PERFORMANCE OF PARALLEL DECODABLE TURBO AND REPEAT ACCUMULATE CODES IMPLEMENTED ON AN FPGA PLATFORM

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

BY

ENES ERDİN

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

SEPTEMBER 2009

Approval of the thesis:

PERFORMANCE OF PARALLEL DECODABLE TURBO AND REPEAT ACCUMULATE CODES IMPLEMENTED ON AN FPGA PLATFORM

submitted by **ENES ERDİN** 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 Department, METU** _______________

Examining Committee Members:

Prof. Dr. Yalçın TANIK
Electrical and Electronics Engineering Dept., METU _______________

Assoc. Prof. Dr. Ali Özgür YILMAZ
Electrical and Electronics Engineering Dept., METU _______________

Asst. Prof. Dr. Behzat A. ŞAHİN
Electrical and Electronics Engineering Dept., METU _______________

Asst. Prof. Dr. Çağatay CANDAN
Electrical and Electronics Engineering Dept., METU _______________

Güzin KURNAZ, Ph.D.
Digital Design Engineer, TÜBİTAK-SAGE _______________

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: ENES ERDİN

Signature :

ABSTRACT

PERFORMANCE OF PARALLEL DECODABLE TURBO AND REPEAT ACCUMULATE CODES IMPLEMENTED ON AN FPGA PLATFORM

Erdin, Enes
M.S., Department of Electrical and Electronics Engineering
Supervisor : Assoc. Prof. Dr. Ali Özgür Yılmaz

September 2009, 75 pages

In this thesis, we discuss the implementation of a low latency decoding algorithm for turbo codes and repeat accumulate codes and compare the implementation results in terms of maximum available clock speed, resource consumption, error correction performance, and the data (information bit) rate. In order to decrease the latency a parallelized decoder structure is introduced for these mentioned codes and the results are obtained by implementing the decoders on a field programmable gate array. The memory collision problem is avoided by using collision-free interleavers. Through a proposed quantization scheme and normalization approximations, computational issues are handled for overcoming the overflow and underflow issues in a fixed point arithmetic. Also, the effect of different implementation styles are observed.

Keywords: Repeat-Accumulate Codes, Turbo Codes, Parallel Decoder, FPGA, Xilinx

ÖZ

PARALELLEŞTİRİLMİŞ TURBO VE TEKRARLA-BİRİKTİR KODLARININ FPGA PLATFORMU ÜZERİNDE GERÇEKLENMESİ VE BAŞARIMI

Erdin, Enes
Yüksek Lisans, Elektrik-Elektronik Mühendisliği Bölümü
Tez Yöneticisi : Doç. Dr. Ali Özgür YILMAZ

Eylül 2009, 75 sayfa

Bu tezde turbo kodlar ve tekrarla-biriktir kodları için düşük gecikmeli bir kod çözme algoritmasının donanımsal tasarımı ve tasarım sonuçlarının saat hızı, kaynak tüketimi, hata düzeltme yeteneği ve veri hızı açısından incelmesi gerçekleştirilmiştir. Çözücüdeki gecikmeyi azaltmak için paralelleştirilmiş çözücü mimarisi önerilmiş ve bahsi geçen kodlar için sonuçlar, alan programlanabilir kapılar dizisinde (FPGA) incelenmiştir. Hafıza çakışma problemi, çakışmasız karıştırıcılar kullanılarak önlenmiştir. Ayrıca önerilen nicemleme ve düzgeleme yaklaşımlarıyla sabit noktalı hesaplamalarda oluşabilecek alttaşma ve üsttaşma sorunları da çözülmüştür.

Anahtar Kelimeler: Tekrarla biriktir kodlar, Turbo kodlar, Paralleştirilmiş Çözücü, FPGA, Xilinx

To My Family,
To My Wife

ACKNOWLEDGEMENTS

I am most thankful to my supervisor Assoc. Prof. Dr. Ali Özgür Yılmaz for sharing his invaluable ideas and experiences on the subject of my thesis. I learnt too much from his innovative ideas. I feel myself privileged to have had him as a mentor.

I would like to extend my thanks to all lecturers at the Department of Electrical and Electronics Engineering, who greatly helped me to store the basic knowledge onto which I have built my thesis.

I want to thank to Çağlar Kılcıoğlu who was a good friend in the laboratory. He had done great job in writing our paper.

I would like to thank to Onur Dizdar for his patience in debugging the testbed environment which took many hours of him.

I am very grateful to TÜBİTAK-SAGE for providing tools and other facilities throughout the production of my thesis.

I would like to forward my appreciation to all my friends and colleagues who contributed to my thesis with their continuous encouragement.

I would like to express my deep gratitude to my family, who has always provided me with constant support and help.

Special thanks to my wife for all her help and showing great patience during my thesis.

TABLE OF CONTENTS

ABSTRACT iv

ÖZ v

DEDICATON vi

ACKNOWLEDGEMENTS vii

TABLE OF CONTENTS viii

LIST OF TABLES xi

LIST OF FIGURES xii

CHAPTERS

1 INTRODUCTION 1

2 TURBO CONCEPT 3

2.1 TURBO CODES 5

2.1.1 Turbo Code Encoder Structure 5

2.1.2 Convolutional Encoding 6

2.1.3 Interleaving 8

2.1.4 Turbo Code Decoder Structure 8

2.2 TURBO-LIKE CODES 10

2.2.1 Low Density Parity Check Codes 10

2.2.2 Repeat Accumulate Codes 11

2.2.3 Repeat Accumulate Code Encoder Structure 11

2.2.4 Repeat Accumulate Code Decoder Structure 13

3 TESTBED SETUP 16

3.1 ML-402 FPGA Evaluation Board 16

3.2 Software Used For Debugging and Implementation 20

3.2.1 Xilinx ISE and XST 20
3.2.2 Implementation steps of an ISE project 21
3.2.2.1 Synthesis 21
3.2.2.2 Translate Process 21
3.2.2.3 Mapping Process 22
3.2.2.4 Place and Route Process 22
3.2.3 BitGen 23
3.2.4 ChipScope Analyzer 23
3.2.5 MATLAB 23
3.2.6 MODELSIM 24
3.3 Overall System Setup 26
3.3.1 System 26
3.3.2 LFSR Noise Generator 27
3.3.3 Error Counter 33
3.3.4 UART Module 33
4 IMPLEMENTATION ISSUES 36
4.1 Channel Model 36
4.2 BCJR Decoder 37
4.3 FPGA Implementation of BCJR Decoder 41
4.3.1 Center to Top Algorithm 42
4.3.2 Observation Quantization 42
4.3.3 Addition and Subtraction Operations 46
4.3.4 Node (α, β) Metric Normalization 48
4.3.5 max^* Approximation 49
4.4 Memory Collision Free Interleavers 50
4.5 Encoder/Decoder Design of Parallel Decodable Turbo Codes 53
4.5.1 Encoder Design 53
4.5.2 Decoder Design 54
4.6 Encoder/Decoder Design of Parallel Decodable Repeat Accumulate Codes 56

4.6.1 Encoder Design 56
4.6.2 Decoder Design 56
5 RESULTS FOR THE PERFORMANCE OF PARALLEL DECODERS 58
5.1 Implementation Results 58
5.2 Simulation Results 60
5.2.1 Bit Size (K) Selection 60
5.2.2 $NormMax$ Selection 61
5.2.3 Interleaver Size 64
5.2.4 Memory Complexity 67
5.2.4.1 PDTC memory structure 67
5.2.4.2 PDRAC memory structure 68
5.2.5 Transmission Bit Rate 68
6 CONCLUSIONS 71
REFERENCES 74

LIST OF TABLES

TABLES

Table 2.1 Rate, Threshold and Shannon threshold comparison for RA codes [12] 14

Table 3.1 Some LFSR generator polynomials with varying size of shift registers. 29

Table 3.2 The registers and their meaning in the design of the UART transmitter. 34

Table 3.3 The registers and their meaning in the design of the UART transmitter. 35

Table 4.1 Xilinx ISE synthesis report for parallel turbo encoder 54

Table 4.2 Xilinx ISE synthesis report for parallel turbo encoder 56

Table 5.1 Implementation results for PDTC decoder using log-MAP decoders 59

Table 5.2 Implementation results for PDTC decoder using max-log-MAP decoders 59

Table 5.3 Implementation results for PDRAC decoder using log-MAP decoders 60

Table 5.4 Another approach to PDTC decoder implementation : Implementation results for PDTC decoder using log-MAP decoders [2] 60

Table 5.5 Another approach to PDTC decoder implementation : Implementation results for PDTC decoder using max-log-MAP decoders [2] 60

Table 5.6 Performance comparison of the proposed decoder structures 67

Table 5.7 Comparison of the proposed decoder structures 69

LIST OF FIGURES

FIGURES

Figure 2.1 Encoder Structure of a Turbo Code 5

Figure 2.2 Parallelized Turbo Code Encoder Structure 6

Figure 2.3 A Rate $1/2$ Convolutional Encoder 6

Figure 2.4 The FSM representation of the convolutional encoder shown in Figure 2.3 each bit arrival(I) contributes to a state transition and reveals two output bits (O) which are shown in I/OO format [9] 7

Figure 2.5 Trellis description of a convolutional encoder. The initial and the final states are the all-zero state [9] 7

Figure 2.6 Turbo Decoder 9

Figure 2.7 Parallelized Architecture for turbo code decoder 10

Figure 2.8 Repeat Accumulate code encoder 11

Figure 2.9 State transition and trellis diagram of the accumulator 12

Figure 2.10 Parallelized Repeat Accumulate Encoder 13

Figure 2.11 Iterative RA Code Decoder with APP algorithm 14

Figure 2.12 Parallelized Repeat Accumulate Decoder 15

Figure 3.1 The structure of a CLB in a Xilinx FPGA 17

Figure 3.2 The effect of dividing logic with flip-flops, pipelining 18

Figure 3.3 ML402 board used in the study 19

Figure 3.4 A generalized system model for testing the decoders 26

Figure 3.5 An LFSR with seed 0101100011001101 28

Figure 3.6 The 16th,14th,13th,10th bit are added and the result is forwarded to the beginning of the register 28

Figure 3.7 With a clock trig the found result is registered as the first bit of the register, the content of the register is shifted once towards right 28

Figure 3.8 Normally distributed noise generation by LFSR 29

Figure 3.9 The hystogram of a pseudo-random Gaussian noise generator obtained by collection of 10000 samples 31

Figure 3.10 A histogram of a noise sequence generated by MATLAB's randn function 32

Figure 3.11 The bit alignment in a UART transmission 33

Figure 4.1 A general block diagram of a communication system 36

Figure 4.2 Forward recursion in calculation of $\alpha^*_{l+1}(s)$ 40

Figure 4.3 Backward recursion in calculation of $\beta^*_l(s')$ 41

Figure 4.4 α and β values are initialized initially at time 0 43

Figure 4.5 α and β values are computed independently and in a recursive manner 43

Figure 4.6 α and β values first meet at time 10 and at this time all information for computing the first LL values are ready 44

Figure 4.7 An illustration of how a memory collision may happen in an encoding process 51

Figure 4.8 RCS-random interleaver is a good approach for memory collision freee interleaver design including the good properties of S-random interleavers [9]. 52

Figure 4.9 The interleaver operation taking place in the FPGA. Each address request is decoded, the requested RAM and the corresponding location is found and the requested data is forwarded to the demanding encoder. . . . 53

Figure 5.1 SNR vs BER for log-MAP based turbo decoder. 4 iterations for 2000 frames of 160 bits through 4 parallel MAP decoders. 61

Figure 5.2 SNR vs BER for max-log-MAP based turbo decoder. 4 iterations for 2000 frames of 160 bits through 4 parallel MAP decoders. 62

Figure 5.3 SNR vs BER for PDRAC decoder. 8 iterations for 2000 frames of 160 bits through 4 parallel MAP decoders. 62

Figure 5.4 *NormMax* values for log-MAP turbo code decoder for different bit representations. The average of 6000 packets of 160 data bits with 4 parallel decoders. 63

Figure 5.5 *NormMax* values for max-log-MAP turbo code decoder for different bit representations. The average of 6000 packets of 160 data bits with 4 parallel decoders. 63

Figure 5.6 *NormMax* values for repeat accumulate codes for different bit representations. The average of 6000 packets of 160 data bits with 4 parallel decoders. 64

Figure 5.7 SNR vs BER for RA with 4 parallel sub-decoders decoding 1344 bits in total with 8 iterations. 65

Figure 5.8 SNR vs BER for turbo decoder with 4 parallel max-log-MAP decoders decoding 1344 bits in total with 4 iterations. 65

Figure 5.9 SNR vs BER for turbo decoder with 4 parallel log-MAP decoders decoding 1344 bits in total with 4 iterations. 66

CHAPTER 1

INTRODUCTION

In wireless communication systems channel coding is one of the most important tools. By the help of strong channel codes the quality in communication can be highly improved. By recent developments and improvements in communication systems technology, reliable and high speed data transfer became an important issue. From satellite communications to wireless local area networks(WLAN), large bandwidth and high speed transfers with a minimum error probability are desired.

Since Shannon determined the maximum achievable rates for AWGN channels, many studies in channel coding have been conducted. One of the most significant studies conducted in this area is the study of Gallager introducing low-density parity check (LDPC) codes in 1963. However, LDPC codes were not popular due to their iterative docoders were impractical at those times. The most important study which ushered a new era in coding theory was introduced by Berrou et al. with the name turbo codes [4]. Right after the introduction of the turbo structure, this idea is applied to other coding schemes and this yielded to invention of many classes of codes, broadly called turbo-like codes. The family of Repeat-Accumulate codes is a well-known type of turbo-like codes [5].

Although turbo and repeat accumulate codes are efficient in terms of bit error ratio (BER) vs. signal to noise ratio (SNR) performance, their decoders introduce large decoding delays due to their iterative decoding scheme. As the number of iterations are increased, a better error performance is usually obtained but the time of decoding increases in proportion to the iteration number. In order to decrease such huge latencies various ideas have been implemented like building many decoders operating in

parallel which is a widely used technique in literature. This approach significantly decreases the decoding delay with almost no loss in terms of BER [9]. For further decrease in decoding latency, certain algorithms such as center to top algorithm are also utilized inside the marginal a-posteriori (MAP) decoders [13].

Parallelization is a powerful tool for decreasing the decoding latency. While constructing a decoder structure operating in parallel memory collision problems can occur. Decoders operating in parallel attempt accessing information residing in the same memory segment. Such problems can be avoided with the implementation of suitable collision free interleavers as studied in the thesis.

In this thesis, our aim is observing the performance of parallelized encoder and decoder structures for turbo and repeat accumulate codes implemented on an FPGA platform and investigating the parameters which affect their operation. The performances of the decoders will be evaluated in terms of BER performance, FPGA resource usage, maximum achievable FPGA clock speed, and data throughput.

The outline of the thesis can be summarized as follows. In Chapter 2, a general description on turbo and repeat accumulate codes is given. We explain the basic encoder and decoder structures for these codes and give a brief description for building decoders and encoders operating in parallel. In Chapter 3, the environment in which the implementations are carried on is described. In Chapter 4, we explain the implementation steps for obtaining a MAP decoder that is later used for constructing parallelized decoders. In Chapter 5, we demonstrate the SNR vs. BER performances of the proposed decoder architecture and discuss the implementation results. Also, the maximum throughput of the decoders are calculated in this chapter. Finally in Chapter 6, we conclude the thesis and provide suggestions for future work.

CHAPTER 2

TURBO CONCEPT

The noisy-channel coding theorem, stated by Claude Shannon in 1948 [16], opened a new era in communications. The theorem basically states that, one can transmit information reliably at information rates (R) smaller than a specific rate referred to as the channel capacity (C). The theorem implies that information transmission with arbitrarily small rate is possible with the conditon $R < C$. The theorem was the starting point of the Information Theory. As the years passed, many studies for obtaining the minimum available error rate over a noisy channel are conducted as attempts for achieving the Shannon Limit.

In order to enjoy a reliable communcation for wireless systems *Forward error correction (FEC)* schemes are used. FEC codes are designed to improve the decisions that the receiver makes by giving it enough information to correct some of the errors that the channel introduced into the signal. The technique is adding redundancy to the information by channel coding.

Channel coding can be thought as a process in which redundant bits are added to a series of bits which are to be transmitted to some receivers. The aim in this redundancy operation is to mitigate the effect of the noise on the transmitted signal. Since these bits are processed by some rule, the receiver side is expected to correct the erroneous bits as much as possible by the help of these redundant bits. There are many different coding techniques for different kind of situations. Bursts of errors, thermal noise or fading channel effects are some examples for these situations [19]. Channel codes can be broadly divided into two categories:

- **Block Codes** : Repetition codes, BCH codes, Reed Solomon codes are the most well-known codes in this category. These codes operate under fixed-size bit blocks. The messages of k bits are mapped to codewords of length n bits. The code rate, R, for an (n, k) block code is then given by

$$R = \frac{k}{n}. \tag{2.1}$$

For k bits of information there exist $n - k$ bits of redundancy.

- **Convolutional Codes** : They can operate under varying size of blocks. Their encoders and decoders are usually less complex compared to that of block codes. This type of codes constitute the basis of this thesis.

After the genesis of Information theory, a number of capacity achieving codes have been invented. The oldest of these codes as first introduced by Gallager in his doctoral thesis in 1963 [8]. The class announced by Gallager was the low density parity check (LDPC) codes, but these codes did not gain popularity up until the invention of turbo codes. Berrou et al. introduced turbo codes [4] in 1993. Turbo codes attracted the attention of the researchers with its good error performance. After these developments a return to LDPC codes occured and people restarted studying them. The class of repeat-accumulate codes introduced in [5] is the result of these efforts.

Turbo codes enjoyed a grand fame with its good error performance and reasonable complexity. After a few years of its invention almost everyone in the area of coding theory agreed that it is a pioneering achievement in the area. As a result, it is accepted as among the coding techniques for next generation wireless communication systems such as Wideband CDMA (WCDMA) and 3rd Generation Partnership Project (3GPP) for IMT-2000.

Another important class of codes are repeat-accumulate (RA) codes. RA codes which is a special type of LDPC codes, is first introduced by Divsalar et al. in [5]. RA codes are known for their low complexity decoder and good error performance.

In this chapter brief information about turbo codes and repeat accumulate codes will be given. The parallelization idea for the encoders and the decoders of these codes also will be explained. The parallel decoder and encoder structure under investigation will be presented.

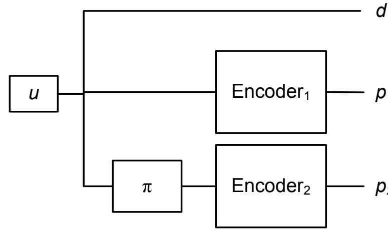


Figure 2.1: Encoder Structure of a Turbo Code

2.1 TURBO CODES

2.1.1 Turbo Code Encoder Structure

In general the encoder structure of a turbo code is parallel concatenation of two encoders. Figure 2.1 depicts the structure of a turbo code encoder. The information bits, a sequence of bits u, are passed through encoders and bypassed. d is the bypassed version of u which is also called as the systematic part. p_1 is the parity bit sequence which is obtained by passing u through an encoder, a convolutional encoder in our case, and called *parity bits* throughout this thesis. The block π represents the interleaver block by which the turbo codes gains its power. p_2 represents the parity bits obtained by encoding of "interleaved" data bits which is called the *interleaved parity bits*. The codes Berrou et al. used was convolutional codes and their scheme was called parallel concetenated convolutional codes (PCCC).

For mitigating the encoding decoding latency parallelization of encoders and decoders are suggested in the literature. The parallelized form of the encoder structure is not too much different from the usual form. The parallelized form of turbo encoders can be seen in Figure 2.2. At this point the most crucial subject is the design of a collision-free interleaver block which will be handled in the preceding sections. For ease of demonstration u, p_1, p_2 and π can be thought as matrices of size [n x N], where n is the codeword length passed through a single encoder.

