University of Alberta

A Power Evaluation Study for an 802.16e LDPC Encoder Implementation with Wireless Implantable Medical Devices

by

Samer Chomery 

A thesis submitted to the Faculty of Graduate Studies and Research
in partial fulfillment of the requirements for the degree of

Master of Science

Department of Electrical and Computer Engineering

Edmonton, Alberta
Fall 2008

Your file Votre référence
ISBN: 978-0-494-47190-6
Our file Notre référence
ISBN: 978-0-494-47190-6

To My Parents

Abstract

Low-density parity-check (LDPC) codes are forward error control (FEC) codes that are being increasingly adopted in emerging communication systems due to their ability to provide capacity approaching coding performance and the availability of practical VLSI implementations. Low-time-complexity encoding of LDPC block codes has been a challenge. The LDPC codes supported in the 802.16e (mobile WiMAX) standard are designed to take advantage of efficient time and circuit encoding techniques. This thesis explores the design and implementation of an LDPC encoder that is compatible with the 802.16e standard.

Wireless implantable medical devices (IMDs) are an important emerging communications application. One design challenge of IMDs is the low-power consumption required to achieve the necessary battery life. This thesis evaluates the power consumption of the implemented LDPC encoder for use in implant wireless transmit data links. To achieve the necessary power efficiency we need a lower power implementation than the FPGA encoder presented.

Acknowledgments

First and foremost, I would like thank the Lord for blessing me and for providing me with a wonderful life, and especially for the opportunity to pursue my education and career in Canada.

Second, I am grateful to my parents, Esam and Aysar Chomery, for the love and support they've given me all my life, and for their sacrifices without which I wouldn't be where I am today. I thank my brother Ziyad who was always there for me to talk to and watch sports with. I am very thankful to my fiancé Dima Oweis, who stood by me every step of my graduate program and helped and encouraged me during the difficult times. Thank you for your love.

Third, I wish to express my immense gratitude to my supervisors, Dr. Stephen Bates for his guidance, advice, patience, and for sharing his knowledge and technical insights with me. I am also grateful to my co-supervisor Dr. Duncan Elliott, who provided guidance and many contributions to my learning experience.

Fourth, I would like to show my appreciation to everyone at TRLabs; Dr. John Nielsen, Dr. Robert (Bob) Davies, Dr. Grant McGibney, and the support staff at the Calgary lab who helped introduce me the field of wireless communications and encouraged me to pursue further research opportunities. I thank everyone at the Edmonton lab for providing all the learning resources I needed and an excellent working environment. An additional thank-you goes to Luke Chong for making sure my computing experience always ran smoothly.

Fifth, my appreciation goes out to all my friends and colleagues at the Department of Electrical and Computer Engineering at the University of Alberta and TRLabs for the many helpful discussions and late night work-and-pizza sessions.

I'll finish by giving thanks to the organizations and institutions whose financial aid has been instrumental in allowing me to do this work. To NSERC, TRLabs, and the University of Alberta I am humbly grateful.

Table of Contents

List of Tables

List of Figures

List of Abbreviations

Symbol	Description
3G	Third Generation of development in wireless communication networks, especially mobile phone networks
ALT	Almost Lower Triangular Form of a Matrix
ANSI-C	The C language programming standard published by the American National Standards Institute
API	Application Programming Interface
ARQ	Automatic Repeat-Request Codes
ASIC	Application-Specific Integrated Circuit
AWGN	Additive White Gaussian Noise Information Channel
BAN	Body Area Network
BCH	Bose-Chaudhuri-Hochquenghem Code
BER	Bit Error Rate
BP	Belief Propagation
BPL	Body Path-Loss Factor
BRAM	Block Read Access Memory
BW	Bandwidth
CC	Convolutional Code
CDMA	Code-Division Multiple-Access
CRC	Cyclic Redundancy Check
CTC	Convolutional Turbo Code

DL	Downlink
DLL	Dynamic Link Library
DUT	Device Under Test
DVB-S2	Digital Video Broadcast – Satellite – Second Generation
ECC	Error Control Codes
EIRP	Effective Isotropic Radiation Power
EM	Electromagnetic
EVDO-Rev. B	Evolution-Data Optimized – Revision B, a 3G wireless standard
FCC	Federal Communications Commission
FDTD	Finite-Difference Time-Domain
FEC	Forward Error Correction
FER	Frame Error Rate
FF	Flip-Flop
FIFO	First-In First-Out
FPGA	Field-Programmable Gate Array
FSM	Finite State Machine
GF	Galois Field
GUI	Graphical User Interface
HDL	Hardware Description Language
HSPA	High-Speed Packet Access
IEEE	Institute of Electrical and Electronics Engineers
IMD	Implantable Medical Device
IP	Internet Protocol

ISM	the Industrial, Scientific and Medical Frequency Band
LAN	Local Area Network
LDPC	Low-Density Parity-Check Code
LDPC-BC	Low-Density Parity-Check Block Code
LDPC-CC	Low-Density Parity-Check Convolutinal Code
LLR	Log-Likelihood Ratio
LNA	Low-Noise Amplifier
LSB	Least-Significant Bit
LUT	Lookup Table
MAN	Metropolitan Area Network
MEMS	Microelectromechanical system
METAIDS	Meteorological Aids Service
MICS	Medical Implants Communication Service
MIMO	Multiple-Input Multiple-Output
MSB	Most-Significant Bit
MUX	Multiplexer
NCD	Native Circuit Description File
NICT	National Institute of Standards and Technology
OFDM	Orthogonal Frequency Division Multiplexing
PAN	Personal Area Network
PDA	Personal Digital Assistance
PLE	Path-Loss Exponent
PLL	Phase Lock Loop

QC	Qausi-Cyclic
QoS	Quality of Service
QPSK	Quadrature Phase-Shift Keying
RAM	Random Access Memory
RF	Radio Frequency
RICA	Reconfigurable Instruction Cell Architecture
RNF	Receiver Noise Figure
SNR	Signal-to-Noise Ratio
TG6	The WPAN Task Group 6 for Body Area Networks
TU30	Typical Urban 30 Information Channel
UL	Uplink
USB	Universal Serial Bus
UWB	Ultra-Wide Band
VCD	Value Change Dump
VHDL	VHSIC Hardware Description Language
VHSIC	Very High Speed Integrated Circuit
VLSI	Very Large-Scale Integration
WAN	Wide Area Network
WiMAX	Worldwide Interoperability for Microwave Access
WMAN	Wireless Metropolitan Area Network
WPAN	Wireless Personal Area Network
XOR	Exclusive-OR

Chapter 1

Introduction and Thesis Overview

1.1 Project Overview

Channel coding is a technique used to mitigate the effects of noise on the transfer of information in communication systems. The development of channel codes that achieve good coding performance has been the focus of researchers since the late 1940s. Several code families have been developed that provide effective performance in various communication systems and applications [3]. In today's expanding market of portable wireless communications, it is important for communication vendors to develop low-circuit-complexity channel codes for hardware implementation. Several channel coding classes have recently emerged as high-performance, low-complexity codes such as turbo codes and low-density parity-check (LDPC) codes [4]. These codes are quickly becoming the coding techniques of choice for emerging wireless communication systems.

LDPC codes provide comparable coding performance to the widely adopted turbo codes while being more suited for high-throughput, low-latency applications. In addition, LDPC codes offer a wider range of tradeoffs between performance and decoding complexity [13]. The low-complexity decoding of LDPC codes is one of the main reasons for their popularity. This is due to the utilization of the belief propagation (BP) algorithm, which is an iterative decoding algorithm that is well suited for efficient VLSI implementations.

The encoding of the block version of LDPC codes has been criticized because of the apparent high time complexity involved. Direct encoding of LDPC block codes requires the multiplication of vectors with dense matrices, and hence is of $O(n^2)$ complexity. The complexity increases further with larger packets (code lengths). However, the code's performance also improves with larger code lengths. Thus, several code construction methods and algorithms have been proposed in the literature that aim to achieve linear time-complexity encoding of LDPC block codes [30, 31, 32, 34]. This issue highlights one of our project motivations; to gain further insight into efficient encoding algorithms for LDPC block codes and their hardware implementations.

One approach to investigating LDPC encoding implementation and its practicality is to explore the use of LDPC codes in a defined communication system. For our project, we chose the IEEE 802.16e (WiMAX) standard [10]. WiMAX is an emerging wireless metropolitan area network standard that provides fixed and mobile broadband wireless access. WiMAX employs several new technologies such as multiple-input multiple-output (MIMO) antenna arrays, space-time coding, and bandwidth and network scalability. The mobile version of WiMAX adopts LDPC block codes as optional channel coding schemes. The standard supports a set of block codes with six code rates and nineteen code lengths per rate. The supported codes are designed for efficient encoding and present a suitable technology platform for our LDPC encoder implementation project.

The above discussion frames the main goal of this project; to design and implement a low-time-complexity low-circuit-complexity LDPC block encoder, while maintaining compatibility with the IEEE 802.16e standard. The encoder is implemented as an

intellectual property (IP) core on an FPGA device. This provides the ability to use the encoder as a "drop-in" module for various digital designs, as well as provides a working hardware platform for future research into new implementation ideas. Several design challenges exist:

- Taking advantage of the proposed efficient encoding algorithm and the compact structure of the adopted codes in the standard.
- Incorporating all supported code rates and lengths into a single low-circuit-complexity encoder design, with the ability to switch between code rates and lengths during operation.
- Minimizing hardware utilization and memory requirements.
- Developing a low and consistent latency encoder.

Beyond implementation, another research opportunity exists in studying the performance and suitability of LDPC encoders in emerging applications. One such application is the use of high performance codes with implantable medical devices (IMDs). Wireless IMDs are not a new technology; however, the potential treatment benefits that could be gained through utilizing high-throughput wireless transceivers in IMDs are significant [60]. An example of such benefits is the ability to produce non-invasive "swallow-able" imaging devices that are able to capture and transmit images of a patient's gastrointestinal tract for better diagnosis and treatment. The main challenge facing implant designers is the requirement to build low-power transceivers that are able to provide high-throughput with high data reliability [67]. Therefore, another goal for this project is to characterize the power consumption of the implemented 802.16e encoder, as well as a convolutional LDPC encoder, and evaluate their suitability for low-power

medical implant use. We consider the encoder-only scenario at the implant. Furthermore, since designs on FPGA devices are known to consume higher power than the same designs on power-efficient ASICs [101]. We also estimate the power consumption of these encoders when implemented on ASIC devices. In order to conduct this evaluation study, several issues must be addressed:

- Identifying a mathematical model for evaluating the power efficiency of LDPC encoders for medical implant applications.
- Researching reasonable approximations for the scaling of FPGA power consumption levels to ASIC levels.
- Identifying possible power improvement options for our 802.16e LDPC encoder.

1.2 Thesis Overview

In Chapter 2, we elaborate on the concepts mentioned in Section 1.1. Key concepts in information theory and channel coding are briefly reviewed. LDPC codes are introduced with a discussion of decoding and encoding algorithms and implementations. The IEEE 802.16e standard is introduced with a focus on LDPC codes' performance. The last section of Chapter 2 discusses the main challenges faced by wireless IMD designers including low power consumption and the modeling of wave propagation through the human body.

In Chapter 3, we discuss our implementation of the 802.16e LDPC encoder. The structure of the supported LDPC codes and the proposed encoding algorithm by the standard are presented. We describe the design and verification processes applied and

expand on the software modeling process. The hardware components of the design are detailed and they include the matrix-vector multipliers, memory storage and controllers, and the master Finite State Machine (FSM).

Chapter 4 mainly describes the form and usage of the encoder as an IP core illustrating the various input/output ports used to interface with it. In addition, Chapter 4 describes the behavioral, timing, and hardware verification processes used to confirm the functionality of the design. The throughput and hardware utilization of the encoder are also recorded.

Chapter 5 conducts a wireless medical implant power consumption study. The software tools used to estimate the power consumption of the encoders are presented. We also introduce a convolutional LDPC encoder design based on [91]. The power consumption of both the block and convolutional encoders is estimated and analyzed. We derive a mathematical model to evaluate the minimum encoder transmit energy required to achieve a certain coding performance level. The FPGA designs are evaluated against an uncoded system. ASIC approximations of the same designs are performed and evaluated for IMD use. Additionally, the encoders' power estimates are compared to industry-leading devices and possible power improvement options are presented.

Finally, the results are summarized and the future work for this project is discussed in Chapter 6.

Chapter 2

Background and Literature Review

2.1 Information Theory and Channel Coding

In communication systems information is transmitted from a source to a destination. The transmitter converts the information into a form suitable for transmission through the physical medium be it a copper wire, a fiber optic link, or radio frequency waves propagating through space. The receiver detects the transmitted signal and converts it back to the original information message. These physical media are *information channels* that carry the transmitted information messages. Ensuring the integrity of the transmitted messages in these channels would be a trivial task if it wasn't for the presence of *noise* that may corrupt the information during transmission. In this context, *noise* can be defined as any unwanted signal or effect in addition to the transmitted signal in the channel. The sources of noise vary from one channel to the other; interference from other transmissions, thermal effects generated by the electronic devices being used, environmental interference, etc. When the transmitted information messages are corrupted by noise, the receiver may not be able to recover the original information and may interpret these messages in error. *Information Theory* and *Channel Coding* are the fields of engineering that deal with the definition and analysis of information in communication systems as well as the various techniques used to combat the effects of noise on these systems.

It was Claude Shannon's revolutionary 1948 paper "A mathematical theory of communication" [1] that laid the foundations for the field of *Information Theory*. In his work, Shannon quantified the information that is transferred from source to destination in a communication system. He was particularly concerned with defining or measuring the information-carrying capacity of a channel. Realizing that the presence of noise in a channel reduces its information-carrying capacity, Shannon presented one of his most significant contributions; Theorem 17 in his paper, known as the *Information Capacity Theorem.* In it he defined the channel's capacity as the rate of discrete information units (or bits) that can be transmitted reliably (i.e. without error) over the channel measured in bits per second. While many mathematical models have been developed to describe the behavior of transmissions through various noisy information channels, the Gaussian channel is a fundamental and useful channel model for various communication channels such as wireless links and fiber optic cables [2]. This channel models the noise as a zero-mean Gaussian random variable, which means that it assumes a Gaussian distribution of noise amplitude with a constant noise power over a limited channel bandwidth. The output of the channel is defined to be the addition of the noise and the channel's input. That is the channel output is perturbed by *additive white Gaussian noise* (AWGN). Shannon used the AWGN channel model to state and prove the famous theorem mentioned above. In it [1], he stated that the information capacity of a channel of bandwidth W hertz, subjected to AWGN of power N watt, when the average transmitter power is limited to P watt is given by:

$$C = W \log_2\left(1 + \frac{P}{N}\right) \text{ [bits per second]} \tag{2.1}$$

P/N is the ratio of the transmitter signal power to the noise power, hence called the signal-to-noise ratio (SNR), and is one of the most important parameters communication engineers must consider when designing a communication system. With this closed form equation, Shannon asserted that it was theoretically possible to reliably transmit information bits with rates upper-bounded by channel capacity in the presence of noise in an information channel. However, he gave no constructions of schemes that could achieve or approach this capacity.

In order to mitigate the effects of noise on the transmitted information, communication systems use *Channel Coding* techniques. Figure 2.1 illustrates a simplified model of a communication system with the channel coding components highlighted.

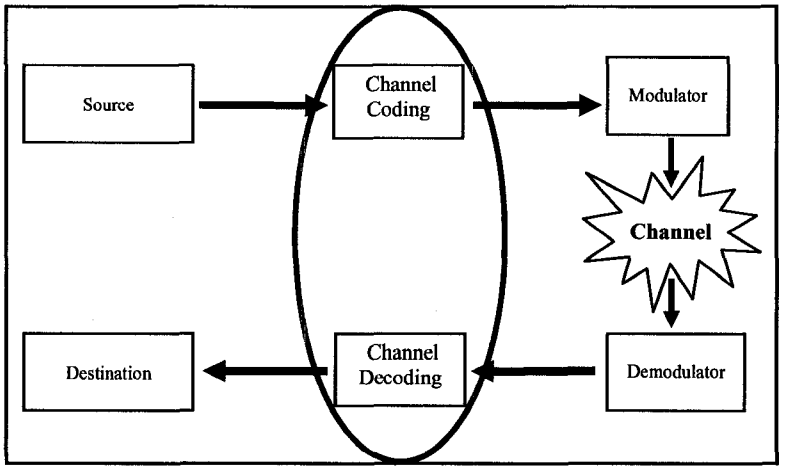


Figure 2.1: Simplified model of a communication system

Channel coding involves the adding of redundancy to the transmitted messages (or symbols) in order to minimize the probability of interpreting these messages in error at the receiver. The process of adding redundancy at the transmitter is known as *encoding* and is reversed by a *decoding* process at the receiver. The receiver's failure to interpret a transmitted symbol correctly is termed a *decoding error*. Simple addition of redundancy,

such as repetition of transmitted symbols, is not the most efficient channel coding technique as it wastes channel bandwidth and system resources. Hence, channel coding goes beyond that, and attempts to encode the transmitted messages with the minimum number of symbols while still providing adequate protection against decoding errors.

Understanding the notion of channel capacity, and channel coding techniques, Shannon proceeded to state and prove his most famous result: the *Channel Coding Theorem.* This is theorem 11 in his 1948 paper [1]. In it he explained that there exist channel codes that make it possible to achieve reliable transmission, with as small an error probability as desired, if the rate of transmitted information is less than the channel capacity. Furthermore, he explained that it is not possible, with any code, to achieve such a low error probability if the rate of information transmission exceeds the channel capacity. This bound became known as the *Shannon limit.*

Finding codes that attempt to achieve channel capacity is only part of the problem of channel coding. Developing practical algorithms for encoding and decoding such codes is another challenging part. The search for "good codes" which provide performance approaching the Shannon limit with reasonable encoding/decoding complexity, has occupied the research community for years. Many channel code families have been developed and implemented [3]. However, certain classes of channel codes have emerged as superior ones in terms of their coding performance and implementation complexity. Turbo codes are one example of such codes [4]; Low-Density Parity-Check codes are another example. Both of these codes benefit from the advances of integrated circuits technology, and can now be constructed and implemented as high-performance and practical channel coding schemes.

2.2 Low-Density Parity-Check Codes

2.2.1 Introduction

Low-Density Parity-Check (LDPC) codes are a class of forward error control codes, and were first proposed by Gallager in the 1960s [5, 6]. However, in order to be simulated and implemented, they required processing resources that were impractical at the time. Despite their coding potential they were largely forgotten until their rediscovery in 1996 by MacKay and Neal [7]. They noticed that their work with sparse random matrices reinvented Gallager's codes. In their paper, they also mentioned that the general assumption regarding concatenated codes' superiority in practical systems may have contributed to the slow development of LDPC codes. Their results, however, showed that LDPC codes outperform the classical convolutional and concatenated codes, and provide performance very close to that achieved by turbo codes.

In recent years, LDPC codes have received a tremendous amount of research focus and popularity. They have been adopted into several communication standards such as the Digital Video Broadcasting standard for Satellite Television (DVB-S2) [8], IEEE's 802.3an [9], 802.16e (WiMAX) [10], and 802.11n [11]. One of the reasons for their popularity is that LDPC codes are *capacity-approaching* which means they provide performance that approaches the Shannon limit. Some studies have shown that LDPC codes can get to within 0.0045 dB of the Shannon limit [12]. This result far exceeds any figures achieved with classical forward error control (FEC) codes of the same decoding complexity. Turbo codes are the other class of FEC codes that provides similar results. However, as discussed in [13], LDPC codes are more suited for high-throughput low-latency applications than turbo codes. And LDPC codes offer a wider range of tradeoffs

between performance and decoding complexity. In addition, unlike turbo codes which are heavily patented by France Telecom [14], LDPC codes provide a relatively more open technology access for both industry and academia. Another reason for LDPCs' popularity is that they could be decoded with a low-complexity iterative algorithm known as the *Belief Propagation (BP) algorithm*, which is suited for efficient VLSI implementations. Therefore with excellent performance versus costs of implementation, LDPC codes represent an important option for emerging and next-generation communication systems.

2.2.2 Overview of LDPC codes

2.2.2.1 LDPC Code Structure

There are two different yet related types of LDPC codes, Block and Convolutional LDPCs. The original LDPC codes proposed by Gallager are block LDPC codes. Convolutional LDPC codes were proposed in 1999 by Felström and Zigangirov [15], and are receiving increasing research focus. However, since the convolutional codes are out of the scope of this project, we will focus on the block codes' structure and relevant encoding and decoding architectures in the following sections.

Before describing the code structure, it is important to define several relevant terms from information theory. In block FEC codes, the encoder accepts a block of message bits and generates a block of coded bits at the output. The resultant bit sequence is termed a *codeword.* If a user information message of length k bits is to be encoded to generate a codeword of length n bits, the number of added redundancy bits (termed code bits) is equal to n - k bits. The *code rate* (R) can be defined as the ratio of the number of original information bits to the total number of codeword bits. And since the codeword is always larger than the un-coded bit-sequence, this number must always be between zero and one.

$$R = \frac{k}{n} \tag{2.2}$$

A block code is termed systematic if the output codeword contains the original information bits in an unaltered form. Therefore, the codeword of a systematic block code may take the form presented in Figure 2.2.

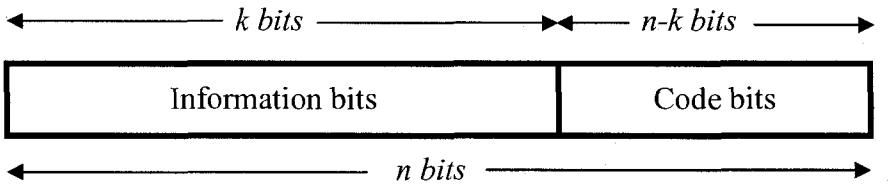


Figure 2.2: Codeword structure of a systematic block code

As the name of LDPC codes indicates, the *parity-check* is a key concept involved in their structure. In fact, Gallager introduced them as a special case of the general family of parity-check codes [5]. To form an LDPC codeword, k information bits are combined with (n - k) *parity bits*. Each parity bit is generated by performing a modulo-2 sum operation (XOR) on several pre-specified information bits. Parity-check works through detecting and correcting single-bit errors in transmission. Taking Gallager's example, Figure 2.3 shows a simple *parity-check* matrix. As indicated on the figure, parity bit 5 is generated by XOR-ing information bits 1, 2, 3 of a length n = 7 codeword. If bit 3 is corrupted by noise and arrives in error at the receiver, the decoder will be able to *correct* this error knowing the other errorless bits, namely bits 1, 2, and 5. This could be done by performing another XOR operation on these bits. It could immediately be seen that a *parity-check* matrix is a compact way of defining which bits participate in which parity-check operation according to the position of the ones. This means that the parity-check matrix defines all the parity-check equations.

