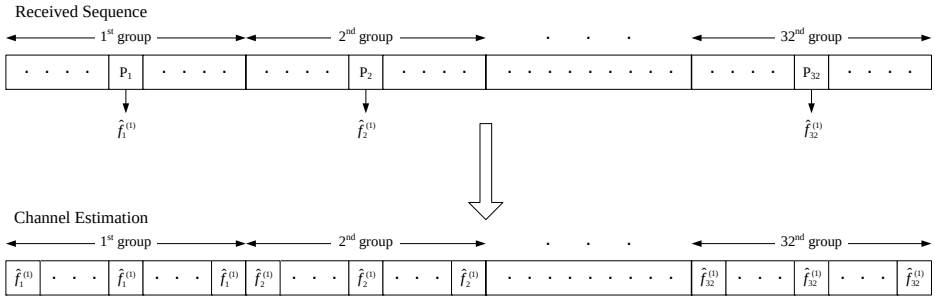

Figure 4.7: First iteration of the channel estimator (constant fading coefficient during one group period).

In the second (and next) iterations, the step size β, is included in the LMS-based estimation algorithm. In our fixed-point domain, the β value is represented with 6 bits, 1 for integer part and 5 for decimal part. The integer bit is used to represent the sign and is always 0 since $\beta > 0$ [15]. As β is a multiplier in (3.5) and (3.7), the representation bit number is increased during this iteration due to this multiplication. Not to lose the accuracy, the received values and the fading coefficients estimated in the first iteration $\{\hat{f}_k^{(1)}\}$ are represented by $L+5$ bits. The second iteration of the channel estimation algorithm operates as follows. The forward and backward channel estimates are initialized using the first estimation results. To get a $L+5$ bit representation, the first estimation results are shifted 5 times to the left and without changing

the sign of that estimation result.

$$\hat{h}_1^{(2)} = \hat{f}_1^{(1)} \text{ x } 100000, \tag{4.22}$$

$$\hat{g}_{544}^{(2)} = \hat{f}_{544}^{(1)} \text{ x } 100000. \tag{4.23}$$

After that shifting operation, the resultant values possess a decimal part of 5 bits and an integer part of L bits. Hereafter, we will denote this representation with "L.5". The integer part remains as L bits since the integer part in the step size is always 0, i.e., $\beta < 1$ in our design.

The second factor that might effect the representation consistency is the feedback coming from the turbo decoder. Due to the complexity avoidance issue, a hard-decision mechanism is used on the feedback path. This choice brings no more additional computations in the LMS algorithm, but just an inversion if the calculated LL value is negative[3]. Since the inversion process does not affect the representation bit number, the notation L.5 keeps its form.

In the expressions (3.6) and (3.8), the error terms are computed by a subtraction operation. For this operation to work properly the received sequence $\{r_k\}$ is also represented in L.5 bits in this iteration. However, this sequence was represented by L bits in the first iteration. Enlargement to L.5 bits is achieved in the same manner as in the initialization of $\{\hat{h}_k^{(2)}\}$ and $\{\hat{g}_k^{(2)}\}$ sequences. After the representations in the subtraction is matched, a *clipsubtract* operation is applied, as described in Section 4.1.4, different than the ordinary subtraction operation. However, this time, the $plus\_inf$ and $minus\_inf$ values change to $2^{L+5-1} - 1$ and $-\left(2^{L+5-1} - 1\right)$, respectively.

After having a consistent representation during this iteration, the multiplication (second) operations in (3.5) and (3.7) can be resolved easily. If we call these terms as $mult_{f,k}^{(q)}$ and $mult_{b,k}^{(q)}$ for forward and backward error terms, in the second iteration we have

$$mult_{f,k}^{(2)} = \beta e_{f,k}^{(2)} \hat{a}_k^{(1)} \tag{4.24}$$

$$mult_{b,k}^{(2)} = \beta e_{b,k}^{(2)} \hat{a}_k^{(1)}. \tag{4.25}$$

In the multiplication equations given above, the $\hat{a}_k^{(1)}$ term has no effect on the bit number of the result but affects the sign. So, we will omit this term. The error terms ($e_{f,k}^{(2)}$ and $e_{b,k}^{(2)}$)

[3] Since $\{\hat{c}_k^{(q)}\} \in \{-1, 1\}$, the product is affected just in sign.

have the representation of $L.5$, while β has 1.5. Therefore, the multiplication result has the representation of $(L+1).10$. However, one of the bits in the integer part is unnecessary in this representation since the integer bit of the β term is always "0". After removing this unnecessary bit, we get a representation of $L.10$. These multiplication results are summed up with $\{\hat{h}_k^{(2)}\}$ and $\{\hat{g}_k^{(2)}\}$ to get the resultant values $\{\hat{h}_{k+1}^{(2)}\}$ and $\{\hat{g}_{k+1}^{(2)}\}$. To match the resultant representation, the multiplication terms are needed to be represented in the form of $L.5$. For that reason, the last (rightmost) 5 bits are discarded. Although discarding these bits decrease the precision of the multiplication result, this effect can be neglected when the quantization process before the decoding operation is taken into account[4]. The representation used in each step is shown in Figure 4.8 for the first two steps in the forward LMS computation. The final summation is realized with a *clipsum* operation, $plus\_inf = 2^{L+5-1} - 1$ and $minus\_inf = -\left(2^{L+5-1} - 1\right)$.

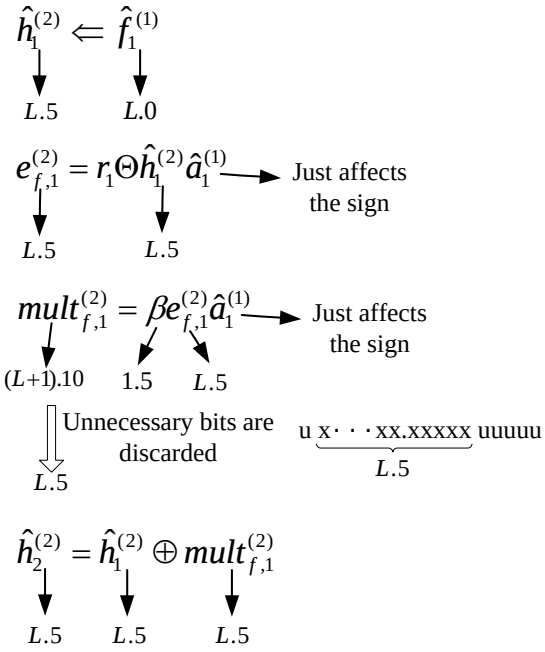

Figure 4.8: The representations in the fixed-point domain of the forward branch of the $2-way$ LMS algorithm.

For the following iterations (if any), the operations described for the 2^{nd} iteration are carried out in the same manner. Although the precision is increased by 5 bits in these iterations, this does not affect the storage capacity. This increase in the representation bit number

[4] The decoders operate on metric values represented in very few number of bits

is effective just for the computation registers. The results are stored in the memories with the width of L bits by discarding the 5 decimal bits at the rightmost place for these iterations. After computing the forward and backward estimation sequences $\{\hat{h}_k^{(q)}\}$ and $\{\hat{g}_k^{(q)}\}$, the final estimates of the fading channel coefficients are calculated as in (3.9). To realize this averaging operation, summation result is first expressed by $L + 1$ bits without any loss. Instead of division by 2, the rightmost bit is discarded and the result is expressed in L bits again.

The duration of the estimation process is an important parameter for the system. To reduce this duration, some implementation tricks are used as follows. In the iterations other than the first one, the 2-way LMS algorithm is used for the estimation issue. As it is obvious in (3.5) and (3.7), the forward and backward recursive computations are independent of each other. Therefore to speed up this LMS algorithm, we can apply the CTT (center to top) algorithm described in Section 4.1.1. By applying the CTT algorithm the estimation process of a sequence with length 544 is completed in 544 cycles rather than 1088.

Up to this point the computations are described for the in-phase (I) components. After carrying out the same operations on the quadrature components (Q) we can calculate the LL values to feed the turbo decoder.

As a design consideration we choose $L > K$, i.e., the operations carried out in channel estimation operations are more precise when compared to the decoding ones. The reason of that choice can be explained as follows. Implementation of a channel estimator on FPGA is realized with the use of sequential logics mostly. Hence, the clock speed and the resource consumption of the design are not affected so much for increasing values of L. On the other hand, the observations received by the turbo decoder is quantized to K bits in the *observation quantization* process described in Section 4.1.2. Not to affect the performance of the decoder, choosing an L value larger than K seems more reasonable when the performance of the overall system is considered. However, choosing a very large L value will bring some unnecessary complexity in the estimator side since the representation bit number is eventually rolled back for the decoder operations.

4.2.3 *LL* Computation

As seen in Figure 3.2, after the channel estimation process, the *LL* values to be used by the turbo decoder are computed. During this computation the channel estimates $\{\hat{f}_k^{(q)}\}$ are used together with the received sequence $\{r_k\}$. As mentioned before, both of the sequences consist of complex values and each value is expressed as

$$r_k = r_{k,I} + ir_{k,Q}, \tag{4.26}$$

$$\hat{f}_k^{(q)} = \hat{f}_{k,I}^{(q)} + i\hat{f}_{k,Q}^{(q)}. \tag{4.27}$$

The *LL* computation is achieved in [16] by multiplying the received sequence by the complex conjugate of the channel estimations and with a factor of $2/N_0$. This multiplication factor comes from the $L_c/2$ term in branch metric equation given in (2.11) where $L_c = 4E_s/N_0$ and $E_s = 1$ in BPSK modulation. As a result, the *LL* computation is accomplished in the way

$$LL_k = \frac{2}{N_0}\Re\left\{r_k \hat{f}_k^{*(q)}\right\} \tag{4.28}$$

$$= \frac{2}{N_0}\Re\left\{(r_{k,I} + ir_{k,Q})\left(\hat{f}_{k,I}^{(q)} - i\hat{f}_{k,Q}^{(q)}\right)\right\} \tag{4.29}$$

$$= \frac{2}{N_0}\left(r_{k,I}\hat{f}_{k,I}^{(q)} + r_{k,Q}\hat{f}_{k,Q}^{(q)}\right). \tag{4.30}$$

However, since the turbo decoder used in the system is a max-log-MAP decoder, the exact values are not needed for the proper operation of this decoder [19, 20]. That's why, the term $2/N_0$ in (4.30) is dropped to reduce the computational complexity. After modifying the *LL* computation operation, we get

$$LL'_k = r_{k,I}\hat{f}_{k,I}^{(q)} + r_{k,Q}\hat{f}_{k,Q}^{(q)}. \tag{4.31}$$

In (4.31), there exist two multiplication terms which operate on two L-bit numbers. The results of these multiplications can be represented by $2L$ bits without any loss in accuracy. Since the representations will be represented in K bits after the *observation quantization* process, there is no need to represent the *LL* values with $2L$ bits. That's why, the multiplication results can be represented in any number of bits B, with $B > K$ not to affect the performance of the decoder. As a design choice, the multiplication results are considered as L bits by regarding the rightmost (least significant) L bits.

CHAPTER 5

TESTBED PLATFORM

In this thesis the motivation is basically the hardware implementation of a parallelized turbo decoder and a channel estimation algorithm as described in Chapters 2 and 3. In order not to deal with problems that one faces with actual wireless modules, the designs are carried on a stand-alone operating environment. Expected characteristics of a real environment is realized on the FPGA with out any loss of generality.

5.1 Testbed Hardware

We designed our system and carried out the experiments on the ML-402 Virtex-*IV* Evaluation Platform with the help of some test and measurement equipments like the oscilloscopes and function generators. The overall testbed setup is given in Figure 5.1.

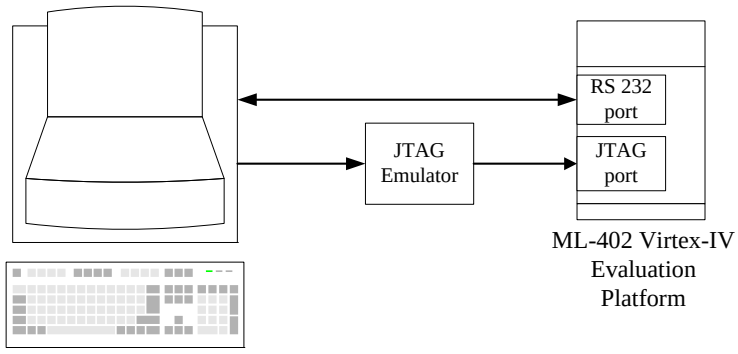

Figure 5.1: The overall testbed.