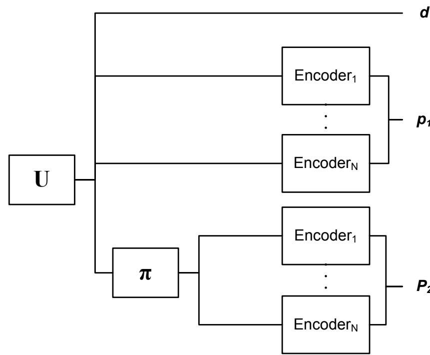


Figure 2.2: Parallelized Turbo Code Encoder Structure

2.1.2 Convolutional Encoding

In convolutional encoding the output bit streams are generated with a state transition matrix and an input bit stream. Their operation principle can be thought as a finite state machine in which n bits of input corresponds to k bits of output. k/n results in the rate, R, of the encoder. Algorithmically there are two main parts of a convolutioanal encoder, a shift register and binary adder blocks. The number of locations in the shift register is indicated by m_i and the constraint length of the encoder is defined as $\max(m_i + 1)$ [19]. Another important parameter in convolutional codes is $minimum\ free\ distance$, d_{free}, defined as the minimum Hamming distance between any two output sequences. Figure 2.3 depicts a rate 1/2 convolutional encoder with

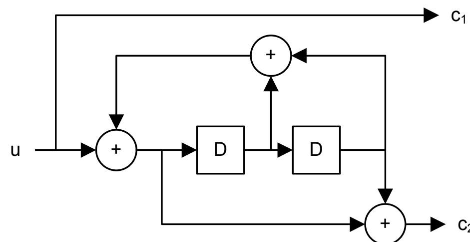


Figure 2.3: A Rate 1/2 Convolutional Encoder

$m_1 = 2$, with one information bit u and two coded output bits c_1 and c_2. Since the shift register is composed of 2 storing elements, this encoder has $2^2 = 4$ states. Its constraint length is 3 and minimum distance is 5. We already stated that the convolutional encoders can be thought as finite state machines (FSM) Figure 2.4 explains

how they can be treated as state machines. Figure 2.5 shows the trellis diagram of

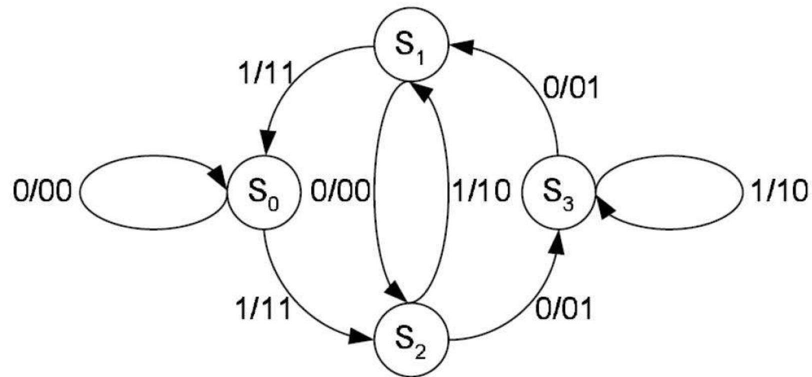


Figure 2.4: The FSM representation of the convolutional encoder shown in Figure 2.3 each bit arrival(I) contributes to a state transition and reveals two output bits (O) which are shown in I/OO format [9]

the encoder and shows how the transitions may occur over time. The encoder used as an example in Figures 2.3, 2.4 and 2.5 will be the default encoder in the encoder of parallel decodable turbo codes (PDTC).

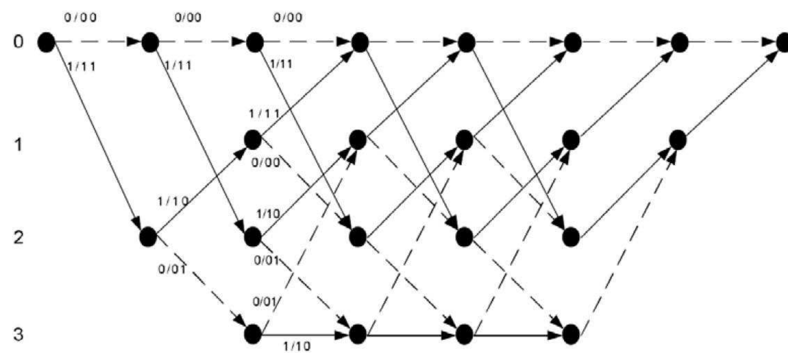


Figure 2.5: Trellis description of a convolutional encoder. The initial and the final states are the all-zero state [9]

After the encoding of a frame is finished, the final state of the encoder can be adjusted to be in a known state for getting a better performance in decoding, as shown in Figure 2.5. In general both the initial and the final states of the encoder is adjusted to be the all-zero state where all of the shift registers are zero. If the final state is also wanted to be controlled then *termination bits* must be added to the frame. The length of the termination bits must be m_i at least.

Many convolutional encoders do not employ feedback, and thus can be thought as finite impulse response (FIR) filters. Recursive convolutional encoders have a feedback component which makes the encoder behave as a infinite impulse response (IIR) filter. Our example is a recursive convolutional encoder with feedback.

2.1.3 Interleaving

Interleaving means changing the place of a bit in the sequence to a newer place such that the initial and the final location of the bits are related to each other with some rule. In wireless channels, transmission suffers from fading problems, which results in bursts of errors. A well defined interleaver decreases the probabability of error by distributing the erroneous consecutive bits far from each other. So at the output of the interleaver the errors seem to be independent of each other. Besides, interleaving enhances performances of turbo codes by reducing the number of low weight codewords [6].

The *S-random* interleaver will be the interleaver type to be used in the designs. The steps for producing an *S-random* interleaver can be given as follows [1]:

1. All the mappings occur randomly with equal chance of selection
2. The randomly selected order is accepted only if it is in a distance greater than S for all of the S previously selected orders. Otherwise, it is not accepted and a new random order is generated, until this condition is satisfied.

The parameter S is predetermined and usually satisfies $S \leq \sqrt{K/2}$, where K is the interleaver size [6]. *S-random* interleavers have good spreading characteristics compared to other interleavers and provide good BER performance when used with convolutional codes.

2.1.4 Turbo Code Decoder Structure

In the original turbo code study the scientists used a modified version of the Bahl et al. (known as BCJR [14]) algortihm [4]. The iterative turbo decoder can be seen

in Figure 2.6. The decoder given is the decoder for the turbo code generated from

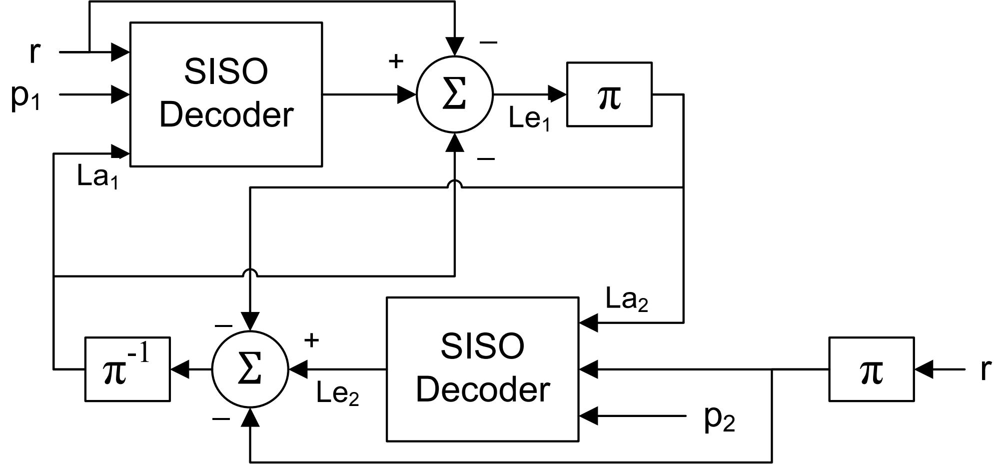


Figure 2.6: Turbo Decoder

recursive systematic convolutional (RSC) codes. r represents the channel observation corresponding to the systematic data, p_1 corresponding the parity bit produced by the use of systematic part and p_2 to the parity bit produced by the use of the interleaved version of the systematic data. The soft in soft out (SISO) decoders can be any decoder. Soft output Viterbi algorithm (SOVA) decoders and the BCJR-MAP decoders are two commonly used decoders among many. In our study a MAP decoder implemented by the BCJR algorithm will be used.

Decoding latency is a big issue in iterative decoding of turbo codes. In order to decrease the latency, a parallelization of decoders may be proposed likewise in the encoder part [9]. The parallelized decoder architecture for a turbo code is given in Figure 2.7. The number N of parallel processing SISO decoders decrease the decoding latency approximately N-folds. Although there is a small performance loss as N increases as observed in [9], the significant latency enhancement justifies parallelization.

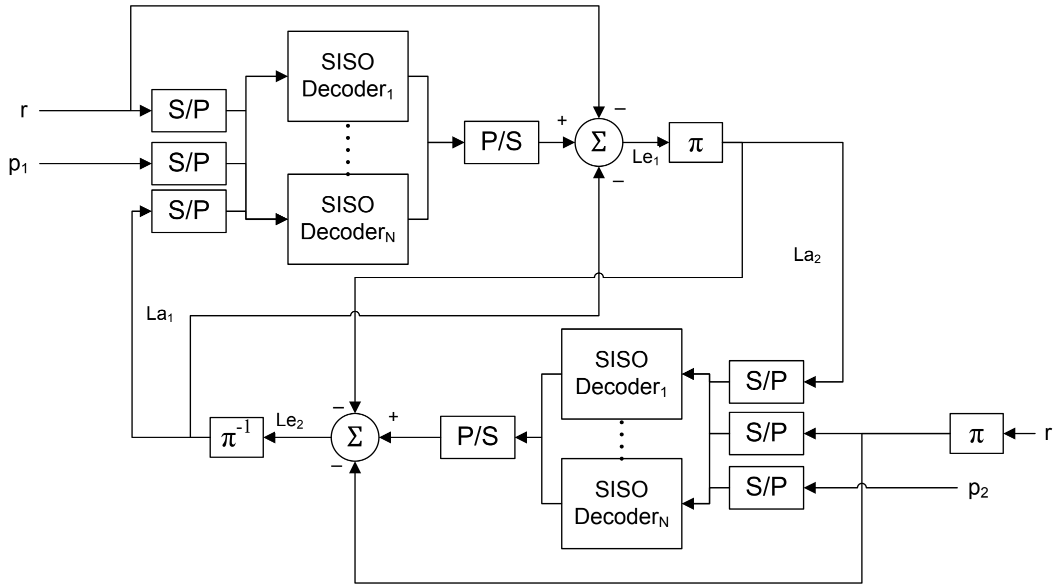


Figure 2.7: Parallelized Architecture for turbo code decoder

2.2 TURBO-LIKE CODES

In this section parallelization of the repeat accumulate codes are discussed. Repeat accumulate (RA) codes are considered to be a sub-class of low density parity check (LDPC) codes. Although there are studies on the parallelization of turbo codes, the same can not be told for repeat accumulate codes.

2.2.1 Low Density Parity Check Codes

Low density parity check (LDPC) codes introduced by Gallager [8] are the first known channel coding family that performs close to the Shannon limit. When Gallager introduced this type of coding in 1960’s, researchers did not give importance to these codes because of its large decoding complexity. After the invention of turbo codes a return to Gallager’s study occured. Nowadays, many studies on analysis of LDPC codes of different variants are taking place.

2.2.2 Repeat Accumulate Codes

Repeat Accumulate codes are first introduced by Divsalar et al. in 1998 [5]. After the introduction of turbo coding principle Divsalar used this concept and invented the RA codes. An RA code can be decoded iteratively and its iterative decoding performance is considerably good despite its low complexity, whereas its coding is simple and the decoder structure is suboptimal [12]. Additionally, RA codes achieve the ultimate Shannon limit -1.592 dB as the code rate goes to zero on the AWGN channel.

2.2.3 Repeat Accumulate Code Encoder Structure

The RA encoder consists of concatenation of a Z-times repeating repetition encoder and an accumulator. If the information bits are transmitted, this type is called systematic RA code. Sometimes the repetition part works in an irregular way, that is, it repeats each bit Z_i times where Z_i is a variable parameter for each uncoded bit at time i, on operation. Irregular repeat accumulate (IRA) codes are formed in this way. IRA codes are actually better codes which excite the curiosity of the coding theorists.

The basic encoder structure of a non-systematic RA code can be seen in Figure 2.8. The information bits are repeated Z times and forwarded to an accumulator. Before the accumulator there exists an interleaver which is one of the most important part in the code since the existence of the interleaver brings the power of the RA codes, as in turbo codes.

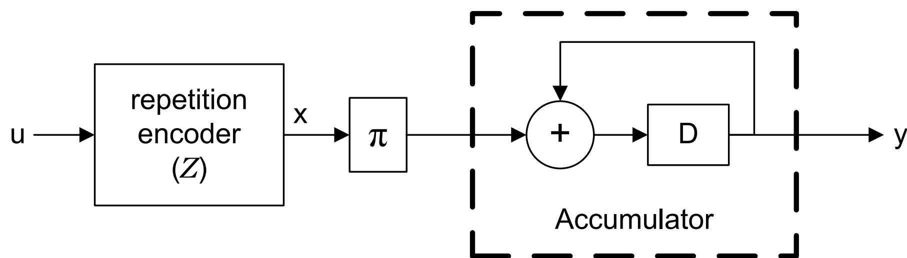
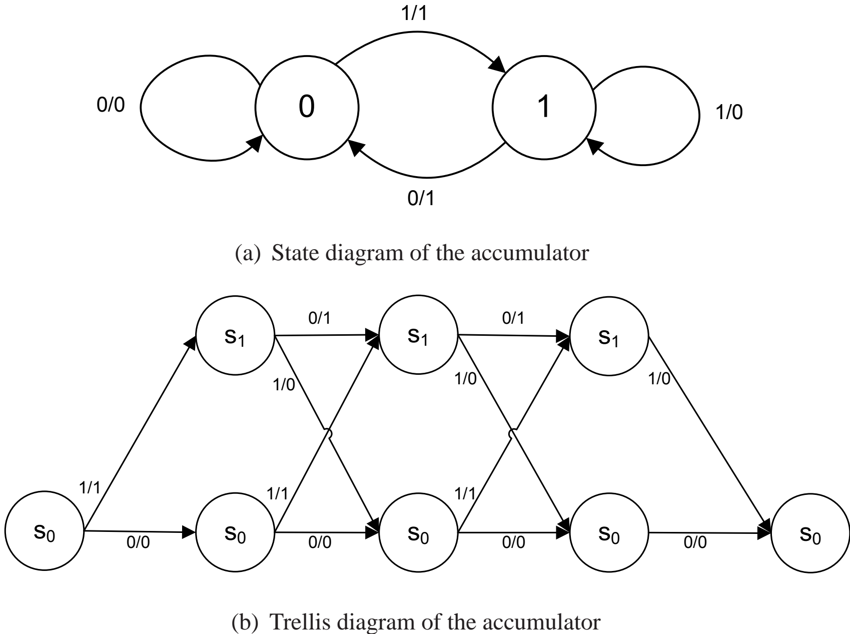


Figure 2.8: Repeat Accumulate code encoder

The accumulator is the part which makes the RA codes simpler compared to other

coding schemes like LDPC or Turbo codes. The accumulator, as it can be seen in Figure 2.8, performs a modulo-2 adding operation. It sums up the current bit with the previous bit and produces what is called a parity bit. From one perspective it can be thought as a 2-state convolutional encoder with transfer function $1/(1+D)$ whose state transition diagram is given in Figure 2.9(a) and trellis diagram in Figure 2.9(b). From another perspective it can be seen as a block code with inputs $[x_0, \ldots, x_{n-1}]$ and



(a) State diagram of the accumulator

(b) Trellis diagram of the accumulator

Figure 2.9: State transition and trellis diagram of the accumulator

outputs $[y_1, \ldots, y_{n-1}]$ whose equations can be given as

$$
\begin{aligned}
y_0 &= x_0 \\
y_1 &= x_0 + x_1 \\
y_2 &= x_0 + x_1 + x_2 \\
&\vdots \\
y_{n-1} &= x_0 + x_1 + x_2 + \ldots + x_{n-1}.
\end{aligned}
$$

The performance derivations of RA codes are done by using its block code behavior but its opearating principle is easy to understand with its convolutional form.

One way for enabling parallelization at the receiver is by the parallelization of the encoders. A parallelization scheme for the encoders can be seen in Figure 2.10. A

number M of repeaters are processing in parallel and forwarding the results to an interleaver. An accumulator cluster consisting of N parallel processing accumulators encodes the repeated bits.

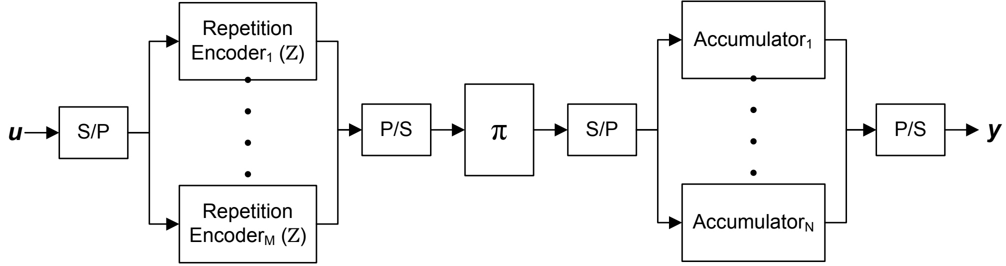


Figure 2.10: Parallelized Repeat Accumulate Encoder

2.2.4 Repeat Accumulate Code Decoder Structure

RA codes can be decoded using variable techniques [15]. Majority-logic (MLG) and bit-flipping (BF) decoding are two examples for hard decision decoding. A posteriori probability (APP) decoding and iterative decoding based on belief propagation (IDBP) which is also known as sum-product algorithm (SPA) are soft decision decoding techniques. Weighted BF decoding is a compromise between hard decision and soft decision decoding. Techniques including hard decision decoding are out of our scope since soft decision decoding algorithms usually provide better performance. The SPA algorithm is the most widely used decoding technique for decoding of RA codes. In the SPA decoding, Tanner graphs [18], introduced by Tanner, are used by the information passing algorithm, generally known as belief propagation. In this study we will focus on using APP decoding by using a BCJR-MAP decoder.

Theorem 3.2 stated in [12] says that if Z goes to infinity then the SNR threshold value, γ_Z, which is the lowest bit SNR value for error free transmission, approaches $log2$, that is, RA codes achieve the Shannon limit for the AWGN channel. Table 2.1 shows a comparison between Z, the achievable SNR threshold value for error-free communication and the corresponding Shannon limit for the related rate. Decoding latency is again an issue for the iterative decoding of RA codes. In order to decrease the effect of this inherent latency, a parallelized architecture for the decoder is pro-

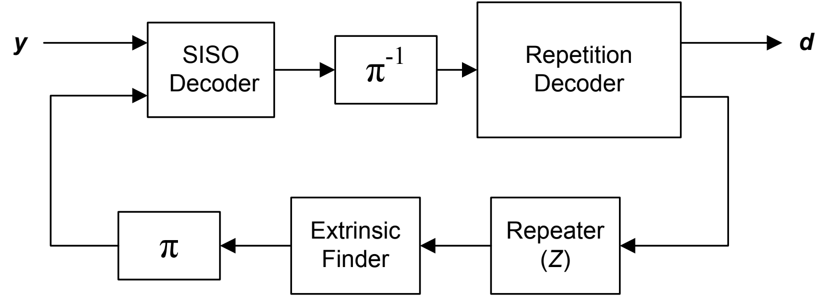


Figure 2.11: Iterative RA Code Decoder with APP algorithm

Table 2.1: Rate, Threshold and Shannon threshold comparison for RA codes [12]

Z	R	γ_Z (dB)	Shannon (dB)
3	1/3	0.792	-0.495
4	1/4	-0.052	-0.764
5	1/5	-0.480	-0.963
6	1/6	-0.734	-1.071
$\vdots$	$\vdots$	$\vdots$	$\vdots$
∞	0	-1.592	-1.592

posed. The decoder block can be seen in Figure 2.12. N SISO decoders operating in parallel first decode the incoming data sequence since this part was encoded by the accumulator in the transmitter. The likelihoods generated by SISO decoders are passed through a deinterlaver and decoding continues with the M number of repetition decoders. If there are termination bits in the received sequence, which improves the error performance of the code N must be the same as that of in the encoder part (as it can be remembered there were N number of parallel encoders in the PDRAC encoder). On the other hand, M has no need to be the same as M in the encoder part, because only the repeated bits are related to each other. As M increases the decoding latency decreases significantly however, this time memory operations must be handled carefully. For the subsequent iterations decoded bits taken from the repetition decoders are passed through repeaters and then an extrinsic finder calculates the a priori probabilities for the next iteration of the SISO decoders.