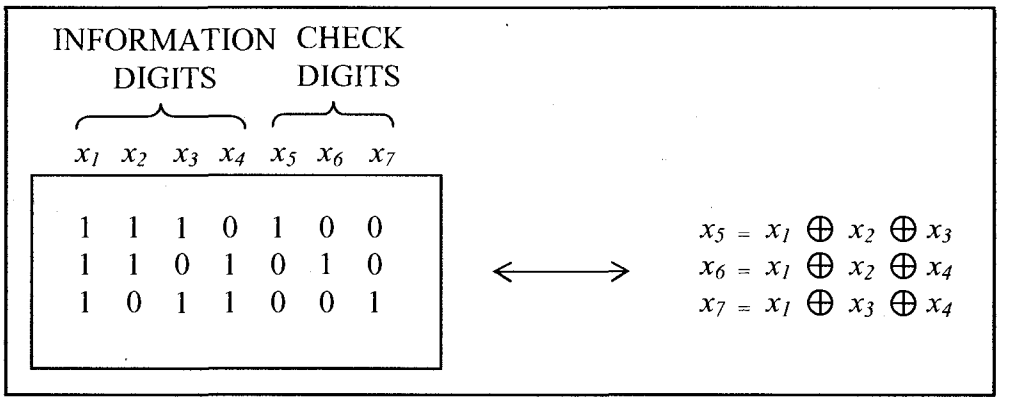


Figure 2.3: Example of a parity-check matrix, figure taken from [5]

In the previous example if a parity-check operation is considered alone, an error in more than one transmitted bit could not be corrected and would result in a decoding error. The parity-check matrix attempts to solve this by specifying each bit to participate in several parity-check operations so that if one failed the others may not. This greatly increases the strength of the parity-check code in combating bit errors. Supported by these concepts, LDPC codes are defined as the *set of codewords* that satisfy all the parity-check equations defined by the parity-check matrix. Algebraically, this means that for any codeword vector x of length n bits, and its parity-check matrix H:

$$x \cdot H^T = 0 \tag{2.3}$$

Where H^T is the transposed version of the matrix H. Another matrix could be defined through manipulations of H as the dual or the *null* of H, and is called a *generator matrix* G satisfying:

$$G \cdot H^T = 0 \tag{2.4}$$

The generator matrix of a code is used to generate all the codewords of that code. Therefore, the LDPC encoding process of a user information bit vector u can then be reduced to the following matrix multiplication:

$$x^T = u^T \cdot G \tag{2.5}$$

Where x^T is the transpose of the codeword vector x. So it can be concluded that both a parity-check matrix and its dual generator matrix, can completely specify an associated LDPC code. Figure 2.4 illustrates an example of a parity-check matrix. This matrix defines an (n,j,k) code, where n is the code length, and j and k are the number of ones in each row and column, respectively. It is important to note that if the number of ones in each row and column – also termed the *weight* – stays constant across the matrix, it is then a *regular* LDPC code. An *irregular* LDPC code has a matrix where the number of ones is not fixed in all rows or columns.

$$\mathbf{H} = \begin{Bmatrix}
1&1&1&1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0 \\
0&0&0&0&1&1&1&1&0&0&0&0&0&0&0&0&0&0&0&0 \\
0&0&0&0&0&0&0&0&1&1&1&1&0&0&0&0&0&0&0&0 \\
0&0&0&0&0&0&0&0&0&0&0&0&1&1&1&1&0&0&0&0 \\
0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&1&1&1&1 \\
1&0&0&0&1&0&0&0&1&0&0&0&1&0&0&0&0&0&0&0 \\
0&1&0&0&0&1&0&0&0&1&0&0&0&0&0&0&1&0&0&0 \\
0&0&1&0&0&0&1&0&0&0&0&0&0&1&0&0&0&1&0&0 \\
0&0&0&1&0&0&0&0&0&0&1&0&0&0&1&0&0&0&1&0 \\
0&0&0&0&0&0&0&1&0&0&0&1&0&0&0&1&0&0&0&1 \\
1&0&0&0&0&1&0&0&0&0&0&1&0&0&0&0&0&1&0&0 \\
0&1&0&0&0&0&1&0&0&0&1&0&0&0&0&1&0&0&0&0 \\
0&0&1&0&0&0&0&1&0&0&0&0&1&0&0&0&0&0&1&0 \\
0&0&0&1&0&0&0&0&1&0&0&0&0&1&0&0&1&0&0&0 \\
0&0&0&0&1&0&0&0&0&1&0&0&0&0&1&0&0&0&0&1
\end{Bmatrix}$$

Figure 2.4: Parity-Check matrix for (20, 3, 4) LDPC code

Sparseness is an important characteristic of an LDPC code's parity-check matrix. A sparse matrix is one that has a small number of nonzero entries. When dealing with regular LDPC codes, increasing the block length while keeping the number of ones per row and column constant lowers the density of the ones. This is where the *low-density* part of the name originates. It has been shown that as n is made large, the density of the ones approaches zero and the performance of the code increases [4, 6]. The sparseness

and low-density concepts have their implications on encoding and decoding, and translate into efficient decoding algorithms, as will be discussed further in section 2.2.2.2.

In order to produce "good codes" that have a low probability of decoding errors, several construction methods were proposed, and could be categorized as *randomly constructed* codes and *algebraically structured* codes. Randomly constructed LDPC codes were proposed first, such as the ones in [5, 7, 12, 16], and they work through randomly placing 1's and 0's in the parity-check matrix subject to some constraints. For example the number of 1's in each row and column could be required to be constant. These studies show that powerful randomly-constructed codes demonstrating excellent theoretic performance thresholds can be found. However, randomly-constructed LDPC codes are often hard to implement with low time complexity. Therefore, a wealth of algebraically-structured LDPC codes have been developed to tackle the complexity issue; examples include [17]-[20]. These codes use algorithmic structures and aim to strike a balance between the low-complexity encoding/decoding needed for hardware implementations and the performance expected from LDPC codes.

2.2.2.2 Decoding and Current Implementations

Decoding of LDPC codes has its roots in the work of Gallager [6]. In particular, he demonstrated that an iterative algorithm could be used to decode LDPC codes. LDPC decoding algorithms could be explained more easily with the help of elegant graphical representations known as *bipartite* graphs.

A bipartite graph contains two disjoint sets of nodes where there are no connections (edges) between nodes of the same set. Tanner used bipartite graphs to graphically represent the parity-check matrix of an LDPC code [21], which is why bipartite graphs

for LDPC codes are sometimes referred to as *Tanner graphs*. A Tanner graph features two sets of nodes named *variable nodes* and *check nodes*. Each bit in the codeword of length n is represented by a variable node, and each row of the parity-check matrix H is represented by a check node. An edge exists between the i^{th} variable node and the j^{th} check node if and only if the parity-check matrix entry $h_{ij} = 1$. It can be immediately seen that the Tanner graph can completely specify an LDPC code. The same sparseness principle is transferred over from the parity-check matrix to the Tanner graph and allows for algorithmic efficiency of LDPC codes. Figure 2.5 illustrates an example for a (6, 2, 3) LDPC code that is graphically represented with a Tanner graph.

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

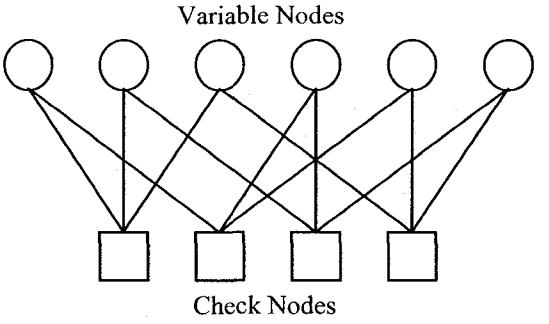


Figure 2.5: An example of a (2,3) regular LDPC code and its bipartite (Tanner) graph

The main algorithm used for decoding LDPC codes is the *Belief Propagation* Algorithm. The BP algorithm is a subclass of *message passing algorithms*. To visually describe the BP algorithm, we use the Tanner graph representation. The BP algorithm attempts to correctly decode the received codeword through several iterations. During each iteration of the algorithm, *messages* are passed between check nodes and variable nodes along the connecting edges. The passed messages are the probabilities or *beliefs* of the logical bit values at the variable nodes. The algorithm takes advantage of the inter-dependencies that exist in the graph (i.e. the parity-check matrix) and adjusts the *beliefs*

of the values at the variable nodes. Therefore, during each iteration, variable nodes learn more about their values from other nodes and the parity-check relations.

The algorithm starts by initializing the variable nodes' values with a metric called the *log-likelihood ratio (LLR).* The log-likelihood ratio is an advantageous metric since it is used to quantify the level of confidence in the value of the received bit being a logical 1 or logical 0. So if the probability of the received bit x_r being a logical one given the actual received value y is denoted as $P(x_r = 1 \mid y)$ and the probability of it being a logical zero is denoted as $P(x_r = 0 \mid y)$, then the LLR $\Lambda(x_r)$ is defined as the logarithm of the ratio between these two probabilities:

$$\Lambda(x_r) = \log\left(\frac{P(x_r = 1 \mid y)}{P(x_r = 0 \mid y)}\right) \tag{2.6}$$

It is noted that the LLR value largely depends on the noise power of the channel. The higher the noise power, the lesser the confidence we have in the received value.

After the initialization, the rest of the BP algorithm can be divided into two stages.

- **Stage 1**: The check nodes construct and send their messages to the variable nodes. The check nodes utilize their *knowledge* of parity-check relations to adjust the LLRs and send them back to the participating variable nodes. Mathematically, this means all check nodes connected to variable node i send the adjusted LLR value Λ_i to variable node i:

$$\Lambda_i = 2\tanh^{-1}\left(\prod_{j/i}\tanh\left(\frac{\Lambda_j}{2}\right)\right) \tag{2.7}$$

Here the notation j/i indicates that the product includes all the LLRs except the one of the target variable node i.

- **Stage 2**: At the variable node, the incoming adjusted LLRs are summed together and are ready for use by the check node in the following iteration.

The algorithm proceeds until the decoded codeword correctly satisfies all the parity-check equations or a fixed number of iterations is reached. The BP algorithm is a powerful algorithm, and its time complexity grows linearly with the block length of the code [7, 9] which makes it feasible to build decoders for reasonably large code lengths.

High-performance, low-complexity implementations of LDPC decoders have been achieved and make these codes an increasingly compelling choice for emerging communications systems. The first published implementation was the Howland and Blanksby decoder [22] where a 690-mW and 1-Gbps throughput decoder was implemented on an ASIC chip for a rate-1/2 LDPC code. Since then, a wealth of decoder architectures and implementations has been published. The decoder proposed in [23] and implemented on an FPGA chip in [24], utilizes an architecture-aware LDPC code design approach to achieve various savings in the message-transport interconnect and memory overhead. A group at Rensselaer Polytechnic Institute, NY, proposed a decoder architecture that is designed for fast VLSI implementations [25]. Their design utilizes the partially-parallel decoder structure where a certain number of variable nodes or check nodes are mapped to single decoding unit. These decoding units are then multiplexed using time-division to achieve better decoding speed while keeping the decoder interconnect complexity low. In [26] flexible decoder architectures were proposed for the family of irregular LDPC codes. While in [27], a decoder architecture for array-code-based LDPC codes was described. Some implementations have targeted specific channel types such as magnetic recording channels [28]. Almost all variations of the decoders

proposed employ the BP algorithm and offer tradeoffs in throughput, complexity, scalability, and hardware flexibility.

2.2.2.3 Encoding and Current Implementations

While the decoding of LDPC codes is a major advantage for these codes over other error-control codes, the encoding side has received criticism because of the apparent high time complexity involved. In this context, complexity is measured by the number of mathematical operations required to encode a bit. As noted by equation 2.5, a straightforward encoding of LDPC codes requires matrix-vector multiplication and hence is *quadratic* in the block length. This makes practical hardware implementations of the encoder almost impossible at large packet lengths (usually on the order of few thousand bits). By contrast, turbo codes can be encoded in linear time. Another problem with traditional encoding of LDPC codes is the fact that to encode an information bit which arrives at time t, the encoder may require an information bit which arrives at time $t + \Delta$ adding *latency* to the encoding process. This also means that while the total number of output codeword bits is constant, the rate of bit production may not be even, requiring the buffering of output bits.

Early implementations of LDPC encoders, such as the one proposed in [29], used straight forward techniques, where a vector of information bits is multiplied by a dense generator matrix resulting in quadratic encoding complexity in the block length of the code. However, several researchers and information theorists worked through various algebraic methods to address this issue and several solutions have been proposed.

One approach used the idea of cascaded codes, where several levels of bipartite graphs are cascaded in a tree structure [30]. By choosing the number of cascaded codes

and the size of each code carefully, it is possible to construct codes for which low complexity encoding and decoding are achievable in linear time. A disadvantage with this approach is the fact that the individual cascaded stages are codes with smaller lengths than the overall code. This results in a loss of performance when compared with a standard LDPC code of the same total length.

In [31] another approach was presented. While constructing the code and in addition to the normal constraints set for the row and column weights of the parity-check matrix, another constraint is set that forces the matrix into the *lower triangular* form. This matrix leads to faster encoding times. However, since some of the "randomness" of the code is lost due to this constraint, a loss of performance is present.

Another construction of LDPC codes was presented in [32] that enables low-time-complexity encoding. In this approach, half the parity-check matrix is generated randomly while the other half is deterministic; the resulting codes are called semi-random LDPC codes. The authors of [32] propose a recursive encoding method with complexity growing linearly with the block length of the code. They further claim that semi-random LDPC codes perform comparably to fully random regular codes of similar dimensions. More recently, further research is focusing on studying semi-random LDPC codes for wireless applications. For instance in [33], methods of constructing semi-random LDPC codes that support various code rates are investigated, and their application to newer diversity-based communication systems is explored.

A significant step towards achieving linear-time LDPC encoding was realized through the work of Richardson and Urbanke [34]. They noted that although direct encoding seems to have a quadratic complexity, the actual number of operations required

is manageable even for long block lengths. Furthermore, they managed to take advantage of the sparseness of the parity-check matrix through "optimization" steps to achieve efficient encoding. These steps require *preprocessing* the parity-check matrix via row and column permutations to achieve an *almost lower triangular* (ALT) form. Illustrated in Figure 2.6, this form is *almost* lower triangular since the matrix features a distance from *true* lower triangular form, and this distance is termed the *gap* (g). It is important to know that these preprocessing steps are applied after constructing the code, thereby preserving the sparseness and randomness of the matrix, and the strength of the code. The detailed steps of the algorithm are presented in [34].

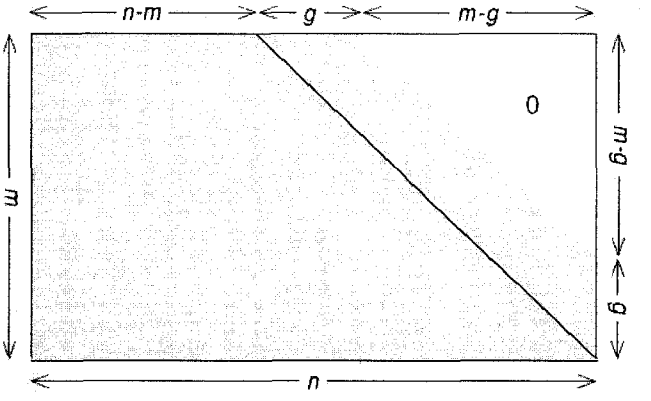


Figure 2.6: A matrix in almost lower triangular (ALT) form

The proposed algorithm involves splitting the parity-check matrix into smaller sub-matrices and performing several matrix multiplication steps. Most of the multiplications use sparse matrices and have linear time complexity except the multiplication by the dense sub-matrices Φ and T^{-1} (see [34] for details). The complexity of the overall algorithm is upper bounded by $O(n+g^2)$, and the smaller the gap the better the encoding time efficiency. In fact, the authors of [34] show that for sufficiently large n, the gap is

smaller than $\sqrt{n}$ resulting in linear time complexity. In practical systems, g is usually a small constant allowing for efficient encoding implementations.

An algebraic structure useful for improving the computational efficiency of LDPC encoding is Quasicyclic (QC) codes. QC LDPC codes perform very well when decoded using the BP algorithm [18], and as discussed in [35], these codes have structures that can achieve computational efficiency and good hardware implementations. A QC code is one where each codeword is a "quasicyclic" shift of another. That is, if a codeword is segmented into strings of length z bits, and all segments are circularly shifted by the same amount, the resulting vector is also a codeword. An ingenious method of constructing QC LDPC codes is through the use of *circulant* matrices [17, 18]. A circulant matrix is a z x z square matrix where each row is a circular shift of the row above it, and the first row is a circular shift of the last. Therefore, QC LDPC codes may be defined by a parity-check matrix H that is composed of blocks of circulant matrices and zero matrices, as shown in Figure 2.7. Such a code is termed a *block-circulant* LDPC code.

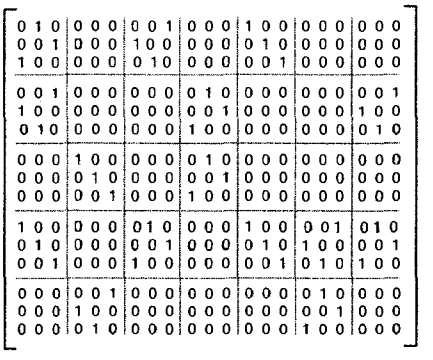


Figure 2.7: An example of the internal structure of a matrix composed of circulant identity matrices, figure taken from [36]

Important advantages could be realized by using block-circulant parity-check matrices. Particularly, the time efficiency of the encoding algorithm of [34] can be improved. The multiplication with sub-matrix Φ, which is proportional to g^2, is made

faster by minimizing the size of the gap g while preserving the strength of the code. The authors of [36] propose that picking g equal to the size of the circulant matrix z is enough for practically good code performance, while at the same time, the size of g can be made small to reduce the computational complexity. The block-circulant approach is especially advantageous in the case of multiplication with sub-matrix T^{-1}. Using this approach, T is a lower triangular matrix composed of smaller block matrices with the diagonal made of identity matrices. Thus according to [36], the direct multiplication with T^{-1} can be replaced with a multi-stage back substitution operation. This effectively converts the dense multiplication into a series of sparse vector-matrix multiplications and additions which significantly reduces the computational complexity.

Unlike the decoding side, where the BP algorithm has become the standard decoding technique, the encoding side exhibits more variations that incorporate the methods discussed earlier. In [37] an FPGA implementation of an LDPC encoder based on the ALT algorithm was created and achieves 44 Mbps output rate running on 143 MHz clock frequency. Encoder designs based on block-circulant matrices were presented in [35], along with a hardware implementation achieving a rate of a 100 million symbols/second. A hardware implementation based on the semi-random technique is presented in [38] and achieves low computational complexity with 99.5 Mbps output rate.

More recent implementations of LDPC encoders combine both the ALT algorithm and the block-circulant ideas to achieve high-efficiency high-throughput encoders. Furthermore, since LDPC codes are quickly becoming the codes of choice in most emerging communication standards, new implementations focus on building *standard-compliant* encoders. In [39], a CMOS implementation of a second-generation digital

video broadcast (DVB-S2) compliant codec is presented. This codec supports BCH codes as well as LDPC codes with code lengths up to 64800 bits and achieves up to 135 Mbps throughput. Another example is the 480 Mbps throughput LDPC codec built for ultra-wide band (UWB) transceivers and presented in [40].

A main focus of our work is the implementation of an IEEE 802.16e compatible LDPC encoder. However, it is worth noting that the next-generation wireless local area network standard, 802.11n, features LDPC codes of a somewhat similar structure to those supported by the 802.16e standard. Therefore, LDPC encoder implementations that support the 802.11n standard are of particular importance to our study. In [41] an LDPC encoder implementation was presented and is compatible with the 802.11n standard. This implementation exploits the structure of the supported LDPC code to achieve efficient encoding with barrel-shifter based multipliers. Another 802.11n compatible encoder architecture is presented in [42], where the encoder multiplication steps are handled via random-access memory (RAM) registers. While not entirely similar to the 802.16e standard, these implementations provide us with useful insights into techniques leveraging the efficient structure of the supported LDPC codes.

More directly related to the 802.16e standard, a group at the University of Edinburgh developed a real-time programmable LDPC encoder for the 802.16e standard [43]. The encoder is implemented on Reconfigurable Instruction Cell Architecture (RICA) processor which executes ANSI-C code of the encoding algorithm. This implementation achieves a throughput of 10 to 19 Mbps. A hardware implementation of a fully compatible 802.16e encoder is produced by Xilinx Inc. as an Intellectual Property (IP) core [44]. The Xilinx encoder core's throughput depends on the FPGA chip and the clock

frequency used. This IP core targets the Xilinx FPGA chip family only, and the internal encoder structure is not released for public access.

2.3 Wireless Communications and Low-Density Parity-Check Codes

2.3.1 Brief Overview of Wireless Communications

Wireless communications are the transfer of information over a distance without the use of electrical conductors such as copper wires or fiber optic cables. The information is transferred via the controlled propagation of *electromagnetic waves* through space. The distance of transfer may be as little as a few meters and as large as millions of kilometers depending on the frequency and power of the waves, as well as the technology that utilizes them.

Due to the open and un-controlled nature of the wireless communication channel, there exist several sources of interference and effects that tend to disturb the transmission and processing of wireless signals. The wireless channel experiences large-scale effects such as *path loss* (attenuation of signal with distance), and small-scale effects such as *multipath fading* and the *Doppler Effect* (for details see [45, 46, 47]). These effects cause degradation of the quality of the received signal which leads to higher probability of transmission errors. The characteristics of the wireless channel, together with additive noise and interference, need to be mitigated by various techniques such as diversity and channel coding.

Though there are many types of wireless communications, our focus is on wireless networking. Modern communication networks, where data is transmitted and accessed by

multiple network clients, are categorized based on coverage area to Personal Area Networks (PANs), Local Area Networks (LANs), Metropolitan Area Networks (MANs), and Wide Area Networks (WANs) [48]. Wireless versions of such networks exist and continue to evolve through technology innovations.

2.3.2 The IEEE 802.16 wireless networking standard (WiMAX)

The computer networking industry has many network vendors, carriers, and suppliers. Typically, various vendors have different ideas and strategies on network implementations and usage. In order to coordinate technology efforts, many networking standards have been developed and adopted by industry [48].

IEEE's 802.16 standard (also known as WiMAX) is a wireless metropolitan area network (WMAN) standard. This standard was first introduced to provide fixed broadband wireless access, and was ratified by IEEE in 2004 to be formally numbered 802.16-2004 [49]. The standard was later modified to include support for mobile broadband access in an amendment which was ratified in 2005, and formally numbered 802.16e-2005 [10]. WiMAX employs several new technologies such as multiple-input multiple-output (MIMO) antenna arrays, space-time and LDPC codes, quality of service (QoS) algorithms, and channel bandwidth and network scalability. Further details of the 802.16 standard can be found in [49, 50, 51].

The 802.16 standard supports theoretical data rates of 70 Mbps over a distance of 50 Km [49]. However, under typical network traffic conditions these limits are rarely achieved. In a comparison study done by the WiMAX forum in [52], WiMAX's performance was compared to existing and evolving third-generation (3G) mobile networks. Figure 2.8 presents the comparison results where DL is the *downlink*

connection from the base station to the mobile device, and UL is the *uplink* connection from the mobile device back to the base station. The study shows the distinct advantage that WiMAX has over other 3G metropolitan wireless networks in both throughput and spectral efficiency.

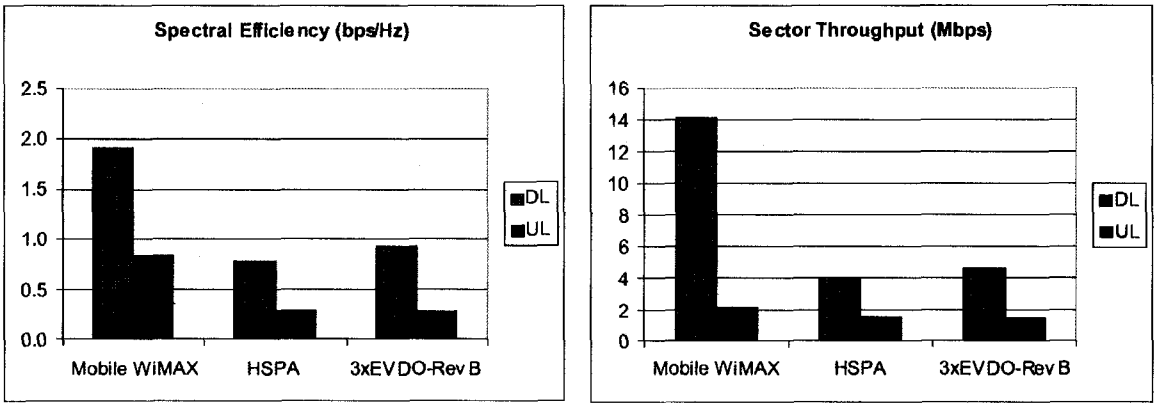


Figure 2.8: Spectral efficiency and sector throughput comparison of HSPA, EVDO-Rev. B, and Mobile WiMAX under the same traffic conditions, figure taken from [52]

In our project, we will be implementing an encoder system that is compatible with the LDPC codes supported in the mobile WiMAX standard.

2.3.3 LDPC codes in Wireless Communications

Several research studies have evaluated aspects of using LDPC codes in wireless communications. In [53] an information theory approach was taken to study the performance of LDPC codes in several types of wireless channel models such as a basic linear Gaussian channel, a partial-band jamming channel, and a Raleigh fast fading channel. The authors of [53] demonstrated that properly designed LDPC codes can be used to achieve reliable transmission over these channels making these codes an appropriate class for various kinds of wireless channels. And in [54] a group at Kansas State University explored the performance gains obtained from combining LDPC coding

with spread spectrum techniques in conventional orthogonal frequency division multiplexing (OFDM) systems. The authors simulated the performance of an un-coded OFDM system versus a convolutional coded system as well as an LDPC coded system in typical outdoor wireless environments including urban, rural and hilly terrain. The effects of code spreading were considered. Figure 2.9 presents their results in the typical urban environment. It can be seen that LDPC coded systems require less SNR to achieve a similar bit error rate (BER) over un-coded ones. Similar coding gains were recorded for the other environments.