The communication to the board is accomplished with two connections. First one is the connection between the USB port of the PC and the JTAG port of the evaluation board via a JTAG emulator. This connection is used to write a ".BIT" file to the FPGA. Also, this connection is used by the ChipScope during the debugging process (to be described in Section 5.2.3). The second connection is maintained between the *RS*232 ports of the PC and the evaluation board. This connection is controlled by MATLAB on the PC side and used to collect data during run time.

5.1.1 ML-402 Evaluation Platform

A Field Programmable Gate Array (FPGA) is a kind of logic chip that can be configured by the user after its manufacturing. Unlike a logic gate, which has a fixed function, an FPGA has an undefined function at the time of manufacturing. FPGAs are very similar to PLDs (programmable logic device), but they differ at the number of gates they contain and their memory structures. Although PLDs are limited to hundreds of gates, FPGAs may contain up to millions of gates. Since the FPGAs have volatile memory due to their static random access memory (SRAM) based structures, they need to be programmed after power up. FPGAs are composed of *configurable logic blocks* (CLBs). These logic blocks can be configured to perform complex combinational functions or to implement simple logic gates like *AND* and *OR*. A common CLB contains the elementary structures such as look-up tables (LUTs), flip-flops, multiplexers, etc. as shown in Figure 5.2. Other than CLBs, FPGA may contain Random Access Memory (RAM) modules for data storage tasks.

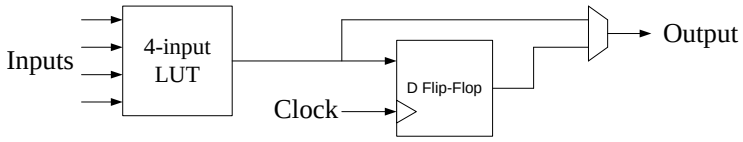

Figure 5.2: A common CLB architecture.

In Xilinx FPGAs, the CLBs are called "slices" and they differ from the common CLBs with their contents. The slice structure of a Xilinx FPGA is shown in Figure 5.3. These logic blocks are connected to each other with programmable switches. If there is a relation between

slices then these switches will be ON, else OFF. When it is needed to store a data, the output of LUTs can be multiplexed to the slice flip-flops. Other than storage, flip-flops introduce pipelining by breaking an operation into two parts in order to obtain a fast propagating function. As a result of having these combinatorial logic parts, FPGAs have a great potential in operating parallel processes simultaneously.

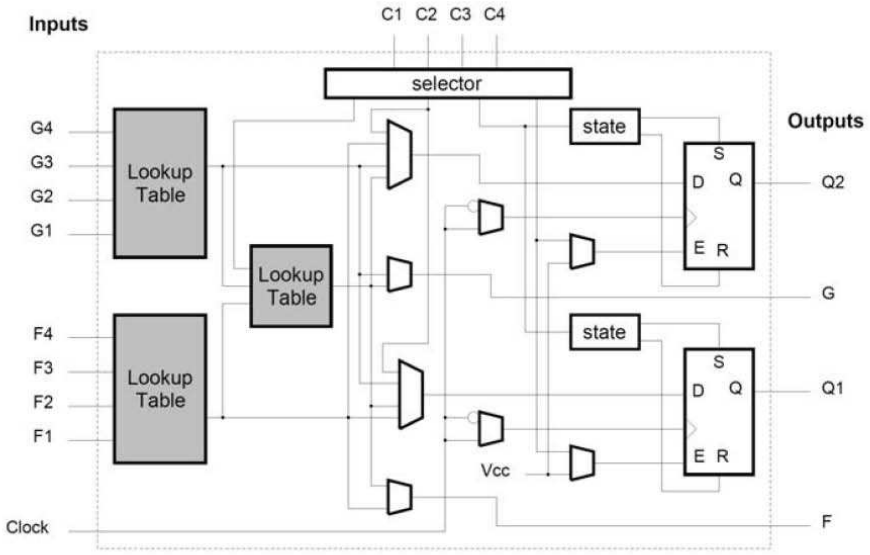

Figure 5.3: Slice structure for Xilinx FPGAs.

Our evaluation platform is the Xilinx ML-402 Evaluation Platform. It contains a XC4VSX35-FF668-10 Xilinx FPGA chip. The basic specifications of this chip is given in Table 5.1. The block RAM resources are 18 kb true dual-port[1] RAM blocks, programmable from 16k x 1 to 512 x 36, in various depth and width configurations by concatenating the blocks vertically and/or horizontally as desired.

In addition to the FPGA chip, ML-402 contains several other features and interfaces. A short list of these interfaces is given below [22].

- **DDR SDRAM :** The board includes an external 64 MB of DDR SDRAM using two Infineon HYB25D256160BT-7 chips. Each chip has 16-bits wide data port and two of

[1] Dual-port RAMs contain two ports which operate independently and synchronous with two different clocks. Two different pieces of information can be written (read) to (from) two distinct addresses of the RAM concurrently.

Table 5.1: Specifications of a XC4VSX35-FF668-10 FPGA chip

Structure	Count	Explanations
Slice	15,360	Function generators, storage elements,multiplexers, etc.
BRAM (kb)	3,456	Variable size RAM blocks for data storage
Xtreme DSP Slice	192	18-bit x 18-bit signed multiplier and 48-bit accumulator

them form a 32-bit data bus which is capable of running up to 266 MHz [22]. In the presence of a microprocessor, these RAMs can be used to store data for stand-alone operations. Besides, these RAMs can be used as the processor memory which includes instructions in the presence of a soft microprocessor core.

- **ZBT Synchronous SRAM :** The board contains a 256K x 36 bits synchronous ZBT RAM which provides a high speed low-latency external memory to the FPGA. This module is also available for data storage.

- **RS**-232 **Port with Direct FPGA connection :** The ML-402 board contains an DB-9 serial port which allows the FPGA to communicate with other devices. An interface chip changing the voltage-levels are also included. The RS-232 connection is one of the most widely used communication methods and is known for its low-weight transmitter/receiver structure. In our setup, RS-232 is used for simulation purposes. In the run-time, the FPGA communicates with the PC through this port.

- 10/100/1000 **Tri-Speed Ethernet PHY :** The board contains a Marvell Alaska PHY device operating at 10/100/1000 Mbps. This port makes it available to reach the board through ethernet connection. However, to accomplish that, a small processor and an ethernet controller are needed.

- **Compact Flash and SystemACE:** The board contains a Xilinx System ACE CompactFlash (CF) configuration controller. A maximum of eight configuration images on a single CF card can be supported by the SystemACE controller. Using the switches on the board, the configuration file to be loaded on the FPGA can be selected.

- **Differential Clock Input And Output With SMA Connectors :** High precision clock signals can be fed to FPGA by 50 Ω SMA connectors. This structure allows the FPGA

to be fed by function generators.

5.2 Software Used For Simulation, Implementation, and Debugging

In this section, some of the software programs used during the studies will be described very briefly. These programs are used at different phases of the setup procedure. We can group these phases as simulation, implementation, and debugging.

5.2.1 ISE Design Suit and XST

ISE (Integrated Synthesis Environment) is the *integrated development environment* (IDE) designed by Xilinx. This environment provides a GUI (graphical user interface) to the designers. XST (Xilinx Synthesis Tools) is the software used for synthesis, implementation, and loading a project on a Xilinx FPGA. During implementation of the system on a Xilinx FPGA, we have used Xilinx ISE software. The implementation steps of a general Xilinx project is divided into steps, each having certain inputs and outputs. Brief descriptions of these synthesis, implementation, and loading steps are as follows [23].

5.2.1.1 Synthesis

The project is firstly synthesized before the implementation process. Synthesis acts an important role that is analogous to the compilation process in a software design. In this step, the hardware description language (HDL) is converted into register transfer level (RTL) descriptions. In this description level, the design is represented in terms of logical blocks, like gates. The operation following that conversion is the optimization process. In this process, the unused signals are removed and the number of slices is reduced by removing the slices which do the same job with an existing one. Xilinx XST produces a ".NGC" file in binary format.

5.2.1.2 Translation

Translation is the first step of the implementation process. In this step, a native generic database (NGD) file is generated by combining all of the input netlists and design constraint information provided by ISE. The generated file contains all the information needed for mapping the design on the destination FPGA.

5.2.1.3 Mapping

The mapping process starts to operate after the translation finishes its job and creates the NGD file. In this step, a design rule check (DRC) is run over this file and the design is mapped into the Xilinx FPGA as specific hardware blocks. It is possible to see an error message if there is a mismatch in the specified constraints. This kind of errors mostly occurs when the LVDS (low voltage differential signaling) drivers are used in the design. The results of the mapping process are output to a Xilinx native circuit description (NCD) file.

5.2.1.4 Place and Route

After the mapping of the design, the place and route (PAR) step starts work. This operation places and routes the design in accordance with the previously generated NCD file. Constraints have a great importance in this step since the design is placed on the FPGA to satisfy the given constraints. This importance can be exemplified as the following. If a constraint on the clock speed is given to be at least 100 MHz, the placement of the blocks on the FPGA slices are done in a way to satisfy this clock speed in the worst case. As the output, a NCD file which matches all the design specifications is generated.

5.2.1.5 BitGen

BitGen is the programming file generator for Xilinx FPGAs. This program is embedded in the Xilinx ISE tool. After the implementation of the design finishes, BitGen accepts the previously generated NCD file and produces a ".BIT" file which is in the desired format for programming a Xilinx FPGA via a JTAG connection.

5.2.1.6 Impact

Impact is another program which is embedded in the ISE tool. Other than loading the FPGA, it has some additional features like [24],

- verify configuration data for single devices,
- create PROM, SVF, STAPL, System ACE CF, and System ACE MPM programming files.

As the final step, Impact loads the ".BIT" file to the Xilinx FPGA. If all the given steps are completed successfully, the logic blocks in the FPGA are arranged in the correct form to perform the desired function described in the HDL.

5.2.2 ModelSim

ModelSim is an advanced simulation and debugging tool designed for ASIC (application-specific integrated circuit) and FPGA designs. ModelSim has 3 major distributions:

- ModelSim PE (Personal Edition),
- ModelSim SE (Special Edition),
- ModelSim LE (Linux Edition).

For some FPGA vendors, there exist some special distributions of ModelSim. Modelsim XE is the distribution of ModelSim for Xilinx ISE. It is replaced with the original Xilinx simulation tool after installation and fully integrates to ISE. This distribution has two different license choices. One of the licenses is free and can be obtained from the webpage of Xilinx through a member account. The other license is a full one which simulates the codes 1000 times faster when compared to the free licensed version of ModelSim.

ModelSim simulations can be grouped into two, behavioral simulations and post-route simulations.

1. **Behavioral Simulation :** Behavioral simulation simulates the behavior of the code without considering the gate delays and clock skews. The results of this simulation are

"perfect" when compared to the real operation of the design. Synthesizing the design is enough for this simulation to run.

2. **Post-Route Simulation :** This simulation can be run after the place and route step of the implementation. That's why, post-route simulation is highly reliable and generally simulates the design in the best way. However, this simulation method is slower when compared with the behavioral one. In this simulation model, logic block structure of the design on the actual FPGA is simulated. So, latency introduced by the gates and skew in the clock can be observed. Although the design is simulated in the best way by this process, it becomes harder to debug the code since the simulation runs on the optimized form of the design.

5.2.3 ChipScope

ChipScope is a real-time debugging tool designed by Xilinx. This tool embeds a special block in the FPGA in order to track the signal changes and report them in through the JTAG emulator in run-time. The signals to be observed in real-time have to be determined before synthesizing the project since the ChipScope uses some additional slices and RAM on the FPGA.

There may be some special cases that the overall design functions in an unwanted manner. Simulations may not be adequate to see these problematic conditions. In that case, ChipScope may be used to collect the relevant data to resolve the problem. In our studies, we have used ChipScope whenever we face with such a situation.