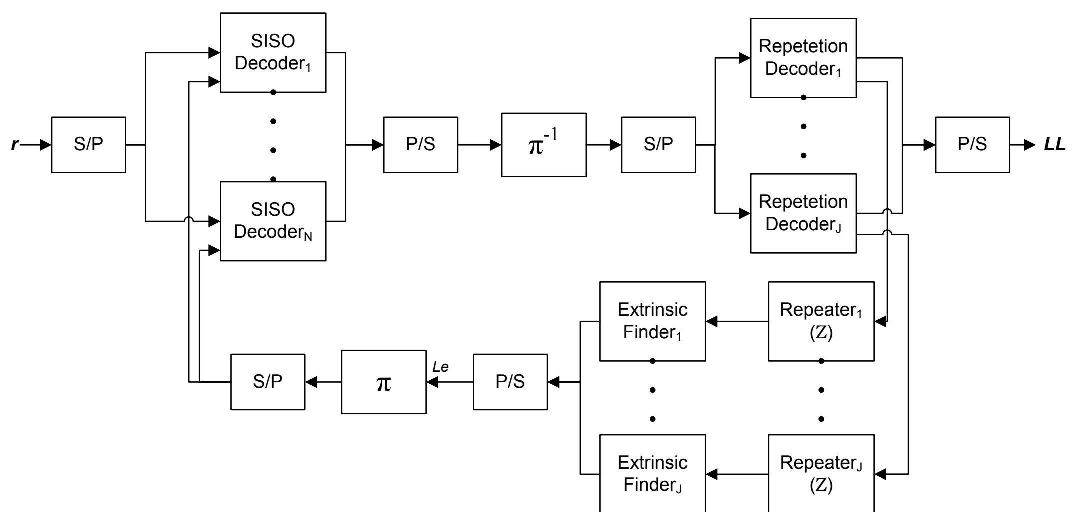


Figure 2.12: Parallelized Repeat Accumulate Decoder

CHAPTER 3

TESTBED SETUP

In this thesis the motivation was basically the hardware realization and comparison of parallelized turbo and repeat accumulate decoders. Initially the study was a continuation of a previous thesis [1]. In this previous study an integrated testbed environment was implemented. However, because of some unresolvable and unexpected problems occured later on the testbed, the designs are carried on a stand-alone operating environment. The characteristics of a real environment is simulated on the FPGA platform.

3.1 ML-402 FPGA Evaluation Board

FPGAs are *reconfigurable logic devices* composed of smaller logic blocks. The building blocks of the FPGA are called *Configurable Logic Blocks* (CLB). A CLB (denominated as *Slice* in Xilinx[1]), shown in Figure 3.1, is the smallest building block of a Xilinx FPGA and for all of Xilinx FPGAs the CLB structure is the same [2]. A slice is composed of two four-input LUTs, six various size multiplexers, and two flip-flops (FFs). Although the logic operations are done with gates in the schematic designs, these gates are embedded into the LUTs in the hardware. When the inputs of the LUTs are excited, the output yields a result which is adjusted to yield the same result as the logic circuits would yield in the schematic design.

All of the logic blocks are connected to each other with programmable switches.

[1] To get more information about the famous FPGA manufacturer visit www.xilinx.com
[2] This structure will change after the not yet manufactured Virtex6 and Spartan6 products

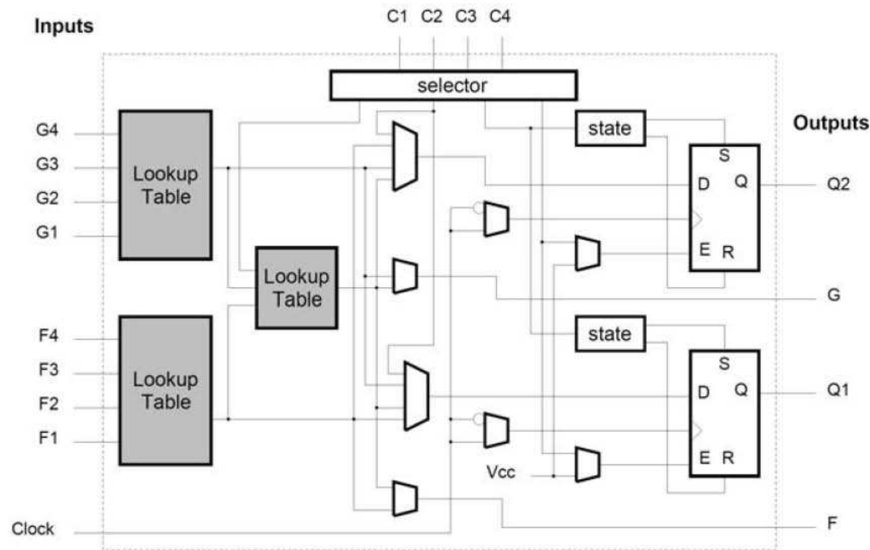
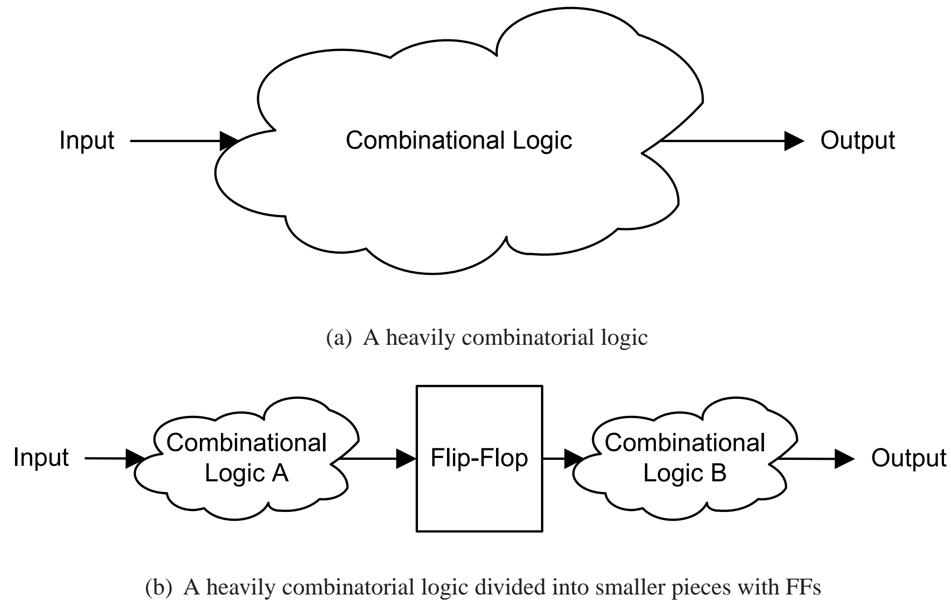


Figure 3.1: The structure of a CLB in a Xilinx FPGA

If there is a relation between slices then these switches will be ON, else OFF. The results of the LUTs can be multiplexed to the slice flip-flops in the case of a need for storing the result. One other important role that flip-flops introduce to the design is pipelining. Pipelining is the most powerful tool for obtaining a fast operating module. Since FPGA is formed by transistors each transistor has a certain delay in response to excitements. If large combinatorial logical blocks are used then these delays can reach tremendeous levels. If these large logic cells are analyzed well and partitioned into smaller blocks by placing flip-flops in between, these partitions will enhance the operation speed of the design. Figure 3.2(a) and Figure 3.2(b) shows this situation. If the large logical block is divided into smaller blocks A and B, with the necessity that the block A does not need the result coming from block B instantaneously, delays will decrease and maximum available clock speed will increase. In this case the output of the design is postponed by one clock period.

Another important block that is available in FPGAs is the block RAM, shortly BRAM. A BRAM in a Xilinx FPGA consists of 18x1024 bits[3]. By cascading these RAMs in parallel or in serial one can obtain RAMs of different sizes. The advantage

[3] This size of BRAMs are the same for all Xilinx FPGAs



(a) A heavily combinatorial logic

(b) A heavily combinatorial logic divided into smaller pieces with FFs

Figure 3.2: The effect of dividing logic with flip-flops, pipelining

of RAMs is that they are cheaper compared to slices and for storing purposes they provide a larger area. However, their operating speed is slower than that of slices.

Some FPGAs also include embedded microprocessors, for example, some Xilinx FPGAs include PowerPC cores. By the help of well picked peripherals it is easy to convert a microprocessor into a fully operating microcontroller. Besides, Xilinx MicroBlaze and Picoblaze are available as soft processor blocks, for the FPGAs which do not include PowerPC. As explained before, since FPGAs are very flexible devices in terms of logical operations, a processor composed of slices can be easily embedded into an FPGA. The advantage of such a solution is that, FPGAs without hard processors are cost effective. The disadvantage of it is, a soft core processor consumes some of the resources of the FPGA so available number of logic blocks decreases.

FPGAs also include additional blocks such as clock management blocks which can be used for multiplying/dividing clocks and/or mitigating the clock skews, dedicated multi-gigabit input/output ports, input/output (I/O) buffers those are compatible with many electrical standards such as LVTTL, LVDS, LVPECL etc., dedicated fast binary multipliers and so on.

The board we used in designing the encoders and decoders was ML-402 Virtex4

Evaluation Board, shown in Figure 3.3. This board carries a Xilinx Virtex4 SX35 FPGA on it which can be thought to be an average capacity FPGA compared to others in the industry. Besides containing an FPGA, the ML-402 board contains other chips which can be used extensively for many different applications.

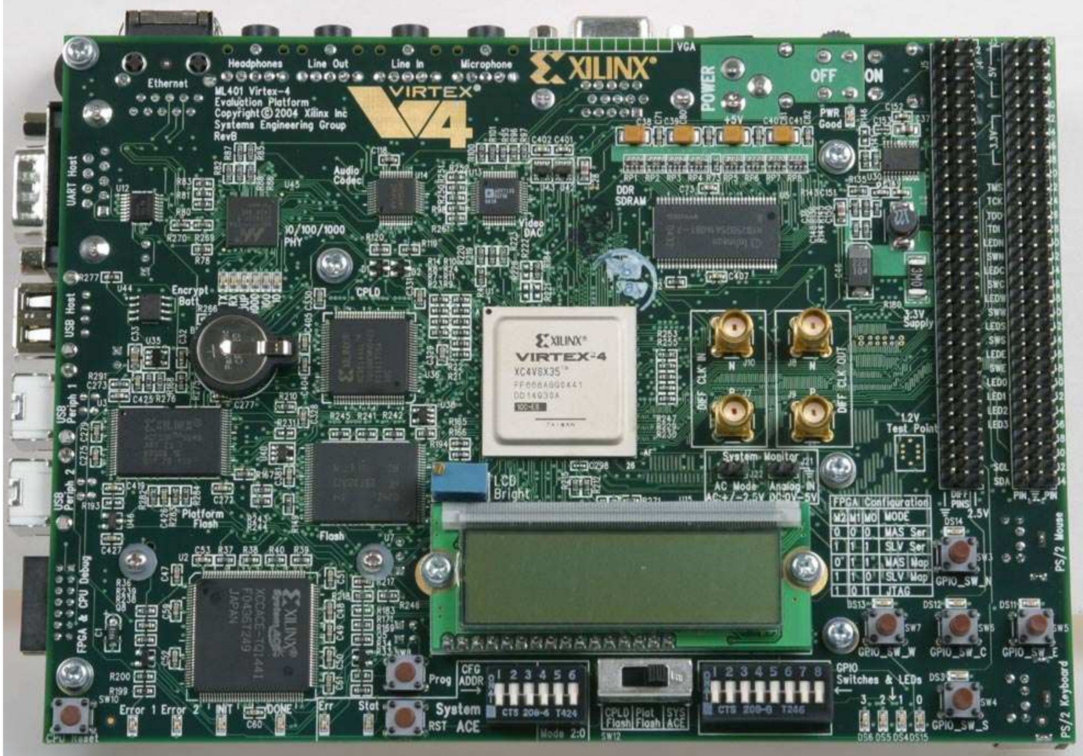


Figure 3.3: ML402 board used in the study

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 them form a 32 bit data bus capable of running up to 266 MHz [11]. In the presence of a microprocessor these RAMs can be used for external data storage in 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 bit synchronous ZBT RAM. The ZBT RAM provides a high speed low-latency external memory to the FPGA. This RAM can also be used for temporary external storage.

10/100/1000 Tri-Speed Ethernet PHY : The board contains a Marvell Alaska

PHY device operating at 10/100/1000 Mbps. By the use of a small porocessor and an ethernet controller the board can be reached through ethernet connection. One application can be incorporating the device into a local area network and reaching to it over ethernet connection.

Differential Clock Input And Output With SMA Connectors : High precision clock signals can be fed to FPGA by the use of 50 Ω SMA connectors. This functionality allows the FPGA to be fed by function generators. For example a demodulator output can be connected to the board, hence, further decoding process can be applied.

RS-232 Port with Direct FPGA connection : The ML-402 board contains an DB-9 serial port allowing the FPGA communicate with another device using serial data. An interface chip changing the voltage-levels are also included. The RS-232 serial port is one of the most widely used communication protocol and is known for its low-weight receiver/transmitter structure. In the thesis RS232 is commonly used for simulation purposes. The FPGA communicated with a PC through this port and the PC interpreted the results coming from the FPGA.

Compact Flash and System ACE: The board contains a Xilinx System ACE Compact Flash (CF) configuration controller. Through the JTAG port both the hardware and the software data can be downloaded to the CF. Maximum eight configuration images on a single CF card can be supported by SystemACE controller. By the help of switches available on the board, the address of each configuration can be selected and then System ACE controller loads the FPGA with that configuration. Besides being used as a configuration storage, a CF can also be used as a FAT filesystem storage device, i.e. harddrive.

3.2 Software Used For Debugging and Implementation

3.2.1 Xilinx ISE and XST

Xilinx ISE (Integrated Synthesis Environment) is the Integrated Development Environment (IDE) designed by Xilinx as a graphical user interface (GUI) for synthesizers. Xilinx synthesis tools (XST) is one of the synthesizers developed by Xilinx

for complete synthesis and implementation of an FPGA project. The free version of ISE which is called the Webpack Edition supports a limited number of Xilinx FPGAs which are generally small in size. The unlimited version supports all of the FPGAs fabricated by Xilinx.

3.2.2 Implementation steps of an ISE project

The implementation steps of a Xilinx project is divided into steps. Each step has certain inputs and outputs.

3.2.2.1 Synthesis

By synthesis, “logic synthesis” is meant and it is an important step before implementation. Since FPGA design is a hardware process unlike compilers, these tools are named as synthesizers. The Synthesis operation basically converts a hardware description language (HDL) into register transfer level (RTL) composed of logic blocks like gates related to the design architecture. Another job done in the synthesis level is the optimization of the design. Optimization is done in a way that the synthesis tool either wipes out unused signals and entity ports, or it reduces the number of gates if two or more gates do exactly the same job. In XST, optimization of the design can be limited by special built-in constraints. For example, a general issue about this topic is that a two signal exactly created by the same logic will be optimized by XST by deleting one of these signals. However, these two signals will survive by setting the “remove equal logic” constraint accordingly.

3.2.2.2 Translate Process

Translate process is the first step in the implementation process. The translate process produces a Xilinx native generic database (NGD) file which includes all of the netlists and design constraint information for implementataion. A netlist is the combination of the blocks such as counters, adders, multipliers, comparators so on and connections between them. This process combines these pieces of information in a way that the

logic is mapped into the target FPGA.

3.2.2.3 Mapping Process

This step follows the translate process. The mapping process takes the previously created NGD file, runs a design rule check (DRC) over this file and maps the logic into the FPGA-specific hardware blocks. If one or more of the constraints is not applied properly, an error will pop up at this step and report that the constraint is not applicable. For example, if a buffer compatible with low-voltage differential signaling (usually called as LVDS buffer) is instantiated in an FPGA project its differential ports must be tied accordingly since these ports are connected to deterministic pin locations. However, if mistakenly the pin locations for that LVDS buffer is tied into irrelevant pin locations the synthesis process will not issue any error since the LVDS buffer instantiation is done in a correct way. Additionally, the translate process will not issue an error too since synthesis is fine and the "declaration syntax" of the pin locations are also correct. In the mapping process the software will check whether the LVDS receiver input pins are suitably placed or not. Since the LVDS pins are dedicated for each LVDS buffer, the mapping process will issue an error because of the failure in the constraints of the location of the LVDS pins and the implementation process will stop. The result of a successful map process will be written in a Xilinx native circuit description (NCD) file.

3.2.2.4 Place and Route Process

The place and route (shortly called as PAR) process is executed after the mapping process finishes and takes a mapped NCD file and places and routes the design. This process can be thought as an auto-router like in a printed-circuit board design software. Since the blocks and the constraints are known from the previous mapping process, it tries to connect all blocks in accordance with the netlist and the constraints. This process places all of these FPGA blocks in such a way that all limitations are satisfied, all I/O pins are connected, and the design will not go into an erroneus state with the specified clock speed. The output of this process is an NCD file suitable for

the operation of BitGen software.

3.2.3 BitGen

BitGen is a programming file generator for Xilinx FPGAs. After the implementation process finishes this software takes the NCD file and produces a .bit file which is suitable for programming an FPGA. If the programming finishes successfully, the FPGA will act in the way it is wanted by the HDL code.

3.2.4 ChipScope Analyzer

ChipScope is an advanced real-time debugging and verification tool designed by Xilinx. The ChipScope tool embeds special low-level soft core blocks into the design or into the netlist of the FPGA in order to track the signal changes. After PAR finishes, these blocks are ready to send data to the PC via a special port called the JTAG port. These cores, can be adjusted by software to be triggered in real time at certain conditions. When the set conditions are met, the states of the selected signals are examined and stored for a period of time. When the desired number of samples are taken these values are sent to the PC and the states of the signals are observed by the help of a GUI.

BRAMs of the FPGA are used for storing the signals.A maximum of 255 signals can be observed for a maximum of 16384 samples. Of course these are the achievable maxima allowed by the software. If the FPGA is not a large one, large Chipscope blocks are impossible to embed because of resource limitations.

3.2.5 MATLAB

MATLAB is used in various phases of the study. Since it provides a high level programming environment, a code can be changed and tried in minutes by the help of this software. It also provides a visual interface into the variables so that, by the help of the naked eye most of the problems can be seen and solutions can be produced. MATLAB is used firstly for the implementation of the encoders and the decoders. By

verifying that the results of the encoders and decoders are as desired, the discretized (fixed-point rather than floating-point) versions of them are written and simulated. After this step, the FPGA implementation and simulation is carried on in an easier way. It must be noted that any HDL is low level compared to any of the programming or scripting languages like C, C++, MATLAB since HDL deals with RTL. For example, an algorithm designed in a programming language in a few hours can be fully simulated on a HDL platform over weeks. The benefit of MATLAB can easily be seen. Another situation MATLAB was used is the generation of look-up tables. The approximations performed to decrease the complexity of the designs are also simulated in MATLAB, so that, the designs continued in a more confident way. After programming the FPGA with the implemented decoders, MATLAB is also used for to observe how the decoders operate on the FPGA. A controller module written in the FPGA was listening the commands transmitted from MATLAB through RS232 and was returning information such as the bit error rate, the frame error rate, the SNR, and the number of decoded packets. Such received information were processed in MATLAB, and illustrative results and plots were obtained.