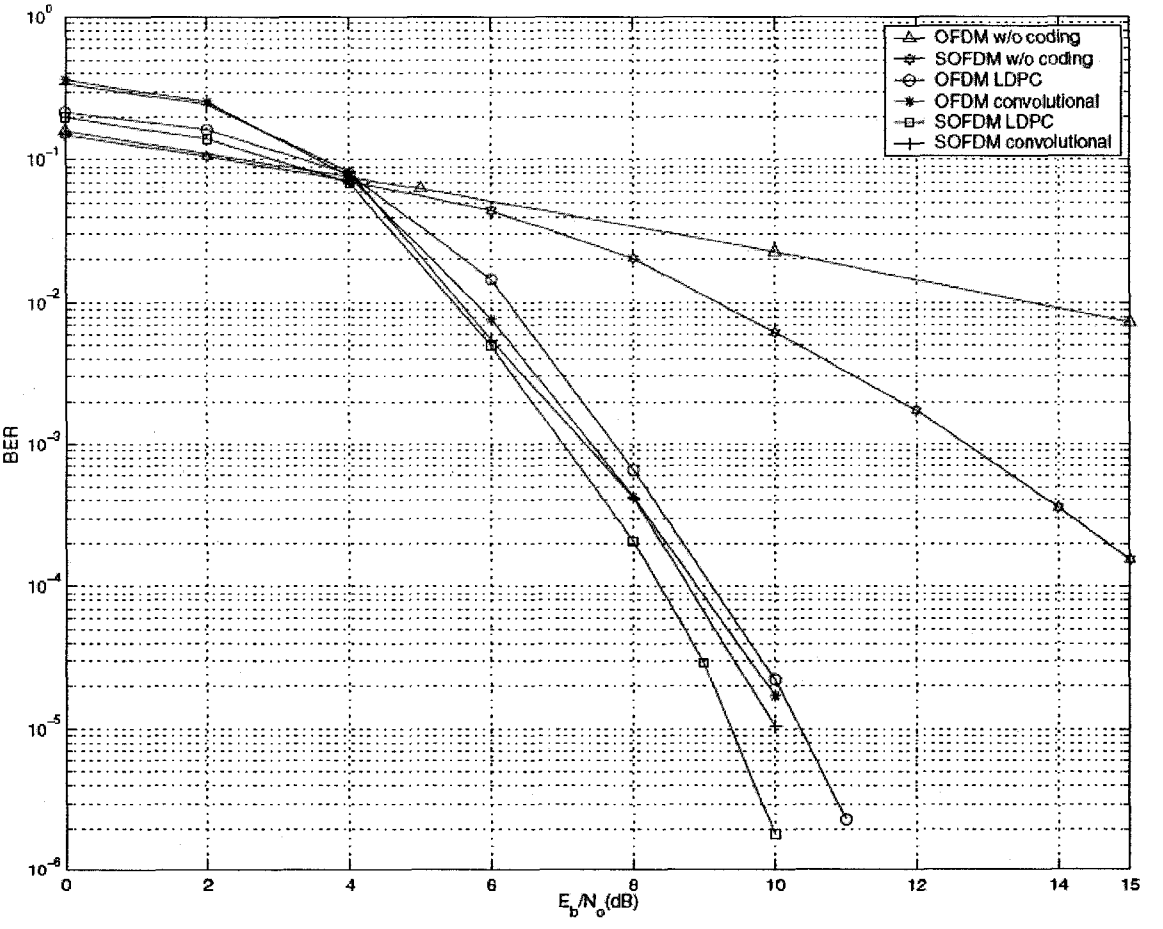
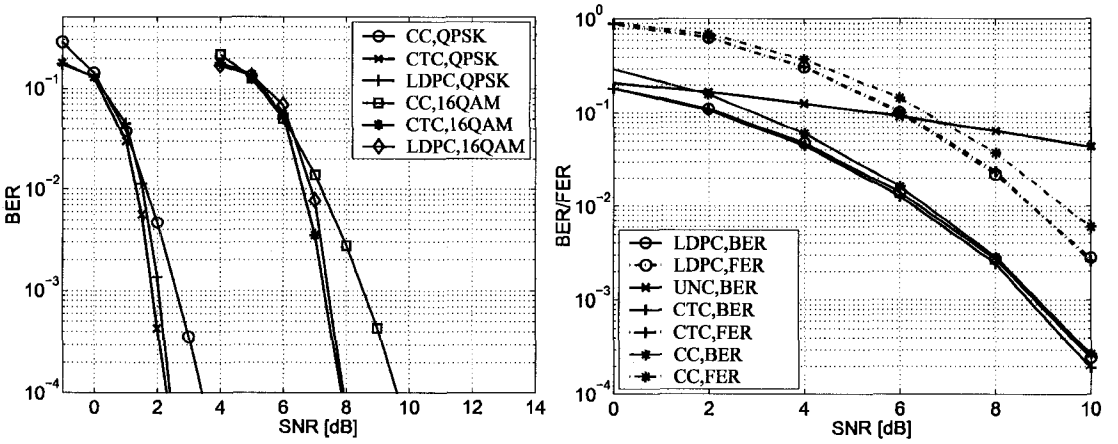


Figure 2.9: Performance of un-coded and LDPC coded OFDM and spread OFDM (SOFDM) systems in Typical Urban wireless channel, BPSK, R = 1/2, N = 1024, figure taken from [54]

On the wireless networking front, LDPC codes represent a viable option for most next-generation networks and standards. Hence many studies were conducted to evaluate the design and performance of LDPC codes for specific networking applications. Wireless sensor networks is one such application where studies such as [55, 56] have demonstrated that LDPC codes accomplish their typical coding gains and perform much better that other channel codes. In addition, LDPC codes provide the appropriate platform for low power implementations required for these networks. Other studies focused on wireless LANs such as the next-generation IEEE 802.11n [11] standard, and WMANs such as CDMA networks [57].

The use of LDPC codes in WiMAX networks has been studied extensively, and several designs, variations, and enhancements have been proposed to achieve various performance targets and tradeoffs. For example the study in [58], compares the performance of the supported LDPC codes from the 802.16e-2005 standard to a slightly optimized version. In [59], simulations were carried out to plot the performance of the 802.16e LDPC codes against other important FEC codes supported by the standard, namely convolutional codes and convolutional turbo codes. Figure 2.10 presents the study's results, where the code's BER performance was plotted using different modulation techniques. In part A, a codeword length of 576 bits and a code rate of 1/2 were used to simulate the codes on an AWGN wireless channel model. In part B, the same parameters were used to simulate the performance on the TU30 channel, modeling a mobile user in a *typical urban* environment moving with a velocity of 30 Km/h, while communicating with a base station on a carrier frequency of 3.5 GHz. Part B also includes the *frame error rate* (FER) performance curves.



A. AWGN Channel B. TU30 Channel (QPSK)

Figure 2.10: 802.16e supported FEC codes comparison, R = 1/2, N = 576, figure taken from [59]

It can be seen from the figure that the LDPC code achieved the best performance results. While comparable to the convolutional turbo code's (CTC) performance, a big advantage of the LDPC codes over their counterparts is their decoder's low circuit-complexity. Moreover, increasing the block length would provide further coding gains. However, long block lengths require more processing time which could prove challenging for systems requiring low latency codecs. The structure of these standard supported LDPC codes will be further described in Chapter 3. It is worth noting that these results correlate well with the results of Figure 2.9 demonstrating the good performance of LDPC codes in practical wireless systems.

The studies recorded in literature build a solid base for the LDPC coding choice in wireless networking, and the results encourage further investigation of these codes for next-generation technologies.

2.4 Wireless In-Body Medical Communications

2.4.1 Overview of In-Body Medical Communications

Advances in integrated circuit technology have expanded the options of modern medicine to provide new and innovative treatments. Such treatments include the use of sensor devices to monitor a patient's health data (i.e. cardiac data, blood parameters, etc.), imaging devices to monitor and evaluate parts of a patient's gastrointestinal tract and other organs, and various other temperature and muscle sensing devices. Monitoring and sensing devices may require the patient to spend a considerable amount of time in a medical facility while being treated by health professionals. In addition, expensive and invasive operations may be required to capture and transfer the data out of the human body using such links as flexible cables. These reasons were among the motivations behind the design and implementation of miniaturized portable as well as implantable medical devices. *Implantable medical devices* (IMDs) have been widely used in the treatment of many diseases such as heart diseases and neurological disorders. Pacemakers, implantable defibrillators, and hearing aids are examples of such devices.

The increasing need for IMDs that can provide ongoing and cost-effective monitoring and treatment, highlights the need for wireless solutions. Typically, a wireless-enabled medical implant may be remotely controlled from an external station while transmitting diagnostic or monitoring data during operation. The integration of wireless communications with IMDs holds several advantages over wired solutions such as enhanced patient convenience, ease of data capturing and collection, reduced risk of infection, lower costs, and enhanced mobility, all of which contribute to an improved overall healthcare experience. Traditionally, wireless IMDs used inductive RF links to

carry the data out of the human body. This required the use of a magnetic wand in the proximity of the IMD to inductively power-up the device then capture the transmitted data [60]. These devises typically operated in low-frequency ranges (i.e. 10s to 100s of kilohertz) providing data rates of less than 50 kb/sec. While proving robust and reliable, the practical antenna sizes and the inductive structure of these implants provided limited power ranges resulting in short range RF links; often requiring external receivers to have contact with the skin of the patient directly on top of the IMD's location.

In order to overcome these limitations, new low-power RF technologies are being developed to operate at much higher frequencies such as in the 400 MHz range to achieve high data rates and longer range communication links. The fast development of ultra-low-power RF transceivers may contribute to a revolution in the wireless IMD industry. Innovative medical implants could include endoscopic camera capsules that are swallow-able, heat or acidity sensors that can transmit their data to a base station meters away, and even miniaturized medication pumps such as insulin pumps for diabetes patients. Wireless implants are being studied for *body area networks* (BANs). As part of the larger field of *wireless personal area networks* (WPANs), BANs define the wireless communications of devices in the vicinity or inside the human body. The IEEE 802.15 working group [61] and its BAN task group (TG6) [62] represent the most prominent committees working on standardizing communications in this space.

Wireless IMDs are designed to operate in the un-licensed Industrial Scientific Medical (ISM) frequency bands as well as the dedicated Medical Implants Communication Service (MICS) band. In addition to allocating the MICS band, the federal communications commission (FCC), as well as European regulators, have applied

specific rules and regulations for the operation of IMDs in this spectrum [63, 64]. We present a summary of these rules.

- The available frequency band is from 402 to 405 MHz for high data rate, short range (up to 10m) wireless links between an implant and a monitoring device.
- No voice communications are allowed on this band.
- Up to 10 channels are allowed with a maximum bandwidth (BW) of 300 KHz per channel.
- The Effective Isotropic Radiated Power (EIRP) is limited to 25 μW or -16 dBm.

The MICS spectrum is shared with the Meteorological Aids Service (METAIDS) used for weather devices such as weather balloon transmissions [65]. Therefore, MICS systems are specified as indoors-only. The characteristics of signal propagation through the human body (as it will be explained further in the next section) in addition to the international availability of this frequency band makes it well suited for wireless IMD implementations.

2.4.2 Design and Implementation Challenges of In-Body Wireless Communications

When designing a medical implant with an on-board RF transceiver for wireless communications, several requirements and challenges exist.

- **Small Form Factor:** The total size of the device is an important consideration. In a medical usage, less obtrusive and more patient-friendly

devices are desired. Therefore Designers must strive to integrate as many components on-chip as possible with minimal external parts and circuitry.

- **Low Power Consumption:** This is one of the main challenges facing medical implant designers as some of these devices are expected to operate for prolonged periods of time (reaching years) without the need to re-charge or replace their batteries. Replacing the battery is not a trivial process for IMDs. In addition to battery lifetime issues, wireless transceivers are designed to consume very low power in order to allow for extra functionality such as higher data-rates or more sophisticated sensing functions. Low dissipated heat is also a health concern and a motivation for lower power devices.

  Combining the above small size and low power requirements, modern RF modules/transceivers developed for IMDs have sizes smaller than 5 x 5 x 10 mm [60] and aim to consume less than 10mA of current at 3V during operation [66]. Moreover, these RF modules are designed to spend most of their time in an ultra low power state termed *deep sleep,* while periodically "sniffing" a wake up signal from an external control station. This *duty-cycling* further helps in conserving power and prolonging the battery lifetime.

- **Low Cost:** while striving to be commercially viable, wireless IMDs face higher cost challenges than non-medical devices. For example an oscillator crystal normally costing 25 cents may cost up to 10 dollars when targeting a medical implant device [67]. The high cost of components used for medical devices stems from the relatively small number of suppliers willing to be in the medical market and its strict regulations. These include the various tests,

verifications, and documentations that medical components must undergo to be certified for medical use. Therefore IMD designers aim at integrating as many components as possible on-chip reducing the size and cost, and improving device reliability.

- **Propagation Through and Out of the Human Body:** The human body is not an ideal medium for RF wave propagation due to several reasons.
    - As recorded in initial TG6 studies [68, 69], the partially conductive nature of the human body results in *high attenuation* caused by power absorption. The attenuation effect depends on several factors including the number and types of tissues traversed by the RF waves. These tissues consist of materials with different *relative permittivity* (also known as the dielectric constant), thickness, and *electrical conductivity*. Attenuation also depends on the frequency used, and decreases exponentially with both distance and frequency. From [69], Figure 2.11 illustrates this relationship.

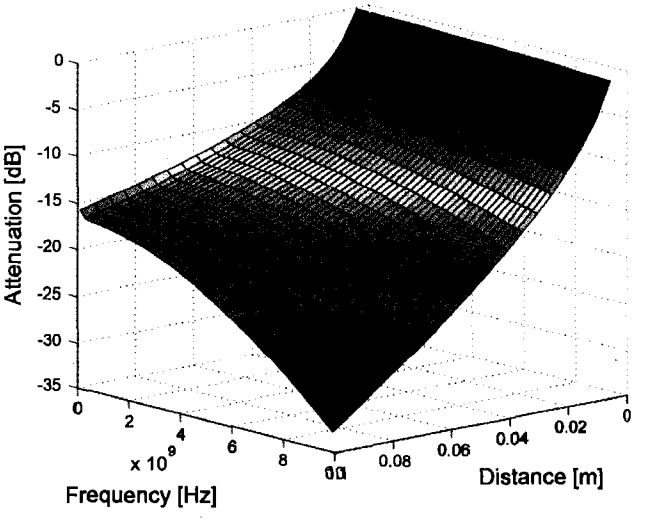


Figure 2.11: Measured body attenuation data vs. frequency and distance, figure taken from [69]

- Wave *distortion* is another challenge facing implant designers. RF waves travel with different velocities in each type of tissue, and reflect on boundaries of body tissues [69]. These waves may also experience bone and skin shadowing effects [70]. Such conditions may result in central frequency shifting and radiation pattern destruction [68]. Like attenuation, distortion is influenced by the characteristics of the tissues and the frequency used.

Being a central factor, several studies have focused on studying the best frequency that minimizes the effects of body attenuation and distortion. The authors of [71] investigated the radio propagation properties from an implant at various frequencies; specifically, the 402 MHz and 868 MHz frequencies, as well as the microwave frequency of 2.4 GHz. Recorded dielectric properties of human tissues were combined with a human body simulator built from animal tissues to yield measured as well as simulated propagation results. These results demonstrated that a higher frequency incurs higher path loss, and that the choice of an optimum frequency is important to radio range, device size, and power consumption. Confirming these results, the authors of [72] determined that path loss would increase rapidly at frequencies higher than 1 GHz, and that the 300-470 MHz band experiences relatively small loss. Based on these results and considering practical antenna sizes, the 400 MHz band represents an appropriate frequency choice for wireless medical implants.

Beside the required small antenna size, other design challenges exist. The antenna's radiation pattern and performance are affected by its shape and orientation. These, in turn, depend on the implant's shape and size, as well as the implant's in-body location. In fact, the exact field that an implant antenna generates depends on the thickness and type of surrounding tissue. Heating body tissues from radiated electromagnetic (EM) waves must also be considered when designing an implant antenna. Therefore, implant antenna challenges have been the focus of intensive study for years. Several antenna designs, shapes, and sizes have been proposed such as disk-shaped, helix-shaped, etc. [67, 68, 73].

As mentioned above, the MICS standard limits the EIRP to 25 μW or -16 dBm. Therefore, characterizing the in-body wireless channel and understanding the wave propagation model is essential to measuring and designing the implant's radiated power. Due to the various attenuation and distortion effects of the transmission medium, along with the variability of the medium from one person to another, developing an accurate propagation model for the human body is a difficult and complicated process. As well there are ethical issues associated with testing implant systems in humans.

Researchers from the Japanese national institute of information and communication technology (NICT) [69] conducted a propagation measurement experiment using the body of a pig. They submitted their results to the IEEE 802.15 BAN task group in which they derived a frequency response equation for the propagation channel that correlated well with the

measured data. They also found that the presence of various tissues results in a stochastic dispersive factor in the equation. Furthermore, they deduced that the effect of multipath propagation is negligible.

Channel characterization is an important step towards developing path loss models with which the radiated power can be specified. However, the lack of an accurate path loss model for the human body pushed scientists to use simulation data to characterize the path loss. In his Ph.D. thesis, A. J. Johansson performed several measurements and analysis of in-body propagation [74]. A human body torso simulator tool was used and is termed a *phantom*. The MICS standard defines a physical phantom as an acrylic plastic cylinder with a diameter of 30 cm. The standard also defines the thickness of the phantom and its filling tissue simulating liquid. While being easy to build and use, the phantom is not very anthropomorphic and certain measurement inaccuracies are to be expected. Measurements of the path loss from an implanted antenna in a phantom operating at 403 MHz were recorded. Simulations of these measurements were also conducted using the finite-difference time-domain (FDTD) method. In this study, the minimum path loss was found to be 23 dB with a mean of 34 dB. The maximum path loss figure was found to be 51 dB for a patient in bed with varying position. An excess loss margin of 15 dB is applied to the mean in order to cover patient orientation, antenna misalignment, obstruction of line of sight, and polarization losses which brings the body path loss figure to 49 dB. These

figures compare well with industry accepted body path loss figures of 40-45 dB [75].

In medical implant applications, the uplink (implant to monitor/base station link) is the power critical path. The monitor contacts the implant infrequently sending wake-up signals and control commands, or requesting patient's data. In comparison, the implant contacts the base station frequently sending relatively larger and more critical blocks of patient data. Therefore, the radiated power through and out of the human body is a focus when analyzing path loss. Research studies conducted to characterize the path loss of in-body medical transceivers point towards modeling the path loss using a modified version of the well-known Friis transmission formula. As presented in Equation 2.16, Friis' formula defined the received power P_{RX} at the receiver antenna based on an unobstructed line-of-sight transmission in free space.

$$P_{RX} = P_{TX} G_T G_R \left(\frac{\lambda}{4\pi d} \right)^2 \tag{2.16}$$

Here P_{TX} is the transmitter power, and G_T and G_R are the transmitter and receiver antenna gains, respectively. λ represents the wavelength used and d is the separation distance between the Tx and the Rx, which is assumed to be larger than the near field ($\lambda/2\pi$). It could be seen from this equation that the power losses increase with increasing distance and frequency. To model the in-body losses into Friis' equation, an in-body path loss factor is introduced to reflect these additional losses. Presented in [72] is a modified version of Friis' equation that incorporates this factor.

$$P_{RX} = P_{TX} G_T G_R \left(\frac{\lambda}{4\pi d} \right)^2 e^{-2\alpha d} \tag{2.17}$$

The body attenuation factor $e^{-2\alpha d}$, as included in Equation 2.17, uses the real part α of a complex figure called the *propagation constant* γ.

$$\gamma = \alpha + i\beta \tag{2.18}$$

The propagation constant is the logarithmic rate of a change of an electromagnetic wave that varies with time in a sinusoidal fashion. The real part α is termed the *attenuation constant* while β is the *phase constant*. The dielectric properties: relative permittivity ε and the electrical conductivity σ of any medium are used to calculate the attenuation constant of that medium. Therefore, in order to accurately represent the attenuation of waves inside the body, a database of human tissue dielectric properties is needed to reference these constants for a range of useful frequencies. Such a database is kept in the online records of the FCC and could found in [76]. Other studies further illustrated the significance of the tissue dielectric properties in modeling the path loss in the human body. In [77] it was determined that the near-filed loss is a significant component of the total body loss. However, both the near-field and far-field loss components depend on the body propagation constant, and hence, on the tissue dielectric properties. The body path loss equation may then be modified to reflect the near-field and far-field loss components instead of the propagation constant.

Accurately modeling the propagation through the human body and its path loss is a challenging task. In addition to the in-body attenuation and distortion effects discussed here, other noise sources must also be considered when

designing a complete system such as, reflection losses in a crowded environment, noise at the receiver, thermal noise, and the effect of incorporating error control codes into the operation of the transceiver. On this note, we refer the reader to Chapter 5 for a more detailed discussion of power requirements of implantable medical devices with ECC solutions.

- **Reliability of Transmission:** Ensuring the reliability of wirelessly transmitted data is an important design aspect of medical implants. As discussed above, the implant's RF transmission environment features multiple attenuation, noise, and interference sources. It is a reasonable design guideline to assume that a raw wireless data transmission for a typical implant may not achieve a better BER than 1×10^{-3} errors/bit [60, 67]. Therefore, in order to accommodate critical medical applications, a form of error correction mechanism is required which would improve the implant's BER performance. Automatic repeat request (ARQ) codes are an option, however in poor wireless channel conditions they result in slowing the data transfer rate due to multiple packet re-transmissions. Forward Error Control codes are another suitable option that does not require re-transmission while providing strong performance. Therefore, a data rate (and power consumption) tradeoff exists between packet re-transmission and additional coding capability [60].

The design of suitable FEC codes for wireless medical implants involves tradeoffs of its own. Adding a sophisticated encoder/decoder system to the RF transceiver increases the total power needed to transmit a data packet while enhancing the transmission reliability. Therefore, a tradeoff exists between the

additional power consumption from extra bit overhead, and the reliability gains acquired from coding. Typically, a small increase, for example 30%, in coding bit overhead may result in an order of magnitude improvement in the BER, for example from 1×10^{-3} to 1×10^{-4} errors/bit. Such an improvement could mean, depending on the data rate used, years added to the average time between errors. These overhead and power tradeoffs will be studied further in Chapter 5. The ZL70101 integrated circuit produced by Zarlink Inc. [86] is an ultra low power transceiver chip that utilizes the *Reed-Solomon* error control codes for reliable transmission. Reed-Solomon codes are especially good for combating burst errors and interference while maintaining low system overhead and achieving high data rates [60].

Interference is another adversity source affecting transmission reliability. Immunity against interference from surrounding RF signals as well as other possible in/on body medical devices is an essential design target. When considering the analog components of RF transmission, immunity places several constraints on filters, low-noise amplifiers (LNA), and synthesizers. These constraints may directly conflict with the low-power design targets of implants requiring skilled RF design expertise.

Finally, depending on the application of the implant and the associated data traffic, different levels of QoS may be required. For instance, QoS requirements are different depending on the data rate used, and whether the transmission is real-time versus best-effort. The type of data transmitted also plays a role being general versus emergency data [70]. Therefore a QoS

mechanism is a necessary ingredient of the overall reliability of wireless IMDs.

- **Safety:** Patient safety is a top design priority for medical implants. One of the most important concerns regarding safety is the biocompatibility of implant materials. These materials must be nontoxic and must not react with body liquids and tissues [67]. Titanium is an example of such material where it forms a thin layer of passive titanium oxide on its surface that prevents it from reacting with any body fluid. Passive coatings may also be applied for further safety. While very low resistivity metals, such as copper and silver, may be preferred for better RF performance, biocompatibility limits the available choices of metals. Good choices include platinum or platinum iridium [67].

  Other safety aspects, as mentioned in previous design challenges, include the generated heat from RF radiation and absorption which must be addressed by implant designers. Interference with other implants or external RF sources may severely degrade the transmission quality and is a definite safety issue.

  A final note about safety is the fact that wireless transmissions are non-contained, which makes it possible for non-authorized persons to access private patient data. Therefore a form of data protection, such as encryption, must be applied in order to preserve patient privacy and enhance the overall safety of wireless medical implants.

- **Regulatory Issues:** In addition to the above technical design challenges, wireless implant developers must deal with the complicated process of regulatory certification. Implant designs have to pass through multiple testing

and approval steps. Various countries adopt different, and sometimes conflicting, standards. Therefore, wireless implant developers and designers must be aware of all applicable standards and requirements in order to build efficient and useful devices.

2.4.3 Current In-body Communication Solutions

Research into wireless solutions for medical applications has been rapidly increasing. Current state-of-the-art wireless systems for medical applications utilize various wireless technologies and architectures. Low data rate applications implement inductive as well as battery powered implants. High data rate applications resort to various wireless technologies to implement the wireless link. *Ultra Wide Band transceivers* show a great potential in the medical implant area due to their low radiated power, high bandwidth, and security capabilities. Off-the-shelf transceivers may also be used in medical applications. Bluetooth-based medical telemetry systems represent examples of this approach which is known as *telemedicine*. Data gathered from such devices can be transferred to PDAs and personal computers for continuous monitoring. Another area influencing wireless medical communications is *micro-electromechanical systems* (MEMS) which have many applications ranging from building medical sensors to ultra low power RF transceivers. The authors of [79] present a more detailed description of each of the areas mentioned here and discuss the recent advances and future trends in low power wireless medical systems.