5.2.4 MATLAB

MATLAB is used in different phases of this thesis study. Before implementing the system on the FPGA board, all the blocks are first generated in MATLAB environment. These blocks are simulated until the desired results are obtained and the implementation process on the FPGA board starts after then. The encoder and decoder blocks, and the channel estimation algorithm, described in Chapters 2 and 3 respectively, are implemented on MATLAB to see how they behave in the existence of an AWGN (additive white Gaussian noise) and fading channels. The MATLAB environment presents a fast verification method for the decoding and

estimation algorithms. To collect data from the board and to change the design parameters in the run-time, MATLAB is used on the PC side. For these run-time purposes, MATLAB uses the RS232 port of the computer. The collected data on the PC side is processed to obtain performance results and related figures again by MATLAB.

5.3 Miscellaneous Components

In this section, we will describe the procedure carried out during the test phase of the proposed systems and algorithms.

The test setups are shown in Figures 5.4 and 5.5. During the performance tests of PDTC decoder architecture under AWGN, the setup given in Figure 5.4 is used. In this setup, the most compelling part is the generation of the AWGN channel on an FPGA. The performance tests of the PDTC decoder in a fading channel with a channel estimator is completed in the setup in Figure 5.5. In this setup, fading channel is generated on the PC side by MATLAB and the resulting received signals are directly given to the ML-402 board.

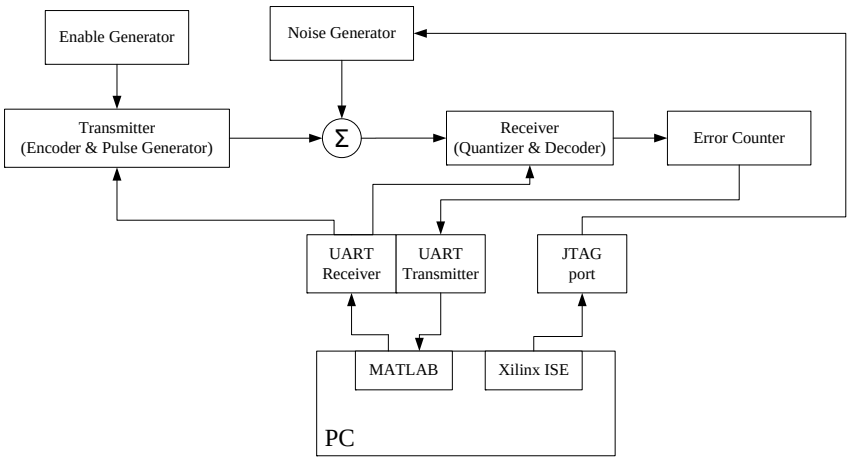

Figure 5.4: The test setup used to see the performance of the proposed PDTC decoder structure in an AWGN channel.

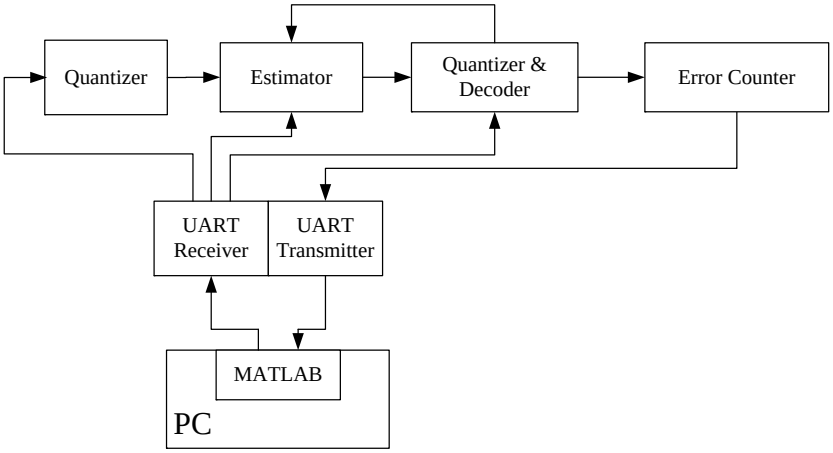

Figure 5.5: The test setup used to see the performance of the proposed joint estimation and decoding algorithm in a fading channel.

5.3.1 Enable Generator

The “Enable Generator” module produces an enable signal periodically. This period is adjusted not to interrupt the decoder operations. The generated enable signal triggers the encoder block and transmission of a packet starts after that point.

5.3.2 Encoder and Pulse Generator

“Encoder and Pulse Generator” module is triggered by an enable signal produced by the “Enable Generator”. After this enable signal is received, the module starts to produce the encoded version of a known sequence. When the sequence is ready to be transmitted, the BPSK modulation is applied to this sequence with an amplitude value of “A_s” (to denote the signal amplitude). This value is given to the system by the UART module to achieve some test parameters by varying the SNR value. Giving an amplitude of A_s results in a BPSK modulated signal with the energy of $E_s = A_s^2$.

5.3.3 Noise Generation

To obtain the performance of the proposed decoder architecture in an AWGN channel, the most important point is to realize such a channel on the FPGA. In our study, a random data generation algorithm is needed in order to generate a realistic environment. However, a true random number generation is impossible since the state of the data generator affects the future data and future states of this generator block, and this effect disrupts the randomness of the generated data. There are many different methods for generating random data and they vary as to how unpredictable the generated data pattern is or how statistically random the data is. In most of the generators, an initial state is assumed to begin from. This initial state is called the "seed", and the numbers are generated by the usage of this *seed*. Although this method creates long runs of data with good random properties, after some point the sequence repeats itself. That's why, the generators using this kind of number generation methods are called "pseudo-random number generators".

One of the most popular pseudo-random number generating algorithms is the "linear feedback shift register" (LFSR) method. The word "linear" comes from the binary linear operations (*XOR*) used in the method. This generator updates the register bit contents by shifting the register in a way and a feedback mechanism operates to update the discharged content. An LFSR with a register width of 16 bits is shown in Figure 5.6.

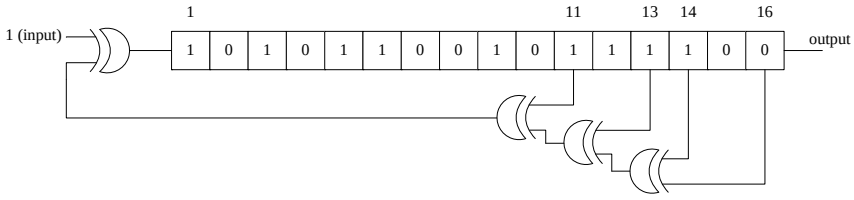

Figure 5.6: An LFSR of width 16 bits with a seed of "1010110010111100".

The arrangement of taps for feedback in an LFSR can be expressed as a polynomial with coefficients "0" and "1". This polynomial is called the *feedback polynomial* or *characteristic polynomial*. If a content is not used in the feedback path, then the corresponding coefficient in the polynomial is 0. The feedback polynomial of the LFSR given in Figure 5.6 can be written

as,

$$x^{16} + x^{14} + x^{13} + x^{11} + 1 \tag{5.1}$$

where the constant term "1" in the polynomial corresponds to the input to the first bit ($x^0 = 1$). In a feedback polynomial, the first bit is always connected as input and the last bit is always included as a tap. The algorithms running based on a seed repeats itself after a while, and so does LFSR since the register used has a fixed width. The maximum number of obtainable different numbers from an LFSR with shift register of width n can be told to be $2^n - 1$. An LFSR providing this maximum number of different outputs is called as a *maximum-length* LFSR. This maximum number of different obtainable output values is called the *period*. In other words, the LFSR of width n is called a *maximum-length* LFSR if its period is equal to $2^n - 1$. To obtain a *maximum-length* LFSR, some restrictions exist on the feedback polynomials, such as

- Number of taps must be even.
- The set of taps must be relatively prime.
- There can be more than one maximum-length tap sequence for a given LFSR width.
- For an n-bit LFSR, if the tap sequence $[n, a, b, c, 0]$ is a maximum-length tap sequence then the sequence $[n, n - c, n - b, n - a, 0]$ constructs a maximum-length LFSR, too.

Some feedback polynomials for different register widths are listed in Table 5.2 with the corresponding repetition periods. All these polynomials generate maximum-length LFSRs, but they are not the only ones for the given widths.

Table 5.2: Some LFSR feedback polynomials with varying width of shift registers.

Bits (n)	Feedback polynomial	Period ($2^n - 1$)
10	$x^{10} + x^7 + 1$	1024
11	$x^{11} + x^9 + 1$	2047
12	$x^{12} + x^{11} + x^{10} + x^4 + 1$	4095
13	$x^{13} + x^{12} + x^{11} + x^8 + 1$	8191
14	$x^{14} + x^{13} + x^{12} + x^2 + 1$	16383
15	$x^{15} + x^{14} + 1$	32767
16	$x^{16} + x^{14} + x^{13} + x^{11} + 1$	65535

Operation steps of an LFSR with a 16-bit register are shown in Figures 5.7, 5.8, and 5.9. The feedback polynomial given in (5.1) is used in this design. The tap locations are shaded in these figures and *XOR* operations are depicted with $\oplus$. The seed of the LFSR is given as "1101001000010001". This seed can be chosen for any bit sequence with length of 16 without any restrictions. The shifting operation and linear operations are triggered by an external clock source. After the bit contents in the tap locations are *XOR*ed (input "1" is included), LFSR waits for an external clock for the shift operation. When the external clock triggers the LFSR, the contents are shifted once to the MSB side[2] and the result of *XOR* operations is written to the first bit location. The content in the 16^{th} bit locations comes out of the register and this content is given to the output.

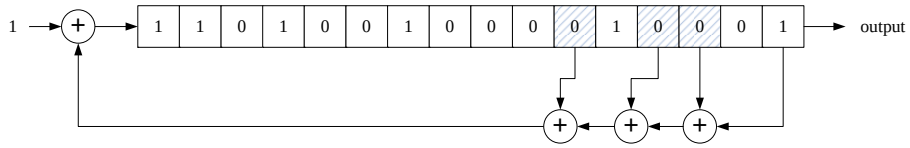

Figure 5.7: An LFSR of width 16 bits in the initial state with a seed of "1101001000010001".

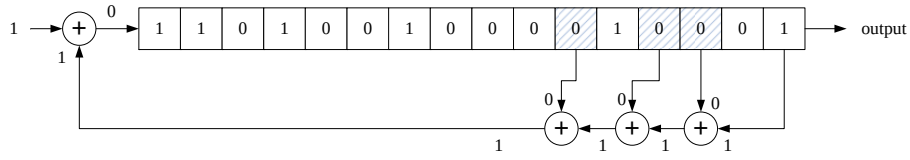

Figure 5.8: After completing the *XOR* operations, LFSR waits for an external trigger to update the register contents.

The VHDL code that performs an LFSR with a feedback polynomial as given in (5.1) is as follows.

...

[2] The MSB side for the given figures are on the right-hand side, since the 16^{th} bit is in the rightmost location.

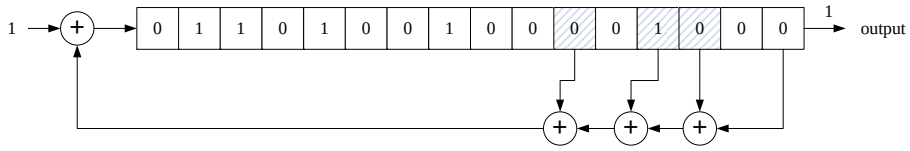

Figure 5.9: When the trigger occurs the contents are shifted to the right, left-most content is updated with the result of *XOR* operations, and the right-most content is given to output.

```
    signal reg_content   : std_logic_vector(15 downto 0);
    constant signal seed : std_logic_vector(15 downto 0) := "1101001000010001";
  ...
process(clk)
begin
    if rising_edge(clk) then
        if rst = '1' then
            reg_content <= seed;
        else
            reg_content <= (reg_content(5) xor reg_content(3) xor reg_content(2) xor
                            reg_content(0) xor '1') & reg_content(14 downto 0);
        end if;
    end if;
end process;
```

In probability theory, the "central limit theorem" (CLT) states that the sum of a sufficiently large number of independent random variables, each with finite mean and variance, will be approximately normally (Gaussian) distributed [15]. If we implement sufficiently many LFSRs and sum up the output values of them, this variable is expected to have distribution very close to Gaussian (normal) distribution.