3.2.6 MODELSIM

Modelsim is an advanced simulation and debugging tool for ASIC and FPGA projects provided by Mentor Graphics which is one of the leaders in the electronic design automation (EDA) industry. ModelSim recently started supporting many hardware description languages (HDL) including VHDL, Verilog HDL, SystemC, SystemVerilog. ModelSim has 3 major distributions, ModelSim SE, ModelSim PE, and ModelSim LE. Special distributions are also distributed for FPGA vendors. ModelSim provides ModelSim XE (Xilinx edition) for Xilinx. ModelSim XE is distributed with 2 licenses, one is a free but limited license and a full license which must be purchased from Xilinx. The full version is 100 times faster than the free version and the free version also additionally slows down if the HDL code is more than 1000 lines. In order to use ModelSim with ISE and a Xilinx FPGA a compiled form of the Xilinx FPGA blocks (these blocks are FPGA specific blocks that only the FPGA vendor may distribute the simulation models) must be available. Basically, ModelSim compiles a HDL code into a form that is suitable for the operation of ModelSim. In order to use

the HDL code in consistence with the FPGA specific block, the compiled version of the Xilinx blocks must be added to the library of the ModelSim.

ModelSim is a very helpful software for debugging a project with its well-designed GUI. The best approach for debugging a code is simulating the design module by module. Writing successful testbenches are important at this step. The integration of modules to each other will be less painful after successful tests.

Simulations can be divided into two main parts. One is functional simulation and the other is timing simulation.

A functional simulation simulates the behavior of the code. The timing in the simulation will be perfect compared to the real world behavior. As the name implies, this simulation simulates the functionality of the HDL code. No optimizations or simplifications occur, you see what you write. This kind of simulation is the fastest since no gate delays, IOB delays, clock skews, and setup-hold violations are observed.

A timing simulation (or equivalently "Post place and route simulation") can be done after the PAR process finishes. This simulation style is the most reliable one. If the post-PAR simulation is successfully applied, it is highly predictable that the design will operate after it is loaded into the FPGA. The timing simulation is slower compared to the functional one. In this simulation all of the components and routings used in the FPGA are replaced by its simulation models, so the skew and latency of a signal can be easily observed. The setup and hold times of the flip-flops, gate delays, and IOB delays are tracked and erroneus situations are reported by the simulator. In this simulation it is highly predictable that the designer will not be able to see all of the codes written. That is simply because all of the design is created after optimization steps. So the designer must be aware that if the functional simulation results are not alike the post-PAR one, the code must be rechecked. A synchronous design is the most reliable design because most probably it will operate without timing failure.

ModelSim can also be used via a console. It supports TCL (Tool command language) scripts so the software can be used without the need of a graphical interface and the simulations speed up.

3.3 Overall System Setup

In this section the setup that is used to implement and simulate the decoder performance will be explained. The important step in this part is the realization of a channel on the FPGA. Additionally how the FPGA get into contact with the outer world will be dealt.

3.3.1 System

A general system model can be seen in Figure 3.4. The decoder block consists of a decoder and some auxiliary modules for proper operation which will be described in the upcoming sections.

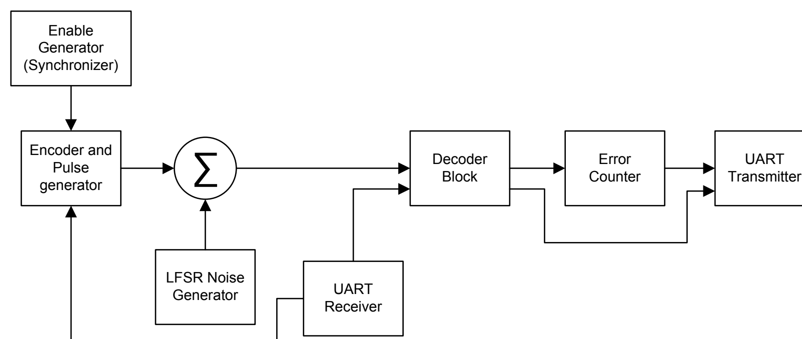


Figure 3.4: A generalized system model for testing the decoders

The "Enable Generator (Synchronizer)" block perodically produces an enable signal which triggers the encoder block. Because of the chain structure of the design and the periodic behavior of this module, it can also be called the synchronizer. After the encoder block receives an enable signal, it starts to produce the encoded version of a known sequence. When the encoded sequence is ready, this module also produces a ready signal to indicate that the sequence is ready for decoding. The encoded data is scaled by a parameter received from the outer world via UART. This parameter is $\sqrt{E_s}$ which is the amplitude of the encoded data. Hence, the energy of each bit in the sequence is E_s. After the encoded data is produced and multiplied by $\sqrt{E_s}$, a pseudo-random noise is added to the sequence. The pseudo-random noise is generated by the

LFSR noise generator module. Afterwards, the noisy sequence is forwarded to the decoder block. The information produced by the decoder block, which will also be discussed in the upcoming sections, is transmitted to the outer world through UART port.

3.3.2 LFSR Noise Generator

While testing the implemented decoders, varying inputs must be fed into the decoder along with many different noise realizations for a proper test operation. In order to generate a realistic environment in our study, random data generation algorithms are used in the FPGA. In the random number generation procedure an initial state called the *seed* is assumed, and the numbers are generated by the use of this seed. Since the state of the generator can be known in any time because of its seed-based structure the numbers generated are actually pseudo-random in nature. Then we have the chance to test the design for so many different inputs. By obtaining a gaussian like distribution we also have the opportunity of creating an AWGN channel in the FPGA. Among various random number generator algorithms a *linear feedback shift register* (LFSR) based one is chosen here because of its simple structure and wide usage. An LFSR is called linear becaues it is composed of binary linear operations, basically *xor* (exclusive or). Besides, it has a feedback structure in which a generated bit value is fedback to the shift register again. The feedback operation is done under a special characteristic equation. As the size of the shift register varies, the characteristic equation changes. Table 3.3.2 shows some characteristic feedback polynomials and corresponding register widths [3].

Figures 3.5, 3.6, and 3.7 shows the operation cycles of an LFSR composed of a 16 bit shift register. The shaded bit locations are called as taps which also demonstrates the locations those are described in the characteristic polynomial, namely 16th, 14th, 13th, and 10th locations. The initial state of the LFSR is called the seed of the LFSR and it can be any sequence of bits. As the name implies, the shift-register operates with the existance of a trigger signal, generally a clock. When the clock triggers the register, the *xor*-ed value is fed into the initial bit location and all of the bits are shifted. The last bit is the result coming from the shift register. When a seed is

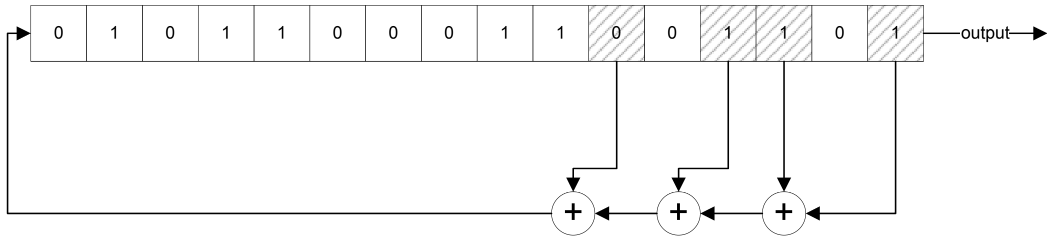


Figure 3.5: An LFSR with seed 0101100011001101

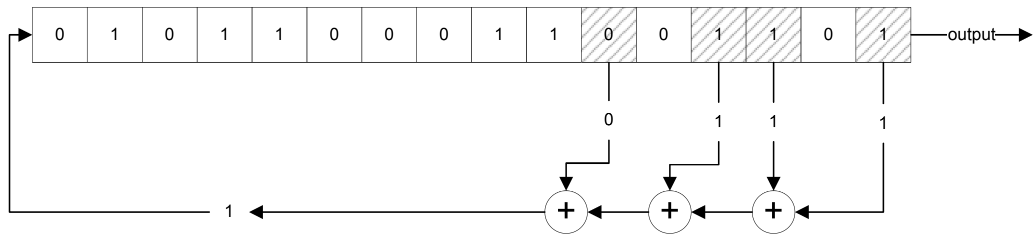


Figure 3.6: The 16th,14th,13th,10th bit are added and the result is forwarded to the beginning of the register

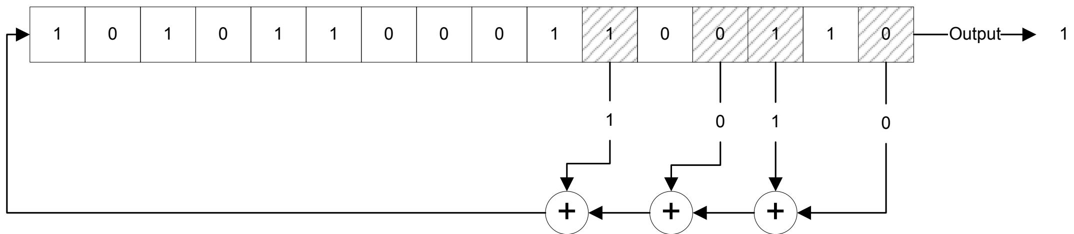


Figure 3.7: With a clock trig the found result is registered as the first bit of the register, the content of the register is shifted once towards right

Table 3.1: Some LFSR generator polynomials with varying size 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
17	$x^{17} + x^{14} + 1$	131071
18	$x^{18} + x^{11} + 1$	262143
19	$x^{19} + x^{18} + x^{17} + x^{14} + 1$	524287

suitable to obtain all $2^n - 1$ numbers then this seed is called as the maximal.

If many LFSRs are implemented with different seeds[4], then the output of each LFSR will be independent of each other. If the outputs of these LFSRs are summed up as shown in Figure 3.8 then by the Central Limit Theorem a pseudo-random noise generator whose probability distribution is close to that of the Gaussian distrubiton will be obtained.

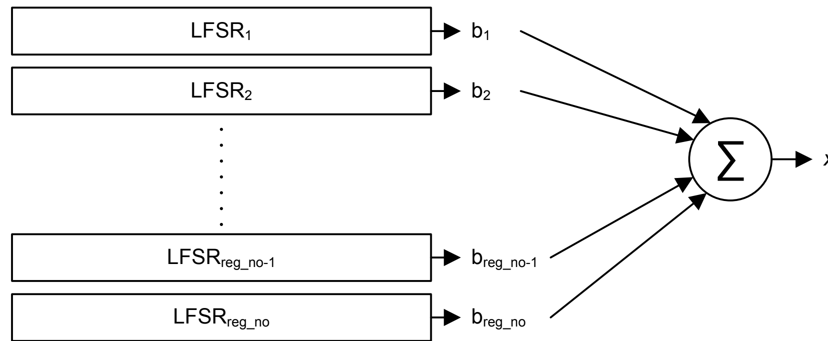


Figure 3.8: Normally distributed noise generation by LFSR

The mean and the variance of the random noise generator can be evaluated through some equations. Let x be a random variable that is obtained by the summation of the

[4] Special care should be taken when assigning seeds to the LFSRs. A bad choice of seeds may lead to incorrect results due to auto-correlation etc.

output bits of reg_no number of LFSRs. The mean of x is

$$E(x) = E\left(\sum_{reg_no} b_i\right) \tag{3.1}$$

Since summation is a linear operation and each b_i is independent of each other the expectation function can go into the summation and (3.1) yields

$$E(x) = \sum_{reg_no} E(b_i). \tag{3.2}$$

The probability of b_i to be either 0 or 1 is equal, $1/2$ so (3.2) can be calculated as

$$E(x) = \sum_{reg_no} \frac{1}{2}, \tag{3.3}$$

$$E(x) = \frac{reg_no}{2}. \tag{3.4}$$

The variance of x is found by the well known variance equation that

$$\begin{aligned} Var(x) &= E(x^2) - E^2(x), \\ &= E\left(\left(\sum_{reg_no} b_i\right)^2\right) - \frac{(reg_no)^2}{4}. \end{aligned} \tag{3.5}$$

Here comes a square of a summation and this equation must also be put into a linear form for ease of calculation. If the squared term is written in an open form it will be seen that there will be reg_no number of b_i^2's and other terms will be in the form of $b_i b_j$. The variance of x can be rewritten in the following form:

$$Var(x) = E\left(\sum_{reg_no} b_i^2 + \sum_{i,i\neq j}^{reg_no} \sum_{j}^{reg_no} b_i b_j\right) - \frac{(reg_no)^2}{4} \tag{3.6}$$

$$= \sum_{reg_no} E(b_i^2) + \sum_{i,i\neq j}^{reg_no} \sum_{j}^{reg_no} E(b_i)E(b_j) - \frac{(reg_no)^2}{4} \tag{3.7}$$

$$= \frac{reg_no}{2} + (reg_no - 1)reg_no\frac{1}{4} - \frac{(reg_no)^2}{4} \tag{3.8}$$

$$= \frac{reg_no}{2} + \frac{reg_no^2}{4} - \frac{reg_no}{4} - \frac{(reg_no)^2}{4} \tag{3.9}$$

$$= \frac{reg_no}{4}. \tag{3.10}$$

In the design of the pseudo-random gaussian noise generator 40 LFSRs ($reg_no = 40$) of 16 bit locations with different seeds are generated [5]. At each rising edge of a

[5] 16 bit LFSR corresponds to 65536 cycles in period. For long tests, this period seems to be too small for noise generation. However, if the length and the period of the generated transmitted data frames do not coincide with the period of the LFSRs, this would not significantly affect the results here due to the use of highly randomized turbo codes.

clock the usual LFSR operation is carried on. The outputs of the LFSRs are summed up and the sum is taken as a normally distributed random variable. Since the produced numbers are fairly uncorrelated from each other, summing up consecutive results of the generator results in a Gaussian distribution. Figure 3.9 shows the histogram of the outputs of such a generator obtained by MATLAB simulation. On the other hand, Figure 3.10 shows the histogram of a normal distribution obtained by MATLAB's randn function which has the same variance of that illustrated in Figure 3.9.

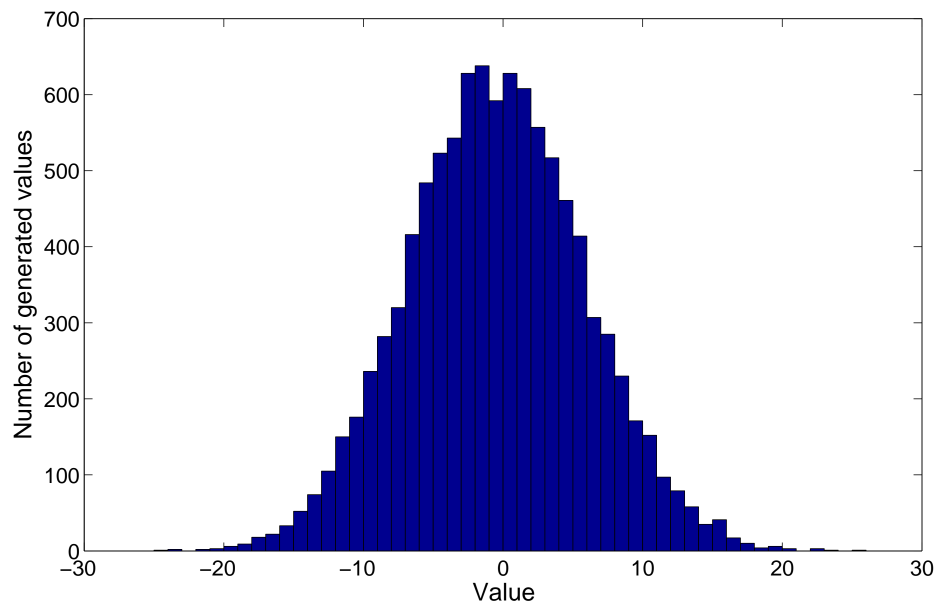


Figure 3.9: The hystogram of a pseudo-random Gaussian noise generator obtained by collection of 10000 samples

The LFSR random noise generator VHDL code is as

```
process(clk4X)
begin
 if rising_edge(clk4X) then
  if rst = '1' then
    for i in 1 to reg_no loop
     seed(i) <= file_sonucu(i*16 downto (i-1)*16+1);
    end loop;
   else
    for jj in 1 to reg_no loop
     seed(jj) <= seed(jj)(14 downto 0) & (seed(jj)(10) xor
                 seed(jj)(12) xor seed(jj)(13) xor seed(jj)(15));
    end loop;
```

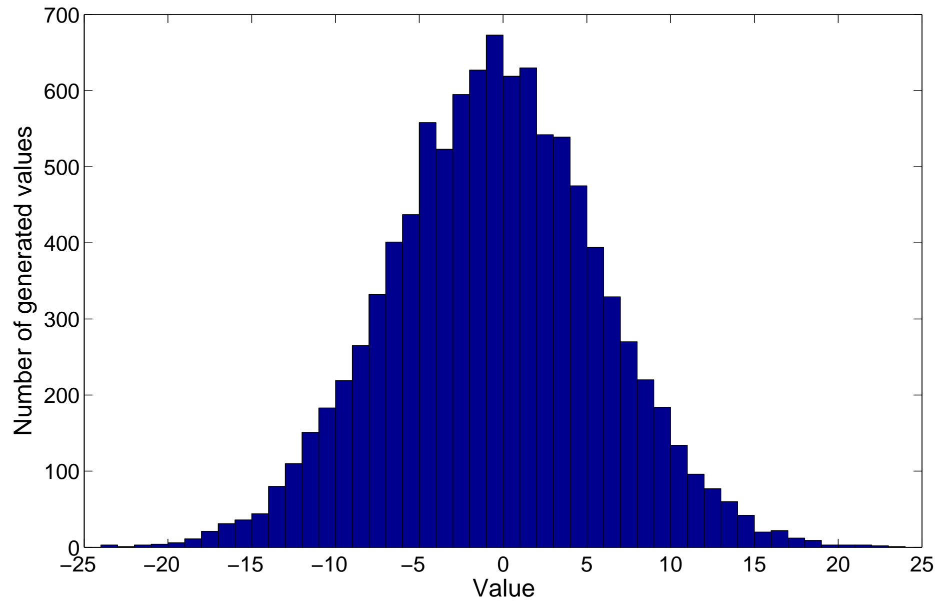


Figure 3.10: A histogram of a noise sequence generated by MATLAB's randn function

```
    end if;
  end if;
end process;
```

$file_sonucu$ is an array of strings composed of 1's and 0's previously generated by MATLAB and written into a text file. When the synthesis happens, the tool reads that file and initializes all of the LFSRs with the desired seeds. reg_no is the number of LFSRs taking place in the noise generator. It is 40 in our studies but it can be adjusted with the necessity that $file_sonucu$ must also be renewed. Since there are 40 registers the summation results a normal distribution with mean 20, hence in order to make the mean of that random variable 0 a 20 is always subtracted from the summation result. The variance of the sequence generated by this code is 10. The process operates with a clock 4 times faster than the usual operating clock (clk4x represents this notation). The reason for this is that 4 consecutive random numbers are added up to obtain a noise with a larger variance, that is, 4 times of a single sequence, so the variance of the new distribution is 40. Since the LFSR clock is 4 times faster than the usual clock, the generated random numbers are collected in a first in first out buffer (FIFO) and an adder module at the normal clock side reads 4 of them and sums up the numbers.

3.3.3 Error Counter

The error counter module counts the number of errors occured in a packet. As mentioned in Section 3.3.1 a known packet of data is encoded and transmitted. This uncoded data sequence is also known by this module. Whenever the decoder block starts to produce the bit estimates this module starts to check bit by bit whether the estimation is correct or not and keeps the number of incorrect estimations. Besides calculating the wrong bit decoding, it also counts how many packets are decoded incorrectly. These numbers are fed to the UART transmitter module for reporting to the PC.

3.3.4 UART Module

UART is the acronym for Universal Asynchronous Receiver Transmitter. In our design a full duplex UART is used in conjunction with RS232. A UART takes parallel data and transmits it bit by bit in a sequential fashion. The receiver side understands a new data coming and translates the bit by bit received sequence into a parallel form. The conversion between serial to parallel or vice versa in the transmitter and receivers is accomplished by the use of shift registers.

The protocol is called asynchronous because the transmitter does not send any clock signal to the receiver side. The transmission process starts whenever the transmitter sends a start bit. After the transmission of the start bit the data is transmitted from the least significant bit to the most significant. Optionally a parity bit for error check can also be included after the transmission of the data. A stop bit finishes the transmission of a byte. Figure 3.11 represents the alignment of these bits.