Our focus in this work is on traditional narrow-band wireless systems. On this front several implementations have been presented in literature such as in [80, 81]. In addition a few MICS suitable transceivers are commercially available such as [78, 82]. The

transceiver presented in [78] is an ASIC RF device that provides half-duplex communication with data rates of up to 800 kbps, and operates in the 402-405 MICS spectrum as well as in the 433-434 MHz ISM spectrum. This device employs the Reed-Solomon error control codes achieving a BER of less than 1.5×10^{-10} errors/bit, and uses less than 11 mW of power while in operation mode. Measuring at 7x7x0.85 millimeters and introduced in 2007, the ZL70101 device represents an excellent example of the current state-of-the-art wireless transceivers for medical implants.

The wireless medical device revolution continues, promising to provide innovative solutions that enhance patient treatments and the overall health care experience.

2.5 Summary

In this chapter, we presented a brief overview of the concept of information channels and the effects of noise on the integrity of the data being transmitted. The pioneering work of Claude Shannon in the areas of channel capacity and channel coding established the foundation for the field of information theory. Several channel coding schemes have aimed to enhance the reliability of data transfer and utilize as much channel capacity as possible. Turbo and LDPC codes emerged as coding candidates that provide strong performance with relatively efficient hardware implementations.

We introduced LDPC codes and briefly discussed their structure. The relationship between the parity-check matrix and the Tanner graph was explored illustrating its effect on encoding and decoding these codes. The Belief Propagation algorithm was presented as the standard LDPC decoding algorithm along with several published hardware implementations. On the encoding side, the apparent complexity of LDPC encoding was

discussed showing that direct encoding results in quadratic time complexity. Several solutions have been proposed in the literature. The encoding algorithm by Richardson and Urbanke [34] combined with the block-circulant code construction approach represents an efficient encoding technique that is able to achieve linear time complexity. Several software and hardware encoding implementations were presented.

A brief overview of wireless communications was presented. In addition, the IEEE 802.16 standard was introduced as the main platform for our encoding project. The performance of LDPC codes in wireless systems and environments was investigated showing the superior performance and suitability of these codes for emerging wireless standards.

The last section of this chapter focused on the modern trend of integrating wireless solutions into implantable medical devices. The MICS rules governing the operation and use of such devices were listed. We highlighted the main design challenges facing IMD designers including small-size, low power, low cost, and in-body propagation challenges. Despite the difficulty of specifying a definitive wave propagation model for in-body RF communications, studies have shown that path loss approximations are possible. The need for channel coding in order to improve the reliability of the wireless link was discussed. Finally safety, privacy, and regulatory considerations were mentioned. Today's commercial wireless IMDs are advanced ASIC implementations featuring implemented solutions for many of the design challenges and targets. The area of low power wireless IMDs is promising with potential and encouraging for further research activities.

Chapter 3

The 802.16e compatible LDPC Encoder

3.1 Introduction

The low-complexity decoding of LDPC codes is one of their main advantages over other channel coding schemes, and it has received much research focus in both the theory and implementation fields. The encoding of LDPC codes is typically less researched and has received criticism because of its apparent high computational complexity. In Chapter 2, we discussed in details the problems facing LDPC encoder developers including the $O(n^2)$ nature of direct encoding, buffering and latency issues, and the rate of output bit production. It was also noted that several approaches have been introduced that tackled these problems and offered various tradeoffs in terms of computational complexity and performance [30, 31, 32, 34, 35, 36]. These challenges, combined with the expanding importance of WiMAX in the communications field, provide the foundation of our interest in implementing an 802.16e LDPC encoder system. In this chapter, we discuss the LDPC block code structure adopted in the standard, then introduce our encoder architecture which takes advantage of the code properties and leverages various design ideas from the literature, to arrive at a flexible and compact encoder design.

3.2 802.16e LDPC Code Structure

The 802.16e-2005 (Mobile WiMAX) standard [10] adopts several channel coding schemes including convolutional coding, block turbo coding, convolutional turbo coding,

and low-density parity-check (LDPC) coding. The LDPC code specified in the 802.16e standard is based on a set of fundamental *systematic linear block codes*. These *irregular* LDPC codes are designed to accommodate several code rates and code lengths.

Each supported LDPC code is defined by a parity-check matrix H of size m × n where n is the length of the code (output packet length) and m is the number of the code bits (parity-check bits) in the codeword. Therefore, the number of input bits (systematic bits) required to generate the output codeword is k = n - m bits.

For each code of a certain rate and length, its matrix H can be obtained from a compact base model matrix H_{bm} that takes the structure:

$$H_{bm} = \begin{bmatrix} P_{(0,0)} & P_{(0,1)} & P_{(0,2)} & \cdots & P_{(0,22)} & P_{(0,23)} \\ P_{(1,0)} & P_{(1,1)} & P_{(1,2)} & \cdots & P_{(1,22)} & P_{(1,23)} \\ P_{(2,0)} & P_{(2,1)} & P_{(2,2)} & \cdots & P_{(2,22)} & P_{(2,23)} \\ \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\ P_{(m_b-1,0)} & P_{(m_b-1,1)} & P_{(m_b-1,2)} & \cdots & P_{(m_b-1,22)} & P_{(m_b-1,23)} \end{bmatrix} \quad (3.1)$$

Here P(i,j) represents one of a set of z × z permutation matrices or a z × z zero matrix. This means that H_{bm} is a compact representation of the code's binary parity-check matrix H. Alternatively, we may say that H is comprised of smaller square-matrix *blocks* that belong to the set of P(i,j) matrices. Therefore z is defined as the *block size*, and H_{bm} has a total size of $m_b \times 24$, where $m = m_b \times z$ and $n = 24 \times z$. It could be concluded that H can be directly expanded from the base model matrix H_{bm} via replacing each of H_{bm}'s entries with the appropriate permutation or zero matrix.

The permutations used are *circular right shifts*, and the set of permutation matrices contains the z × z identity matrix and circular right shifted versions of it. Each permutation matrix can be represented by a single circular right shift value. This value represents the size of the shift. Therefore a non-negative entry ($P(i,j) \geq 0$) at (i,j)

represents a circularly right shifted identity matrix by a value equal to P(i,j). Similarly a zero entry represents a non-shifted identity matrix. To distinguish them from permutation matrices, the zero matrices are represented by the value -1.

For each separate code rate, a base model matrix H_{bm} is defined for the largest supported block size of 96 bits (which gives a maximum codeword size of n = 2304 bits). This approach adds another compaction factor to the matrix representation used in the standard, where all code lengths associated with one code rate are represented by a single base model matrix. In order to expand the base model matrix to achieve the appropriate code length, a special *scaling* technique is utilized. The idea is to adjust the value of the block size z which changes the dimensions of the z × z blocks, hence changing the total size of H and achieving the desired code length. Thus, before expanding H_{bm}, its values must be scaled within the supported range of block sizes. The equations used to perform this scaling are provided in [10] and outlined below.

For equations 3.2 and 3.3, let the scaled shift size used at entry (i,j) be P(f,i,j), where f is the index of the block sizes, f = 0, 1, 2,..., 18. In addition, the selected block size is denoted by z_f (also known as the *expansion factor*), and the maximum block size of 96 is denoted by z_0.

For code rates 1/2, 3/4A, 3/4B, 2/3B, and 5/6, the shift sizes, P(f,i,j), for a packet size corresponding to block size z_f are derived by scaling P(i,j) proportionally.

$$P(f,i,j) = \begin{cases} P(i,j) & ,P(i,j) \le 0 \\ \left\lfloor \dfrac{P(i,j) z_f}{z_0} \right\rfloor & ,P(i,j) > 0 \end{cases} \tag{3.2}$$

Where $\lfloor x \rfloor$ denotes the floor function. However, for the special case of code rate 2/3A, the shift sizes, P(f,i,j), for a code size corresponding to block size z_f are derived by scaling P(i,j) using the modulo function.

$$P(f,i,j) = \begin{cases} P(i,j) & ,P(i,j) \leq 0 \\ \mathrm{mod}(P(i,j), z_f) & ,P(i,j) > 0 \end{cases} \tag{3.3}$$

Thus, by selecting the code rate and the block size, the base model matrices are scaled to meet the coding requirements producing the appropriate H matrices. This matrix structure provides flexibility in handling the six different LDPC code rates supported in the standard, as well as the nineteen different block lengths for each code rate. All supported rates and lengths, along with input/output packet lengths and block sizes are presented in Table 3.1.

Output Packet Size, n(bits)	Output Packet Size, n(bytes)	Block Size (bits)	Input Packet Size, k(bytes)			
			Rate = 1/2	Rate = 2/3 (A/B)	Rate = 3/4 (A/B)	Rate = 5/6
576	72	24	36	48	54	60
672	84	28	42	56	63	70
768	96	32	48	64	72	80
864	108	36	54	72	81	90
960	120	40	60	80	90	100
1056	132	44	66	88	99	110
1152	144	48	72	96	108	120
1248	156	52	78	104	117	130
1344	168	56	84	112	126	140
1440	180	60	90	120	135	150
1536	192	64	96	128	144	160
1632	204	68	102	136	153	170
1728	216	72	108	144	162	180
1824	228	76	114	152	171	190
1920	240	80	120	160	180	200
2016	252	84	126	168	189	210
2112	264	88	132	176	198	220
2208	276	92	138	184	207	230
2304	288	96	144	192	216	240

Table 3.1: 802.16e supported LDPC code rates and lengths

The 802.16e LDPC code structure is designed to facilitate the implementation of efficient encoding/decoding architectures. This is evident through the properties of the selected parity-check matrices. All supported matrices are in the *almost lower triangular* form which, as discussed in Chapter 2, enables efficient encoding architectures. Another structure of these matrices is the use of *circulant matrices* to construct *block-circulant LDPC codes*. This property, as we demonstrate in section 2.2.2, can improve the computational efficiency of the encoding and decoding processes. These properties and their effects will be explored further in the following sections as we discuss our encoder's architecture and implementation.

3.3 802.16e LDPC Code Performance

The LDPC codes used in the 802.16e standard are capable of providing high coding performance. As mentioned in section 2.3.3, the simulation study performed at the University of Ulm, Germany in conjunction with Siemens Corp. [59] compared the code performance of the most advanced forward error control (FEC) schemes supported by the standard, namely convolutional codes (CC), convolutional turbo codes (CTC) and LDPC codes. The results show that when compared with the traditional convolutional codes, LDPC codes show significant coding gains and achieve similar performance to the turbo codes. Figure 3.1 illustrates these BER performance results for these FEC codes over an AWGN channel using a code rate of 1/2 and length of 576 bits.

When simulated over the TU30 channel, modeling a mobile user in a typical urban environment moving with a relative velocity of 30 km/h while transmitting on a carrier frequency of 3.5 GHz, the 802.16e LDPC codes continued to show their superiority to the CC codes as recorded in Figure 3.2.

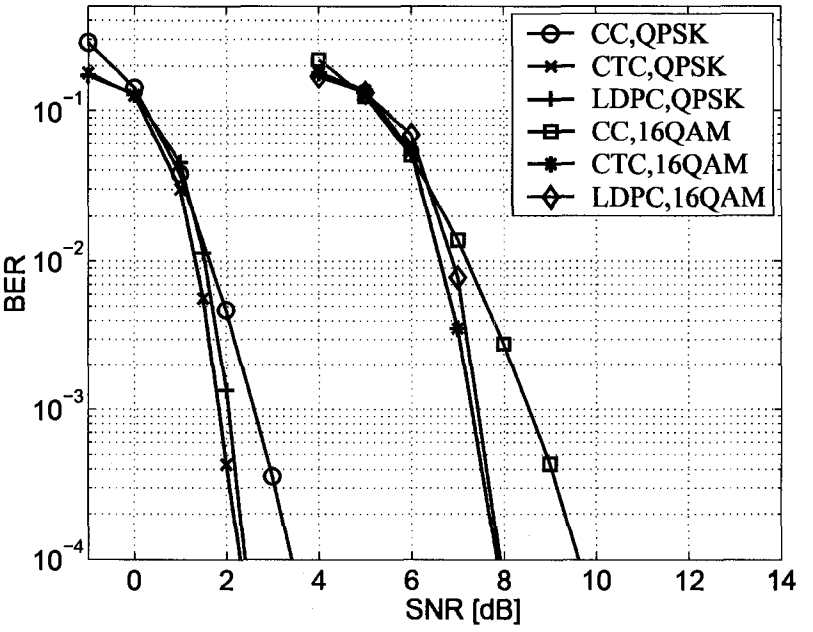


Figure 3.1: FEC codes comparison, BER, AWGN, R=1/2, N=576, figure taken from [59]

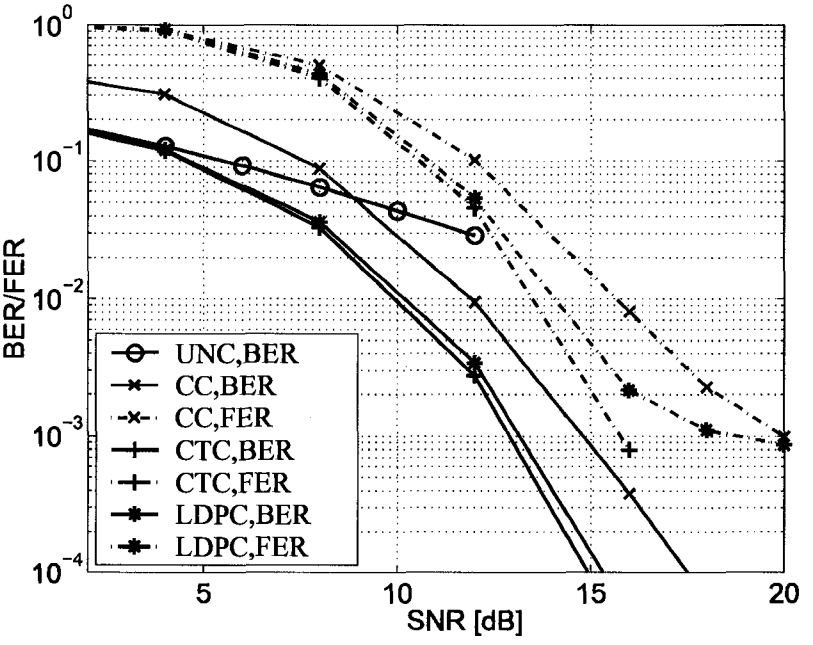


Figure 3.2: FEC codes comparison, BER/FER, TU30, R=3/4, N=576, figure taken from [59]

In Figure 3.3 the simulation results for the TU30 channel model are shown for the LDPC code with rate 1/2 and multiple code lengths. Multiple code rates and lengths for the CC coding scheme are shown.

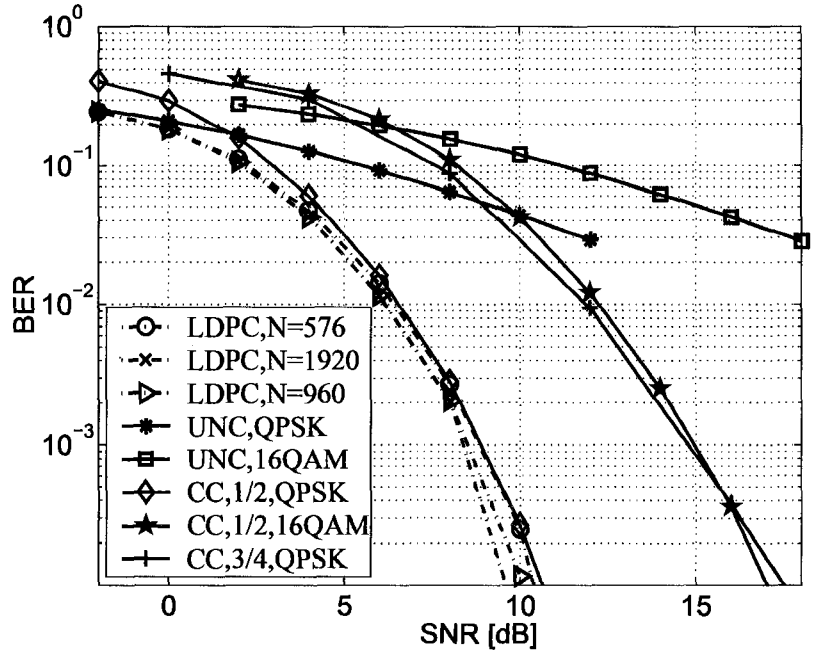


Figure 3.3: FEC codes comparison, BER, TU30, various code rates and lengths, figure taken from [59]

The performance of codes depends on the implementation of the encoder/decoder system, and while software simulations provide a good estimate for the expected performance of codes, a hardware implementation usually reflects a more accurate picture. In [83] the first published implementation for an 802.16e compatible LDPC decoder core included a code performance study of that particular decoder. Figure 3.4 depicts the FER results of communication over an AWGN channel for all supported LDPC code classes. This implementation of the decoder takes advantage of available code properties (such as the layered decoding approach available for code rates 1/2 and 3/4) to improve decoder throughput. Code length n = 2304 bits was used for all simulations. These results correlate well with the simulation results discussed earlier.

LDPC codes proved to be a strong choice for the WiMAX standard and stand to gain increasing popularity in emerging wireless standards.

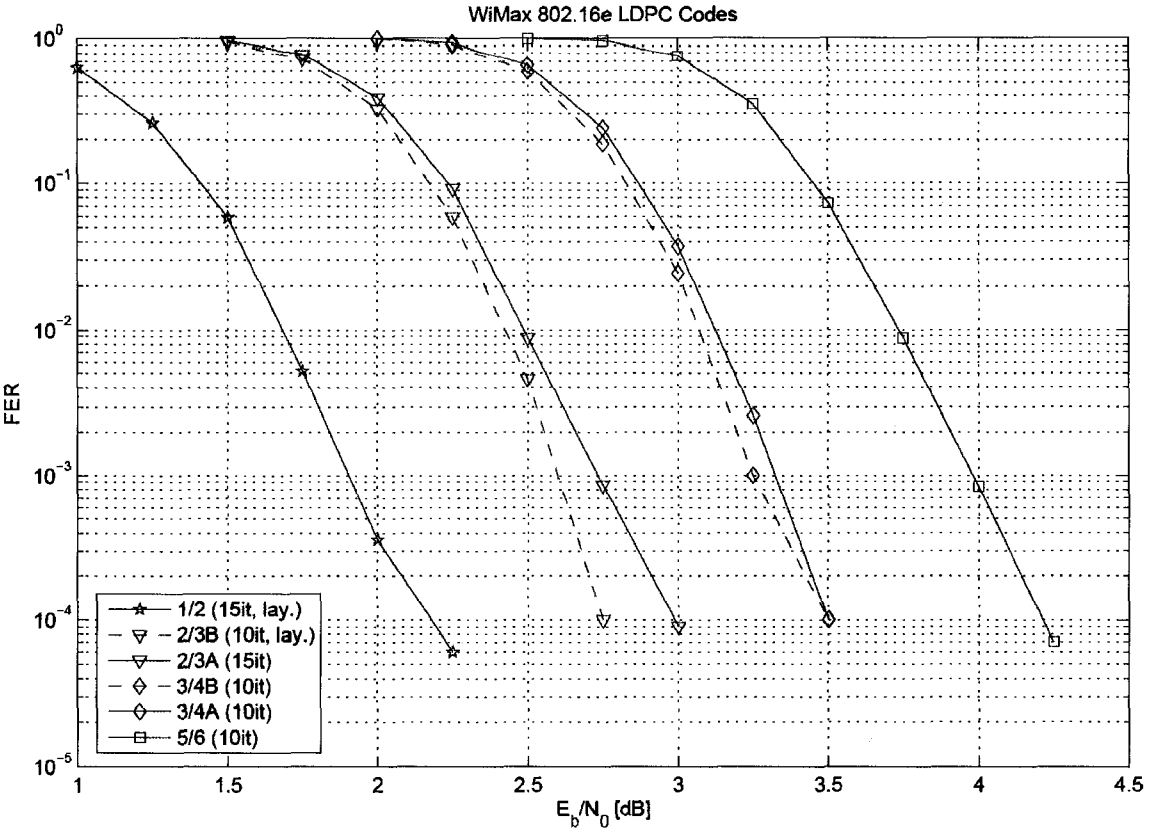


Figure 3.4: FER performance of WiMAX LDPC decoder implementation from [83]

3.4 The 802.16e LDPC Encoder Core

3.4.1 Encoding Algorithm

As shown in Chapter 2, encoding an information block s of length k bits may be done using the code's generator matrix G such that $G \times H^T = 0$, and a codeword x could be generated via the multiplication $x^T = s^T \times G$. However, since G is a dense matrix for LDPC codes, this direct encoding is quite computationally heavy and is quadratic in time. Therefore, efficient encoding methods were researched and proposed in literature. In addition, WiMAX's support for multiple code rates and lengths adds a further significant degree of difficulty to the implementation problem.

The 802.16e standard recommends two methods for *directly* encoding an input packet using the code's parity-check matrix H. Method 2 from Annex G of the standard [10] is based on the efficient encoding algorithm originally proposed by Richardson and Urbanke in [34], which aims to achieve linear encoding complexity. We present a brief summary of this algorithm next.

The algorithm's idea is to preprocess the code's parity-check matrix H into the ALT form using only row and column permutations preserving its sparseness and randomness properties. Fortunately, the standard supplies all six base model (H_{bm}) matrices in ALT form removing the need for this preprocessing stage. When in ALT form, the upper right corner of a matrix is populated by 0s and is some distance from *true* lower triangular form as shown in Figure 3.5. This distance is termed the gap (g) and in this case is set equal to the code's block size z.

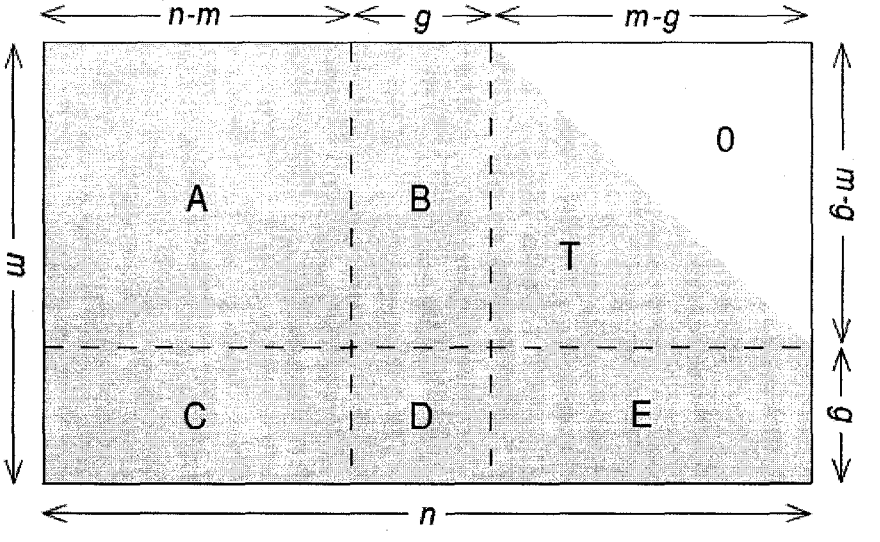


Figure 3.5: The parity-check matrix H in almost lower triangular form

The H matrix has the form

$$H = \begin{bmatrix} A & B & T \\ C & D & E \end{bmatrix} \tag{3.4}$$

Here A is an (m-z) × (n-m) sub-matrix, B is (m-z) × z, T is (m-z) × (m-z), C is z × (n-m), D is z × z, and E is z ×(m-z). The output codeword is defined to have the structure x = [s p1 p2], where s is the input packet and p1 and p2 are the added parity code bits. Given the specific structure of the provided H matrices and after a series of mathematical manipulations (see [34]), we arrive at Equations 3.5 and 3.6

$$p_1^T = \left(ET^{-1}A + C\right)s^T \tag{3.5}$$

$$p_2^T = T^{-1}\left(As^T + Bp_1^T\right) \tag{3.6}$$

Thus, the encoding procedure may be summarized using the following steps:

Step 1) Compute As^T and Cs^T

Step 2) Compute $ET^{-1}(As^T)$

Step 3) Compute p_1^T using Equation 3.7 which directly results from Equation 3.5

$$p_1^T = ET^{-1}\left(As^T\right) + Cs^T \tag{3.7}$$

Step 4) Compute p_2^T using Equation 3.8 which directly results from Equation 3.6

$$Tp_2^T = As^T + Bp_1^T \tag{3.8}$$

The algorithm steps can be graphically illustrated with the block diagram shown in Figure 3.6 (from Annex G of the 802.16e standard).

We choose this algorithm for our design of the 802.16e compatible encoder. We aim at taking advantage of its efficient computational complexity and suitability for hardware implementations.