In our design, 40 LFSRs, all having 16-bit width and different seeds, are implemented on the FPGA. The seeds are generated by MATLAB and written to a file. When the synthesis operation starts, the tool reads the data from that file and initializes all of the LFSRs with the desired seeds. A clock, which is 4 times faster than the overall system clock, is used to trigger this LFSR block. The VHDL code is as follows.

```
  ...
    constant reg_no   : integer := 40;
```

```
  ...
process(clkX4)
begin
    if rising_edge(clkX4) then
        if rst = '1' then
            for ii in 1 to reg_no loop
                reg_content(ii) <= seed_file(ii*16 downto (ii-1)*16+1);
            end loop;
        else
            for jj in 1 to reg_no loop
                reg_content(jj) <= (reg_content(jj)(5) xor reg_content(jj)(3)
                                    xor reg_content(jj)(2) xor reg_content(jj)(0)
                                    xor '1') & reg_content(jj)(14 downto 0);
            end loop;
        end if;
    end if;
end process;
  ...
```

In one cycle of "clk", "clkX4" completes 4 cycles and 4 random numbers are generated. By summing up these 4 numbers, we generate a *pseudo-random noise* which is updated synchronously with the system clock. The main idea behind using a faster clock for LFSR operations is not to consume more resources for implementing an LFSR block with 160 registers. After the reset operation, the registers are updated from a seed file, which contains 40 random seeds generated by MATLAB.

As the generated numbers are sum of 40 bits in each fast clock cycle, their values vary between 0 and 40. Let x be the random variable generated by summing up these 40 bits. The mean of x is found as,

$$E\{x\} = E\left\{\sum_{reg\_no} b_0\right\} \tag{5.2}$$

where b_0 is the right-most bit of LFSRs and the value of this bit is either 1 or 0. Since the LFSRs have independent seeds, they generate independent random numbers. So,

$$E\{x\} = \sum_{reg\_no} E\{b_0\}, \tag{5.3}$$

$$E\{x\} = \sum_{reg\_no} \frac{1}{2} \tag{5.4}$$

$$E\{x\} = \frac{reg\_no}{2}. \tag{5.5}$$

As we use 40 registers, at the output we get a random variable with mean, $\mu = 20$, and

the distribution of this random variable is approximately Gaussian. To realize the effect of an AWGN channel, the mean is normalized to 0 by subtracting 20 from the number generated at each tick of "clkX4". It must be noted that subtracting a constant does not affect the variance of the random variable. When we sum up these 4 generated random numbers during one clock period, the resultant random variable has a mean equal to 0. However, the variance of this random variable will be affected from this summation operation. Figure 5.10 shows the distribution of the proposed noise generator at each "clk" instant. This figure shows how the histogram of the noise generator comes out and approximates the Gaussian distribution given in Figure 5.11.

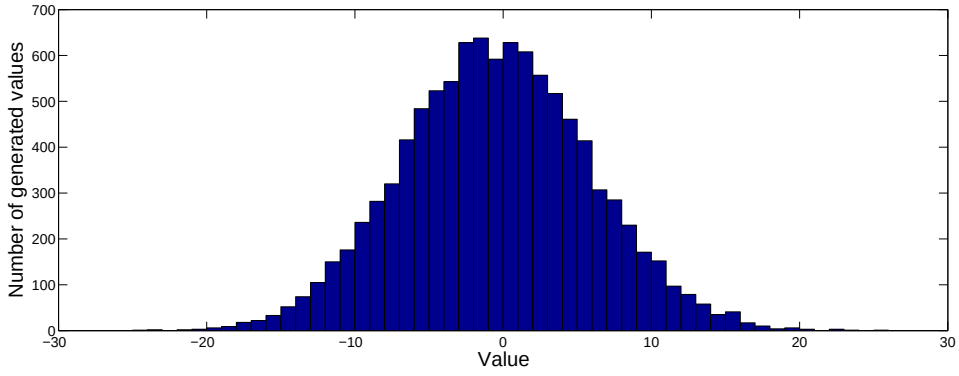

Figure 5.10: The histogram of the generated numbers by the proposed pseudo-random number generation method over 10000 samples.

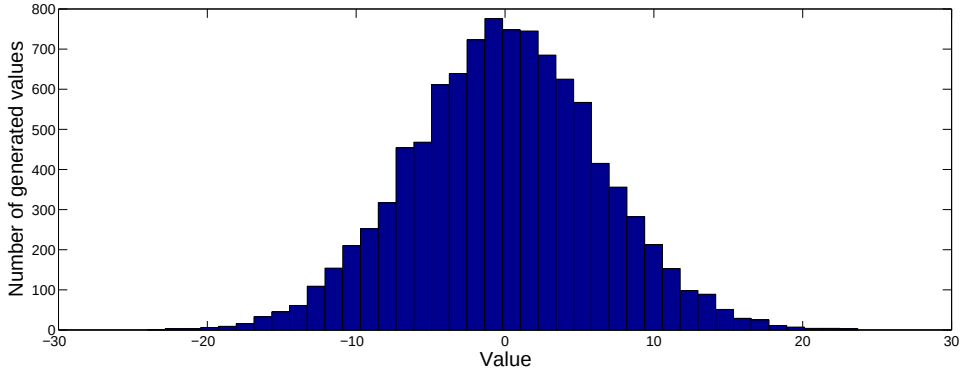

Figure 5.11: The histogram of a noise sequence generated by MATLAB's *randn()* function. The output vector of *randn()* function is multiplied by $\sqrt{40}$ to match the variances.

When we simulate this LFSR implementation on MATLAB, the variance of the generated noise turns out to be 40. That value is the noise power of the AWGN channel created on

the FPGA board and is used while producing the test results.

5.3.4 Fading Channel Generation

To obtain the performance results of the proposed joint channel estimation and decoding structure in a fading channel model, we need to realize such a fading channel or simulate its effects. At first, we had decided to implement a fading channel on FPGA by using the autoregressive (AR) model with an order of one [25]. However, this was a perfect match for the applied LMS algorithm. Due to this matching, the estimator is anticipated to operate with a better performance when compared to the real world fading case. That's why, instead of realizing a fading channel effect on the FPGA board, we simulate a Rayleigh fading model on MATLAB. The modulated signals are passed through a fading channel, noise is added on them, and then the resultant data is given to the board as the received sequence.

In MATLAB, to generate a Rayleigh fading channel object, the function *rayleighchan()* is used. This function takes two values as input, one is the sample time in terms of seconds (T_s), and the other one is the maximum Doppler frequency in Hz (f_D). The generated object is a "single-path" Rayleigh fading channel with the given features. Figure 5.12 shows the path gains of a channel generated by the *rayleighchan()* function.

The generated Rayleigh fading channel object is used to realize the channel effect on a transmitted signal. To realize the effect of channel on a transmitted signal, *filter()* function is used. After the fading effect, AWGN is added to the resultant sequence with proper variance to match the desired SNR value. This sequence is quantized before giving to the FPGA. This quantization process is carried out with optimum quantization decision points. To find the optimum decision points the training sequences with 1000-packet length is used for each SNR. The location of these optimum points vary with the SNR value. After obtaining the optimum quantization decision points for each SNR, the received packet is quantized according to this optimum values. After all these operations, the sequence is given to the FPGA platform through UART module for testing estimation and decoding.

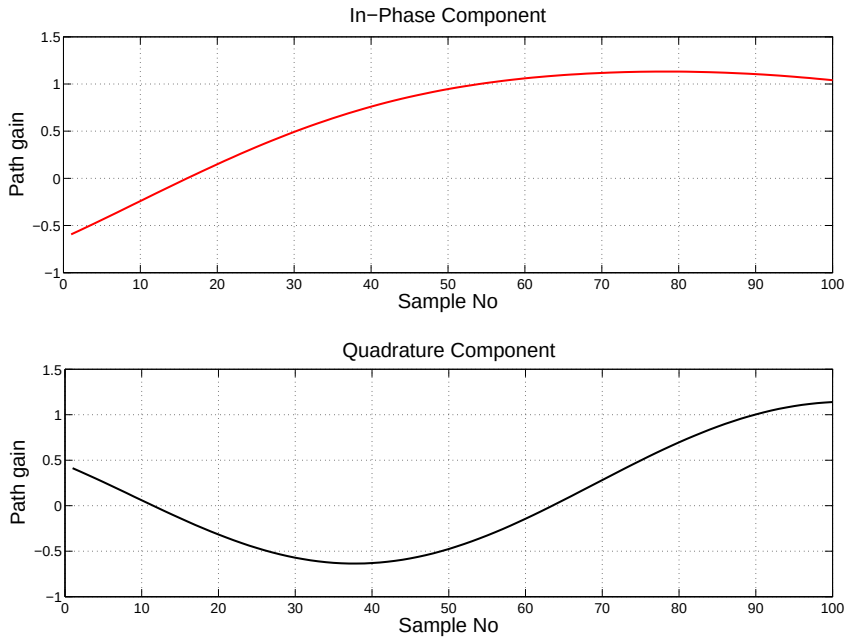

Figure 5.12: The in-phase and quadrature components of a Rayleigh fading channel with the normalized fading rate $f_D T_s = 0.01$, over a sequence of length 100.

5.3.5 Error Counter Module

The *error counter module* counts the number of errors occurred in the packet and the number of packets which is not decoded correctly. The data to be coded in the encoder side is also known by this module in both of the test setups. The operation of this module can be described as follows. Whenever the decoder block starts to produce the bit estimates, this module starts to check whether the estimate is correct or not and keeps the number of wrong estimates until the number of packets reach a predetermined value. Besides, it also counts how many packets are decoded incorrectly[3]. These number of wrong estimated bits and packets are fed to the UART transmitter module for reporting to the PC.

5.3.6 Communication with PC

Communication between the testbed and PC is accomplished by UART (Universal Asynchronous Receiver Transmitter). In our design, a full duplex UART module is implemented and used in conjunction with RS-232. Simply, a UART transmitter takes a parallel data and

[3] Even if only 1 bit is estimated wrong, the whole packet is decoded incorrectly.

transmits it bit by bit in a sequential configuration. In the receiver side, if a new coming data is detected, it receives data bit by bit in serial and translates the received sequence into a parallel form. The conversions between serial to parallel and parallel to serial are accomplished by the use of shift registers.

This protocol is called “asynchronous” because the transmitter does not send any clock signal to the receiver side. The transmission process may start whenever the transmitter sends a start bit. After the transmission of the start bit, the data is transmitted beginning from the least significant bit to the most significant bit. Optionally, a parity bit can also be added to the end of the data for error check. A stop bit finishes the transmission of a word[4]. Figure 5.13 represents the alignment of these bits.

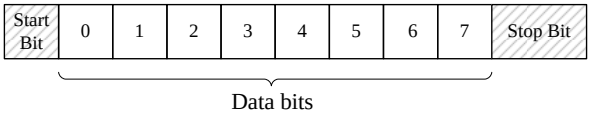

Figure 5.13: The bit alignment of a word used in UART transmission.

The communication between the testbed and PC is handled by a protocol. In this protocol, the register names (or addresses) are represented in UART words. This protocol works basically as follows. The transmitter sends the word representing the register name to be updated in the transmission of the first word. After that, the content of this register is put in the transmission channel. On the receiver side, the update operation is completed according to the register name and content.

In the implementation of UART transmitter on the FPGA, a memory is used to store the data and queue them before the transmission starts. After collection of sufficient data for performance evaluation, a command signal is sent to the transmitter to inform that the memory is ready. After that command, the transmitter module starts to read the memory contents one by one and puts these contents on a shift register employed for parallel to serial conversion. After the word is placed on this shift register, the bits are transmitted serially at an agreed baud rate through the transmit pin of RS-232 port placed on the ML-402 platform. The transmission protocol is given in Table 5.3.

[4] The bit sequence sent in each transmission is called a “word”.

Table 5.3: The registers and their meanings used in the implementation of the UART transmitter module.

Register Address (Decimal)	Register Name	Description
160 161 162 163	Bit_error_counter	Total number of bits estimated incorrectly. This register has a width of 32 bits, and transmitted in 4 episodes.
164 165 166	Packet_error_counter	Total number of packets estimated incorrectly in at least 1 bit. This register has a width of 24 bits.