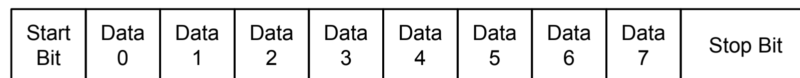


Figure 3.11: The bit alignment in a UART transmission

The transmission of data from FPGA to PC and from PC to FPGA is handled by

a protocol. In the PC to FPGA part a register map mode is used. In this mode some numbers are protected. If these special characters are sent, the next character will be the content of that register. That is, if initially byte A is transmitted and then byte B is transmitted the receiver module on the FPGA checks whether byte A is the address of a register. If it is then the content of register addressed by A is changed by B. If a value is wanted to be assigned to a register, this value cannot be the address of the registers, equivalently, B can not be protected numbers. This check is done in the MATLAB module that if B somehow enters to the forbidden zone the MATLAB code does not send this value and issues an error to the user. The register scheme is shown in Table 3.3.4.

Table 3.2: The registers and their meaning in the design of the UART transmitter.

Register Address (Decimal)	Register Name	Description
171	E_s	The amplitude value defined in Section 3.3.1
172	$Iteration$	Number of iterations that the decoders will run for. This parameter will be discussed in the next sections
173	$Paket_Ust$	High byte of a 16 bit register (*Paket* register) which determines the number of packets to be transmitted during the simulation
174	$Paket_Alt$	Low byte of the 16 bit *Paket* register
175	$NormMax$	$NormMax$ value which will be defined in the next sections

The reception of a byte starts with the coming of a start bit. Up to that time the receiver always checks the signal level at the receiver pin. When a start bit is received the clock gets synchronized with the received data bit sequence. A counter counts for the number of bits received in parallel with the baud rate and after a word is received a check of the stop bit occurs. If all of the control bits are received correctly the byte value is processed.

There is a buffer in the transmitter part. After the buffer is filled in, a command signal is activated to warn the transmitter that the buffer memory is available for transmission. A module starts to read the memory addresses one by one and puts the output of the memory in a shift register. At each baud rate period the contents of this

shift register is shifted towards the LSB and the LSB is sent through the transmit pin of RS232 transmit pin.

In the FPGA to PC part the PC expects to get the values shown in Table 3.3.4. BER [6] is the total number of errors up to the last decoded packet. PER [7] is the number of erroneus packets up to the last packet and Paket is the number of packets that are decoded upto these information are produced.

Mostly counters are used in the UART VHDL code . The sampling counters running in both the transmitter and the receiver is obtained by generic parameters. For the receiver there are two generic parameters. One of them is the clock frequency and the other is the baud rate. The counters are formed by using these variables.

Table 3.3: The registers and their meaning in the design of the UART transmitter.

Value (Decimal)	Description
BER	Total number of erroneus bits
PER	Number of erroneus packets
Paket	Number of processed packets

[6] Here, accept BER as a register name for preventing misunderstandings. Since the decoded number of packets are known dividing BER to number of packets and number of bits in a packet gives the real BER value

[7] The same definition as BER is applicable

CHAPTER 4

IMPLEMENTATION ISSUES

In this chapter, implementation of the parallelized turbo and repeat accumulate encoder/decoder on the FPGA will be discussed. The implementation steps, the parameters which may affect the performance of the designs and the algorithms that are used will be interpreted.

4.1 Channel Model

A general communication system can be depicted as in Figure 4.1. In this system an encoder encodes the uncoded information and passes the result to a modulator. The modulator modulates the signals and the transmit antenna transmits the packet. In the wireless channel the transmitted signals are distorted and the recieve antenna

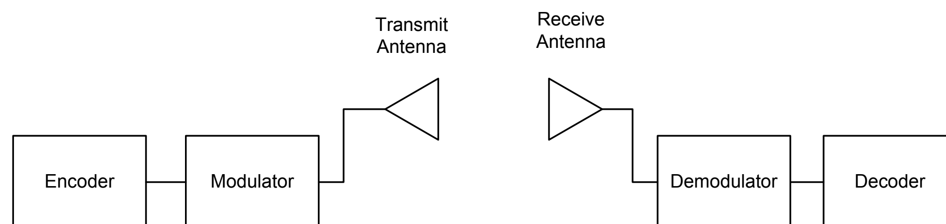


Figure 4.1: A general block diagram of a communication system

receives a distorted version of signals. In this study the channel is assumed to be an additive white Gaussian noise (AWGN) channel, the modulation is assumed to be bipolar phase shift keying (BPSK). The encoder and the decoder blocks will be either

PDTC encodor/decoder or PDRAC encoder/decoder. The generalized channel model is

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

where h_k is the channel gain, x_k is the transmitted signal, n_k is the complex AWGN term with power N_0, and y_k is the received signal. A significant amount of sections in this chapter are borrowed from a submitted paper [7].

4.2 BCJR Decoder

SISO decoders were the building blocks in the parallelized decoder structures presented in Chapter 2. These SISO decoders are *Marginal A Posteriori (MAP) Decoders* in this study. Another famous decoding technique is *Maximum Likelihood* (ML) decoding for which the Viterbi algorithm is a good example. After receiving a codeword $\mathbf{r}$, a codeword $\hat{\mathbf{v}}$ is found corresponding to a transmitted codeword in ML decoding $\mathbf{v}$. The algorithm tries to find the best approximation by minimizing the probability $P(\hat{\mathbf{v}} \neq \mathbf{v} \mid \mathbf{r})$. Hence ML is minimizes the *word error probability* (P_w). In MAP decoding, estimation for bits included in $\mathbf{r}$ is performed. For every transmitted bit u_l, a $\hat{u}_l$ is estimated. That is, the algorithm tries to minimize $P_b = P(\hat{u}_l \neq u_l \mid \mathbf{r})$, hence it is a *bit error probability* minimizing algorithm. When the data bits are a priori equally likely, the performance of ML and MAP decoders are usually very similar. However, when this probability is not the same then MAP, which is computationally more complex than ML, is observed to be superior to ML. A good example where the bit probabilities are not equally likely is iterative decoding. Since at each iteration the bit probabilities are updated by the information from the previous iteration, the probabilities of the bits do change.

The BCJR algorithm [14] is the most popular MAP decoding algorithm, which was also used by Berrou et al. in their famous study on turbo codes [4]. It aims at minimizing the bit error ratio (BER) by maximizing the marginal a posteriori probabilities. In practice, the BCJR algorithm usually calculates *a posteriori log-likelihood ratio* (*a posteriori L-value*) of an information bit. The reason of working in the log-domain will be clarified later in this section.

The log-likelihood ratio (LL) of an information bit u_l can be calculated as

$$LL(u_l) = \ln\left[\frac{p(u_l = +1|\mathbf{r})}{p(u_l = -1|\mathbf{r})}\right], \tag{4.2}$$

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

$$\widehat{u_l} = \begin{cases} +1, & LL(u_l) > 0 \\ -1, & LL(u_l) < 0 \end{cases}. \tag{4.3}$$

In the remaining part of this section the BCJR decoding algorithm steps will be explained without derivation. Detailed derivations can be found in [15].

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

$$\alpha_l = p(s_l = s', \mathbf{r}_{t<l}), \tag{4.4}$$

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,

$$\beta_l = p(\mathbf{r}_{t>l}|s_l = s). \tag{4.5}$$

As the third metric, 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

$$\gamma_l = p(s_{l+1} = s, \mathbf{r}_l|s_l = s'). \tag{4.6}$$

As a result of a few steps 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

$$\alpha_{l+1}(s) = \sum_{s' \in \sigma_l} \gamma_l(s', s)\alpha_l(s'), \tag{4.7}$$

$$\beta_l(s') = \sum_{s \in \sigma_{l+1}} \gamma_l(s', s)\beta_{l+1}(s), \tag{4.8}$$

with initial conditions,

$$\alpha_0(s) = \begin{cases} 1, & s = 0 \\ 0, & s \neq 0 \end{cases}, \tag{4.9a}$$

$$\beta_N(s) = \begin{cases} 1, & s = 0 \\ 0, & s \neq 0 \end{cases}. \tag{4.9b}$$

In (4.9b), N stands for the length of the input sequence[1]. In (4.7) and (4.8), 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 [15]

$$\gamma_l(s', s) = \mathrm{e}^{u_l L_a(u_l)/2} \mathrm{e}^{(L_c/2)(r_l \cdot v_l)}, \tag{4.10}$$

where $L_a(u_l)$ is the a priori bit probability[2], L_c is the channel reliability factor which is equal to $4E_s/N_0$, 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 hypothesized 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 (4.7), (4.8), and (4.10) in an easier way, these operations are usually realized in the logarithmic domain. The log-domain metric values are given as follows:

$$\gamma_l^*(s', s) = \ln \gamma_l(s', s) = u_l \frac{L_a(u_l)}{2} + \frac{L_c}{2}(r_l \cdot v_l), \tag{4.11}$$

$$\alpha_{l+1}^*(s) = \ln \alpha_{l+1}(s) = \ln \sum_{s' \in \sigma_l} \mathrm{e}^{\left[\gamma_l^*(s', s) + \alpha_l^*(s')\right]}, \tag{4.12}$$

$$\beta_l^*(s') = \ln \beta_l(s') = \ln \sum_{s \in \sigma_{l+1}} \mathrm{e}^{\left[\gamma_l^*(s', s) + \beta_{l+1}^*(s)\right]}. \tag{4.13}$$

It can easily be seen that both forward and backward metric calculations can be

---

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

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

simplified more by defining a *max** operation

$$max^*(x,y) = \ln(e^x + e^y) = \max(x,y) + \ln(1 + e^{-|x-y|}), \tag{4.14}$$

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

By using the multiple argument form of the *max** operation, (4.12) and (4.13) can be simplified as

$$\alpha^*_{l+1}(s) = max^*_{s' \in \sigma_l}\left[\gamma^*_l(s',s) + \alpha^*_l(s')\right], \tag{4.15}$$

$$\beta^*_l(s') = max^*_{s \in \sigma_{l+1}}\left[\gamma^*_l(s',s) + \beta^*_{l+1}(s)\right] \tag{4.16}$$

with the initial conditions,

$$\alpha^*_0(s) = \begin{cases} 0, & s = 0 \\ -\infty, & s \neq 0 \end{cases}, \tag{4.17a}$$

$$\beta^*_N(s) = \begin{cases} 0, & s = 0 \\ -\infty, & s \neq 0 \end{cases}. \tag{4.17b}$$

Figures 4.2 and 4.3 illustrate the use of *max** operation in α and β computations, respectively.

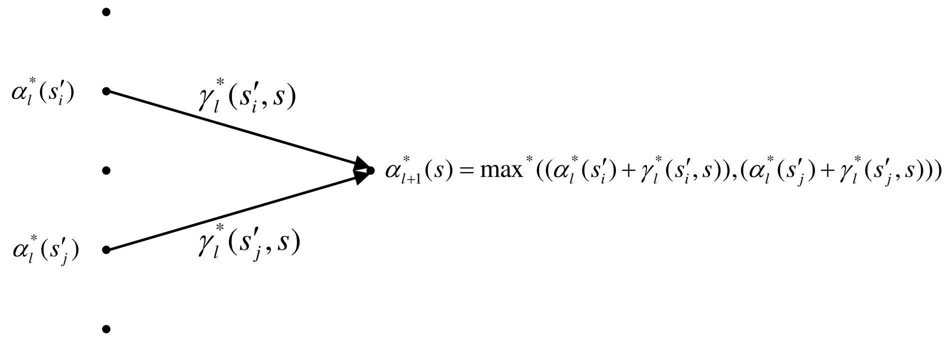


Figure 4.2: Forward recursion in calculation of $\alpha^*_{l+1}(s)$

By skipping the intermediate steps, the log-likelihood formula in (4.2) can be rewritten using the formulas described above as [15]

$$LL(u_l) = \ln\left\{\sum_{(s',s)\in\Sigma_l^+} e^{\left[\alpha^*_l(s') + \gamma^*_l(s',s) + \beta^*_{l+1}(s)\right]}\right\} - \ln\left\{\sum_{(s',s)\in\Sigma_l^-} e^{\left[\alpha^*_l(s') + \gamma^*_l(s',s) + \beta^*_{l+1}(s)\right]}\right\} \tag{4.18}$$

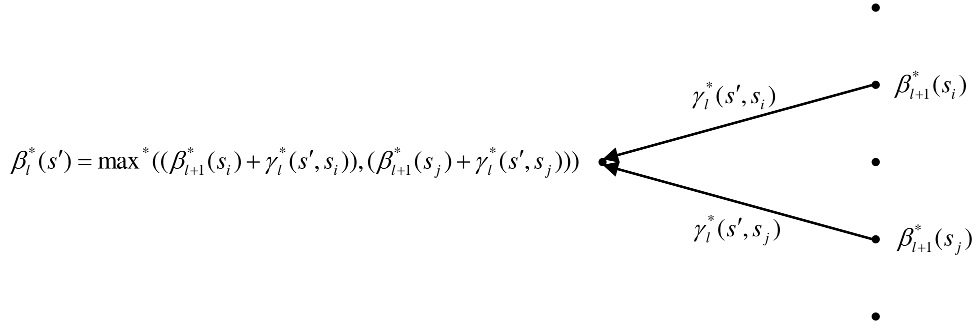


Figure 4.3: Backward recursion in calculation of $\beta_l^*(s')$

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

4.3 FPGA Implementation of BCJR Decoder

FPGA's are very flexible programmable devices. Since all of the programmable devices from microcontrollers to DSP processors are produced by the help of transistors and boolean operators, one can build everything on FPGA's from scratch. In our design there are too many arithmetic operations from usual addition to square root operation each of which must be handled carefully. We could design the decoder in such a way that all of the numbers were represented by floating point numbers and the operations could be handled in floating point form. This implementation would give us the ability of using a large interval among the real numbers. For example, with a single precision floating point we can represent all numbers between 2^{-128} to 2^{127}. But this approach will cause huge latencies in operations hence the achievable bit rate will drastically decrease and consume too many resources. For these reasons, we have to implement the algorithms on a fixed point arithmetic basis. This approach will increase the speed of the decoders tremendously however this time we will face some problems because of the fixed-point arithmetic. Those problems can be divided into two subgroups, outer problems and inner problems. Outer problems are mainly due to the resolution order of the receiver, that is, trying to fix the number of bits, K,

in fixed point representation in such a way that the information loss in the decoder will not be so much. Inner problems are due to the fixed point arithmetic used in the decoder. Those are overflowing and underflowing of summation and subtraction operations, division, and square-root operations in fixed point arithmetic. Besides these problems some additional optimizations need to be performed to increase the computation speed of the decoder.

4.3.1 Center to Top Algorithm

This algorithm is basically for optimization purpose. When the metric calculations in α^* and β^* 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 (like of turbo codes as in our case) since decoding process can begin after receiving the whole packet. By this algorithm, the decoding time can be halved. Consider a decoder running on 20 information bits. At time 0 the metric values are initialized as defined in (4.17a) and (4.17b) and shown in Figure 4.4. As shown in Figure 4.5 α^* and β^* values are calculated without computing any LL value up to time 10. At time 10, both of $\alpha^*_{10}, \beta^*_{11}$ and α^*_9, β^*_9 values are available together with the branch metrics for the time, γ^*_{10} and γ^*_9. So, $LL(u_{10})$ and $LL(u_9)$ are computed and given out as in Figure 4.6. That process, starting from the center of the frame, continues to the end and simultaneously to the beginning of the frame. That is why this algorithm is named as "center to top" [13]. 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.

4.3.2 Observation Quantization

In addition to being an optimization process, this process is basically an outer problem. Although the MAP decoder will be repeated many times and put into parallelized form, the observations will directly be fed to these decoders so the observations are

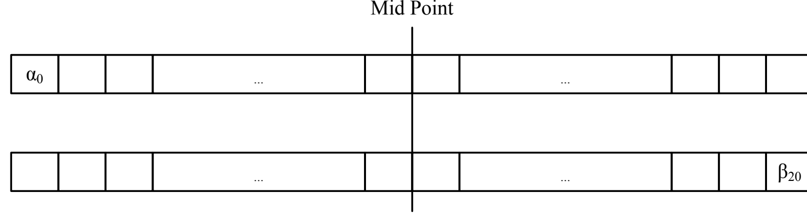


Figure 4.4: α and β values are initialized initially at time 0

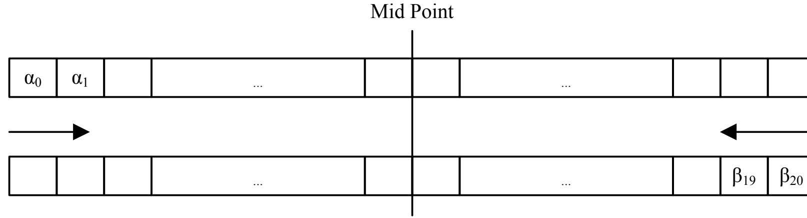


Figure 4.5: α and β values are computed independently and in a recursive manner

also important for the BCJR decoder. 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 AWGN channel for BPSK modulation can be modeled as

$$y_k = h_k x_k + n_k \tag{4.19}$$

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 symbol energy), x_k is the transmitted bit ($x_k = \pm 1$) 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

$$f(y_k|h_k, x) = \frac{1}{\pi N_0} e^{-\frac{|y_k - h_k x|^2}{N_0}} \tag{4.20}$$

The logarithmic form of (4.20) is

$$ln(f(y_k|h_k, x)) = -\ln(\pi N_0) - \frac{|y_k|^2}{N_0} - \frac{|h_k|^2 |x_k|^2}{N_0} + \frac{2}{N_0} \Re\{y_k h_k^* x_k^*\} \tag{4.21}$$

$$= C + \frac{2}{N_0} \Re\{y_k h_k^* x_k^*\} \tag{4.22}$$

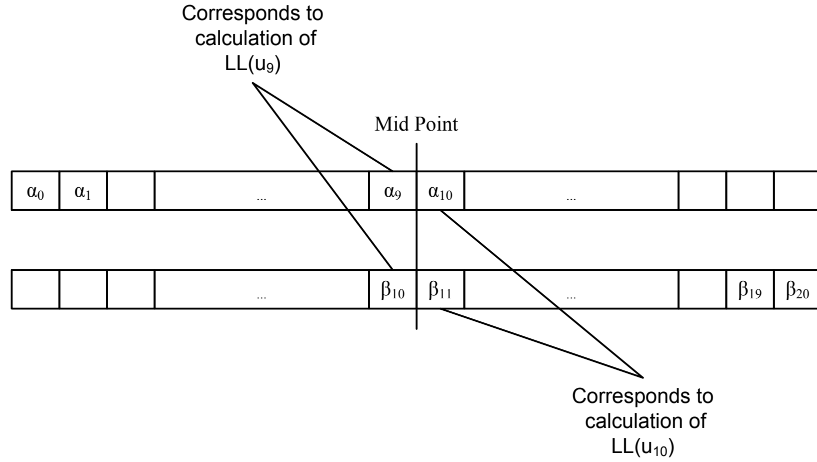


Figure 4.6: α and β values first meet at time 10 and at this time all information for computing the first LL values are ready

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

$$\ln(f(y_k|h_k, x)) \doteq \frac{2}{N_0}\Re\{y_k h_k^* x^*\}, \tag{4.23}$$

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

$$Q_k = Q(\ln(f(y_k|h_k, x = 1))) = \left\lfloor \frac{2/N_0 \Re y_k h_k^*}{q} \right\rfloor \tag{4.24}$$

If we apply the AWGN channel model given in (4.19) on (4.24) for a BPSK modulation we get,

$$Q_k = \left\lfloor \frac{2\sqrt{E_s}/N_0 \Re\{y_k\}}{q} \right\rfloor \tag{4.25}$$

$$= \left\lfloor \frac{2\sqrt{E_s}/N_0 \Re\{(\sqrt{E_s} + n_k)\}}{q} \right\rfloor \tag{4.26}$$

$$= \left\lfloor \frac{2E_s}{N_0 q} + \frac{2\sqrt{E_s}}{N_0 q} n_I \right\rfloor \tag{4.27}$$

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 many equations will blow up 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 $S = \{-NormMax, -NormMax + 1, ..., NormMax - 1, NormMax\}$ 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 v and variance σ^2, obtaining a number p such that $|p| > v + 3\sigma$ 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

$$A = \frac{2E_s}{N_0 q} + \frac{2\sqrt{E_s}}{N_0 q} n_I$$

$$E\{A\} = \frac{2E_s}{N_0 q} \tag{4.28}$$

$$\sigma_A = \frac{2\sqrt{E_s}}{N_0 q}\sigma_{n_I} = \frac{2\sqrt{E_s}}{N_0 q}\frac{\sqrt{N_0}}{\sqrt{2}} \tag{4.29}$$

$$= \sqrt{\frac{2E_s}{N_0 q}}\frac{1}{\sqrt{q}} \tag{4.30}$$

$$= \frac{\sqrt{E\{A\}}}{\sqrt{q}} \tag{4.31}$$

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

$$NormMax = E\{A\} + 3\sigma_A \tag{4.32}$$

$$= E\{A\} + 3\frac{\sqrt{E\{A\}}}{\sqrt{q}} \tag{4.33}$$

By replacing (4.28) in (4.33), we get

$$NormMax = \frac{2E_s}{N_0 q} + 3\sqrt{\frac{2E_s}{N_0 q}}\frac{1}{\sqrt{q}} \tag{4.34}$$

By solving this equation, q can be calculated as

$$q = \frac{\frac{2E_s}{N_0} + 3\sqrt{\frac{2E_s}{N_0}}}{NormMax} \tag{4.35}$$

As it is obvious in (4.35), 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. In our design, we have used a relatively large LUT that stores the q values in 8 bits, 3 for integer part and 5 for the decimal part. That gives a precision of $1/2^5$ and yields a satisfactory performance.