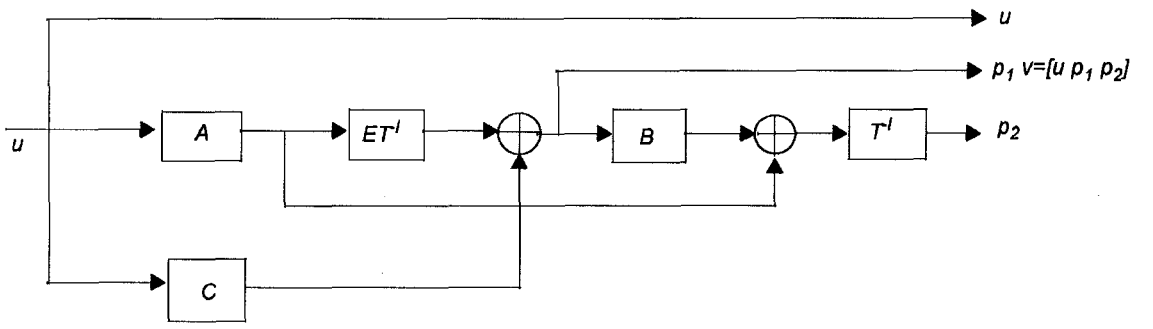


Figure 3.6: Block diagram of the encoding algorithm for the block LDPC code, figure taken from [10]

3.4.2 Encoder Design and Implementation Flow

The process we applied to the encoder design and implementation project can be summarized in Figure 3.7. As illustrated in the figure, the overall process can be divided into three partly overlapping stages.

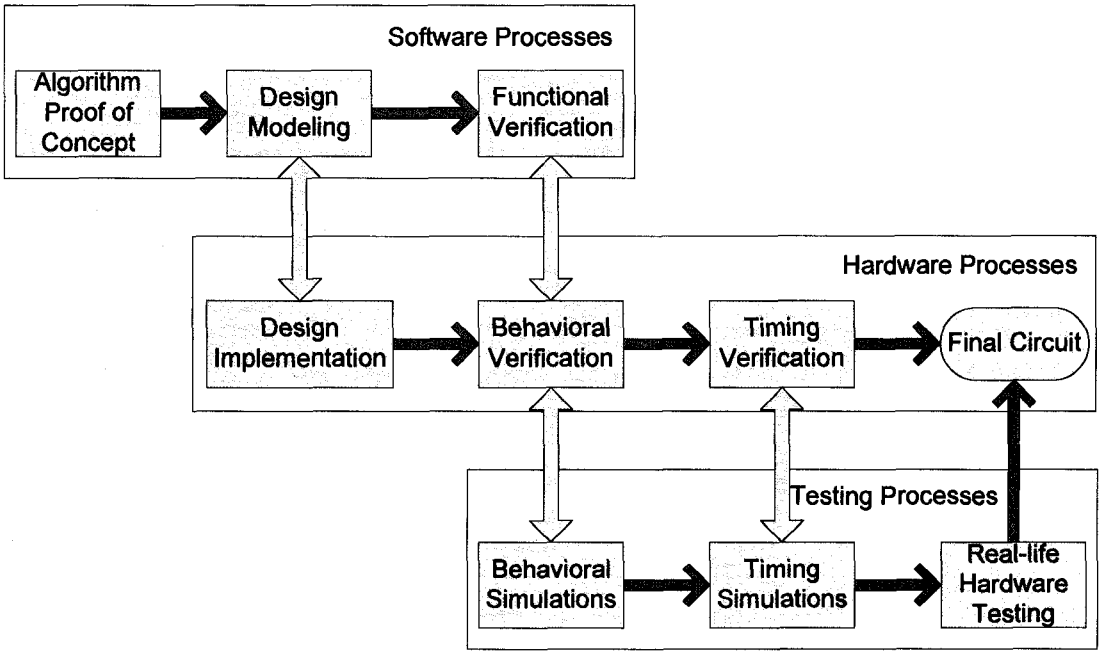


Figure 3.7: Encoder design flow chart

Software Processes: This is the first stage where a direct software implementation of the LDPC encoder is completed to serve as an algorithm *proof of concept*, as well as a

reference model for the final design. Following that, modeling of the hardware design commences. Through this, competing design ideas are explored in software. As shown in the figure, the modeling process overlaps in time and collaborates with the hardware implementation process. After arriving at the final design, the software model is complete, and is ready to serve as a comparative testing model for the hardware's functionality. The functional verification process serves both the behavioral and real-life hardware testing processes.

Hardware Processes: During this stage, initial hardware models are implemented and analyzed against their software counterparts. Design tradeoffs such as area, resource usage, and performance are considered in order to select the most appropriate design idea. After the final design is selected and implemented, behavioral verification starts for the individual encoder components then the full system. Finally, the encoder system is simulated with real-life timing data and test vectors to verify its close-to-completion behavior. Both behavioral and timing verifications are performed in conjunction with the corresponding simulation processes from the testing stage. The tests are performed, and the results are fed back into the hardware stage to modify and improve the implementation.

Testing Processes: During the first two processes of this stage, the general and the timed behavior of the implemented hardware circuit are tested. To accomplish this, *software testbenches* are developed, and feed the software-prepared test vectors to the hardware circuits. The results are then captured and studied. After satisfying the functionality and performance requirements, the design is finalized and the hardware circuit is generated. The final process involves moving the circuit *on-chip* to perform the

real-life hardware tests that confirm the functionality of the finished system. At various points in the design flow, verification results may require moving back to an earlier process to modify the design and/or develop a new design idea. This looping procedure is not shown on the graph, but it is an essential element of the project's development.

The methodology and results of each of the above three stages will be explored in more details in later sections of Chapters 3 and 4.

3.4.3 Encoder Top-Level Design

Combining ideas presented in the code structure and encoding algorithm sections, we developed a top-level design for the encoder system, illustrated below in Figure 3.8.

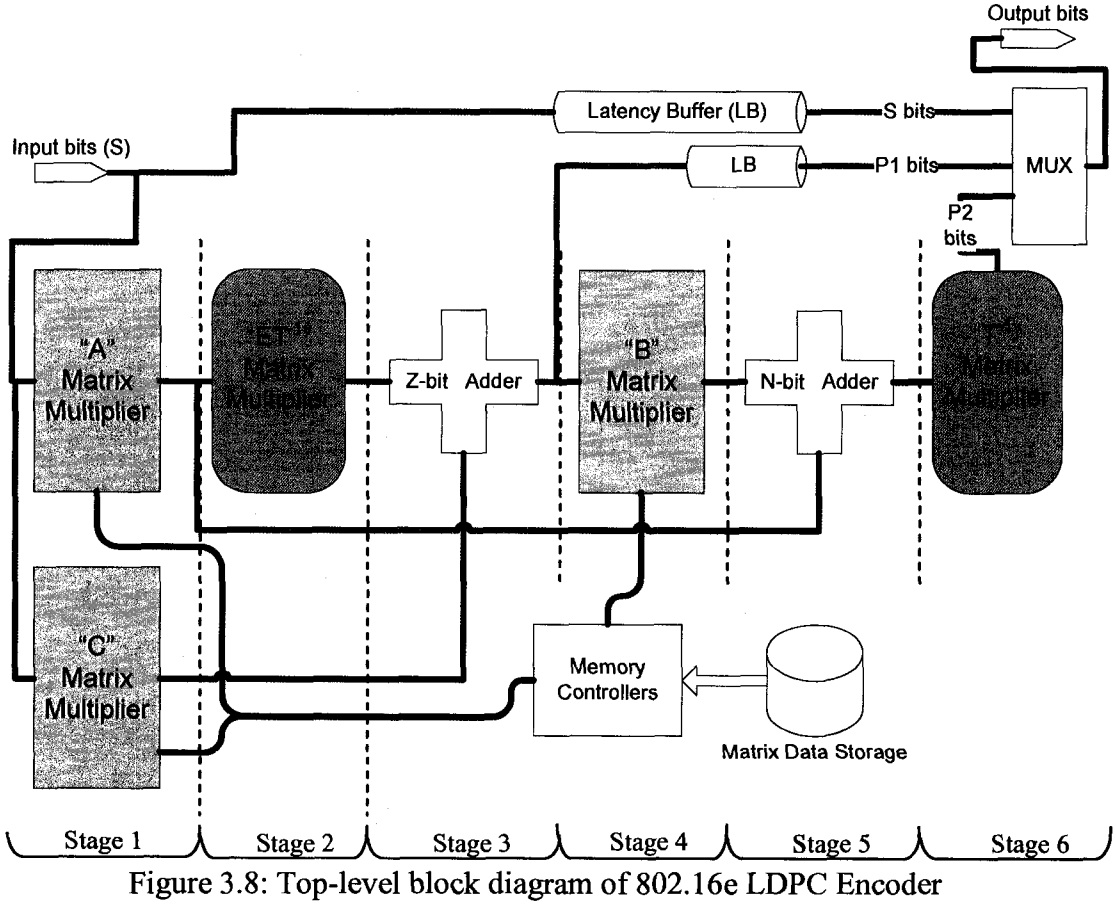


Figure 3.8: Top-level block diagram of 802.16e LDPC Encoder

This design serves as a plan to divide the overall problem into smaller internal components. The idea is to focus the design efforts on building blocks which can be replicated and assembled quickly to construct the full encoder system. The plan also aids in identifying the needed tools to build and test the system, and the algorithm and/or code structure features that may be leveraged to produce an efficient final design.

The first step is to identify the target design requirements. Our aim is to develop and implement an FPGA-based encoder system that functions as an IP *core.* The term IP core refers to a hardware module designed to be integrated as a component in a larger digital design. IP Cores are prime examples of *design reuse* and hence, must be portable and able to be inserted into different technologies and systems. We design our encoder with the ability to digitally interface with other components of a transmitter system. Studying the encoder's specifications as described in the standard's text [10], we identified the following essential encoder requirements. The encoder:

- Accepts k-bit input packets and encodes them into n-bit LDPC output packets (codewords).
- Supports the six LDPC codes, namely 1/2, 3/4A, 3/4B, 2/3A, 2/3B, and 5/6, as well as all code lengths from 576 - 2304 bits for each code.
- Provides the ability to switch code rates and lengths during operation, albeit only between packets.
- Provides low and consistent latency in order to prove useful for communication applications.
- Embodies a low power and small area implementation, initially targeting the Xilinx Spartan-3 FPGA chip series [84].

As depicted in Figure 3.8, the top-level design diagram resembles the structure of the encoding algorithm presented in Section 3.4.1. The main components in the design are matrix multipliers, vector adders, memory storage and controllers, and an overseeing master finite state machine (FSM).

The vector adders are relatively simple components. In software they are binary additions, while in hardware they are arrays of exclusive-OR (XOR) gates. The matrix multipliers are, however, the central components in the design and can be further divided into two classes. The A, B, and C matrix multipliers are the first class. These multipliers process input bit streams and generate product bit streams while receiving the matrix contents from the memory controllers. The second class is the ET^{-1} and T^{-1} matrix multipliers where *no matrix content is required.* Upon studying the code structure provided in the standard, we learned that the ET^{-1} and T^{-1} matrices always take the same non-random structure. This allowed for the design of these multipliers to simulate the content of these matrices using shift registers and *without the need to actually store them.* Design reuse is applied in the development of the multipliers since they feature internal building blocks that can be replicated to assemble the full multiplier.

The contents of the A, B, and C matrices are stored in memory storage components, while the memory controller components handle the data transfer between the storage and the multipliers. Despite them showing in the figure as a single *store-controller pair*, the design actually includes a separate pair for each matrix. The internal structure of the storage and the controllers is another example of design reuse, where one matrix store-controller pair is designed then replicated with minor size modifications to handle the other matrices.

The master FSM, has several responsibilities. The first of which is to handle the data transfer between the various stages as they input and output their bit streams. Other FSM tasks include the buffering of input and parity bits until the output multiplexing process is ready for them. Finally, the master FSM handles the selection of the appropriate code rate and length through the negotiation of user-selected encoder parameters.

The most notable design ingredient is the close integration of algorithm and code features into the design in order to reduce circuit complexity and area, and to improve performance. As discussed above, the structure of the ET^{-1} and T^{-1} matrices assisted in reducing the hardware complexity and memory usage for two main multiplier units. However, the code format and structure goes beyond that and provides the tools necessary to further reduce the memory space needed to store the rest of the matrices. This idea will be discussed in further details in Section 3.4.4.

The algorithm structure is also utilized to achieve several hardware complexity gains. As could be seen from the steps presented in Section 3.4.1, the multiplications of both the A and the C matrices are done first since their products are needed in later stages of the algorithm. However, both of these multiplications can be performed at the same time providing a significant gain of hardware *parallelism*. All of the following stages happen in sequence in order to achieve the correct encoding. However, multiplier units that are finished in earlier stages can be utilized again to perform later multiplications. Specifically, the A matrix multiplier hardware can be reused for the B matrix multiplication stage, and being the largest hardware unit, this provides excellent device utilization and area reduction. The same concept is applied to the ET^{-1} and T^{-1} multiplier

units. Therefore, even though they show up as separate design entities on Figure 3.8, these multiplier entities are in fact reutilized at different stages in the arithmetic pipeline.

3.4.4 Software Implementation and Modeling

The first stage of the project's development is software modeling. During this stage, we completed the initial encoder design and explored its performance. The software model also assists in the development and initial testing of the hardware modules. At the end of the project, software was the platform for the comprehensive testing tasks providing the simulation environment and testing vectors. The tools used at this stage include MATLAB, Microsoft Excel, and various word processors.

The software modeling stage is best explained through the structure and the description of the various software modules produced for the project. The actual code of these modules is available in Appendix A.

In Figure 3.9 below, the hierarchy of the software model of the encoder is presented. The *encoder.m* module implements the encoding algorithm of Section 3.4.1 in a straight forward fashion.

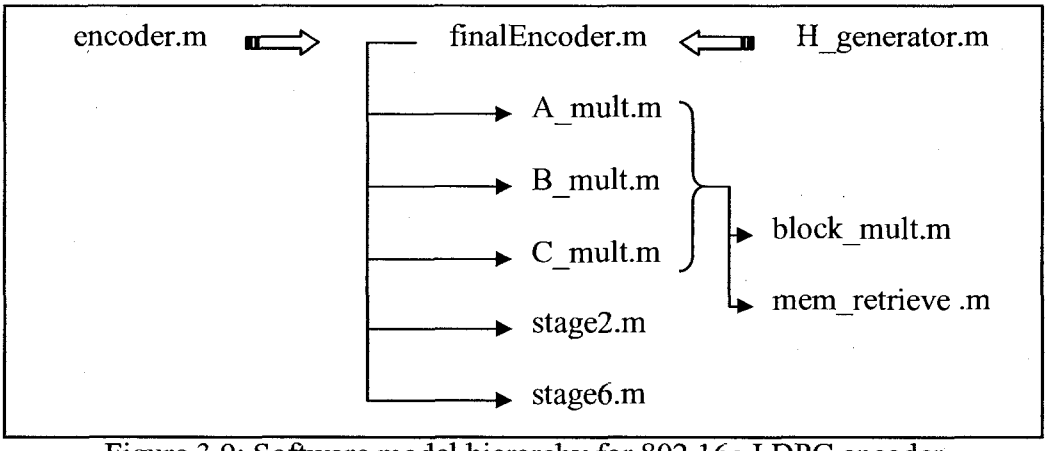


Figure 3.9: Software model hierarchy for 802.16e LDPC encoder

This module serves as a proof of concept and assists in better understanding the algorithm. Furthermore, the *encoder.m* module is integrated into the *finalEncoder.m* module to become a reference point for the final design's performance.

The *finalEncoder.m* module implements the master FSM, and controls all the components necessary to carry on the encoding process. The *stage2.m* and *stage6.m* modules are software implementations of the ET^{-1} and T^{-1} matrix multiplier units, respectively. The rest of the matrix multiplier units are implemented in the *A_mult.m, B_mult.m,* and *C_mult.m* modules. All three of these modules have a similar internal structure utilizing one or more instances of a block multiplier unit along with a memory interface function. The block multiplier, implemented in *block_mult.m*, is one of the central modules in our project. The *mem_retrieve.m* module handles the retrieval of memory contents used in the multiplication process. The contents of all the required matrices are generated using the *H_generator.m* utility. This utility loops through all the supported combinations of code rates and lengths expanding their model H_{bm} matrices into the full binary H matrices and storing them for later access by *finalEncoder.m*. And last but not least, *finalEncoder.m* performs the various vector additions by binary XOR-ing the appropriate vectors.

In addition to controlling the data flow of the algorithm, the software encoder is able to randomly generate input packets for encoding. It also implements several performance statistics measures such as error counters and a pseudo-clock counter for latency estimates. Finally, the encoder performs a simple test at the end to verify the integrity of the produced codewords. This flexible MATLAB implementation provided us with good design feedback and assisted in speeding up the hardware design process.

Testing and functional verification is another main category where software modeling was utilized. However, this discussion is postponed till Chapter 4 where the full software/hardware verification process is described along with its results.

Overall, software proved to be a powerful tool for the encoder's modeling, testing, and verification tasks. But perhaps one of the most significant software contributions was realized through the design and preparation of the encoder's memory storage.

As discussed in the top-level design section, each of the A, B, C matrix multiplier units requires access to the contents of these matrices. Since the target encoder supports all code rates and lengths, a straight forward storage of the expanded binary A, B, C matrices would require 19 versions of each matrix for each of the 6 code rates. This brings the total required storage space for the A, B, C matrices to approximately 55566.1 Kb or 54.3 Mb, which is 30 times larger than the maximum available *block RAM* space of 1872 Kb [85] on the largest chip in the Spartan-3 series. Therefore, it was clear that a different storage approach is needed.

One possible approach is to only store the matrices in their model base numerical format. This is possible since all of the information required to perform a *single block multiplication* is contained in two elements, the input from the user and the shift value from the base model matrix (or subset thereof). The details of the multiplication will be discussed further in Sub-section 3.4.5.1. Using this approach, there are two possible implementation directions. The first direction is to *pre-process the base model matrices* provided by the standard, *scaling them* to support all 19 possible code lengths. The other direction is to *perform the scaling/expansion on-chip.* Obviously, the first direction requires more memory storage than the second. Therefore a tradeoff exists between the

simplicity of the memory controller's circuit and the required memory storage. The following table presents the memory requirements for all three ideas discussed here.

	Fully expanded Binary A,B,C Matrices	Scaled Base Model A,B,C Matrices	Non-scaled Base Model A,B,C Matrices
Bits per Matrix Entry	$z_f \times z_f$	7	11
Total Number of Matrices	342	342	6
Required Memory (Kilo Bits)	55566.1	95.3	7.9

Table 3.2: Three approaches to storing the A, B, and C sub-matrices

Based on these initial calculations, and the available block memory on the Spartan-3 chips we decided to pursue the memory-conservative on-chip scaling technique. The matrix multipliers are designed to receive the properly-scaled shift values and use them to perform the multiplication with the input bits. Thus, in order to supply the multipliers with the correct shift values, the hardware memory controllers must:

- Access the appropriate matrix corresponding to the selected code rate.
- Divide the retrieved shift values by z_0 (i.e. 96).
- Multiply the result by the user-selected z_f value
- Perform the flooring function and forward the final values to the multipliers.

These are processing-intensive steps that require several clock cycles to finish. The double divide-multiply steps may exceed the time available for the encoder's matrix multiplication, which will add latency clock cycles and conflict with one of the main design targets of the encoder. In addition, a more complex memory controller requires more area, and consumes more power. Therefore, we concluded that our design needed to perform as many processing steps *off-chip* as possible reducing the circuit complexity of the on-chip controller. We determined that the division step can be performed ahead of

time in software. The rest of the steps depend on user-selected parameters and must be performed on-chip.

When performing the division in software, the result will be a fraction that needs to be represented in binary before being stored in chip memory. The resultant fraction is always less than one since all shift values are less than 96. Therefore we needed to determine the number of bits, that is the *precision,* required to represent the fractions. Adopting the *fixed-point binary format*, this number must provide enough accuracy to produce the correct results once the fractions are multiplied with z_f on-chip. We developed MATLAB scripts (*Accuracy_driver.m & verify_accuracy.m*) that evaluate the divide-multiply results with a range of precision bits. The minimum required number of bits to represent the fractions resulting in no errors was determined to be 54 bits. This number would result in approximately 56 Kb of storage, but the main problem would be handling the very wide multiplier and memory bus lines needed on-chip. Therefore, in order to reduce this number, *rounding* was introduced. This means that a pre-determined precision is selected to represent the fractions, and when the division is performed in software the result is rounded up towards the closest fraction represented by the available binary bits.

Rounding, however, presented a challenge. If the precision is chosen too large, memory storage space is wasted. If it's chosen too small, rounding errors will occur. To illustrate how rounding works, we discuss the following numerical example.

Let the matrix entry (shift value) be 41 and the selected code length is 1632 bits, which corresponds to $z_f = 68$. Further let the precision used be 8 bits. Applying Equation 3.2, the expected result after scaling is 29.

- 41 is first divided by z_0 (96). 41/96 = 0.427083333. When converted to binary using 8-bit precision the result is 01101101, which is in fact equal to 0.4258.
- The fraction 0.4258 is then multiplied by z_f. 0.4258 × 68 = 28.9544.
- Finally, performing the flooring function $\lfloor 28.9544 \rfloor$, the result is truncated to 28. This is the *wrong* result.

To correct this, a rounding factor is added to the binary representation to adjust the value. The factor is chosen to be the smallest possible fraction. Therefore:

- After rounding. 00000001 + 01101101 = 01101110, which is equal to 0.4311 in decimal representation.
- The fraction 0.4311 is then multiplied by z_f. 0.4258 × 68 = 29.3148.
- Finally, performing the flooring function $\lfloor 29.3148 \rfloor$, the result is 29. This is the *correct* result.

However, changing the code length to 672 bits which corresponds to $z_f = 28$, an *over-rounding* error occurs. Applying Equation 3.2, the expected result after scaling is 11.

- After rounding, the fraction 0.4311 is multiplied by z_f. 0.4311 × 28 = 12.0708.
- Performing the flooring function $\lfloor 12.0708 \rfloor$, the result is 12. This is the *wrong* result.

Despite the 8-bit precision being suitable for the 41/96 fraction when z_f is 68, it produces the wrong result when z_f is 28. Therefore, more precision is needed. Fortunately our problem is limited to the set of values available in the provided matrices. Thus, we developed a MATLAB script (*check_allvalues.m*) that performed a trial-and-error run through a range of precision values while looping through all possible fraction values. The precision value providing enough accuracy was found to be 11 bits. Therefore all

fraction values were converted to 11-bit fixed-point binary representation before porting to hardware. This division and conversion were performed using the *col_maker.m* modules.

Software tools aided significantly in solving the memory design challenge. In addition, we developed software scripts that verified the output of the hardware memory controllers against the output of their software counterparts. These *AB_mem_checker.m* and *C_mem_checker.m* modules can be found along with all the above discussed modules in Appendix A.

3.4.5 Hardware System Components

Before we describe the hardware structure of the system's components, we will discuss in more detail the hardware processes presented briefly in Section 3.4.2. In order to transform ideas or software designs into hardware circuits, a *hardware description language* (HDL) may be used. HDLs provide designers with the ability to describe digital components, their behavior, and their interconnections using software code. The code can then be transformed into hardware circuits using software synthesizers and routers. Finally, the design can be programmed into an FPGA device as it assumes the purpose and function of the programmed hardware design. One of the main widely adopted hardware description languages is Very-high-speed integrated circuits Hardware Description Language (VHDL). In our project, we used VHDL for our hardware implementation purposes, and the Xilinx ISE 8.1i software design suite for our synthesis, debugging, and routing tasks.

For the design of each system component, we followed the same hardware design flow, presented here in Figure 3.10.

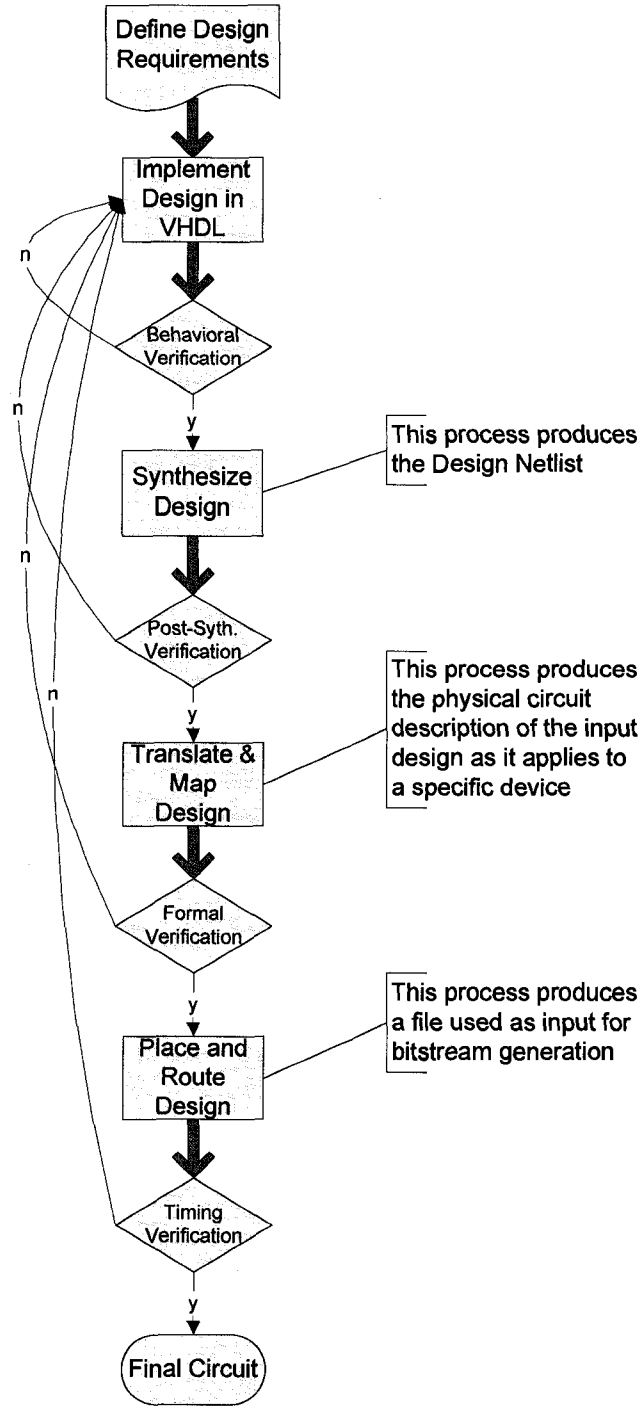


Figure 3.10: The VHDL design flow

After defining the design requirements, the hardware design flow commences with the following processes:

Implement Design in VHDL: In this process a written specification of the design is implemented in VHDL using a suitable text editing tool. A text editor is provided in ISE.