In the UART receiver module, a small block is employed to check the signal level at the receiver pin of the RS-232 port. After a *start bit* is detected, the module starts to receive the sequential signals. During the reception process, a shift register is used to carry out the serial to parallel conversion. After a word reception is complete, the same small block starts to check for a *stop bit*. The reception protocol is given in Table 5.4.

Table 5.4: The registers and their meanings used in the implementation of the UART receiver module.

Register Address (Decimal)	Register Name	Description
170	A_s	The amplitude of the BPSK signals. Used to vary SNR values.
171	Iteration_no	Number of iterations that the decoders will run for.
172 173 174	Paket_no	The number of packets that the simulation go for. This register has a width of 24 bits.
175	NormMax	*NormMax* value to configure the observation quantizer.
175 176 177 178	Rx_I	The in-phase part of the received signal for the fading channel tests. This register has a width of 32 bits.
179 180 181 182	Rx_Q	The quadrature-phase part of the received signal for the fading channel tests. This register has a width of 32 bits.
183	Beta	The β value used in the LMS algorithm.

To catch up synchronization in the UART module, we used counters to indicate the data transmission/reception instants. At each system clock, the counters count up. This counting operation continues till the desired baud rate is reached by the counter. If the counter reaches the number of system ticks to be generated in one baud rate period, this means that the UART transmitter (receiver) can transmit (receive) the next bit of the word. A sample code for the counter blocks of these modules are given below. At each system clock, the counters count up. This counting operation continues till the desired baud rate is reached by the counter. If the counter reaches the number of system ticks to be generated in one baud rate period, this means that the UART transmitter (receiver) can transmit (receive) the next bit of the word. A sample code for the counter blocks of these modules is given below.

```
  ...
constant baud_rate : integer := 115_200;
  ...
generic(
    clk_freq : integer := 36_000_000
      ...
);
  ...

signal counter_limit : integer := clk_freq/baud_rate;
signal counter : integer range 0 to counter_limit;
  ...
```

The signal “counter” is the register used to hold the number of clock ticks counted up to that time. The signal “counter_limit” represents the number of system ticks to be counted during one period of the given “baud_rate”. If “counter” reaches that “counter_limit” value, this means that a new bit can be transmitted/received in the UART module.

CHAPTER 6

NUMERICAL RESULTS

In Chapter 4, the implementation procedure of the two proposed structures is described in detail. In this chapter, the effects of the implementation choices will be shown and the performances of the joint channel estimation and parallelized decoder structure will be discussed. During the tests, the convolutional encoder shown in Figure 2.1 is used as the constituent encoder in the PDTC encoder. On the decoder side, the constituent max-log-MAP decoders have been used with the state diagram given in Figure 2.3. For the joint channel estimation and decoder tests, in the first iteration of the channel estimator zero forcing is applied and in the second iteration *2-way LMS* algorithm is applied.

6.1 PDTC Decoder Performance Results

The choice for the number $NormMax$ (explained in Section 4.1.2) is an important issue for the decoder to work in an optimum way in the fixed-point domain. Choosing a high $NormMax$ value results in saturation in metric calculations while choosing a small value causes a loss in representing observation values. This effect on the performance of the PDTC decoder has been shown in Figure 6.1. The figure is obtained by observing 6000 packets with each containing 160 information bits for $E_b/N_0 = 2.6423$ dB in an AWGN channel. The optimum $NormMax$ values for each K value can be obtained from this figure. It can be seen in the figure that as the K value increases, so does the number of available optimum $NormMax$ values.

Another parameter that affects the performance of the decoder is the selection of the K value. Choosing a large K results in a better performance while at the same time causes the

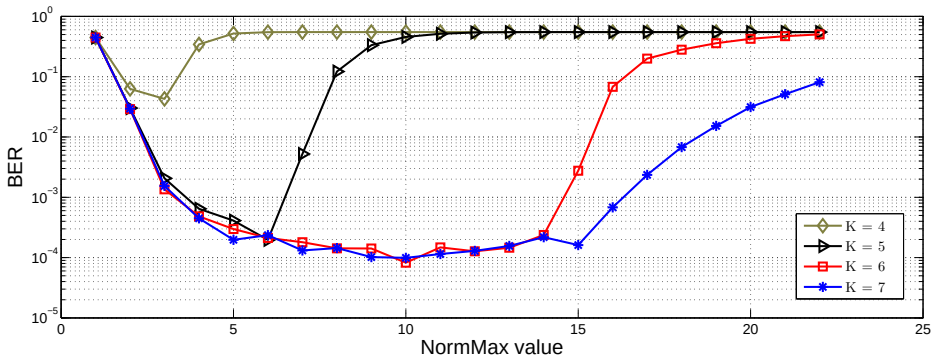

Figure 6.1: The effect of the *NormMax* value on the BER performance of the PDTC max-log-MAP decoder.

decoder to consume more resources on the FPGA and to work on lower clock speeds. Therefore, there is a trade off between the performance, resource consumption, and the speed. The resulting performances are given in Figure 6.2. It must be noted that these comparisons are made by using the best resulting *NormMax* values for each K value after 2000 runs on a 512-bit packet with a fixed iteration number of 4 for different K values. The implementation results are given in Table 6.1 to compare the resource consumption and the maximum available clock speed features for different metric size (K) values. These performance and synthesis results are obtained for the turbo decoder with 4 constituent decoders in each cluster. An important point is that, the resource consumption is approximately linearly proportional to the number N and increasing the N value does not affect the performance of the PDTC decoder [8]. In the determination of the maximum data rate (see Section 6.2) of the system this property will be used.

Table 6.1: Synthesis results of the PDTC decoder with max-log-MAP decoding algorithm

metric size (bits)	Slices used	Slice usage (%)	Max. clock speed (MHz)
4	5709	37	50.053
5	5894	38	48.123
6	6665	43	46.786
7	7566	49	42.865

Figure 6.2 shows that as K increases, the performance approaches the floating-point case. That's why, the selection of the K value in a design must be done carefully to match the BER

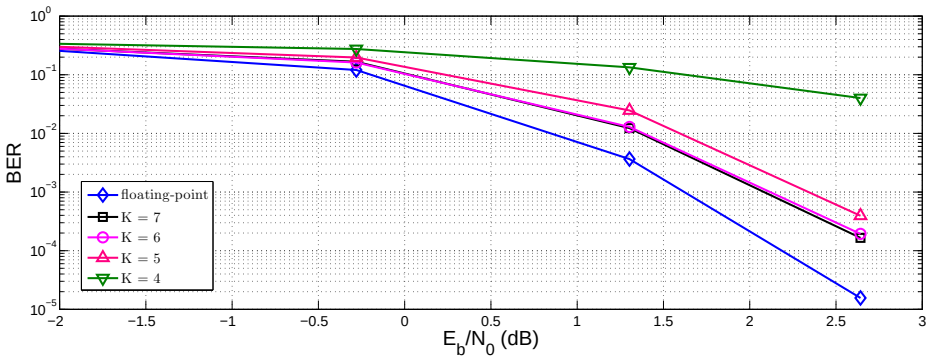

Figure 6.2: Performance of PDTC decoder with max-log-MAP decoding algorithm.

requirement. As it is obvious in the figure, the performance increase is slowed down after $K = 6$. On the other hand, it is shown in Table 6.1 that resource consumption increases as K increases while the maximum available clock speed decreases. So, the choice of K as 6 seems to be a good compromise for this decoder structure when we take all considerations into account. It must be noted that this choice brings a performance worse than the floating-point decoder by 0.5 dB.

The other parameter that affects the performance of the decoding process is the iteration number I. It can be anticipated that the error correction performance will enhance as the decoder runs for more iterations. In [13], some studies have been done to figure out the effect of the iteration number on the performance. For our architecture, this effect is given in Figure 6.3.

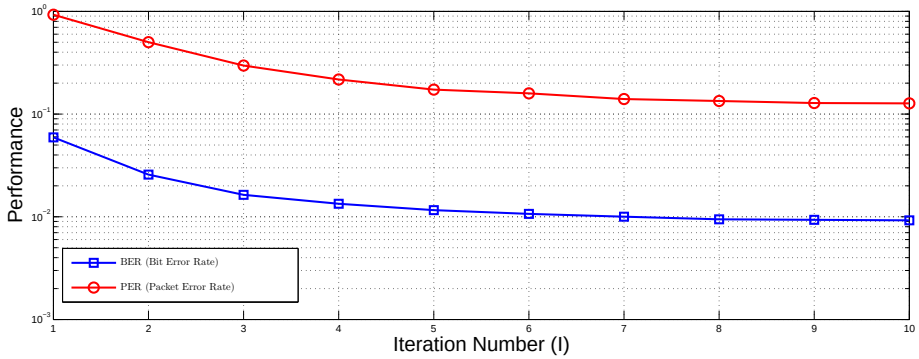

Figure 6.3: Effect of the iteration number on the BER and PER performances at SNR=1.3 dB.

For a better illustration of the effect, the number of errors after each iteration is given in Table 6.2. The data on Table 6.2 is generated on a 512-bit packet with data bit SNR, $E_b/N_0 = 1.3$ dB. The same packet is sent several times over the AWGN channel with the appropriate noise power and the data is collected after simulating a reasonable number of packets.

Table 6.2: Iteration steps and the corresponding number of errors for the PDTC decoder at SNR= 1.3 dB.

	Packets				
Iteration Step	Packet-1	Packet-2	Packet-3	Packet-4	Packet-5
1	23	24	17	26	19
2	13	17	15	16	19
3	8	14	6	14	18
4	8	5	0	10	16
5	4	0	0	3	12
6	3	0	0	2	9
7	0	0	0	0	5
8	0	0	0	0	3
9	0	0	0	0	2
10	0	0	0	0	0

However, a large iteration number may pose a trouble when we work on higher SNR values. When the same packet decoding process described above is simulated with data bit SNR, $E_b/N_0 = 2.65$ dB, we can see that the number of errors decreases up to some number of iterations, and then begins to increase. Some simulation results are given in Table 6.3. The reason of that situation is the saturation (overflow or underflow) of the LL values after some point. To prevent this problem, different early stop algorithms exist in the literature and some are listed in [26] and [27].

In the early stop algorithms, the iteration number varies according to the instantaneous channel condition. This ambiguity in the iteration number makes it impossible to determine an exact decoding latency and so the maximum data rate offered by the proposed system. Not to deal with that uncertainty, we need to use a constant iteration number in our system. As it can be seen in Figure 6.3, as the iteration number increases, the BER performance of the decoder improves. However, larger I means a larger decoding latency and a lower data rate. After considering all these effects, we decide on a value of $I = 4$.

Table 6.3: Iteration steps and the corresponding number of errors for the PDTC decoder at SNR= 2.65 dB.

	Packets			
Iteration Step	Packet-1	Packet-2	Packet-3	Packet-4
1	18	13	16	10
2	7	5	9	3
3	5	3	6	0
4	0	0	0	0
5	4	14	0	0
6	5	16	0	0
7	5	19	1	4
8	20	20	5	5
9	28	23	9	7
10	30	26	11	12

6.2 Decoding Latency Calculation

Large decoding latencies in turbo codes are told to be the drawback of their decoder structures. By making them operate in parallel, a decrease in their decoding latencies is expected. To observe that decrease, decoding latency is better to write in a formula. The decoding latency, τ, for our parallel decodable turbo code decoder structure is,

$$\tau = \frac{D}{N} \text{ x } 2I \tag{6.1}$$

where D is the number of information (data) bits in the packet, N is the number of parallel decoders in a cluster, and I is the iteration number. The $\frac{D}{N}$ term is the decoding latency of a single BCJR decoder operating with the CTT algorithm. The reason of multiplying by $2I$ is that in each iteration the BCJR decoders run twice, one for the uninterleaved form of data and one for the interleaved.

During the decoding latency calculations, we assume that a *ping-pong* buffer structure is used in the receiver side. Each of the *ping* and *pong* structures contains 3 memory blocks in which d, p_1 and p_2 observations are stored[1]. As each structure consists of 3 memory blocks, we name them *ping memory pool* and *pong memory pool*. To decrease the latency and increase the data rate, these memory pools are used as follows. While the first packet is being received, the observations are stored in the *ping memory pool* in the quantized form as

[1] Each of these memory structures contains N distinct memories to be used by each BCJR decoder in the cluster.

described in Section 4.1.6 for d, p_1 and p_2 observations. After the memory blocks are filled and ready to be read, PDTC decoder begins to run. Storing and decoding operations of the first packet (*packet*1) are shown in Figures 6.4 and 6.5, respectively.