4.3.3 Addition and Subtraction Operations

Addition (by the term "Addition" also "Subtraction" is also assumed) is the first problematic operation in decoder structure, because at almost every step of the algorithm there exists an addition operation. Since the bits are represented by limited number of bits, K, an overflow can easily be observed if the addition of two numbers passes 2^{K-1} (Since the observations are represented by K bits in two's complement maximum positive number can be 2^{K-1}). For this kind of erroneus situation a new addition and subtraction operations must be defined. The new defined addition operatin is named as, $clipsum$ and the new subtraction operation is named as $clipsubtract$.

$clipsum$ function,represented by $\oplus$ can be defined as

$$a \oplus b = \begin{cases} MaxInf & \text{if } a \geq MaxInf, \\ MinInf & \text{else if } a \leq MinInf, \\ MaxInf & \text{else if } b \geq MinInf, \\ MinInf & \text{else if } b \leq MinInf, \\ MaxInf & \text{else if } a+b \geq MaxInf, \\ MinInf & \text{else if } a+b \leq MinInf, \\ a+b & \text{else} \end{cases} \tag{4.36}$$

where $MaxInf=2^{K-1}$ and $MinInf=-2^{K-1}$. The new function basically adds a clipping capability to usual summation. Similarly the *clipsubtract* function, represented by $\ominus$ can be defined as

$$a \ominus b = \begin{cases} MaxInf & \text{if } a \geq MaxInf, \\ MinInf & \text{else if } a \leq MinInf, \\ MinInf & \text{else if } b \geq MinInf, \\ MaxInf & \text{else if } b \leq MinInf, \\ MaxInf & \text{else if } a - b \geq MaxInf, \\ MinInf & \text{else if } a - b \leq MinInf, \\ a - b & \text{else} \end{cases} \tag{4.37}$$

Drawback of these operations is that if K is chosen to be too small then the algorithms can get into saturation, to either $MaxInf$ or $MinInf$ values, so value of K is of significant importance. The clipsum function is realised in the FPGA as follows:

```
procedure sum(A,B : in std_logic_vector;C : out std_logic_vector)is
variable summ : std_logic_vector;
begin
  summ := (A(A'high)&A)+(B(B'high)&B);
  if A >= max_inf  then
    C:= max_inf;
  elsif  A <= min_inf then
    C:= min_inf;
  elsif B >= max_inf then
    C:= max_inf;
  elsif B <= min_inf then
    C:= min_inf;
  elsif summ >= max_inf then
    C:= max_inf;
  elsif summ <= min_inf then
    C:= min_inf;
  else
    C:= summ;
  end if;
end sum;
```

The usage of this procedure is as

```
sum(a,b,a_variable);
c <= a_variable;
```

where the operation defines $c = a \oplus b$. It must be noted that since the procedure returns to a variable, this function is not a synchronous operation. We base on aim in using the function in a combinatorial logic. Since there are recursive operations taking place in the algorithm, in order to use a result at the next clock cycle it must be ready before the operation clock. In our case this defines a combinatorial logic.

4.3.4 Node (α, β) Metric Normalization

In (4.16) and (4.15) 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 ($> MaxInf$) or underflow ($< MinInf$). To solve this problem, α^* and β^* values are normalized at each trellis step. After each forward recursion, the 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. Another approach to node metric normalization can be found in [21].

The algortihm can be written as follows. Let $A_l(s)$ be all of the calculated α^* values at time l then we define a new variable α^*_{max} such that

$$\alpha^*_{max} = max_s A_l(s) \tag{4.38}$$

Then the new defined α^* values, α'^*, are

$$\alpha'^* = \alpha^* \ominus \alpha^*_{max} \tag{4.39}$$

The same approach can be applied to β^* metrics. The VHDL code for α^* normalization is divided into two sections. Firstly α^*_{max} is found in a process description, the code is for a 2-state trellis,

```
process(alfa)
begin
if alfa(1) > alfa(2) then
    alfa_max <= alfa(1);
else
    alfa_max <= alfa(2);
end if;
end process;
```

where "alfa" is an array of std_logic_vector. It must be noted that this process is not a synchronous logic operation. The reason to do this in combinatorial is that the normalized form of α^*'s must be ready at the next clock cycle for the recursive operation. New alfa values are calculated as

```
process(...)
...
subtract(alfa(1),alfa_max,alfa_new_var(1));
alfa_new(1) <= alfa_new_var(1);
...
```

It is obvious that the logic is fully combinatorial.

4.3.5 max^* Approximation

The correction term in max^* operation poses a trouble when it is needed to be expressed in fixed-point arithmetic. It is not possible to easily realize the ln function fully in such a system. For that reason, some approximations must be made to implement the max^* operation. There are basically two approximations in the literature. These two different approaches result in log-MAP with tables and max-log-MAP.

If the decoder is a log-MAP decoder then max^* calculation is a more difficult subject, because the correction term, $\ln(1 + e^{(-|x-y|)})$, should be calculated. Since the hardware implementation of such a function is complicated, this term is handled by construction of a LUT in practice. As described in the previous part, the observations are in quantized form, therefore LUT values also have to be quantized accordingly. That is, if the inputs to the max^* function are in a quantized fashion, the other terms generated in the function also should be quantized in parallel with the inputs. The LUT construction function is,

$$LUT(i) = \left\lfloor \frac{\ln(1 + e^{-iq})}{q} \right\rfloor \tag{4.40}$$

where i is the absolute value of the difference of the inputs of the max^* function. The LUT sizes are usually quite small (around 5-6 entries) with reasonable $NormMax$ values.

In a max-log-MAP decoder, the correction term is neglected, that is max^* operation is the same with ordinary max operation. So, the quantization term, q, is useless for

this method. In other words, it can be said that decoder does not need an exact SNR estimation to operate properly. Studies in [20] and [17] have shown that max-log-MAP decoders work without any need on SNR estimation.

4.4 Memory Collision Free Interleavers

The interleaver is the most crucial part in turbo and turbo-like code structures. When parallelization is in effect the interleaver is evenmore important since each sub-encoder and sub-decoder must operate coherently. The importance comes from its structure which must be collision free. For an exemplification of the importance of a collision free interleaver let us assume the operation of a parallelized encoder structure for a turbo code in this section. The encoder under observation consists of $N = 4$ sub-encoders each of which encodes n number of uncoded data bits.

The uncoded data bits are saved in RAMs. An FPGA RAM is not reachable through more than two ports at a time. For that reason, the uncoded data must be stored in N number of distinct RAMs in parallelized encoder structure. The interleaver must be mapping these distinct data RAMs to sub-encoders accordingly. An uncarefully designed interleaver shows how a collision happens in Figure 4.7. In the figure, everything seems fine. All the sub-encoders are accessing to different bit locations hence all of the bits and encoders are one-to-one matched. However, the problem here is that $encoder_1$ tries to reach a bit which is located at 1 at a time instant, where 1 is accessible by $encoder_1$ among locations $\{1, 5, 9, 13\}$. At the same time instant, $encoder_2$ tries to reach a bit at location 2, $encoder_3$ to bit location 3 and $encoder_4$ to bit location 4. When these locations are tried to be mapped with that interleaver design, it is seen that all of these 4 locations are mapped to RAM1. Since RAM1 is not able to serve to 4 different requests, a collision will happen and the sub-encoders will not function properly. Besides functioning, the implementation of such a system is impossible without adding more latency.

For preventing such a collision, a new interleaver scheme has to be defined. The interleaver structure used in this study is the *row-coloumn S-random* (RCS-random) interleaver which is a subclass of *S-random* interleavers [10]. *RCS-random* inter-

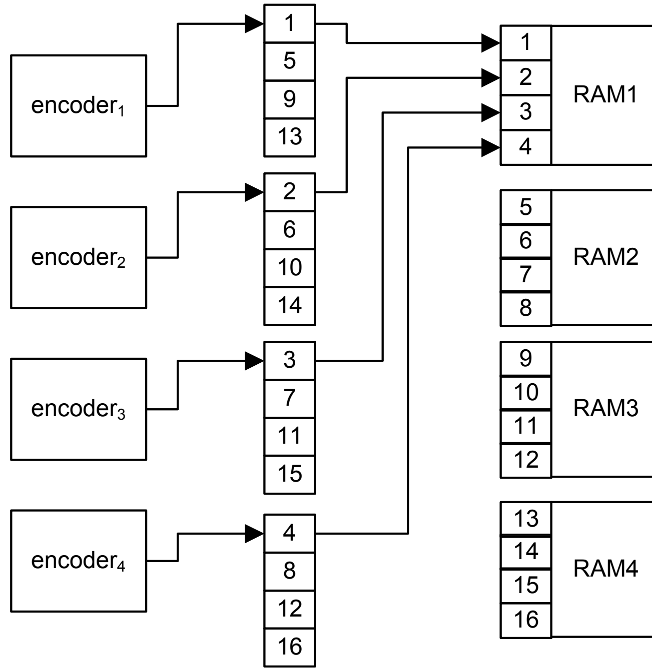


Figure 4.7: An illustration of how a memory collision may happen in an encoding process

leaver can be thought as the combination of too many small *S-random* interleavers designed in a fashion to prevent memory collisions. In this study it means 4 distinct *S-random* interleavers each of which is of size n. This interleaver is prepared by such an algorithm that the data location numbers are aligned into an n -by- N matrix. Firstly, the data in each RAM are permuted, which means the interleaving of the rows of the matrix. Next, the "RAMs" are permuted in an *S-random* fashion for all RAM addresses which means the independent interleaving of columns. The operations are depicted in Figure 4.8. Such an interleaving will obviously create a new interleaver table free of memory collisions. The proof can be done in such a way that, if all of the N encoders have independent RAMs, each sub-encoder will reach to only one RAM, that is, $encoder_1$ only reaches to RAM1, $encoder_2$ only reaches to RAM2 and so on, there will be no memory collision. That corresponds to address permutation, which is row interleaving. If the addresses are distributed in a collision free fashion, distributing the RAMs will not cause any problems. This is true because if the RAMs are distributed, locations will interchange between RAMs, e.g. if $encoder_1$ is reaching to the location x at RAM1 now it will reach to the location x at RAM2 but this

time $encoder_2$ will not reach to location y in RAM2 but it will reach to location y at RAM1 and so on.

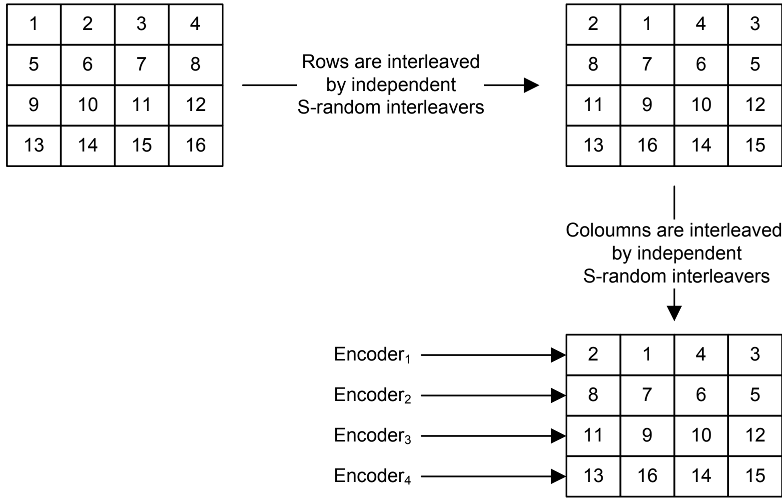


Figure 4.8: RCS-random interleaver is a good approach for memory collision freee interleaver design including the good properties of S-random interleavers [9].

In the construction of a real n by N interleaver MATLAB is used. First N (in the designs N is chosen to be 4) distinct S-random interleavers of size n are formed. In the second step n distinct *S-random* interleavers of size N are formed. Then these numbers are converted to binary form for storing them in the FPGA RAMs. This conversion is done in a comprehensible way such that two *S-random* interleaver numbers are combined giving a single number. The first bits, MSB portion, of this new constructed number gives the RAM numbers. It must be noted that the number of these bits are $\lceil \log_2 N \rceil$. Remaining bits, LSB portion, gives the address of the selected RAM to be read/written. Also this LSB portion is in the order of $\lceil \log_2 n \rceil$. Since interleaver table numbers are only used for reading after once they are stored, these RAMs are nominated as read-only memories (ROM) in FPGA. The operation taking place in FPGA is shown in Figure 4.9.

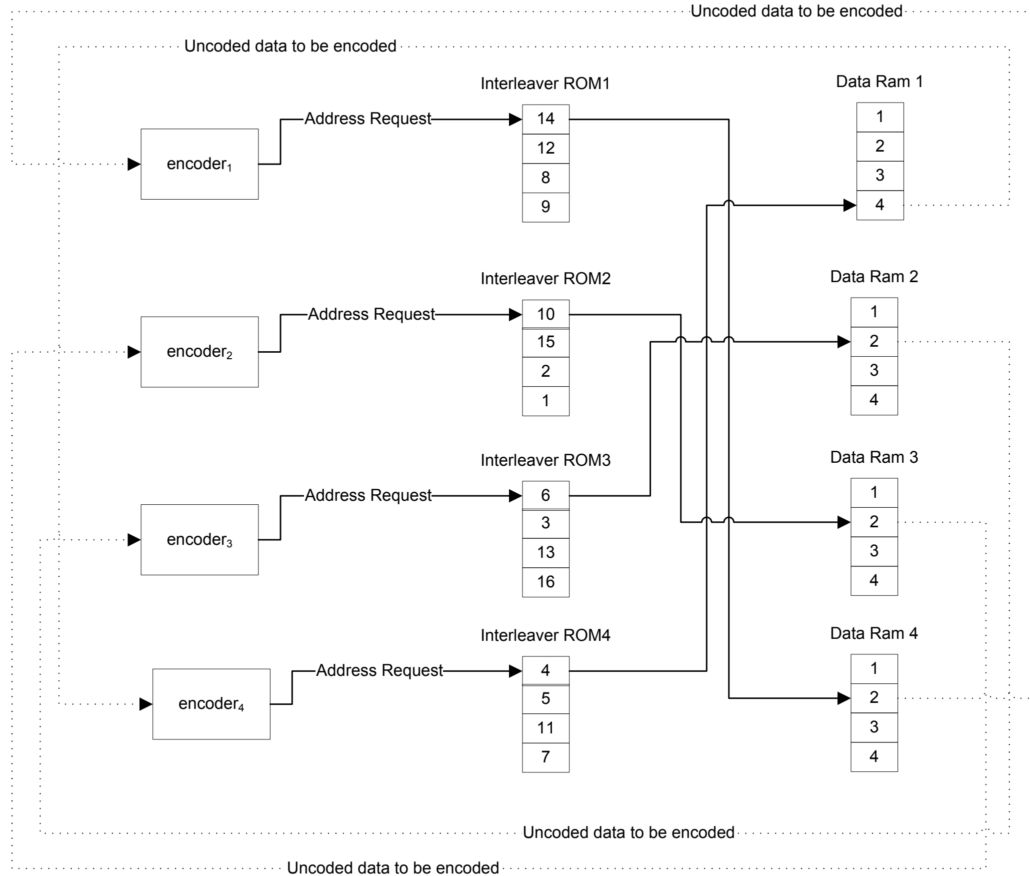


Figure 4.9: The interleaver operation taking place in the FPGA. Each address request is decoded, the requested RAM and the corresponding location is found and the requested data is forwarded to the demanding encoder.

4.5 Encoder/Decoder Design of Parallel Decodable Turbo Codes

4.5.1 Encoder Design

The parallelization of the encoders was introduced in Chapter 2. The design can be summarized as follows. Uncoded data sequence is fed into 4 parallelized convolutional encoders. This part forms the systematic data and the parity bits of the systematic part. At the same time by the use of interleaver a small controller interleaves the uncoded data. The interleaved data part are also fed into 4 parallel encoders so that the second parity terms are obtained. Since termination is on the scene, the last two bits of the interleaved data is also included in the sequence that is to be transmitted.

The sub-encoders are obtained by the help of a finite state machine. Each one of

the sub-encoders starts with an initial state chosen to be the all-zero state and the state of the encoder changes with each arrival of an uncoded bit. The coded bits are stored in a buffer. When n bits are encoded, the sub-encoder adds termination bits to the coded sequence and warns the outer world that the encoding process is done. A small controller, which is responsible for the initialization of the encoders and for the interlaver control, senses that the sub-encoders filled the buffers. When the buffers are full, the encoder transmits the coded data sequence to the next block such as a modulator.

If the number of parallel encoders is N and the interleaver block size is B then total encoding latency for this encoder scheme is

$$\tau_{enc} = \frac{B}{N} + 3 + 2 \tag{4.41}$$

+3 comes from the fact that the interleavers reach to the memories with 3 clock cycle constant latency. +2 comes from the termination bits.

The ISE synthesis report for the defined parallel encoder is given in Table 4.1. The table shows that encoder does not consume much of the logical blocks of the device. However, by the use of the interleavers and additional memory for storing the results of the parallel encoders more BRAMs are occupied.

Table 4.1: Xilinx ISE synthesis report for parallel turbo encoder

Unit Name	**Usage count and percantage**
Slices	670 (4 %)
BRAM	7 (3 %)
Max. clock frequency	178.396 MHz

4.5.2 Decoder Design

Decoder design is more challenging compared to encoder design. The decoder is complex because there are memory read/write operations, summations, subtractions, assertions and all of them are taking place at the same time. The PDTC decoder block diagram was given in Figure 2.7 of Chapter 2. The operation of the PDTC decoder can be summarized as follows. First, by the reception of a sequence an

FSM starts operating, which can be considered as a small controller in the decoder. In the initial states all of the received data are written into dedicated memories for proper operation of decoding. After the received data are written into the memories an assertion for the MAP decoders (sub-decoders) happens telling to the sub-decoders that a decoding process will start. Since all of the parallel decoders are in the idle state, they get ready for the reception of bits for decoding by the assertion of that start signal. All of the MAP decoders are fed by the controller in accordance with the CT algorithm. When the sub-decoders decode $n/2$ number of bits they assert to the controller that first decoded bits are being produced. The controller starts to write these decoded bit probabilities to dedicated memories. When all of the bits are decoded the next step starts. In this step the interleaver comes to the scene. This time the interleaved versions of the received bits are fed to the MAP decoders with the information from the previous step.When this step finishes, an iteration is said to be over. The performance of the PDTC decoder will be discussed in the next chapter.