Synthesize Design: This process checks the code syntax, analyzes the design hierarchy, and produces a design *netlist* saved in an NGC file. A design netlist contains the list of parts and their connections in a circuit. Synthesis is performed using the software tool Xilinx XST which is part of the ISE software suit.

Translate & Map Design: Translate merges the design netlist and constraints into a Xilinx design file, while Map fits the produced design into the available resources on the target FPGA chip. The results are stored in a native circuit description (NCD) file.

Place and Route Design: This process places and routes the mapped design to the timing constraints, and produces a modified NCD file that is used as an input for bitstream generation.

As illustrated in Figure 3.10 above, a verification step is performed after each major design milestone. The hardware circuits are tested using VHDL *testbenches*. These test bench modules receive input test vectors and simulate the operation of the design producing output vectors and/or waveforms. The simulations are performed using the Mentor Grahics' ModelSim XE III software tool. These verification steps are similar to the full-system verifications discussed in Chapter 4, therefore a detailed description is deferred till then.

After all the individual components are implemented and tested, the encoder system is assembled. The encoder moves into the testing processes before a final hardware bitstream is generated to program the FPGA chip. The following subsections describe the structure of the encoder's internal components.

3.4.5.1 The Block Multiplier

The encoding algorithm requires that several bit vectors be multiplied by sparse matrices. These matrices, as we discussed earlier, are comprised of smaller blocks of circulant square matrices. Therefore when multiplying a bit vector by a sparse matrix, the vector can be divided temporarily into segments with a length matching the dimension of the square block matrices. *The multiplication process, in turn, can be split into a series of smaller block multiplications.* Figure 3.11 from [36] depicts a helpful example of the multiplication operation between a larger sparse matrix U and a bit vector X, the result is the bit vector Y. In the figure, 3 × 3 block matrices are chosen for illustration purposes. The minimum block size in the 802.16e standard is, however, 24 bits.

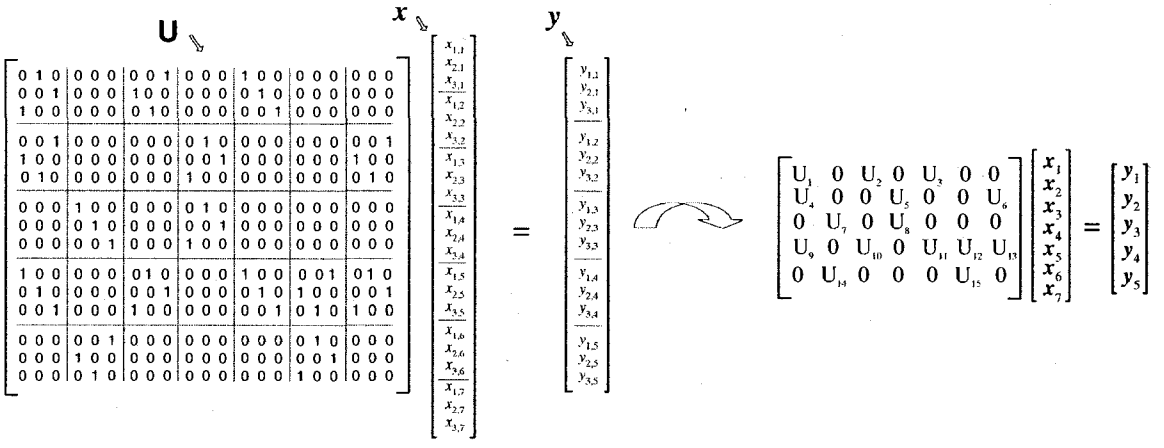


Figure 3.11: Matrix-vector multiplication illustration, figure taken from [36]

Figure 3.11 shows that the block matrices are either zero matrices, identity matrices, or circularly right-shifted versions of the identity matrix. It is then noted that whenever a vector is multiplied by a shifted identity matrix the result is simply the given vector shifted the same number of times that the identity matrix is shifted. For example, matrix Z is circularly right-shifted X times, specifically 1 time in Figure 3.12.

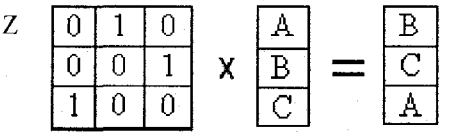


Figure 3.12: An identity matrix shifted once, produces a vector that is shifted once

This means that the '1' bit in the first row is in the X^{th} column, the '1' of the second row is in the X^{th+1} column, and so on. With this in mind, one then realizes that the first row of the result vector is the X^{th} element of the original multiplied vector; the 2nd row is the X^{th+1} element of the original vector and so on. As such it is apparent that, to get the result vector, we need only circularly shift the original vector the same number of times as the multiplicand shifted matrix.

But one must also remember that the block matrices that are being multiplied are building blocks of a larger matrix. Therefore, it is clear that addition is necessary to produce the result of an entire row. This is done through simply adding every intermediate shifted vector, as shown in Figure 3.13.

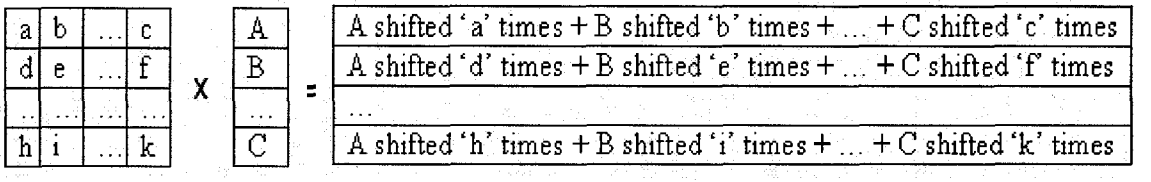


Figure 3.13: After the intermediate vectors are shifted, they are compounded through addition to produce the row result

A straight forward attempt at implementing the block multiplier unit may be based on a shift register design. Since, the 802.16e standard supports 19 different block sizes, the block multiplier must be able to accommodate all sizes with one implementation. Figure 3.14 illustrates one such design where a 96-bit shift register is used.

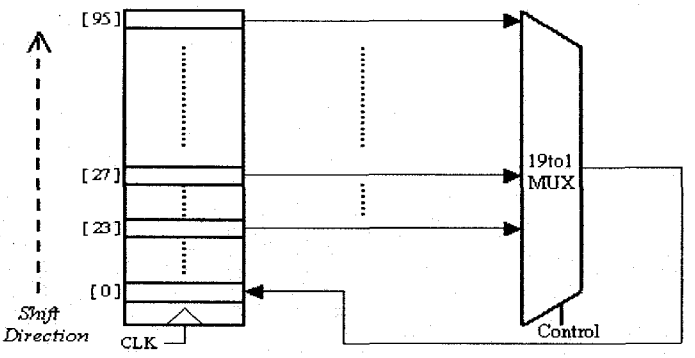


Figure 3.14: Shift-register based block multiplier

The input bits of the multiplicand vector are loaded into the shift register. The bits are then shifted upwards as many times as the shift value associated with the multiplicand block matrix. While the shift register is designed to accommodate the maximum block size of 96, a 19:1 multiplexer (MUX) component is used to create a "variable length" shift register. As the bits are shifted upwards, the bit slot at location 0 gets constantly filled with the content of the top-most bit that is selected by the MUX. The multiplexer selects the top-most bit according to the block size specified by the user, thus effectively capping the top of the shift register at the right location and creating a looping mechanism for the bits.

While the shift register design is an effective way to perform the block multiplication, it suffers from two severe performance problems. The first problem is *high latency*. Most encoder implementations, require the input bit stream to arrive serially (i.e. one bit at a time). The above implementation requires the entire input bit vector to be present before the multiplication (i.e. shifting) can start, hence it must wait for several clock cycles before the input is fully loaded into the shift register. This adds a high count of latency clock cycles to the design. The second problem is *variable latency*. Since different block matrices have different associated shift values, the multiplier will need a different number

of clock cycles to produce the results. This makes the multiplier's latency highly variable and affects the overall encoder's latency similarly. Therefore, it is clear that a different design is required to implement the block multiplier.

In order to build an efficient block multiplier with low and consistent latency, we developed a design idea inspired by the concept of *circular pointers*. Circular pointers are known constructs in the field of computer science. They are based on the idea of associating a pointer with a storage unit such as a memory block or a register, whose range of addresses are accessible by the pointer. The pointer traverses the register's address space in a unidirectional manner. When reaching an end, the pointer loops back to the other end, thus creating a virtual circular register. Read and/or write operations are possible at the addresses identified by the pointer. Figure 3.15 explains the idea of the circular pointer.

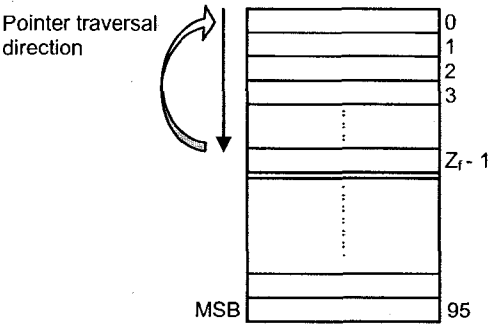


Figure 3.15: 96-bit register with a circular pointer

The above figure shows a 96-bit register designed to support the maximum block size of the 802.16e LDPC encoder. The pointer traverses the register towards the most significant bit (MSB), and may start at any bit address. When reaching the address corresponding to the user-selected block size (Z_f), the pointer loops back to address 0. This mechanism effectively implements the shifting operation required to perform the

block multiplication. As the first bit of the multiplicand vector arrives, it is inserted at an address calculated based on the shift value retrieved from memory. The next bit is inserted at the following address, and so on. The pointer loops back to address 0 after reaching the z_f-1 address, all the while the rest of the bits are inserted accordingly. When the vector's bitstream is done, so is the multiplication (shifting). As such, the multiplication takes exactly z_f clock cycles to complete. This design achieves constant multiplication times *regardless of the shift value*, which significantly reduces and stabilizes the latency of the encoder.

Our block multiplier design is illustrated below in Figure 3.16. The 7-bit wide shift value is received from the memory controllers, and used to calculate the starting address of the pointer. After that, the *Address Generation Logic* block takes over control of advancing and looping the pointer's address. The 7-bit address is transformed into a 96-bit one-hot encoded value via the *96-bit Binary Decoder block*, thus *activating* the single register location where the input bit is meant to be inserted. When a zero matrix is encountered, the shift value received from memory is "1111111". This triggers the *7-bit NAND gate* to generate the *Zero Flag* signal. This flag forces the input low (i.e. zeroes the input via the AND gate), and disables the binary decoder thus conserving power.

As we mentioned earlier, the results of multiplying a vector with one column of a matrix have to be added to the product of multiplying the same vector with the next column, and so on. And as we showed earlier in Figure 3.13, the block multiplication results have to be compounded through addition. Our design accomplishes this through the integration of 96 XOR gates into the block multiplier. Each XOR gate always "adds" the value of the arriving new bit to the current bit value stored at that register location.

Therefore, after all of the input is "clocked" in, the output register will hold the *row result* of the multiplication.

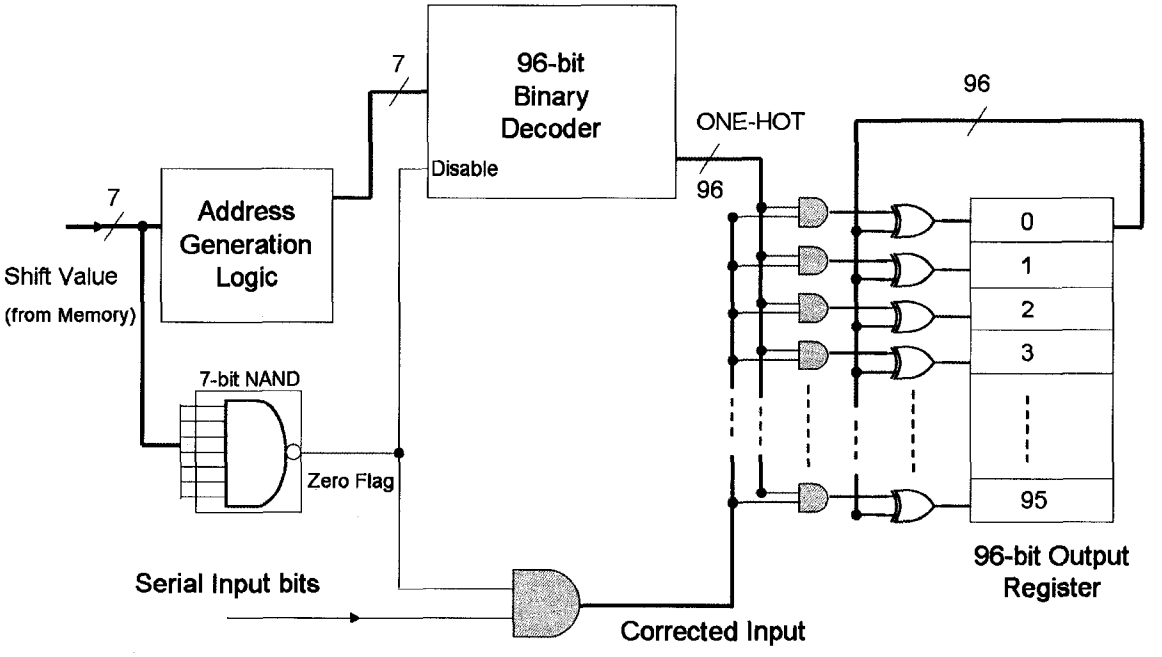


Figure 3.16: The circular-pointer based block multiplier

In order to process multiple rows, as is expected with large matrices (such as the A and B matrices), one must replicate this block multiplier as many times as the matrix has rows. This leads to the next component of the 802.16e LDPC encoder.

3.4.5.2 The Full Matrix-Vector Multiplier

This component is the first type of matrix-vector multipliers required by the encoder to handle the A, B, and C matrix-vector multiplications as outlined in Section 3.4.3. We used the block multiplier presented in the previous section as a building block for this component. The idea is to replicate the block multiplier an appropriate number of times to handle the multiple rows of the large matrix. This is illustrated in Figure 3.17. The challenge is, however, that the dimensions of these matrices differ per matrix, code rate, and code length.

The code length challenge is solved by the flexibility of the block multiplier design. The block multiplier loops the pointer back limiting the size of the register to the block size (z_f). As the block size changes, so does the end address accommodating the full range of block sizes and code lengths.

The code rate and matrix challenge is out of the scope of the block multiplier's design, and is tackled at the full multiplier level. The three matrices in question range in row size from 1 to 11, and in column size from 1 to 20. Noting that the C matrix has a single row at all conditions. The effect of the ranging columns is handled through the re-use of the block multipliers a number of times equal to the matrix' columns. The results of the first column are added in the same register to the results of the next column on route to the full row result. The effect of the ranging rows is handled through the replication of the block multipliers. At this point, we decided to develop two separate multiplier components. The first has only one block multiplier unit inside, and handles the C matrix multiplication. The second, featured below in Figure 3.17, handles the A and B matrix multiplications, and has 11 block multiplier units inside. Despite supporting the

maximum number of rows, only the required number of block multipliers will be in operation. The excess multipliers are turned off preventing any switching activity and reducing the total power consumption.

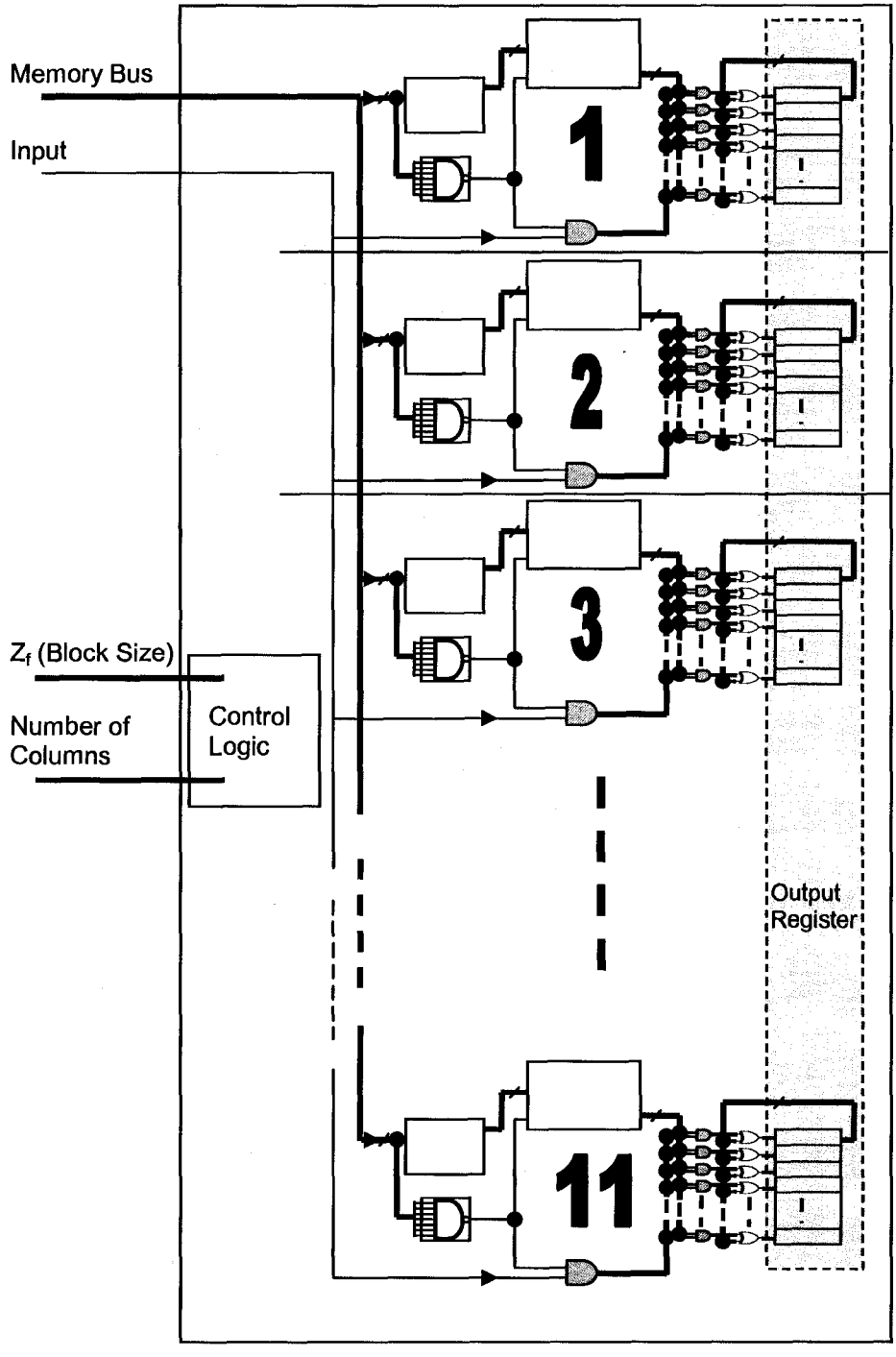


Figure 3.17: The full matrix-vector multiplier

The full matrix-vector multiplier operates according to an internal FSM, briefly explained in the following steps.

1. The multiplier requests the values of one column of the matrix at hand from the appropriate memory controller.
2. The multiplier receives one matrix column of shift values on the memory bus.
3. The multiplier enters a loop for z_f clock cycles supplying the input bits to the individual block multipliers in parallel. The block multipliers perform the shifting.
4. After finishing a column multiplication, the multiplier checks the total number of columns required for the matrix at hand. If the maximum number is reached, the multiplier goes to step 4, otherwise it goes back to step 1 where it requests the next matrix column values.
5. Upon finishing all columns, the row results of the final vector are now ready in the block multiplier register segments. These segments make up the complete output register highlighted in Figure 3.17.

In addition to the buses and signals shown in the figure, other hand-shaking signals, reset signals, and the clock are not shown for clarity.

3.4.5.3 The ET^{-1}, T^{-1} Matrix Multiplier

In section 3.4.3, we alluded to the second type of matrix-vector multipliers required in the encoder system to handle the multiplication with the ET^{-1} and T^{-1} matrices. Unlike the previous matrix-vector multiplier, this component *requires no matrix contents* to perform the multiplication. Instead, it employs an efficient hardware design idea that mimics the contents of the ET^{-1} and T^{-1} matrices producing the correct multiplication results.

The design of this component is based on an observation made at the matrix structure level. It is observed that *for all supported code rates* in the standard, the T, E, T^{-1}, ET^{-1} matrices always *take specific shapes*, illustrated below in Figure 3.18 for a code rate 3/4.

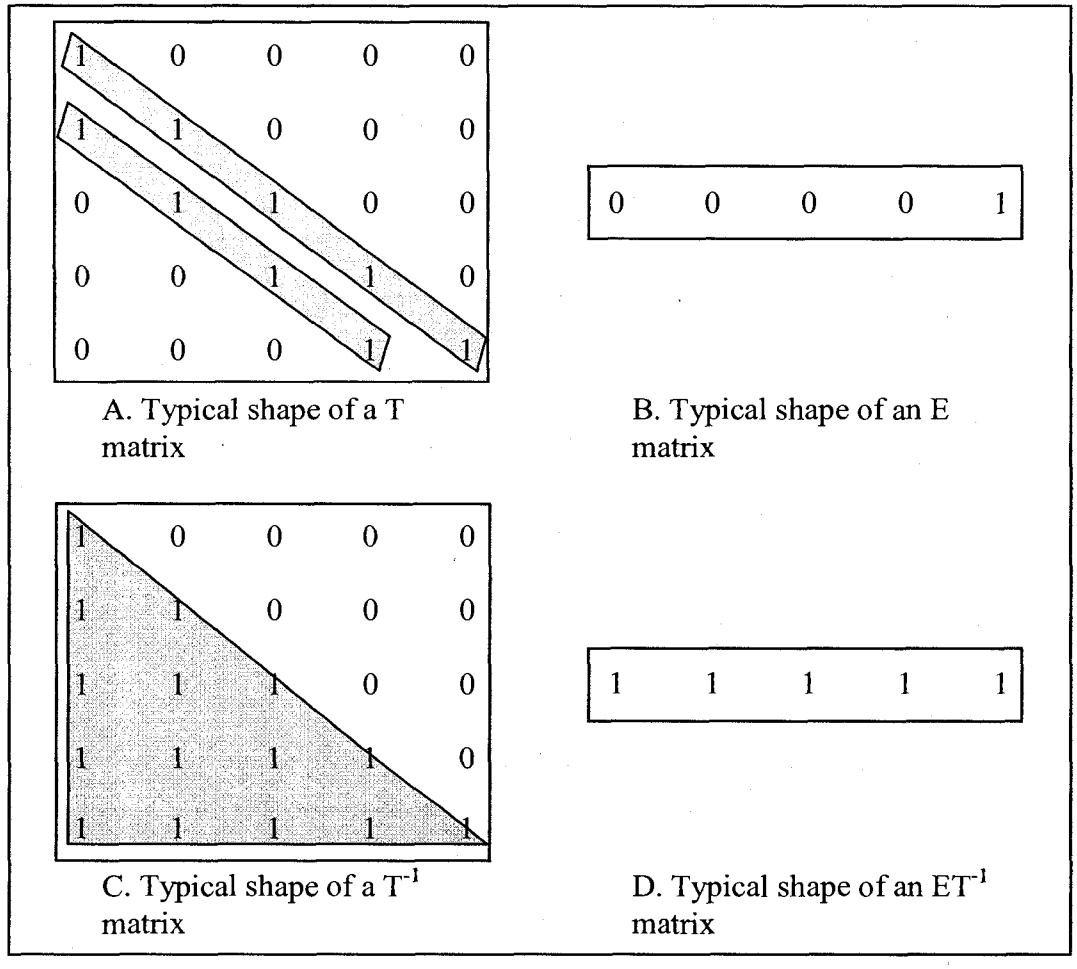


Figure 3.18: Typical shapes of T, E, T^{-1}, ET^{-1} matrices for code rate 3/4

The 1s in Figure 3.18 represent identity block matrices, while the 0s represent zero block matrices. The T matrix is a square matrix with a dual diagonal of identity block matrices. The inverse of this, shown in part C of the figure, is a true lower triangular matrix where all elements below the diagonal are identity block matrices. Similarly, the E matrix always takes the shape of a vector of zero block matrices with the exception of an identity matrix at the last location. And when multiplying the E and T^{-1} matrices, the result is a vector of identity matrices matching in dimensions with the E matrix.