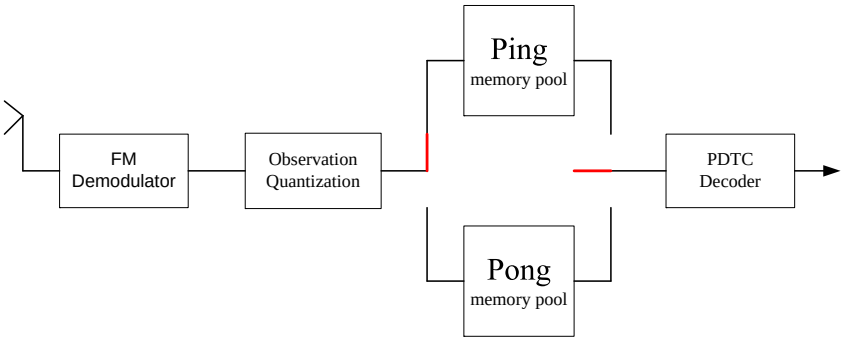

Figure 6.4: Reception of *packet*1 and filling the memory blocks of the *ping memory pool*.

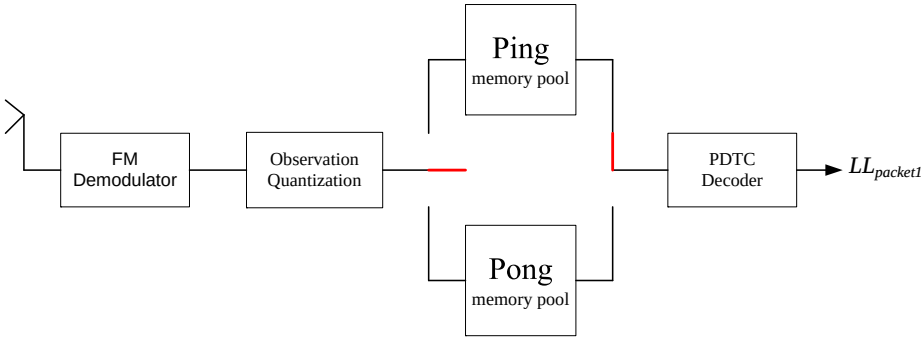

Figure 6.5: Decoding of the *packet*1.

If the second packet (*packet*2) arrives during the decoding of *packet*1, the d, p_1 and p_2 observations are stored in the *pong memory pool*. In this case, the decoding process is not affected by the reception of the new packet as shown in Figure 6.6.

When the decoder finishes its job, it starts operating from the *pong memory pool* and this time the *ping memory pool* becomes available for another packet storage. This structure doubles the memory usage in the system for storing observations but improves the data rate in a significant amount.

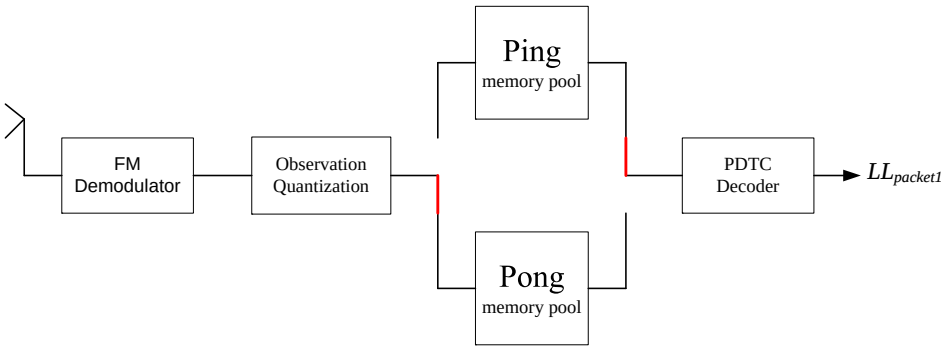

Figure 6.6: Reception of *packet*2 and filling the *pong memory pool* while decoding of *packet*1 still continues.

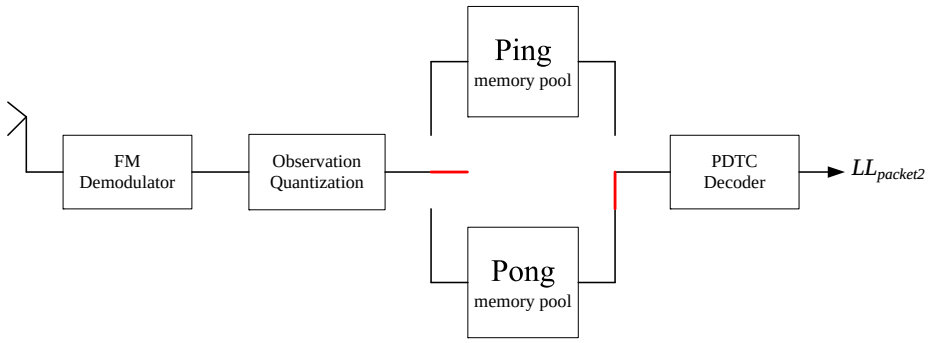

Figure 6.7: Decoding of the *packet*2.

At this point, we can find the maximum available data rate of our system. If we denote the data rate by υ, we can formulate it as,

$$\upsilon = \frac{D \text{ x } f}{\tau} = \frac{D \text{ x } f}{\frac{D}{N} \text{ x } 2I} \tag{6.2}$$

$$= \frac{f \text{ x } N}{2I} \tag{6.3}$$

where f is the maximum available frequency and τ is the decoding latency. To find the exact data rate, we need to decide on the metric representation width (K), iteration number (I), the number of constituent decoders in a cluster (N). In (6.3) it is obvious that the data rate of the system is proportional to N, and inversely proportional to I. Besides, it must be noted that the data rate is independent of D. This means that the data rate will stay constant if we use longer packets to communicate while the overall BER performance will improve as we use larger interleaver tables. In data rate calculation, the f value can be obtained by checking the

Table 6.1 for the selected K value. Similarly, τ value can be obtained from (6.1). During the studies we had already decided on N as 4, I as 4, and K as 6, and used a coding scheme with rate-$1/3$ $(1, 5/7)$ convolutional coding given in Figure 2.1. Using the Table 6.1 and Figure 6.2 with the design choices listed above, we can find the available data rate as

$$\begin{aligned} \upsilon &= \frac{46 \text{ x } 10^6 \text{ x } 4}{2 \text{ x } 4} = 23 \text{ x } 10^6 \text{ bps} \\ &\approx 21.93 \text{ Mbps.} \end{aligned} \tag{6.4}$$

To find a maximum data rate available for the system without any degradation in the performance, we can increase the number of parallel decoders in each cluster, N, to use all of the available resources on the FPGA. Since the resource consumption is proportional to the number of parallel decoders, we can increase this number upto 9 after evaluating the synthesis results given in Table 6.1. Such an implementation on the FPGA allows us the maximum available data rate as

$$\begin{aligned} \upsilon_{max} &= \frac{46 \text{ x } 10^6 \text{ x } 9}{2 \text{ x } 4} = 51.75 \text{ x } 10^6 \text{ bps} \\ &\approx 49.35 \text{ Mbps.} \end{aligned} \tag{6.5}$$

This PDTC decoder structure is implemented by using a different architecture in [28]. In that study, pipelining is applied to increase the operating clock frequency of the FPGA. The synthesis results of that architecture, taken from [28], are given in Table 6.4.

Table 6.4: Synthesis results of the PDTC decoder with max-log-MAP decoding algorithm and pipelining.

metric size (bits)	Slices used	Slice usage (%)	Max. clock speed (MHz)
4	6347	42	87.253
5	6501	42	86.963
6	6994	45	86.949
7	7537	49	85.704

When the synthesis results given in Table 6.4 are considered, it can be seen that the operating frequency of the decoder can be enhanced significantly with pipelining. On the other hand, there is a slight increase in the resource consumption but it is negligable.

6.3 Joint Channel Estimator and Decoder Performance Results

In Figure 6.2, the performances of the PDTC decoder is compared, but the channel is assumed to be AWGN. However, if the channel has a fading effect, the performance results probably change. To find an appropriate K value for the decoder, first we need to compare the performances in the case of a Rayleigh fading channel. For that reason, the effect of $NormMax$ value is compared for different K values to obtain the best resulting $NormMax$ values over a fading channel. The resulting performances for different $NormMax$ values are given in Figure 6.8 after running on $200,000$ packets. During the tests, CSI knowledge is assumed to be known at the receiver side and PDTC decoder parameters are set as $N = 4$, $I = 4$.

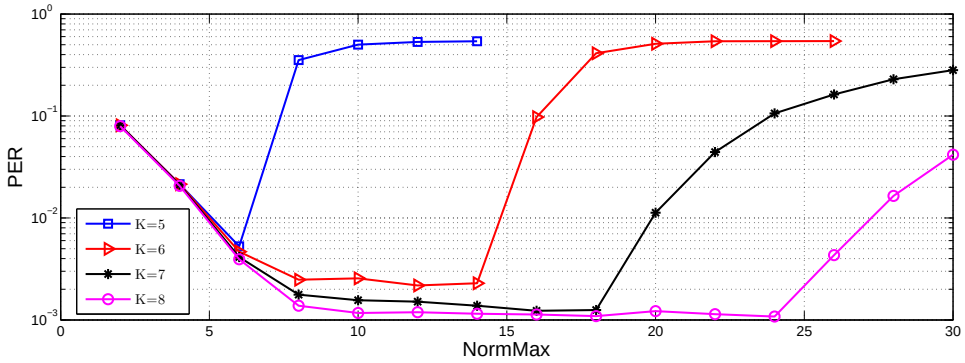

Figure 6.8: The effect of the $NormMax$ value on the performance of the PDTC decoder over a Rayleigh fading channel with the normalized fading rate $f_D T_s = 0.01$ at $E_b/N_0 = 8$ dB with the assumption of perfectly known CSI at the receiver side.

After obtaining the best resulting $NormMax$ for each K value, the effect of K value on the PER performances can be tested. The performances given in Figure 6.8 are acquired for SNR= 8 dB. These results probably change for different SNR values and different best resulting $NormMax$ values can be obtained. However, it is not easy to optimize all these parameters at the same time to get the best performances. For that reason, the same $NormMax$ values are used in the tests for all SNR values. The performances of the PDTC decoder for different K values are given in Figures 6.9 and 6.10. These figures are obtained over $200,000$ packets of length 512 bits for two different Rayleigh fading channels by running the PDTC decoder with $N = 4$ and $I = 4$. During these tests, known channel state information (CSI) is assumed at the receiver side.

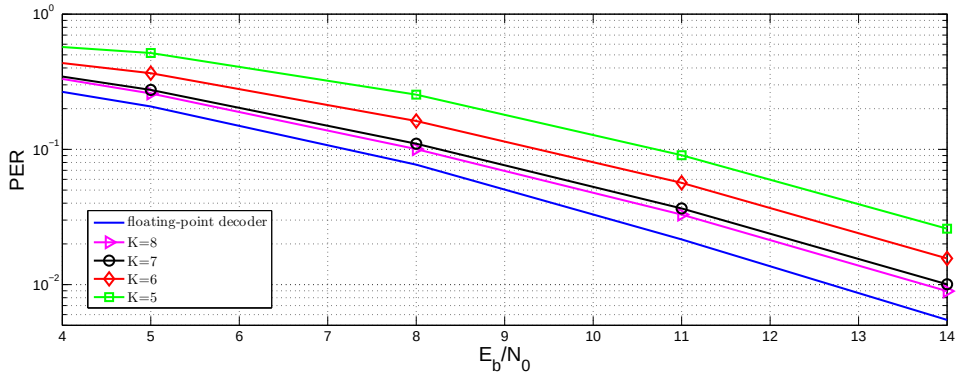

Figure 6.9: Performance of the PDTC decoder over a Rayleigh fading channel with the normalized fading rate $f_D T_s = 0.001$ with the assumption of perfectly known CSI at the receiver side.