One important parameter in the operation of the decoders is finding the a priori probabilities, (L_a), for the next operation cycle in each iteration. A priori probabilities in the PDTC decoder is the interleved or deinterleaved version of the extrinsic probabilities, L_e. L_e values are calculated in the MAP-decoders and its general formula is [15]

$$L_e = LL - r - L_a \tag{4.42}$$

where LL is the log-likelihood of the decoded bits, r is the channel observation probability in the log domain which corresponds to interleaved/deinterleaved systematic data, and L_a is the a priori log-likelihoods that is processed in the sub-decoders. This subtraction operation is for preventing a feed-forward mechanism that may destabilize decoding. If a feed-forward occurs in the decoder, the operation will not be reliable. For example, for an iteration of 20, r will accumulate 20 times and it will be dominant in LL, which will cause decreasing the performance of the decoder dramatically.

4.6 Encoder/Decoder Design of Parallel Decodable Repeat Accumulate Codes

4.6.1 Encoder Design

The encoder of the PDRAC is composed of two parts. The first part has the repetition encoders and the second part has the accumulators. In implementing the repetition part, each bit is suspended for an amount of time by the help of a small counter. If the repetition is to be done 3 times, that is $Z = 3$, then a counter counting up to 3 controls the repetition of the bits. The accumulator part is almost the same as that of the turbo encoder. As explained before, an accumulator is a simple delay and add operator in modulo-2. However, in order to control the termination bits of the encoders, an FSM is extracted and used in the accumulator, that is, accumulator is acting as a convolutional encoder. As the $Z \times n$ of bits are encoded then the accumulator adds a termination bit to the sequence and forwards to the transmitter. The implementation results of the PDRAC encoder is given in Table 4.2

Table 4.2: Xilinx ISE synthesis report for parallel turbo encoder

Unit Name	**Usage count and percantage**
Slices	550 (3 %)
BRAM	7 (3 %)
Max. clock frequency	192.433 MHz

4.6.2 Decoder Design

The decoder design of the PDRAC is similar to the decoder of the PDTC. The differences between them is firstly the MAP decoders in the first cluster are BCJR decoders of a two-state convolutional code. Secondly, in the second cluster a repetition decoder is introduced in place of the MAP decoders. The repetition decoders work in a fashion that they sum up Z-consecutive log-likelihoods for the final decision of the bits. The lack of the second MAP decoder cluster and introduction of the repetition decoders drastically decreases the logic consumption of this decoder.

Calculation of the a priori likelihood, L_a, is an important concern in PDRAC de-

coder and it is derived from L_e, the extrinsic information. Calculation of L_e is slightly different than that done in PDTC decoder and it is done in the MAP-decoders. The equation for calculating L_e in the MAP-decoders is

$$L_e = LL - L_a \tag{4.43}$$

where LL is the log-likelihood and L_a is the apriori-likelihood. There is not an r in this equation since the repeat-accumulate code we used is a non-systematic repeat accumulate code. The L_e Calculated by this equation is the extrinsic log-likelihood information of each "repeated bit". If L_e is deinterleaved, obtaining De_L_e, it yields a new sequence that, every consecutive Z number of values give the extrinsic likelihoods for the repeated bits. The summation of these likelihoods in a cross manner yields the deinterleaved a priori likelihoods. Interleaving deinterleaved a priori likelihoods yields a priori likelihoods, L_a, for the next iteration. Let us visualize the "cross summation" term with an example. Let Z be 3 and deinterleaved version of extrinsic likelihoods for the first bit, u_1, be L_{e1}^1, L_{e2}^1, L_{e3}^1. Now, the deinterleaved a priori likelihood for the first repetition of u_1 will be $L_{e2}^1 + L_{e3}^1$, that is the summation of the second and the third values of the extrinsic likelihood. Similarly, deinterleaved a priori likelihood for the second repetition of u_1 will be $L_{e1}^1 + L_{e3}^1$, that is the summation of the first and the third values of the extrinsic likelihood and so on. These calculation are done in the FPGA by the *a priori-finder* module. The performance of the PDRAC decoder will be conducted in the next chapter.

CHAPTER 5

RESULTS FOR THE PERFORMANCE OF PARALLEL DECODERS

This chapter discusses the FPGA results of the PDTC and the PDRAC decoders implemented on the Xilinx Virtex4-SX35 FPGA. The floating point simulation results will be given in addition to fixed point simulation results with varying block lengths. Unless a different situation is indicated, the number of parallel branches will be 4 for both PDTC decoder and PDRAC decoders. The number of repetitions for the PDRAC operation is chosen to be 3.

5.1 Implementation Results

The PDTC and PDRAC decoders are implemented for varying number of bit representation of the observations. The PDTC decoder is synthesized for two kinds. One is log-MAP based PDTC decoder and the other is the max-log-MAP based PDTC decoder. PDRAC decoder is synthesized by only using log-MAP decoder. The reason for this is not to face performance loss in PDRAC decoder.

In Table 5.1 the synthesis results for log-MAP based parallelized turbo decoder, in Table 5.2 the synthesis results for max-log-MAP based turbo decoder are given. Additionally, in Table 5.3 the synthesis result for PDRAC is given.

The distinct difference between log-MAP and max-log-MAP algorithms is that the former uses a LUT. As described in Chapter 4, LUT is generated by using the q value of the quantization. LUT insertion in decoders introduce extra resource usage. These

Table 5.1: Implementation results for PDTC decoder using log-MAP decoders

K (number of bits)	Slice usage	Slice usage Percentage (%)	Maximum achievable Clock Frequency (MHz)
5	8663	56	65.557
6	10595	68	60.070
7	10807	70	55.491

Table 5.2: Implementation results for PDTC decoder using max-log-MAP decoders

K (number of bits)	Slice usage	Slice usage Percentage (%)	Maximum achievable Clock Frequency (MHz)
4	6347	41	87.253
5	6501	42	86.963
6	6994	45	86.949
7	7537	49	85.704

extra resources, i.e. slices, are included in the results given in Tables 5.1 and 5.3.

It is obvious that LUT insertion degrades the design performance in terms of both resource usage and maximum clock speed. The reason of that can be explained as follows. LUT can be thought as a large multiplexer which is controlled by the q value and the inputs of the max^* operation. Additionally the results of the LUT have to be added in the max^* operation in order that max^* result can be ready at the next clock cycle, that is, a combinatorially operating large multiplexer degrades the resource usage and combinatorial addition degrades the maximum operating frequency. Also, it must be noted that the slice usage increases almost linearly with icreasing number of parallel sub-decoders, N.

Another important tool that can be used in FPGA implementations is pipelining. As described in Chapter 3 pipelining provides convenience for increasing the clock speed of the design. A recent log-MAP and max-log-MAP PDTC decoder conducted in [2] shows how pipelining can be an enhancement for performance. In this design all of the computations were carried right after an information was fetched. However in our design independent blocks were divided into groups by the use of flip-flops. The sysnthesis results for the aforesaid PDTC implementation are given in Tables 5.4 and 5.5.

Table 5.3: Implementation results for PDRAC decoder using log-MAP decoders

K (number of bits)	Slice usage	Slice usage Percentage (%)	Maximum achievable Clock Frequency (MHz)
5	5109	32	68.755
6	6046	39	68.180
7	6219	40	64.218

Table 5.4: Another approach to PDTC decoder implementation : Implementation results for PDTC decoder using log-MAP decoders [2]

K (number of bits)	Slice usage	Slice usage Percentage (%)	Maximum achievable Clock Frequency (MHz)
5	8179	53	36.288
6	10628	69	31.522
7	11309	73	31.352

5.2 Simulation Results

5.2.1 Bit Size (K) Selection

The bit size representation, K, for the observations is an important issue. The decoders can operate on a broader number of set as K gets larger. The available number set for decoder is $\{-2^{K-1}+1, -2^{K-1}+2, \ldots, -2^{K-1}-2, -2^{K-1}-1\}$. The BER performance with respect to K for the log-MAP based PDTC is demonstrated in Figure 5.1 and max-log-MAP based PDTC decoder in Figure 5.2 in addition to floating-point simulation which is done on MATLAB. The BER results for PDRAC decoder is presented in Figure 5.3. In these figures the performance of the PDTC decoders were examined for 4 iterations over 2000 frames where each frame consists of 160 information bits. However, the PDRAC decoder performance was observed for 8 it-

Table 5.5: Another approach to PDTC decoder implementation : Implementation results for PDTC decoder using max-log-MAP decoders [2]

K (number of bits)	Slice usage	Slice usage Percentage (%)	Maximum achievable Clock Frequency (MHz)
5	6104	39	49.873
6	6570	42	47.645
7	7174	46	43.841

erations with the same frame properties of PDTC decoders. From the figures it is understood that as number K increases, the fixed-point performance of the decoders approaches that of the floating-point. The FPGA results are obtained by choosing the most suitable $NormMax$ value, which will be conducted in the upcoming section.

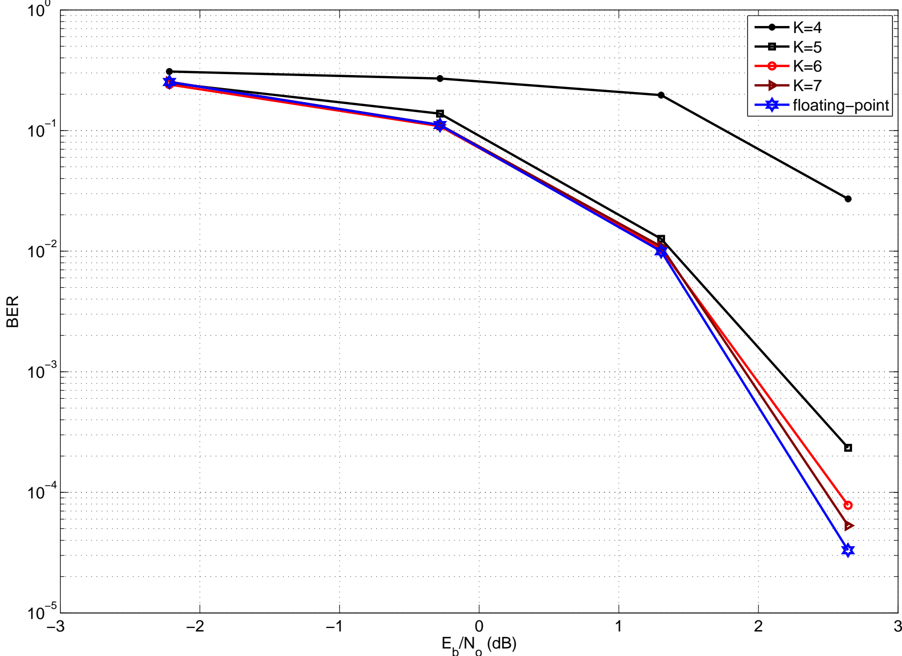


Figure 5.1: SNR vs BER for log-MAP based turbo decoder. 4 iterations for 2000 frames of 160 bits through 4 parallel MAP decoders.

5.2.2 *NormMax* Selection

Selection of $NormMax$ is another important figure in PDTC and PDRAC decoder performance. If $NormMax$ is chosen to be small then most of the information in the observations will be cropped and the decoder will not function satisfactorily. On the other hand, a large $NormMax$ will cause the decoder perform around the saturation values. Hence the selection of $NormMax$ must be done carefully. Figure 5.4 and Figure 5.5 shows the effect of NormMax on the performance of PDTC and Figure 5.6 show the effect of $NormMax$ on PDRAC decoder.

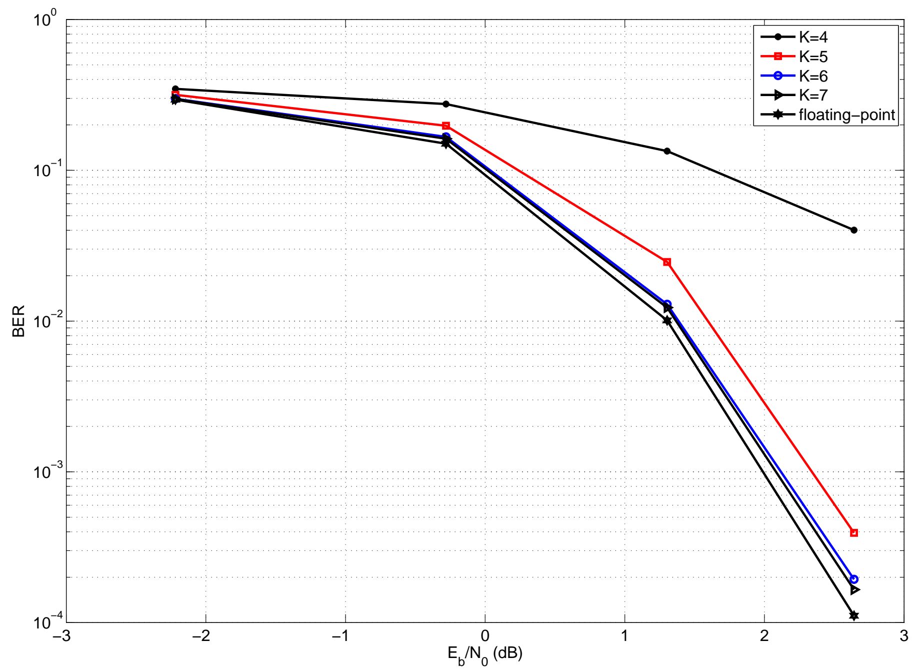


Figure 5.2: SNR vs BER for max-log-MAP based turbo decoder. 4 iterations for 2000 frames of 160 bits through 4 parallel MAP decoders.

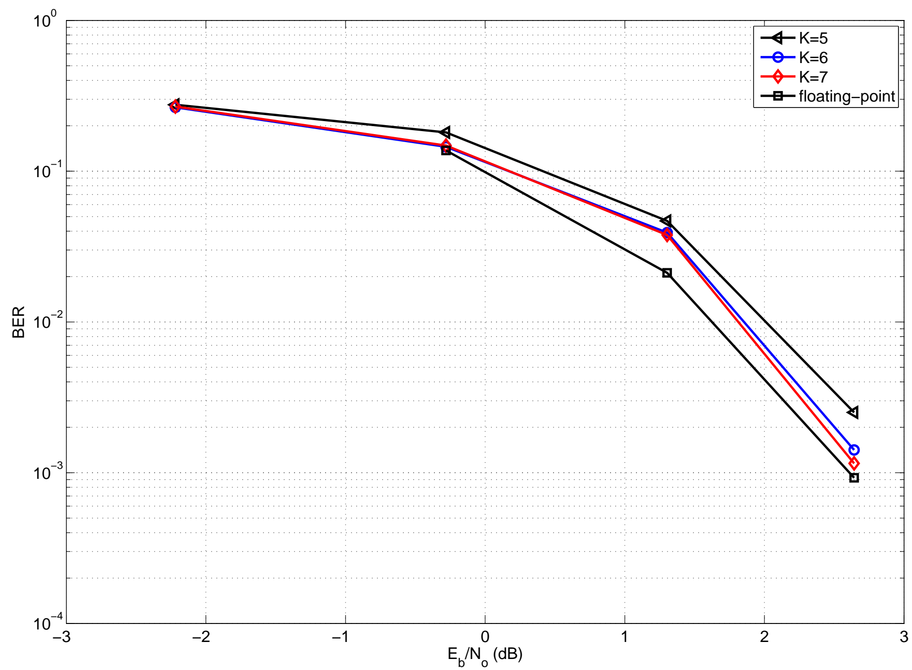


Figure 5.3: SNR vs BER for PDRAC decoder. 8 iterations for 2000 frames of 160 bits through 4 parallel MAP decoders.

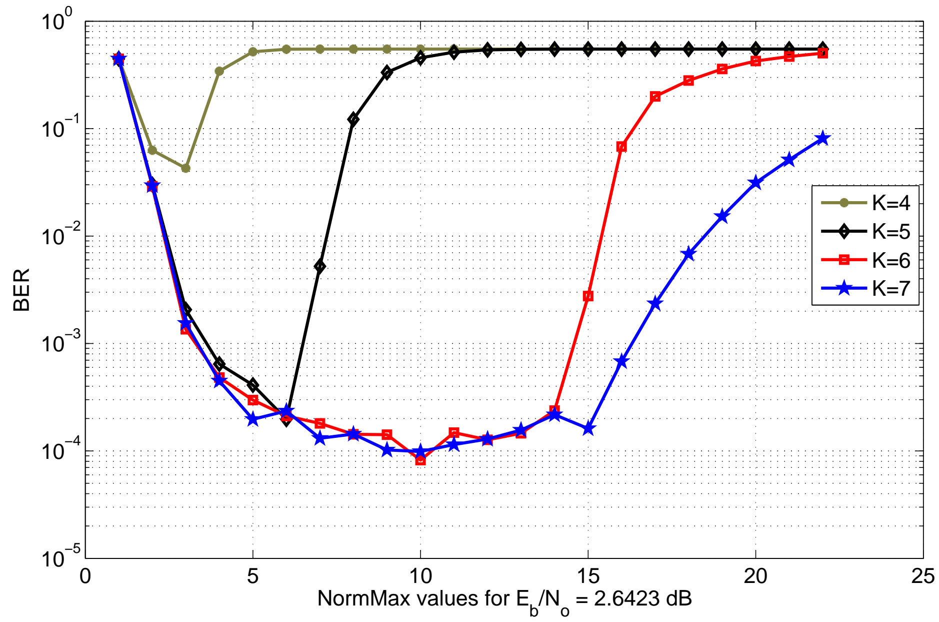


Figure 5.4: *NormMax* values for log-MAP turbo code decoder for different bit representations. The average of 6000 packets of 160 data bits with 4 parallel decoders.

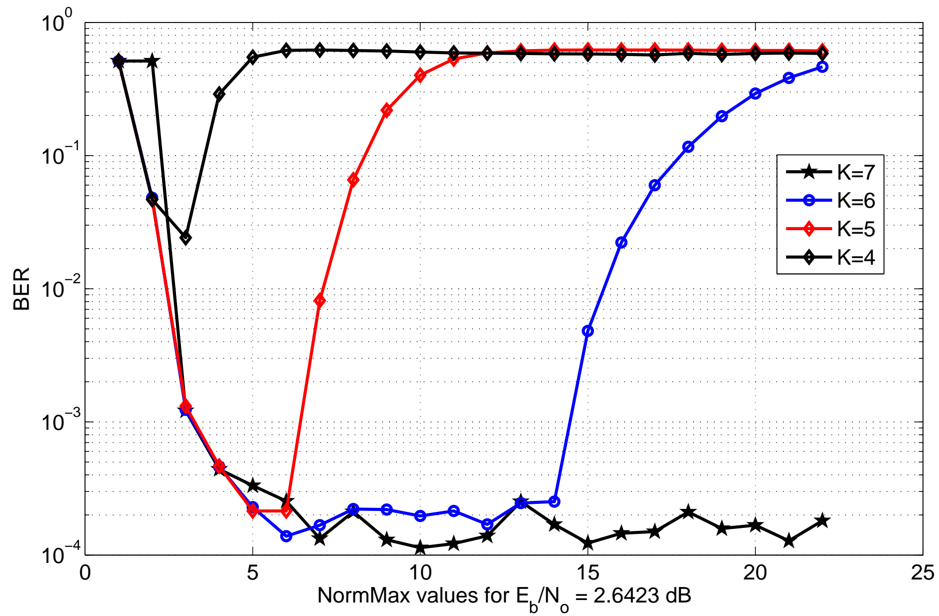


Figure 5.5: *NormMax* values for max-log-MAP turbo code decoder for different bit representations. The average of 6000 packets of 160 data bits with 4 parallel decoders.