These matrix shapes are important to the design of their multipliers. When multiplying the ET^{-1} matrix with a vector X, the result is the sum of the block-size segments of X as shown in Figure 3.19 A. Following the same concept, the multiplication of T^{-1} with X results in a vector whose every segment holds the sum of the corresponding segment from X and all previous X segments. Figure 3.19 B.

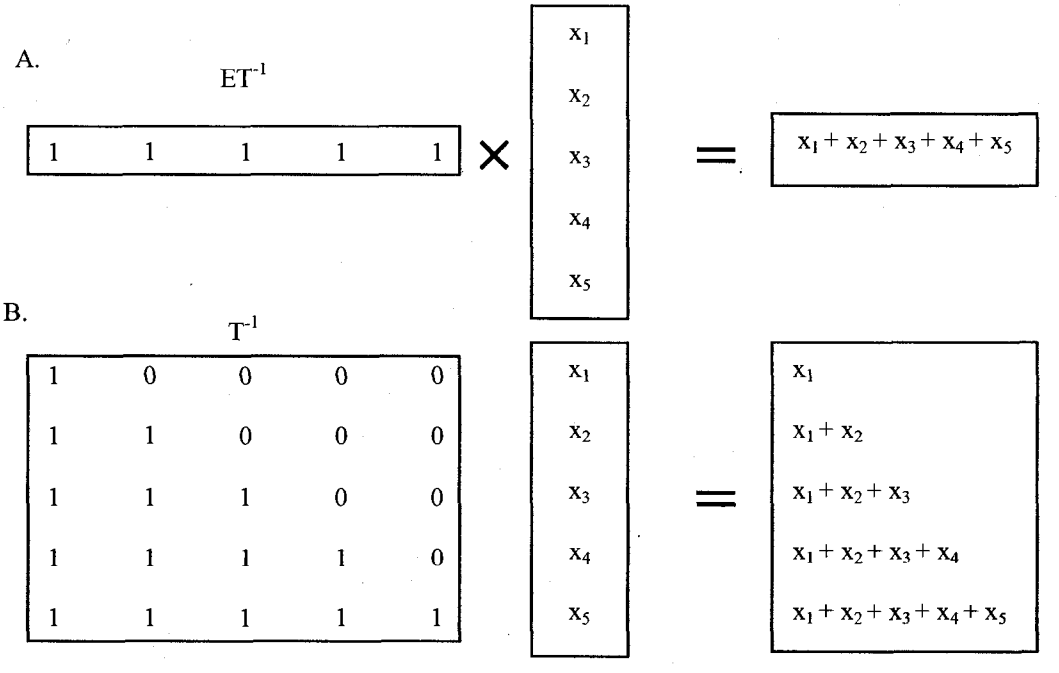


Figure 3.19: Multiplying a vector by each of ET^{-1} and T^{-1}

It is evident from the above observations that the multiplication process can be reduced to a series of additions. We designed a multiplier component that takes advantage of this concept utilizing XOR adders and shift registers. Figure 3.20 shows our design, termed the *large-shifter multiplier* which can be used for both the ET^{-1} and the T^{-1} multiplications.

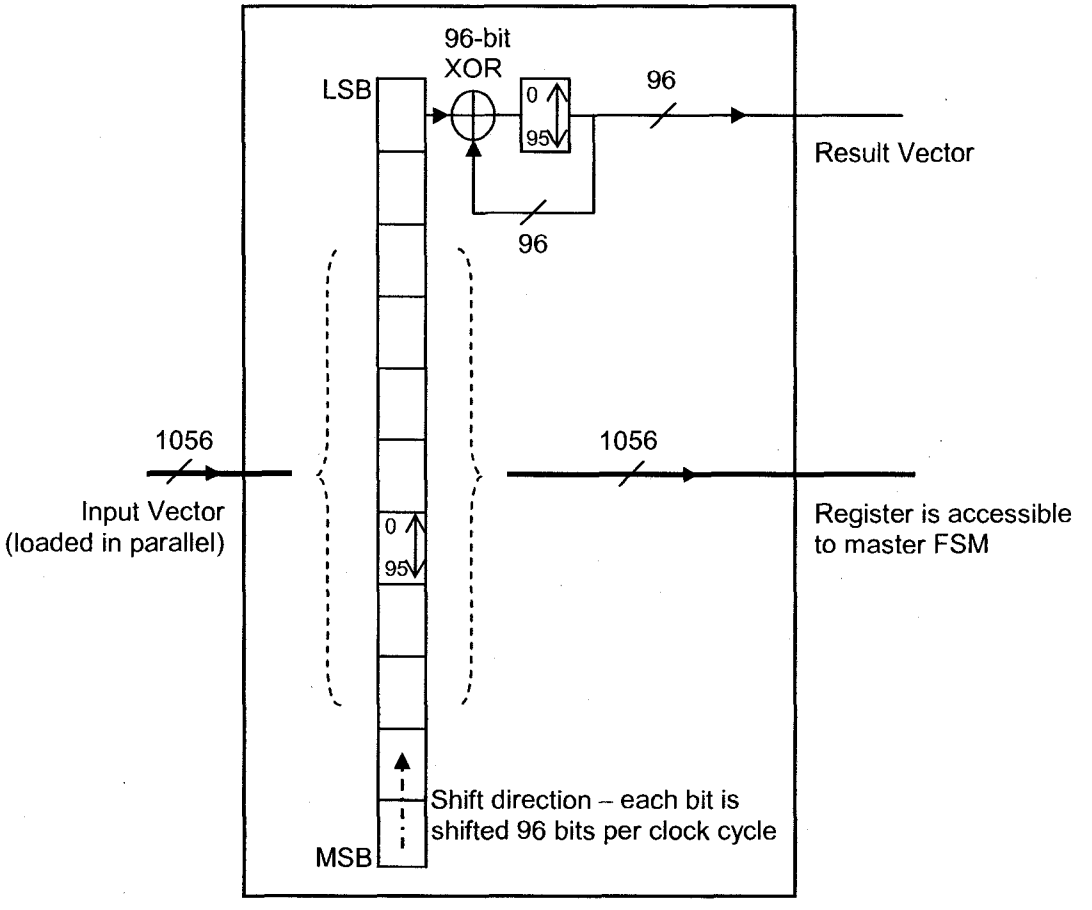


Figure 3.20: The large-shifter multiplier

The multiplicand input vector is loaded into the internal large shift register in a parallel fashion. During each following clock cycle, the input vector is shifted towards the LSB 96 bits at a time. This means that bit 96 becomes bit 0, bit 97 becomes bit 1, and so on. In addition, the most significant segment gets loaded with the contents of the least

significant segment creating a large *circular shift register*. This feature helps conserve the input values for future access by the master FSM. During each clock cycle, the 96 bits stored in the first segment (bits 0-95) of the large register are added to the contents of the small register via a 96-bit XOR array. The use of the XOR gates for addition here, as is the case for all additions in the encoder, is due to the fact that we are working with the binary system or GF(2). The small register is initialized with 0s and thus the very first addition (XOR) operation results in the input segment itself. It is noted that the segments, and hence the registers, are designed to accommodate the largest block size of 96 bits. However, smaller block sizes are processed similarly since the shifting operations do not alter the bit order. After 11 shift-add clock cycles, the small register holds the sum of all segments of the large shift register. This is essentially the result of the ET^{-1} multiplication as presented in Figure 3.19 A. While the number of columns of the ET^{-1} matrix range from 3 to 11 depending on the code rate, 11 cycles of shift-add operations are always performed in order to keep the encoder's latency constant. The extra vector segments are forced low (i.e. filled with zeros) in order not to alter the result.

The same hardware design can be used to perform the T^{-1} multiplication. The only difference is that the intermediate addition results make up the segments of the overall resulting vector, see Figure 3.19 B. Therefore the small register is made accessible to the master state machine, which captures each intermediate result and stores it into an external large register. After 11 shift-add clock cycles, the external large register holds the result of the T^{-1} multiplication.

A key code structure exploit and the component's re-use, position this efficient design to achieve significant memory storage gains as well as resource and area reductions.

3.4.5.4 Memory Storage and Controllers

We demonstrated in Section 3.4.4 that the most memory efficient approach to storing the contents of the A, B, and C matrices is the on-chip scaling method. We discussed that in order to compute the correctly scaled shift values on-chip, we need to store the division fractions in memory using an 11-bit fixed point representation. See Section 3.4.4 for details. Finally, we calculated the total number of bits to be stored in memory using this approach to be 8074 bits or 7.9 Kb.

The design of the full matrix-vector multiplier requires access to the memory contents in a column-by-column fashion. Therefore, the memory controllers transfer the column shift values via the memory bus as requested by the corresponding multipliers. However, the memory values are not ready to be transferred until the memory access, multiplication, and scaling operations are complete. By contrast, the multiplier components are ready to start the multiplication operations as soon as the input bits start arriving at the encoder (i.e. on the next clock cycle). Therefore, the multiplier must wait until the memory processing steps have completed and the values are ready on the memory bus. This waiting delay can be up to 10 clock cycles and adds to the total latency of the encoder, violating one of our main design targets.

We developed a solution for the memory controller latency challenge through redesigning the memory storage scheme. One realizes that only the first column shift values from the A and C matrices experience this latency delay. The rest of the columns, including the single column of the B matrix, do not necessarily need to add any latency to the design. This is possible, because the shortest block multiplication needs at least 24 clock cycles to finish, while the memory controller processing steps need less than 24

clock cycles. Therefore, one can hide the memory processing costs by asking the controller to start preparing the next memory column values while using (multiplying) the current ones. Thus, with the rest of the columns and the B matrix not being a latency factor, there must be a mechanism to make the first column shift values of A and C scaled and ready within one cycle from the input's arrival at the encoder.

We are able to achieve this goal by employing two separate approaches to memory storage. The first columns of A and C are stored in a fully processed format, and scaled to support all code lengths of the standard. This way, these values are ready for immediate access by the multipliers. The rest of the columns along with the B matrix are stored in a non-scaled format and require processing from the memory controllers. This solution is in essence a combination between the second and the third storage approaches presented in Section 3.4.4 and summarized on Table 3.2. In that table, we noted that a fully scaled storage approach requires 95.3 Kb of memory space, while a fully non-scaled approach requires 7.9 Kb of space. This combination solution requires a minimum of 13.1 Kb of space, which is a small storage tradeoff when compared to the 24% of latency gains obtained. With this solution, the total system latency can be reduced from 42 to 32 clock cycles.

For storage, we utilized the available block RAM space on the target FPGA device. Four block RAM components are instantiated. The following table presents their information.

Component Name	Dimensions	Total bit count
C_first_column	95 × 7	665
C_other_columns	79 × 11	869
A_first_column	95 × 77	7315
AB_other_columns	84 × 121	10164

Table 3.3: Summary of the encoder's Block RAM memory components

Adding the bit counts from the above table, the total is 19013 bits or 18.6 Kb. Forcing the increase from the 13.1 Kb figure mentioned earlier, is the internal structure of the block RAMs. With current technology, it is not possible to instantiate a variable-width RAM, which would be ideal to store the various differently sized matrices. For example. The second column from the A matrix with code rate 1/2 is 11 elements, whereas the same column with code rate 2/3 is only 7 elements. Therefore the largest size of 11 elements is used for the block RAM width, and the extra spaces are filled with zeros for shorter columns. A final note regarding the memory storage is the fact that for our particular encoder implementation, code rate 2/3A is not included, accordingly the corresponding matrix information are not stored in memory. The total block RAM space with the 2/3A code rate would be 22710 bits or 22.2 Kb.

The 802.16e encoder includes two memory controller components. One to manage the A and B memory storage and busses, and one to manage the smaller C matrix. The design of the memory controller component is presented in Figure 3.21.

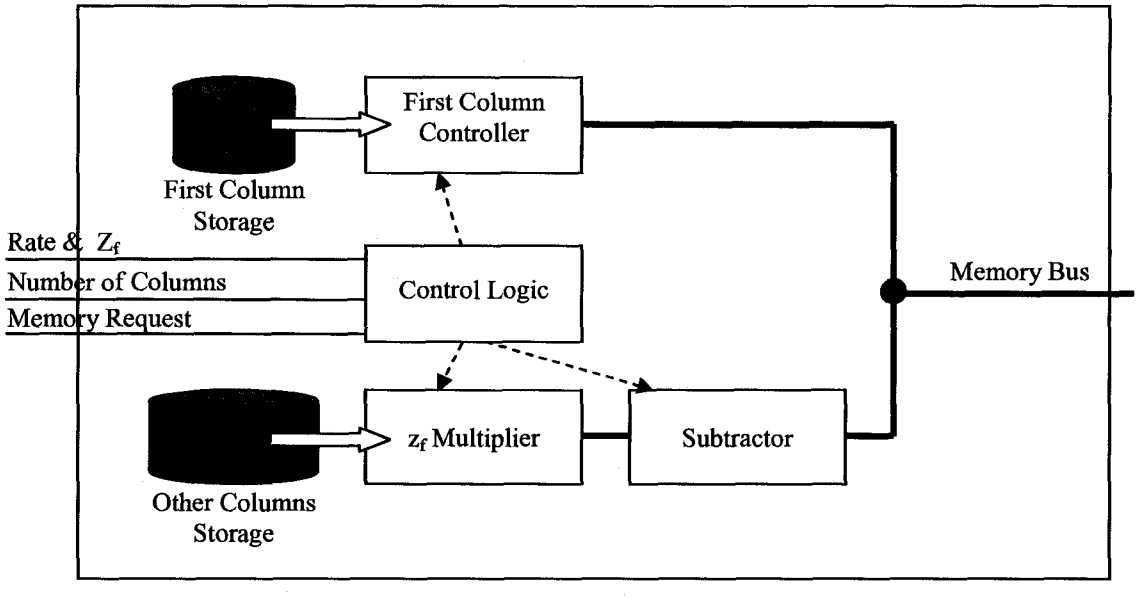


Figure 3.21: The memory controller component

The memory controller processes and prepares the memory values using the following steps:

1. The controller starts by initializing the memory bus to the values supplied by the first column controller. One clock cycle after the code rate and length parameters become available to the encoder, the first column controller accesses the *first column storage*, and produces the correctly scaled values of that column on the memory bus.
2. The controller enters a wait state, until a memory request is received from the associated multiplier.
3. When a memory request is received, the appropriate non-scaled column values are retrieved from the *other columns storage* and forwarded to the internal z_f multiplier.
4. The z_f multiplier performs the multiplication between the 11-bit retrieved fraction and the 7-bit user-selected z_f value. The result is an 18-bit product in a "7.11" fixed-point format.
5. The fractional portion represented by the least significant 11 bits of the product is truncated leaving the 7-bit integer portion. This is equivalent to the floor function. At this stage, the result represents the correctly scaled value from memory.
6. The last processing step includes subtracting the shift value from z_f, to produce the starting address used by the matrix multiplier's circular pointer to insert the first bit. This is explained with the following example.

Let the block size (z_f) be 5, which makes the block matrix a 5×5 matrix. Assume that this matrix is a circularly right-shifted identity matrix by 2, hence

the associated shift value is 2. When multiplying this block matrix by a vector segment A, the result is the vector A shifted twice as shown below. Further assume that the vector holds the illustrated values that match the values of its addresses (i.e. bit 0 holds the value 0, bit 1 holds the value 1, and so on).

$$\begin{matrix} 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \end{matrix} \times \begin{matrix} 0 \\ 1 \\ 2 \\ 3 \\ 4 \end{matrix} = \begin{matrix} 2 \\ 3 \\ 4 \\ 0 \\ 1 \end{matrix} \leftarrow \text{Starting Address}$$

It can be seen in the result, that the A vector's bits are shifted upwards (in address space) twice, and bit 0 is wrapped around to take the address 3. Therefore, when *subtracting* the original shift value of 2 from the block size 5, we get the correct address of 3. This value can then be used as the starting address for the circular pointer implementation.

7. After each memory request signal, the column count is checked. When the last column is reached, the controller's task is over, and it defaults back to the starting state.

Both memory controllers use these steps to process the memory contents in a similar fashion. The AB memory controller receives one more memory request than the C controller in order to retrieve B's only column.

3.4.5.5 The Master Finite State Machine

The master FSM is the *top-level VHDL component* that ties all other components together and controls the flow of the encoding process. It has two main tasks:

- Managing the flow of data between memory and multipliers, as well as controlling the flow of the bitstream along the encoding chain of multipliers and adders.
- Controlling the inputs and outputs of the encoder. This includes configuring the encoder to work according to user parameters, buffering intermediate bit vector results, and producing the output bitstream and signals.

In order to accomplish the above tasks, the top-level module is comprised of two actual FSMs. The first is the *encoding FSM* which handles the encoding data flow, hardware re-use, and output flags. The second is the *latency FSM* which mainly handles the buffering of intermediate vector results and the switching of the output bitstream among the input, p1, and p2 streams. The control logic required to implement these FSMs is contained inside the control logic block in Figure 3.22.

In addition to the FSMs, the top-level module contains the 3:1 output multiplexer component, several registers including the p1 and p2 registers, and the latency buffers. The *latency buffers* are shift registers of different lengths used to delay the arrival of the input and p1 bit streams to the output MUX. This helps in maintaining the serial output of the encoder. Many other less significant components exist in the top-level design but are not shown in Figure 3.22.

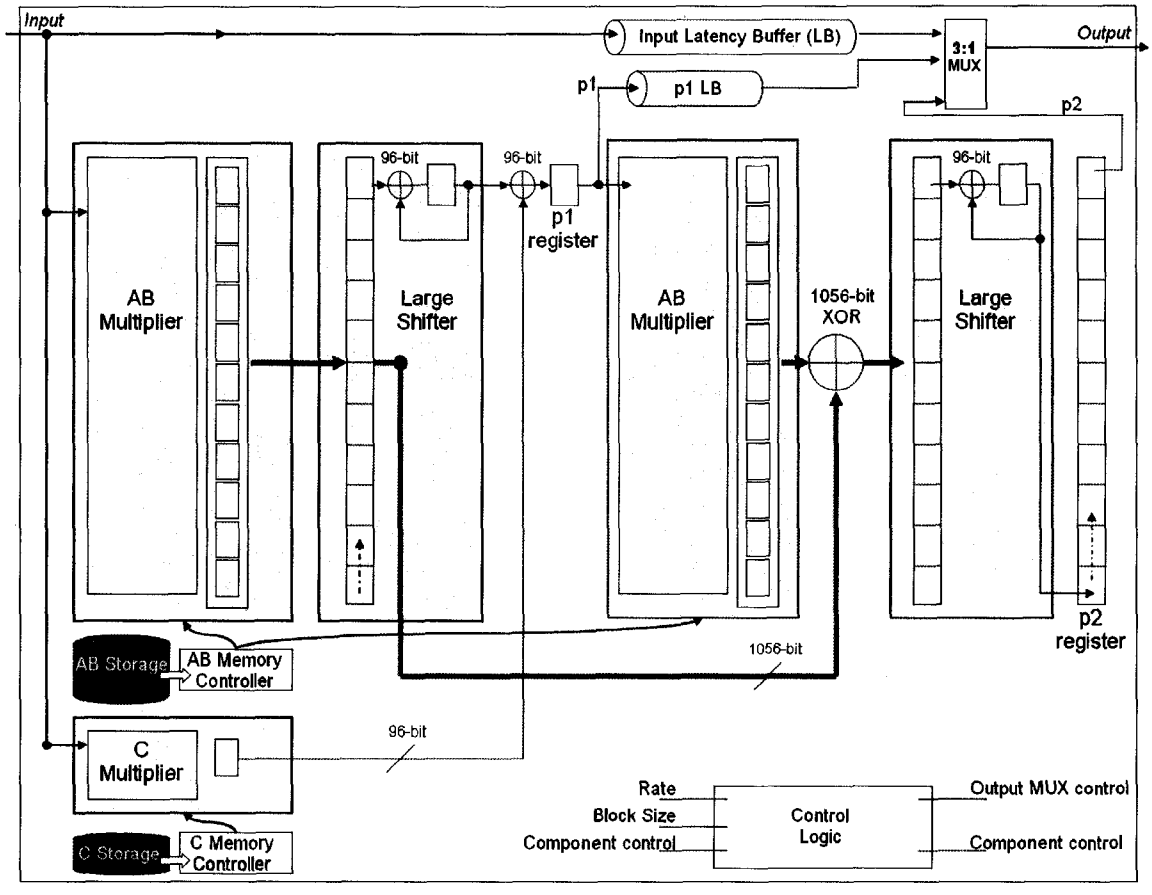


Figure 3.22: 802.16e LDPC Encoder System (top-level) Diagram

The encoding FSM manages the flow of the encoding algorithm utilizing the user-selected parameters such as the code rate and the block size, as well as the internal components' control flags. The following steps detail the encoding FSM's states:

1. The encoder system waits for the start signal from the user design. All signals will be explained in Chapter 4. When the signal is received, the full matrix-vector multiplier components start processing the serial input bits and performing the multiplications. At this stage, the C multiplier, and the larger AB multiplier components are working simultaneously. This is an example of hardware parallelism.

2. When the multipliers are done, a flag signal is raised and the system moves to the second stage. In the second stage, the large shifter component loads the result of the AB multiplier and starts the ET^{-1} multiplication operation.
3. When the large shifter is done, the system moves to the third stage. This is a one clock cycle stage where the C multiplier result and the ET^{-1} multiplier result are added (XOR-ed) to produce the p1 bits. The result is stored into the p1 register. And the system moves to the 4^{th} stage.
4. While switching from the 3^{rd} to the 4^{th} stage, the encoding FSM switches the input sources for both the AB multiplier and the large shifter components. This is done via two multiplexer components (not shown in Figure 3.22) that select the input busses to be connected to the aforementioned components. This is an example of hardware re-use.
5. The 4^{th} stage is a dual-purpose stage. The p1 register is a shift register used to input the bits into the AB multiplier in order to perform the B matrix multiplication. At the same time, the p1 bits are shifted (transferred) into the p1 latency buffer to be eventually outputted by the encoder system.
6. When the AB multiplier is done, the system moves into the 5^{th} stage. This a one clock cycle stage where the A matrix multiplication (saved in the large shifter) is added to the output of the B matrix multiplication (in the AB multiplier) via the 1056-bit XOR array. The system moves into the 6^{th} stage.
7. During the 6^{th} stage the final multiplication with T^{-1} matrix is performed. The output of the previous addition is loaded into the large shifter component which is re-used for this stage. As the shifter is working, the intermediate addition results

are captured by the FSM and stored into an external register. When the large shifter is finished, the p2 bits are ready to output and stored in the p2 register.

8. Despite the algorithm needing only 6 stages to finish, the encoder system needs a final 7^{th} stage to output the p_2 bits. By this stage, the system had already outputted the input bits and the p1 bits and is ready for the p2 bitstream. Providing that bit stream is the encoding FSM through shifting the p2 register contents to the output MUX. Upon finishing all output, the FSM sets the values of several external signals (flags) to indicate the encoding completion to the user design.

While the encoding FSM controls most components and handles the flow of the algorithm, the latency FSM performs a more focused task. This FSM tracks the various stages of encoding, and switches the output MUX at exactly the right clock cycles to preserve a continuous flow of serially outputted bits. The latency buffers are designed to make the input bits finish outputting in time for the arriving p1 bits, and the p1 bits in turn for the ready p2 bits. Specifically the input buffer is a 32-bit shift register while the p1 buffer is a 17-bit one. Since latency is defined to be the number of clock cycles from the arrival of the first input to the production of the first output, the total encoder's latency is dominated by the input's buffer and is equal to 32 clock cycles. This is a low latency design suitable for many communication applications.

This ends our discussion of the components of the 802.16e encoder system. The system is designed, tested, and implemented in hardware as an intellectual property core ready for use with larger digital designs. For further usage and testing details, the reader is referred to the Core's datasheet [86] and Chapter 4.

3.5 Summary

In this chapter, we presented our 802.16e LDPC encoder design. We started with the supported LDPC codes in the mobile WiMAX standard highlighting their compact structure. The use of almost lower triangular and circulant matrices in these codes allows for efficient encoding/decoding hardware architectures. Through various studies, LDPC codes were shown to exhibit good coding performance in mobile WiMAX systems.