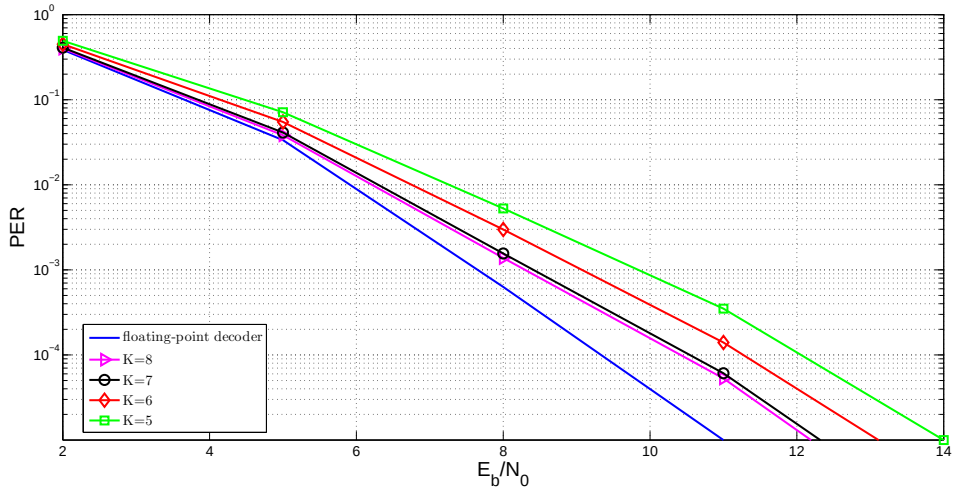

Figure 6.10: Performance of the PDTC decoder over a Rayleigh fading channel with the normalized fading rate $f_D T_s = 0.01$ with the assumption of perfectly known CSI at the receiver side.

Table 6.5: Synthesis result of the PDTC decoder for different K values with max-log-MAP decoding algorithm

K (bits)	Slices used	Slice usage (%)	Max. clock speed (MHz)
5	5894	38	48.123
6	6665	43	46.786
7	7566	49	42.865
8	8621	56	40.461

Since the representation bit numbers of the interest changed a bit, the synthesis results are given as a whole in Table 6.5. From the given performance figures (Figures 6.9 and 6.10), the choice of 7 bits for metric representation in the PDTC decoder seems to be a reasonable choice, different than the AWGN case, when the resource consumption and the clock speed factors given in Table 6.5 are taken into account. In the following tests, this value will be used in the decoder side with the assumption that the K value does not effect the performance of the channel estimation. It must be noted that this choice brings a performance worse than the floating-point decoder by 0.75 dB. In Figures 6.9 and 6.10, it can be seen that the performance differences get larger for increasing SNR. The main reason may be told to be the effect of $NormMax$. This performance difference can be decreased by using the best resulting $NormMax$ values at each SNR as given in Figure 6.11.

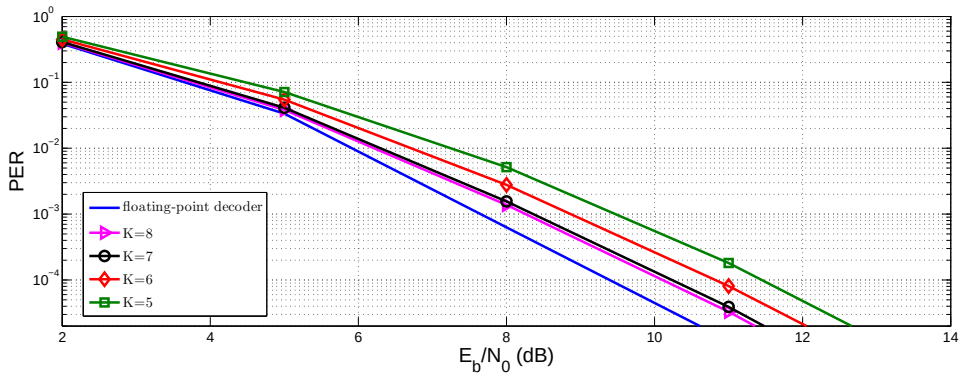

Figure 6.11: Performance of the PDTC decoder over a Rayleigh fading channel with the normalized fading rate $f_D T_s = 0.01$ by using the best resulting $NormMax$ values at each SNR.

After deciding on the K value for the decoder, the next task is the determination of the design parameters for the channel estimator, which are described in Section 4.2.2. The pilot symbol spacing, M, is determined by examining the performance results of the system for two different M values over a Rayleigh fading channel with the normalized fading rate of $f_D T_s = 0.01$ given in [6]. As seen in this comparison, choosing a small M value gives a better performance output. However, the redundancy is increased in the packet by adding more pilot symbols and the overall data rate is degraded. To get a reasonable PER performance together with a high enough data rate, $M = 17$ is chosen after a careful trade off. Hereafter, we make tests on our system considering the normalized fading rate of 0.01.

The β value is an important factor that affects the success of the estimation. However, the most appropriate β value is not the same for all SNR values. Therefore, we need the best resulting β values for each SNR value in the region of our interest. In Figure 6.12, the effect of β value for some SNR values is given. To obtain these results, the estimator is run for three iterations and $L = 20$, $M = 17$ values are used constant not to lose the generality.

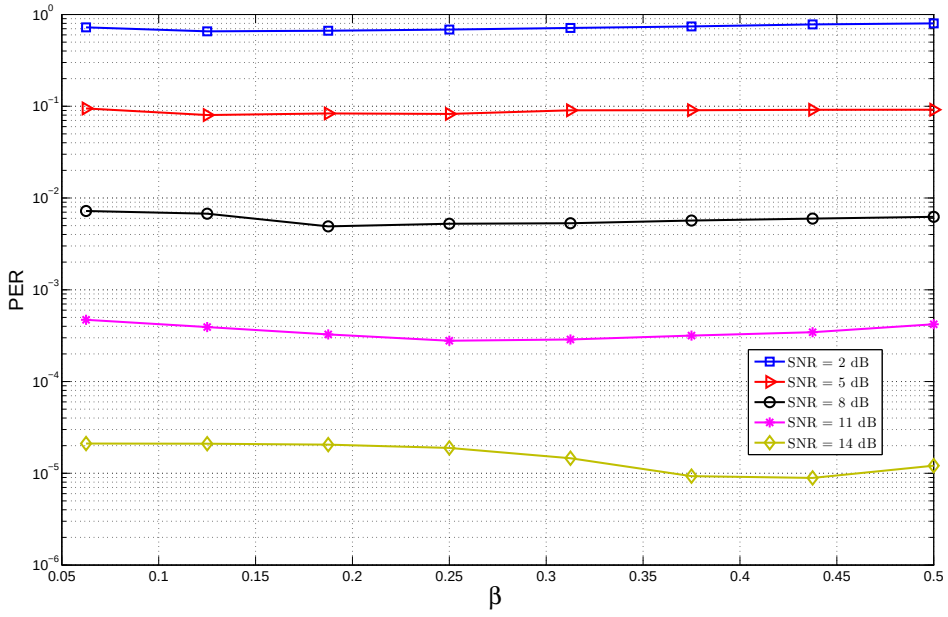

Figure 6.12: Effect of the β value on the estimation and decoding performance over a Rayleigh fading channel with the normalized fading rate $f_D T_s = 0.01$.

As the final design parameter, the L value is needed to be determined. To obtain a reasonable L value, the performance of the overall system for different L values is tested. In this test, the best resulting β values are used for each SNR value. The result of this test is given in Figure 6.13. Since the decoder operates on 6-bit metric values, using a very high L value is meaningless and just increases the consumption of the resources on the FPGA. The synthesis results for the tested L values are listed in Table 6.6.

From the given synthesis results, $L = 15$ seems to be a reasonable choice since the memory usage is nearly doubled after that point. The reason of the increase in the number of used block RAMs after $L = 15$ is that the block RAMs on the FPGA have maximum width of 16 bits. When a wider RAM is needed in the design, two block RAMs are concatenated

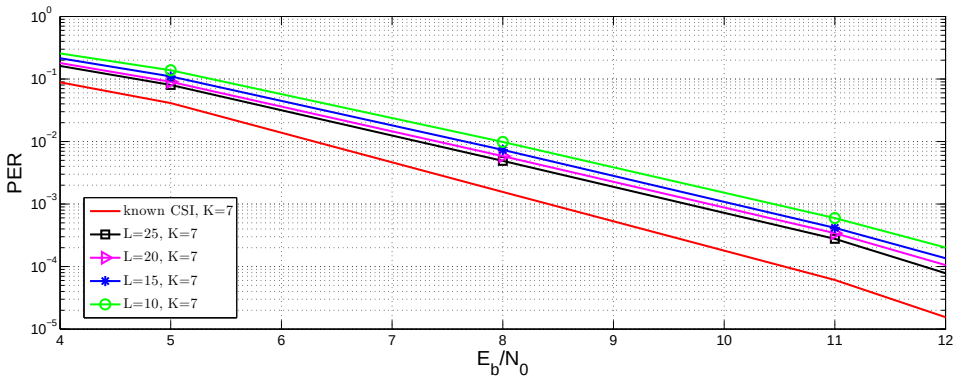

Figure 6.13: Performance of the joint estimation and decoding structure over a Rayleigh fading channel with the normalized fading rate $f_D T_s = 0.01$ for different L values.

Table 6.6: Synthesis results of the estimator with different L values.

L (bits)	Slices used	Slice usage (%)	Block RAMs used	Max. clock speed (MHz)
10	302	1	7	99.668
15	392	2	7	96.313
20	647	4	13	82.218
25	857	5	13	75.753

to generate the required width. When the performances are considered together with the synthesis results, the choice of $L = 15$ seems a reasonable one. There is a performance difference of 2 dB when the $L = 15$ case is compared with the perfect known CSI case. The 2-way LMS algorithm and getting into a fixed-point domain cause that performance difference. It must be noted that, the selection of $L = 15$ does not affect the operating frequency of the overall system because it allows higher clock speeds when compared to the PDTC decoder. The joint structure has the maximum clock speed of 42 MHz.

6.4 Joint Channel Estimation and Decoding Latency Calculation

After completing the overall system setup, we can find the overall latency introduced by the joint estimator and decoder structure. The latency introduced by the PDTC decoder was formulated before as in 6.1. In this section we will deal with the overall system latency.

If D data bits are encoded and transmitted after pilot symbol insertion, we get a packet

length of $P = D/R+32$ where R is the coding rate. However, as we use the $(1, 5/7)$ code given in Section 2.2, the code rate is already determined as $1/3$. If it is assumed that the received sequence is ready on a RAM in the receiver side when the channel estimator starts its first iteration, the packet reception duration can be omitted in the latency calculation. This operation can be realized by using the ping-pong buffer structure as mentioned earlier. If a packet arrives while the operations are in progress on another packet, the unused (ping or pong) memories can be used to store the new arriving packet. After employing such a structure, we can assume that the received sequence is ready in a memory for the first estimation.

In the first estimation step, the estimator does not make any complex operations, but gives the received values and channel coefficient estimates to the LL computing component. However, in every clock cycle at most two readings can be achieved from the memories which store the received sequence and channel estimates even if we use dual port RAM modules. The calculation of LL values can be achieved at the same clock cycle in which the inputs of the module are ready. Therefore, LL computation does not introduce any latency, but outputting the estimates and the received values take $D/2R$ clock cycles. It must be noted that, the pilot values are discarded for this first iteration of the channel estimator. After the LL computation, these values are given to the *observation quantizer* module of the PDTC decoder. Since the estimation process in the next iteration waits for the decoder outputs to compute the new LL values, there is no need to implement a ping-pong structure before the PDTC decoder. However, to decrease the latency introduced by the turbo decoder, the filling operation of the d, p_1, and p_2 memory structures can be completed during the quantization process according to the parallelization in the decoder. Each of these memory structures contain N memory blocks with a length (depth) of D/N where N is the number of constituent decoders in a cluster. If we use dual port RAMs in these structures, the storing and the deinterleaving process jointly takes $D/2N$ clock cycles.