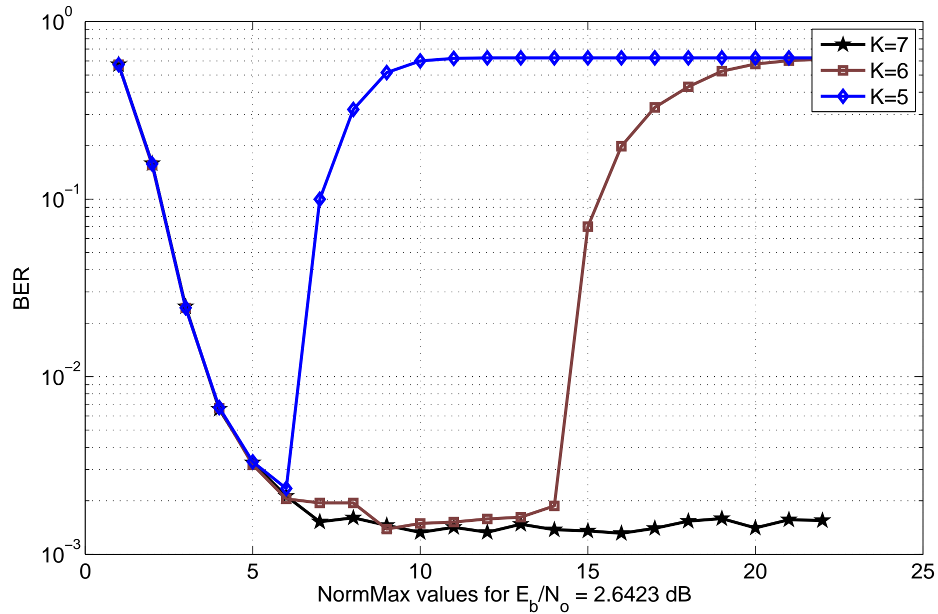


Figure 5.6: *NormMax* values for repeat accumulate codes for different bit representations. The average of 6000 packets of 160 data bits with 4 parallel decoders.

5.2.3 Interleaver Size

As the interleaver size increases the performance of the decoders gets better. This phenomena is called the *Interleaver Gain* in literature. In Figures 5.7, 5.8, and 5.9 the BER performances of the PDRAC and PDTC decoders are depicted. In Table 5.6 these performance results are given for comparison purposes. In these figures and table it must be noted that the SNR curves are approximated with linear interpolation.

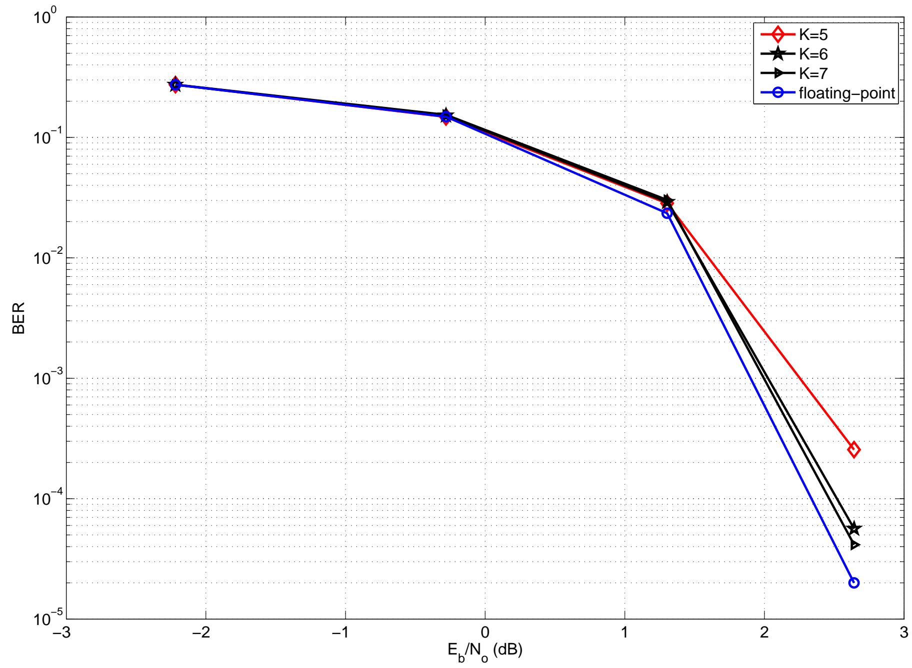


Figure 5.7: SNR vs BER for RA with 4 parallel sub-decoders decoding 1344 bits in total with 8 iterations.

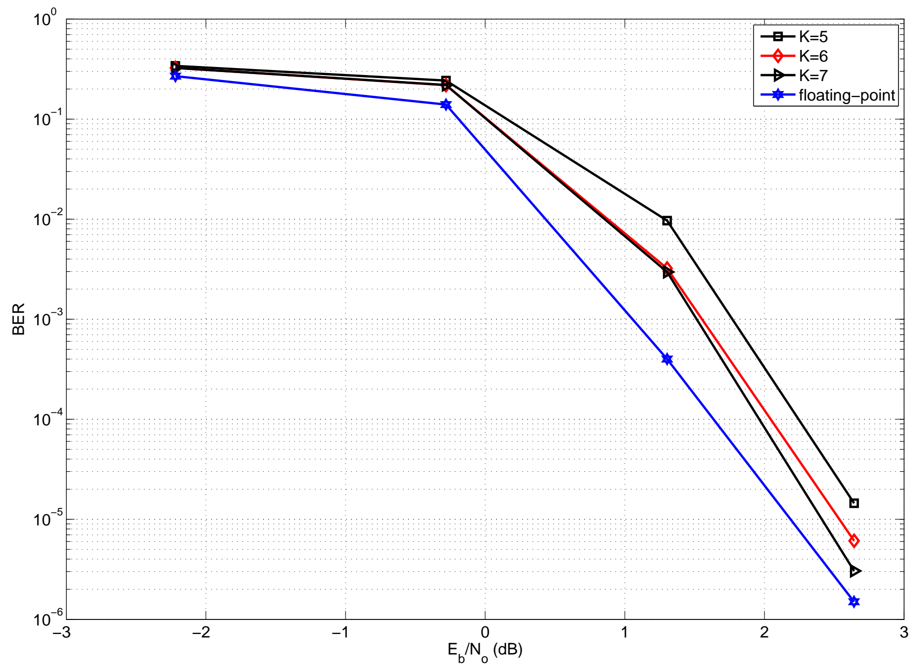


Figure 5.8: SNR vs BER for turbo decoder with 4 parallel max-log-MAP decoders decoding 1344 bits in total with 4 iterations.

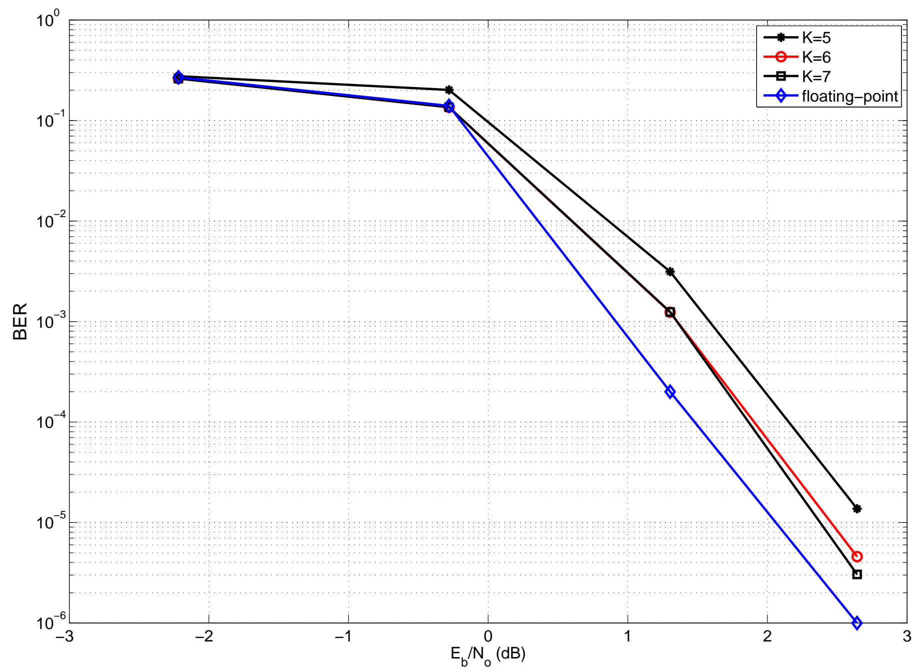


Figure 5.9: SNR vs BER for turbo decoder with 4 parallel log-MAP decoders decoding 1344 bits in total with 4 iterations.

Table 5.6: Performance comparison of the proposed decoder structures

		E_b/N_0 (dB)			
		1.3		2.6	
		Interleaver Size			
Decoder Type	K	160	1344	160	1344
max-log-MAP PDTC decoder	Infinite	1×10^{-2}	4×10^{-4}	1.1×10^{-4}	1.5×10^{-6}
	7	1.2×10^{-2}	3×10^{-3}	1.7×10^{-4}	3×10^{-6}
	6	1.3×10^{-2}	3.2×10^{-3}	1.9×10^{-4}	6.1×10^{-6}
	5	2.5×10^{-2}	9.7×10^{-3}	3.9×10^{-4}	1.4×10^{-5}
log-MAP PDTC decoder	Infinite	9.9×10^{-3}	2×10^{-4}	3.3×10^{-5}	1×10^{-6}
	7	1×10^{-2}	1.2×10^{-3}	5.3×10^{-5}	3×10^{-6}
	6	1×10^{-2}	1.2×10^{-3}	7.8×10^{-5}	4.5×10^{-6}
	5	1.2×10^{-2}	3.1×10^{-3}	2.3×10^{-4}	1.3×10^{-5}
PDRAC decoder	Infinite	2.1×10^{-2}	2.3×10^{-2}	9.2×10^{-4}	2×10^{-5}
	7	3.7×10^{-2}	3×10^{-2}	1.1×10^{-3}	4.1×10^{-5}
	6	3.9×10^{-2}	2.9×10^{-2}	1.4×10^{-3}	5.6×10^{-5}
	5	4.6×10^{-2}	2.8×10^{-2}	2.5×10^{-3}	2.5×10^{-4}

5.2.4 Memory Complexity

The parallel decoder structure requires the observations to be stored in multiple memory segments. We use different memory structures for PDTC and PDRAC decoders.

5.2.4.1 PDTC memory structure

In PDTC we use N MAP decoders operating in parallel. These N decoders need N memory blocks for data bit observations (d for decoders in Chapter 2). Accordingly, N memory blocks are used for parity observations and N memory blocks for interleaved parity observations (p_1 and p_2 for decoders in Chapter 2). In addition to these, N memory blocks are also defined for interleaver (memory collision-free interleavers) tables.

5.2.4.2 PDRAC memory structure

For N parallel MAP decoders, the observations are stored into N memory blocks. Different than the PDTC case, there are no data observations. Similar to the PDTC case, N memory blocks are used to store the interleaver 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. These L_e and L_a notations are eligible 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 illustration which defines 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 both PDTC and PDRAC decoders.

Summing up all yields a usage of $7N$ number of memory blocks for PDTC decoder and $5N$ for PDRAC decoder.

5.2.5 Transmission Bit Rate

Large decoding latencies in turbo and turbo-like codes are told to be the drawback of these algorithms. By making them operate in parallel, a decrease in their decoding latencies is expected. To observe that decrease, decoding latencies are better given in a formula. The decoding latency, τ, for our parallel decodable turbo code decoder stucture (both log-MAP and max-log-MAP) is

$$\tau = \left(\frac{D}{N} + 6\right) 2I, \tag{5.1}$$

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 BCJR decoder operating with the CTT algorithm. The addition of 6 in (5.1) is the result of the latency in BCJR (4) and interleaver structure (2) due to pipelining. 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.

Similarly for PDRAC decoder the decoding latency can be expressed as

$$\tau = \left(\frac{D}{NR} + \frac{D}{2NR} + 6\right) I \tag{5.2}$$

$$= \left(\frac{3D}{2NR} + 6\right) I \tag{5.3}$$

where R is the code rate. The term $\frac{D}{NR}$ in (5.2) is the latency introduced by the BCJR decoders and the term $\frac{D}{2NR}$ is the latency introduced by the accumulate decoders.

During the decoding latency calculations, it assumed that a *ping-pong* buffer structure is used in the receiver side. While a packet is being received, the observations are stored in memory in a quantized form. After that, *ping* memories are filled as described in Section 5.2.4 for d, p_1 and p_2 observations and decoders begin to run. During the decoding process if another packet arrives, the d, p_1 and p_2 observations are stored in *pong* memories. In this case, the decoding process is not affected by the reception of the new packet. When the decoders finish their job, they operate on the *pong* memories and this time the *ping* memories are free for another packet storage. This structure doubles the memory usage in the system for storing observations.

Table 5.7: Comparison of the proposed decoder structures

Decoder	N	I	K (bits)	Clock speed (MHz)	SNR for $BER = 10^{-3}$	Bit Rate (Mbps)
PDTC with max-log-MAP	8	4	6	80	~2.2 dB	61.54
PDTC with log-MAP	6	4	6	60	~2.0 dB	36.73
PDRAC with log-MAP	10	8	7	60	~2.7 dB	15.4

At this point, we can make a final comparison between all the proposed structures in terms of maximum available data rates. If we denote the data rate by υ, we can formulate it as,

$$\upsilon = \frac{D \text{ x } f}{\tau}, \tag{5.4}$$

where f is the maximum available frequency and τ is the decoding latecy. To find the exact data rate, we need to decide on the architecture, number of data bits in a packet (D), metric representation width (K), iteration number (I), the number of constituent decoders in a cluster (N), and the code rate (R). In data rate calculation,

the f value can be obtained by checking the Tables 5.2, 5.1, and 5.3 for the selected K value. Similarly, τ value can be obtained from (5.1), or (5.3) for the decided structure. After observing the BER performances and FPGA resource usage, it is decided to use $K = 6$ for log-MAP and max-log-MAP PDTC decoders and $K = 7$ for PDRAC decoder. Herein we used packets containing 160 data bits, that is $D = 160$. Using the Tables 5.1, 5.2, 5.3 and Figures 5.1, 5.2, 5.3 with the design choices listed above, we can generate Table 5.7 that gives all the information and comparisons needed. The table is constructed with an assumption that the number of parallelly processing sub-decoders are increased in such a way that the FPGA became almost full. The clock speeds are also adjusted for matching the clock speeds that are available in industry.

CHAPTER 6

CONCLUSIONS

Iterative decoding is one of the most effective techniques for obtaining low error rates. However, as the number of iterations increase there will be more latency in decoding which will reduce its applicability. In order to increase the decoding speed of such decoders, parallelization of many decoders is one of the most important ideas in literature. While constructing the thesis work, two iteratively decodable code types are utilized, turbo codes (TC) (known as, parallelly concatenated convolutional codes, PCCC) and repeat-accumulate (RA) codes. In order to decrease the latency at decoding of these codes to reasonable levels, we applied parallelization which yielded to parallel decodable turbo codes (PDTC) and parallel decodable repeat-accumulate codes (PDRAC) respectively. In this thesis work, the performances of the PDTC and the PDRAC decoders are investigated and compared by implementing them on an FPGA.

Marginal a posteriori (MAP) decoders are utilized as soft in soft out (SISO) subdecoders. The algorithm used for the MAP decoder was the Bahl et al. algorithm (BCJR) which is renowned for optimal performance. We have applied the a posteriori probability (APP) decoding in the decoder of PDRAC for observing its performance as in PDTC.

The parallelization idea causes new problems mainly in interleaver design. Besides the complexity of building an interleaver which supports parallel processing, due to limited access to memory blocks available in the FPGA the memory collision problem was the most crucial one. In order to build collision free and effective interleavers, row-column *S-random* interleaver design technique was used.

In the thesis, the aim was observing the performance of aforesaid codes under AWGN channel. The channel was created in the hardware by producing pseudo-random Gaussian noise components through applying linear feedback shift register approximation. The distribution of generated random variables was quite similar to that of random Gaussian noise.

A reasonable way for designing low-latency decoders in an FPGA was using an integer based approximation by representing numbers by K bits in FPGA. However, integer based operations came with some consequences. The first drawback was the limited set of numbers for operations. The set defined at the out of the decoder can be numbers from $-\infty$ to $+\infty$. So this large set is modified by using normalization techniques for proper usage of observations in decoding algortihms. Since the maximum and minimum available numbers in operations are defined by K, overflows or underflows are highly predicted to happen in addition and subtraction operations. So, addition and subtraction operations are redefined for handling these kinds of situations. Next, the implementation of a log-MAP decoder has some fine details which includes *correction term*. The correction term is composed of a logarithmic function and its implementation heavily decreases the performance of the decoders in terms of both latency and resource allocation. In order to overcome this situation an, accordingly defined look-up table is used.

In terms of SNR vs BER performance log-MAP based PDTC decoder was the best operating decoder, but its resource utilization was the largest. max-log-MAP based PDTC decoder's performance is observed to be about 0.2 dB worse than the log-MAP based one. However, its both clock speed and resource allocation was better than log-MAP. If this small performance loss is neglected, max-log-MAP based PDTC decoder seems to be the best match for high-speed communications. The performance of PDRAC decoder was the worst among inspected decoder architectures. The FPGA used in this study is a moderate sized FPGA. For such an FPGA PDRAC decoder is not a good choice and it is about 0.5 dB worse than max-log-MAP based PDTC decoder. In small-sized FPGAs like FPGAs those contain about 7-8k slices, PDRAC decoder can be used unless communications speed more than 10 Mbps is needed.

After this study some future work can be recounted. RA codes became famous for

their good BER performance. If the BER performance of these codes is improved they can be competitive in terms of resource allocation. A sum-product algorithm (SPA) based decoder can be constructed, put into parallelized form, and be implemented. The performance of the provided SPA decoder can be compared to that of the PDRAC decoder. Additionally, after invention of RA codes new RA-based codes appeared. *Irregular Repeat Accumulate Codes (IRA), Accumulate Repeat Accumulate (ARA) codes*, *Accumulate Accumulate Repeat Accumulate (ARAA)*, and *Accumulate Repeat Accumulate Accumulate (ARAA)* codes are variants of it. The BER performance of these codes can also be observed by implementation.

REFERENCES

[1] Tuğcan Aktaş. Parellel decodable channel coding implemented on a MIMO testbed. Master's thesis, Middle East Technical University, August 2007.

[2] Çağlar Kılcıoğlu. FPGA implementation of jointly operating channel estimator and parallelized decoder. Master's thesis, Submitted to Graduate School of Natural and Applied Science for approval, METU, September 2009.

[3] Peter Alfke. Efficient shift registers, LFSR counters, and long pseudo-random sequence generators. *Xilinx Application Notes No 52*, 1996.

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

[5] Dariush Divsalar, Hui Jin, and Robert J. McEliece. Coding theorems for turbo-like codes. *in Proc. 36th Annual Allerton Conference on Communications*, 1998.

[6] S. Dolinar and D. Divsalar. Weight distribution for turbo codes using random and nonrandom permutations. *JPL Progress Report 42-122*, pages 56–65, 1995.

[7] Enes Erdin, Çağlar Kılcıoğlu, and Ali Özgür Yılmaz. An implementation-based comparison of parallelized turbo decoders. *An article submitted to IET Communications*, 2009.

[8] R. Gallager. Low-density parity-check codes. *IRE Transactions on Information Theory*, 1962.

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

[10] Orhan Gazi and Ali Özgür Yılmaz. Collision free row-coloumn S-random interleaver. *IEEE Communications Letters*, 13(4), April 2009.

[11] Xilinx User Guide. ML-402 board user guide. *Xilinx*, 2008.

[12] Hui Jin. *Analysis and design of turbo-like codes*. PhD thesis, California Institute of Technology, 2001.

[13] J. Jung, I. Lee, D. Choi, J. Jeong, K. Kim, E. Choi, and D. Oh. Design and architecture of low-latency high-speed turbo decoder. *ETRI Journal*, 27(5):525–532, October 2005.

[14] L.Bahl, J.Cocke, F.Jelinek, and J.Raviv. Optimal decoding of linear codes for minimizing symbol error rate. March 1974.

[15] Shu Lin and Daniel Costello. *Error control coding*. Pearson Education International, 2004.

[16] Claude E. Shannon. A mathematical model of communication. *Bell System Technical Journal*, 27:379–423, 1948.

[17] T. A. Summers and S. G. Wilson. SNR mismatch and online estimation in turbo decoding. *IEEE Transaction on Communications*, 46(4):421–423, April 1998.

[18] R. Michael Tanner. A recursive approach to low-density codes. *IEEE Transactions on Information Theory*, 1981.

[19] Stephan B. Wicker. *Error Control Systems for Digital Communication and Storage*. January 1995.

[20] A. Worm and P. Hoeher. Turbo-decoding without SNR estimation. *IEEE Communications Letters*, 4(6):193–195, June 2000.

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