In the next estimation iterations, the CTT algorithm is applied to reduce the latency. By employing a CTT structure in the 2-way LMS algorithm, two channel estimates are computed beginning from the center of the packet and that takes P clock cycles to estimate the channel for the whole packet duration. Although estimation process at the pilot locations can be avoided in the first iteration, in the next iterations the estimates at the pilot locations have to be calculated due to the recursive computations in the LMS algorithm. By using the generated estimates, the LL values are calculated instantly, again beginning from the center of the packet.

During the LL calculation process, the estimates and observations at the pilot locations are discarded. The resulting LL value are written to the D/N memory blocks in $D/2N$ clock cycles. Then, the PDTC decoder starts to operate.

After these reckonings, we can write the joint estimation and decoding latency in a formula as

$$\tau = \left(\frac{D}{2R} + \frac{D}{2N}\right) + \left(P + \frac{D}{2N}\right)(I_e - 1) + \left(\frac{D}{N}2I_d\right)I_e \tag{6.6}$$

where I_e is the iteration number of the channel estimator and I_d is the iteration number of the PDTC decoder. The term $\frac{D}{2R} + \frac{D}{2N}$ gives the latency of the first iteration of the channel estimation, $P + \frac{D}{2N}$ gives the latency of the second and later iterations of the channel estimation, and $\frac{D}{N}2I_d$ gives the decoding latency of the PDTC decoder running for I_d iterations.

After having the latency formula, we can calculate the available data rate allowed by the proposed system by using the equation

$$\upsilon = \frac{D \text{ x } f}{\tau}. \tag{6.7}$$

The values of τ and f depends on the design parameters used in the system setup. To calculate the latency τ, we need to decide on the values of D, R, N, I_e, and I_d. On the other hand, to determine the value of f, operating frequency, we need to decide on the values of K and L. To obtain the maximum data rate, we can increase the parallelization number in the decoder as much as the FPGA allows in terms of resource consumption. The PDTC decoder consumes 49% of the resources for $N = 4$ and $K = 7$, and the channel estimator consumes 2% for $L = 15$. This means that the number of parallel decoders in each cluster can be increased up to 8 for data rate maximization. With the design parameters of $D = 160$, $R = 1/3$, $N = 8$, $I_e = 3$, and $I_d = 4$, we get a minimum latency introduced by the proposed joint estimator and decoder as

$$\begin{aligned} \tau &= \left(\frac{160}{2/3} + \frac{160}{16}\right) + \left(160 \text{ x } 3 + 32 + \frac{160}{16}\right)2 + \left(\frac{160}{8}8\right)3 \\ &= 1774 \text{ clock cycles.} \end{aligned} \tag{6.8}$$

To find the maximum available data rate, the operating frequency can be obtained from Tables 6.1 and 6.6 for the chosen $K = 8$ and $L = 15$ values. From the found operating frequencies it can be seen that the limitation on the clock speed comes from the PDTC decoder. Therefore, $f = 42$ MHz is obtained as the clock speed for the whole system by checking

the Table 6.1 for $K = 8$. After deciding on all of the design parameters of the system, the maximum data rate comes out as

$$\begin{aligned} \upsilon &= \frac{160 \text{ x } 42 \text{ x } 10^6}{1774} \\ &= 3.79 \text{ x } 10^6 \text{ bps} \\ &\approx 3.61 \text{ Mbps.} \end{aligned} \tag{6.9}$$

The parallelization idea can also be implemented on the channel estimating process. If we divide the overall estimation process into 4 parallel parts, the length of the memories storing the received sequence and the estimates get $P/4$ and the number of these memories arise to 4 fold of the old one. After this parallelization, it can be anticipated that the performance degrades in a non-significant amount. However, the total latency becomes

$$\tau' = \left(\frac{D/N_e}{2R} + \frac{D}{2N}\right) + \left(\frac{P}{N_e} + \frac{D}{2N}\right)(I_e - 1) + \left(\frac{D}{N} 2 I_d\right) I_e \tag{6.10}$$

where N_e is the number of parallel processing estimators. With the same design parameters mentioned before and $N_e = 4$, we get the overall latency and data rate as

$$\begin{aligned} \tau' &= \left(\frac{160/4}{2/3} + \frac{160}{16}\right) + \left(\frac{160 \text{ x } 3 + 32}{4} + \frac{160}{16}\right) 2 + \left(\frac{160}{8} 8\right) 3 \\ &= 826 \text{ clock cycles,} \end{aligned} \tag{6.11}$$

$$\begin{aligned} \upsilon' &= \frac{160 \text{ x } 42 \text{ x } 10^6}{826} \\ &\approx 7.76 \text{ Mbps.} \end{aligned} \tag{6.12}$$

It must be noted that these results depend on the iteration numbers of the estimator and the decoder. They can be enhanced by using some different methods in the literature, e.g., early stop algorithms for the turbo codes. Also, better performances can be obtained by optimizing the parameters N, N_e, I_e, and I_d.

CHAPTER 7

CONCLUSIONS AND FUTURE WORK

The work on this thesis aims at implementing a jointly operating channel estimator and parallel decodable turbo code (PDTC) decoder on a FPGA-based platform. In this thesis, we applied the parallelization idea on turbo codes to reduce the decoding latency due to their iterative structure. On the other hand, the complexity of the design is increased after this operation. By this idea, the decoding latency is decreased by a factor of the number of parallel branches employed in the decoder. Parallelization is induced by employing parallel encoders operationg on substreams of the data sequence simultaneously. The parallel processing structures are implemented on FPGA with the help of its architecture containing *configurable logic blocks* (CLBs).

However, parallelization process introduces some problems during the implementation. The memory collision problem arises as the most critical one. In literature, there exist some interleavers designed to prevent the memory collision problem. As one of them, memory collision free row-column S-random interleavers are used in this thesis.

During implentation of a PDTC decoder we face with some problems. Implementing a soft-in soft-out (SISO) decoder on an FPGA is the most critical problem since FPGA has limited resources which do not let one easily use floating-point arithmetic or large fixed-point arithmetic. To work on fixed-point arithmetic, a metric quantization scheme is used. Some operations used in the decoding process is modified to work on the fixed-point arithmetic. Parameters effective on the performance of the implemented decoder are presented and a decoding latency is calculated together with the available data rate for the reasonable choices of these parameters. These results are compared with the ones obtained in a parallel study in which the decoder is implemented by using a different architecture which applies pipelining.

It has been observed that the pipelining architecture enhances the clock frequency.

A channel estimator algorithm is implemented to work jointly with the the PDTC decoder structure. For the channel estimation process, the 2-*way LMS* algorithm which uses the pilot symbol assisted estimation method is employed. The operations used in the estimation process are optimized to work on the fixed-point architecture of the FPGA. After adjoining the two structures, the design parameters of the decoder are decided again to obtain a reasonable performance from the joint structure. The latency introduced by this joint structure and the resultant data rate is calculated as well. By applying the parallelization idea on the estimation algorithm, the latency is reduced in some amount to enhance the available data rate.

In addition to the studies described in this thesis, some additional research and improvements to our testbed are set as future goals. These goals can be listed as follows:

- An early-stopping algorithm can be used to prevent the extra iterations of the PDTC decoder. Thereby, the decoding latency can be decreased more and the achievable data rate can be enhanced.
- The performance comparisons of the used channel estimation algorithm with the other algorithms in the literature can be made as an additional study.

REFERENCES

[1] C. E. Shannon, "A mathematical theory of communication," *The Bell System Technical Journal*, vol. 27, pp. 379–423, July 1948.

[2] C. Berrou, A. Glavieux, and P. Thitimajshima, "Near Shannon limit error-correcting coding and decoding: Turbo-codes," *in Proc. ICC'93*, pp. 1064–1070, May 1993.

[3] L. R. Bahl, J. Cocke, F. Jelinek, and J. Raviv, "Optimal decoding of linear codes for minimizing symbol error rate," *IEEE Transactions on Information Theory*, vol. IT-20, pp. 284–287, 1974.

[4] O. Gazi and A. Özgür Yılmaz, "Collision free row column S-random interleaver," *IEEE Communications Letters*, vol. 13, April 2009.

[5] E. K. Hall and S. G. Wilson, "Design and analysis of turbo codes on Rayleigh fading channels," *IEEE Journal on Selected Areas in Communications*, vol. 16, pp. 160–174, February 1998.

[6] Y. Yapıcı and A. Özgür Yılmaz, "Joint channel estimation and decoding with low-complexity iterative structures in time-varying fading channels," *to be presented at IEEE PIMRC 2009, Tokyo, Japan*.

[7] S. B. Wicker, *Error Control Systems for Digital Communication and Storage*. Prentice-Hall, Inc., January 1995.

[8] O. Gazi, *Parallelized Architectures for Low Latency Turbo Structures*. PhD thesis, Middle East Technical University, 2007.

[9] S. Lin and D. J. C. Jr., *Error Control Coding*. Prentice-Hall, Inc., second ed., 2004.

[10] S. Benedetto and G. Montorsi, "Unveiling turbo codes: some results on parallel concatenated coding schemes," *IEEE Transactions on Information Theory*, vol. 42, pp. 409–428, March 1996.

[11] O. Gazi and A. Özgür Yılmaz, "Fast decodable turbo codes," *IEEE Communications Letters*, vol. 11, April 2007.

[12] J. Jung, I. Lee, D. Choi, J. Jeong, K. Kim, E. Choi, and D. Oh, "Design and architecture of low-latency high-speed turbo decoders," *ETRI Journal*, vol. 27, pp. 525–532, October 2005.

[13] S. Yoon and Y. Bar-Ness, "A parallel MAP algorithm for low latency turbo decoding," *IEEE Communications Letters*, vol. 6, pp. 288–290, July 2002.

[14] L. Tong, G. Xu, and T. Kailath, "Blind identification and equalization based on second-order statistics: A time domain approach," *IEEE Transactions on Information Theory*, vol. 40, pp. 340–349, March 1994.

[15] J. G. Proakis, *Digital Communications*. McGraw-Hill Science Engineering, fourth ed., August 2000.

[16] M. C. Valenti and B. D. Woerner, "Iterative channel estimation and decoding of pilot symbol assisted turbo codes over flat-fading channels," *IEEE Journal on Selected Areas in Communications*, vol. 19, pp. 1697–1705, September 2001.

[17] A. Goldsmith, *Wireless Communications*. Cambridge University Press, 2005.

[18] M. C. Valenti and J. Sun, "The UMTS turbo code and an efficient decoder implementation suitable for software defined radios," *International Journal of Wireless Information Networks*, vol. 8, pp. 203–215, October 2001.

[19] T. A. Summers and S. G. Wilson, "SNR mismatch and online estimation in turbo decoding," *IEEE Transactions on Communications*, vol. 46, pp. 421–423, April 1998.

[20] A. Worm, P. Hoeher, and N. When, "Turbo-decoding without SNR estimation," *IEEE Communications Letter*, vol. 4, pp. 193–195, June 2000.

[21] P. H.-Y. Wu and S. M. Pisuk, "Implementation of a low complexity, low power, integer-based turbo decoder," *Global Telecommunications Conference*, pp. 946–951, 2001.

[22] "ML401/ML402/ML403 Evaluation Platform User Guide." http://www.xilinx.com-/support/documentation/boards_and_kits/ug080.pdf, last visited on 14 August 2009.

[23] "Synthesis and Simulation Design Guide." http://www.xilinx.com/support-/documentation/sw_manuals/xilinx11/sim.pdf, last visited on 14 August 2009.

[24] "iMPACT User Guide." http://www.xilinx.com/itp/xilinx4/data/docs/pac/pac.html, last visited on August 2009.

[25] J. G. Proakis and D. Manolakis, *Digital Signal Processing: Principles, Algorithms, and Applications*. Pearson Prentice Hall, fourth ed., 2007.

[26] F. Zhai and I. Fair, "Techniques for early stopping and error detection in turbo decoding," *IEEE Transactions on Communications*, vol. 51, pp. 1617–1623, October 2003.

[27] D. Lee and I. Park, "A low complexity stopping criterion for iterative turbo decoding," *IEICE Transactions on Communications*, vol. 88, no. 1, pp. 399–401, 2005.

[28] E. Erdin, "Performance of parallel decodable turbo and repeat-accumulate codes implemented on an FPGA platform," Master's thesis, submitted to Graduate School of Natural and Applied Sciences of Middle East Technical University, September 2009.