For our encoder design, we adopted the second method, as proposed by the standard, as our encoding algorithm [10]. Our encoder design and implementation flow was discussed, and included three main processes: software, hardware, and testing processes. The encoder top-level design served as a plan to divide the overall problem into smaller design components and allowed for design re-use and hardware parallelism. We focused on the software processes as a development and testing platform. In addition, we illustrated how software was used to design and prepare the encoder's memory storage. We determined that a fair hardware utilization is to store the parity-check matrices' contents in memory as semi-scaled fixed-point values with 11-bit precision.

The hardware components of the encoder system were developed using the VHDL language and a standard VHDL design flow. The main design component is the block multiplier unit. The block multiplier uses a circular pointer solution to implement a vector-matrix multiplication. This design provides constant multiplication times regardless of the input value, which in turn achieves low and consistent latency to the overall encoder. This unit is replicated 11 times to construct the full multiplier component which supports the various code rates in the standard. This multiplier handles the A and B matrix multiplications while the ET^{-1} and the T^{-1} multiplications are handled using a large

shifter design. The large shifter component exploits the specific forms of the ET^{-1} and T^{-1} matrices to do away with matrix storage and perform the required multiplications.

The memory storage-controller pair design was presented next. The controller accesses the memory contents and completes the scaling process before transferring the shift values to the multipliers on the memory bus.

Finally, the master FSM was introduced. Internally, it is comprised of two FSMs; one to handle the I/O and the latency buffers; and one to handle the rest of encoder operations including memory-multiplier interaction and algorithm control. The complete design, presented in Figure 3.22, satisfies all design targets and implements an 802.16e LDPC encoder IP core with latency of 32 clock cycles.

Chapter 4

Encoder Verification and Performance Analysis

4.1 Encoder IP Core - Description and Usage

The 802.16e compatible LDPC encoder discussed in Chapter 3 is implemented in an Intellectual Property (IP) core form. This core is designed to interface with other digital cores in applications such as digital transmitters. Figure 4.1 depicts the block diagram for the core illustrating the input/output interfacing signals.

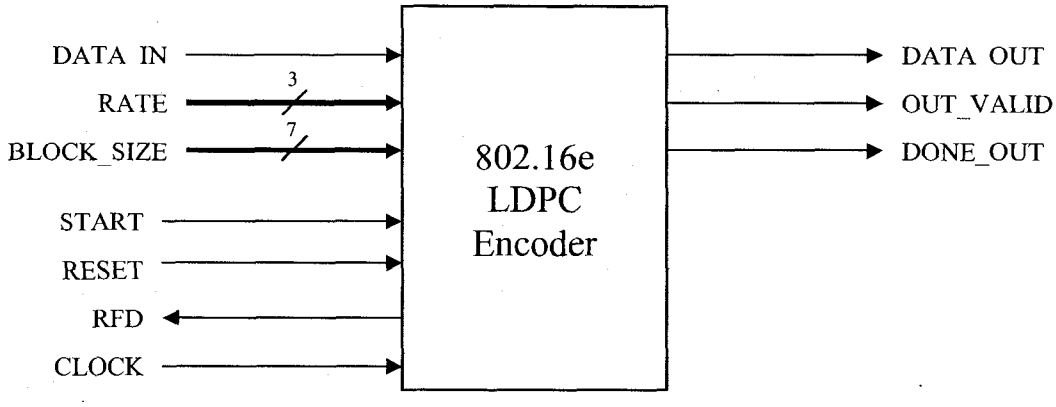


Figure 4.1: 802.16e LDPC encoder core block diagram

The core's signals include data busses where the user design supplies input data, and receives output data. It also incorporates control signals which facilitate interfacing and provide the user control over the operation of the encoder core. Except for the multi-valued RATE and BLOCK_SIZE signals, all other control signals are *active high*, which means they are considered *on* when their value is logical '1'. Table 4.1 provides a summary of the signals (a.k.a ports) featured in this design.

Port Name	Port Width	Direction	Active State	Description
DATA_IN	1	INPUT		Data Input, serially presented packet to be encoded
RATE	3	INPUT		Rate to be used to encode the packet
BLOCK_SIZE	7	INPUT		Block Size to be used to encode the packet
START	1	INPUT	High	Start, to start the encoding process
RESET	1	INPUT	High	Synchronous Reset
RFD	1	OUTPUT	High	Ready For Data, to indicate the system is accepting input bits
CLOCK	1	INPUT	Rising Edge	Clock
DATA_OUT	1	OUTPUT		Data Output, encoded packet output provided serially
OUT_VALID	1	OUTPUT	High	Output Valid, to indicate the presence of valid output bits on DATA_OUT
DONE_OUT	1	OUTPUT	High	Done Output, to indicate the end of the output bit stream.

Table 4.1: System signal pinout

The operation and usage of this core is best described using its ports and how they are utilized, we start with the data and parameter busses:

DATA_IN: On this input bus, the user design supplies the *uncoded data input bits* to the system in a serial fashion starting with the first bit to be encoded.

DATA_OUT: This is an output bus, on which the system outputs the *encoded packet bits* serially.

RATE: Using this input bus, the user design supplies the code rate *run-time parameter* to the system. The encoder uses the value on this bus to select the appropriate coding matrix and code rate for encoding. This bus is read by the system on the start of encoding when the START and RFD signals are asserted. The allowed values for this bus and their definitions can be found in [86].

BLOCK_SIZE: Using this input bus, the user design supplies the block size (z_f) run-time parameter to the system The encoder uses the value on this bus to select the

appropriate block size, and the input and output packet lengths. This bus is read by the system on the start of encoding when the START and RFD signals are asserted. The allowed values for this bus and their definitions can be found in [86].

The user design communicates with, and manages the operation of the encoder core using the following control signals:

START: The encoder core starts the encoding process when the start signal is asserted by the source of the uncoded bits. When START is asserted, the first input bit is consumed by the system. All subsequent input bits will be consumed by the system on the following clock cycles until the end of the input bitstream. The encoder assumes a non-interrupted sequence of input bits. Figure 4.2 is a snapshot of post place-and-route simulation illustrating a typical behavior of the start signal.

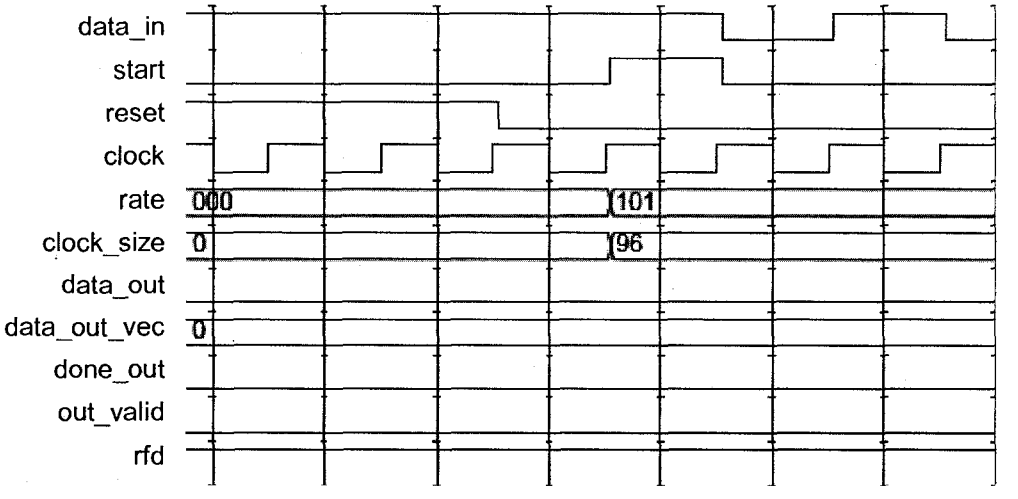


Figure 4.2: Behavior of START signal during encoding of a rate 1/2, block size 96 input packet

RFD: Prior to the start of encoding, the *ready for data* (RFD) signal is asserted by the system to indicate that it is ready to accept data on DATA_IN. RFD is initially asserted but the input bit stream will not start to be consumed until START is asserted for at least one clock cycle. RFD will remain asserted until the entire input packet is consumed, after

that it will remain deasserted while the system is processing and outputting the encoded packet. Figure 4.2 above, shows the initial condition of RFD, and Figure 4.3 illustrates the end of input packet consumption by the system.

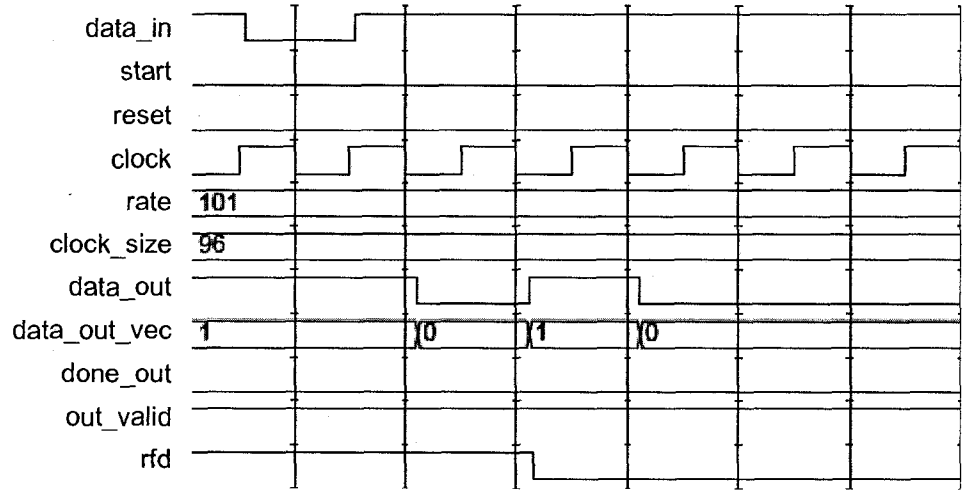


Figure 4.3: RFD Signal on end of input packet, code rate 1/2, block size 96

OUT_VALID: This output signal is asserted by the system to indicate the presence of valid output bits on DATA_OUT. This signal is first asserted 32 cycles after the start of encoding and remains asserted until the end of the output packet. Figure 4.4 illustrates the 32-cycle latency mentioned above.

DONE_OUT: The system asserts DONE_OUT for one clock cycle to indicate the end of the output packet. This signal also indicates the end of the encoding process. When DONE_OUT is asserted, RFD is asserted and OUT_VALID is de-asserted indicating the system's readiness to start encoding another packet.

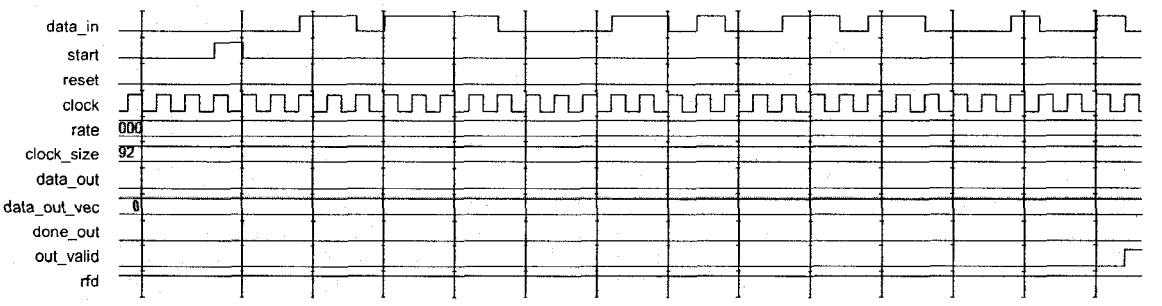


Figure 4.4: OUT_VALID Signal and the 32-cycle latency, code rate 5/6, block size 92

DONE_OUT's behavior is shown in Figure 4.5. Figure 4.6 illustrates a complete encoding waveform for a packet using code rate 1/2, block size 96 and an encoded packet length of 2304 bits.

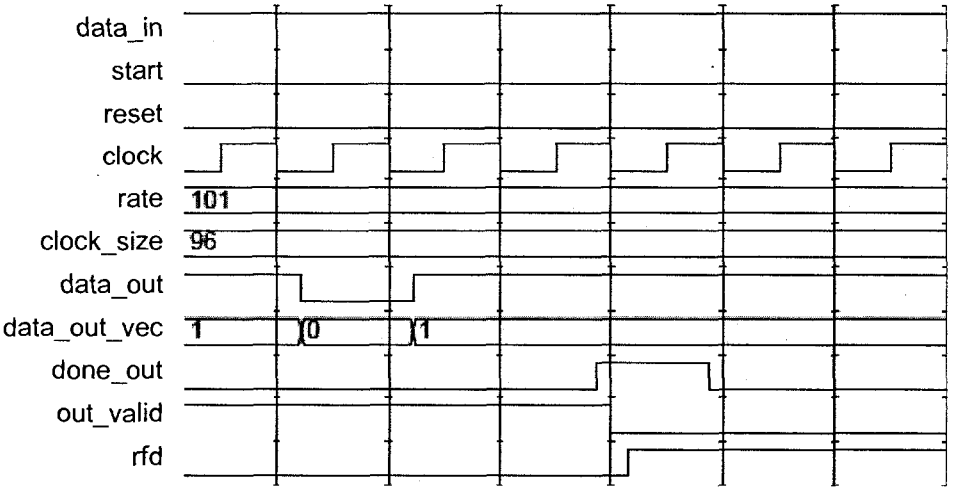


Figure 4.5: DONE_OUT asserted on end of data output, code rate 1/2, block size 96

CLOCK: This pin supplies the clock frequency signal to the design. All components in the encoder core are synchronous to the CLOCK input. This means that all finite state machines, registers, etc. in the design change state only on the rising edge of the clock signal.

RESET: When asserted, this input signal causes all registers to be forced low (zero) and all state machines to be reset. The system returns to the initial reset state one cycle later.

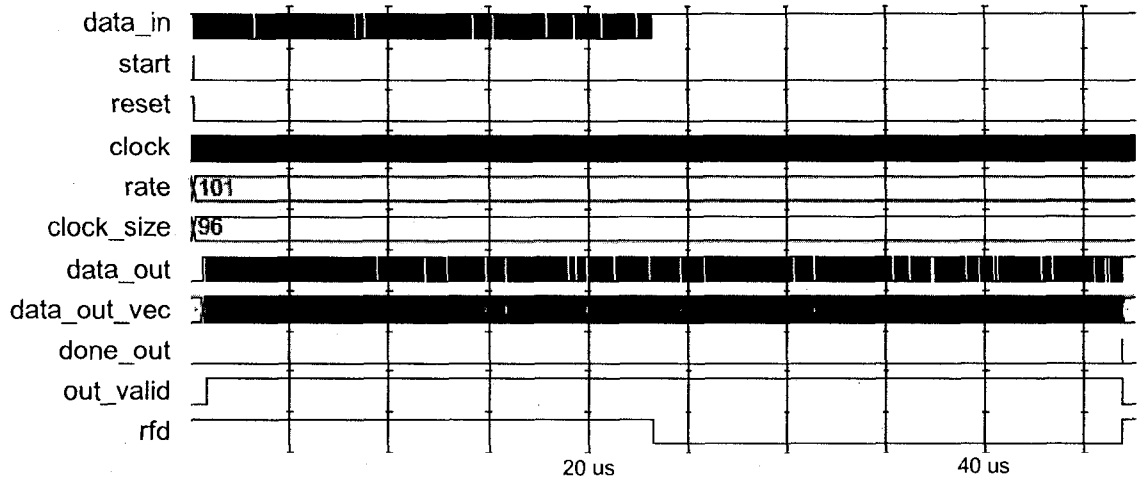


Figure 4.6: Encoding of one packet with code rate 1/2 and block size 96

4.2 Simulation and Testing Method

Testing and verification is the third stage of the overall design flow of the project. As described in Chapter 3, this stage consists of three processes, the first of which is behavioral simulations. The second and third processes are timing simulations and real-life hardware testing. The next three sections will detail these processes.

Each of the encoder's internal components is passed through behavioral and timing simulations as mentioned in Section 3.4.5. However, since the method and tools applied here are the same, we will limit our discussion to the overall system for brevity.

All simulation and testing processes utilize the same general structure presented in Figure 4.7. Software modeling as well as hardware modeling tools are employed. On the software front, we used MATLAB, Excel, and word processors. We developed several modules to automate the testing process. The *input_generate.m* module generates a user-defined random sequence of input bits. The input packets are then fed into both the hardware implementation and the software model. The output of the software encoder is compared with the output of the hardware implementation. The *packet_compare.m* (See

Appendix A) module performs the comparison and reports the number and location of encoding errors encountered as well as several other useful statistics. In order to run the test for several code rates and lengths, not to mention multiple input packets, the *Encoder_script.m* module loops through the required number of times altering the encoder's parameters as needed. The *finalEncoder.m* module, discussed in Chapter 3, behaves as a function that can be invoked by this test script.

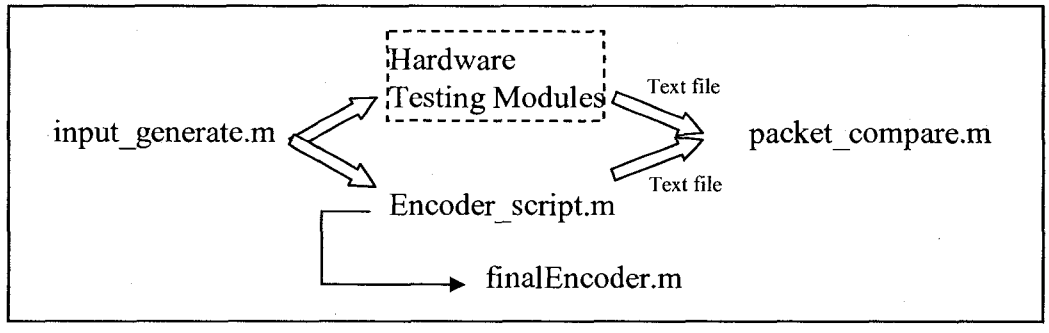


Figure 4.7: MATLAB testing structure

On the hardware front, we used different testing modules and tools depending on the simulation/test being performed. For behavioral and timing simulations we used the ModelSim XE III simulation tool, while real-life testing was performed on the XEM3010 hardware prototyping platform provided by Opal Kelly [87]. The details of the *hardware testing modules* are presented according to the corresponding simulation/testing process in the following three sections.

4.3 Behavioral Simulations

The aim of this process is to analyze the general behavior of the system and identify any defects or re-design requirements. Behavioral simulations do not include the timing information of the system, rather they focus on whether or not the system produces the correct results in an ideal environment. They are performed in conjunction with the hardware behavioral verification process, as well as the software functional verification process (see Section 3.4.2).

Using the ModelSim software tool, we developed several *testbenches* as our hardware testing modules. Testbenches are VHDL modules designed to act as a user for the design under test (DUT). They provide sources for all the ports necessary to operate and interface with the DUT. It is also possible to capture the output data and store it into text files for later processing. One of the most useful features of simulation software is the ability to examine the *waveforms* generated by the testbench modules and the DUT. This provides the ability to track the changes on every signal, bus, and register in order to verify the functionality of the design.

For our behavioral simulation purposes, we developed the *Encoder_tb.vhd* testbench, available in Appendix B. This module interfaces with the encoder's top-level VHDL module; and together with ModelSim, it simulates its behavior over a comprehensive set of run-time conditions and input values. We used a total of 100 input packets to be encoded with all supported code lengths and rates. The largest possible input packet occurs at rate 5/6 and block size 96, and is equal to 1920 bits. The resulting output encoded packet length is 2304 bits. Using the largest input packet length helps accommodate all other possible lengths. Therefore, the total input bit sequence required is

$$1920 \text{ bits/packet} \times 100 \text{ packets} = 192000 \text{ input bits} \quad (4.1)$$

Both the *Encoder_script.m* script and the *Encoder_tb.vhd* testbench include a code loop that feeds the encoder the generated 100 packets for each code length and rate. The following MATLAB pseudo code illustrates the idea:

```
% loop through all code rates
for i = 1:4
      if (i == 1)
            R = 1/2;
      elseif (i == 2)
            R = 2/3;
      elseif (i == 3)
            R = 3/4;
      else
            R = 5/6;
      end

      % loop through all block sizes in steps of 4
      for zf = 24:4:96

            % loop through all 100 input packets
            for i = 1:100

                  %%% Encoding %%%

            end
      end
end
```

While the MATLAB implementation is relatively direct, dealing with a large input sequence for a VHDL testbench provided a challenge. There are several approaches to accomplishing this. One involves creating an ultra large shift register initialized with the values of the 192000 input bits. Then, the bits can be shifted into the encoder during operation. However, the Xilinx ISE tools experienced difficulty handling this large testbench implementation. Another approach involves storing the input bits in an external text file. The bits can be read and shifted into the design during operation. However, the high count of text file accesses and the limited VHDL text file interaction tools make this approach less than ideal. Our solution involved creating a separate block RAM entity that

houses the entire input bits vector. The block RAM is not synthesized with the design and is strictly for testing purposes. This provided a simple and fast way of accessing the input bits multiple times while operating the encoder.

Initial behavioral simulation runs uncovered several design defects. We used the information provided by the simulation waveforms to refine the design. The final encoder design *passed all behavioral simulation runs correctly.* The figures presented in Appendix C depicts several captured waveforms of the comprehensive behavioral simulation discussed above, highlighting different aspects of the encoder's operation. Specifically, they present:

1. The waveform of the full behavioral simulation test.
2. A zoomed waveform illustrating the encoder working with rate 1/2.
3. A waveform of a full packet encoded with code rate 1/2 and block size 36.
4. A waveform of a full packet encoded with code rate 5/6 and block size 92.
5. A waveform illustrating the dynamic change of encoding rate.
6. A waveform illustrating the dynamic change of block size.
7. A waveform illustrating a packet-to-packet transition.

4.4 Timing Simulations

The aim of this process is to analyze the timing performance of the system. Timing simulations consider the effects of clock frequency, components mapping, and routing. They provide a fairly accurate idea of the performance of the design with a particular clock frequency, and assist in estimating the system's throughput and power consumption. Timing simulations are performed in conjunction with the hardware timing verification process (see section 3.4.2).

Unlike behavioral simulations, timing simulations are complex calculations that involve many parameters including the target device timing parameters, routing information, gate fan-out and fan-in, setup times, and hold times. Therefore, timing simulations require a considerably longer period of time to complete. It is not feasible to conduct the same comprehensive simulation runs used in behavioral mode. For our timing simulation purposes, we developed the *Encoder_tb_packet.vhd* testbench, available in Appendix B. This testing module interfaces with the encoder's top-level VHDL module; and together with ModelSim, it simulates timing behavior while encoding an input packet. The encoded output packet is captured and verified against the software model's results.

Our initial timing tests revealed a few errors. Most of them were fixed through ensuring some input signals are registered in the design. Adding the registers shortens the critical path and enhances the timing performance. We also dealt with several issues unique to timing simulations:

a. Unlike behavioral ones, timing testbenches must incorporate an artificial delay time when changing an input signal. This is necessary in order not to violate device hold times. We picked a delay of 1 ns for all changing input signals to the encoder.
b. It is necessary to define all signals from time zero. If not done, this will result in later undefined signal values, and hence simulation errors.
c. Setting constraints on the clock period may help the routing software achieve better timing results.

After making the minor modifications to the design and testbench, the final encoder passed the timing tests. Figure 4.8 depicts the waveforms generated by simulating the encoder running on a 50 MHz clock frequency with code rate 1/2 and block size 24. Figure 4.9 depicts the waveforms generated using code rate 5/6 and block size 96 with the same clock frequency. In addition, Figures 4.2 to 4.6 above represent further examples of timing simulations for the 802.16e LDPC encoder.

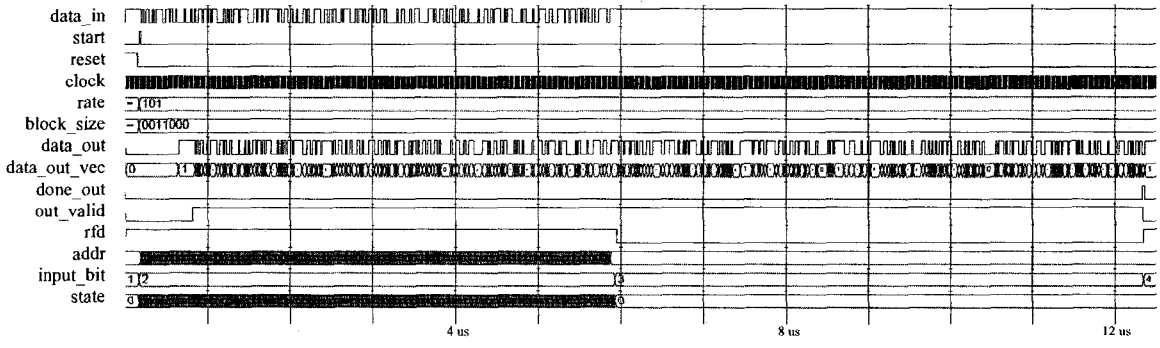


Figure 4.8: Post place-and-route simulation with code rate 1/2 and block size 24

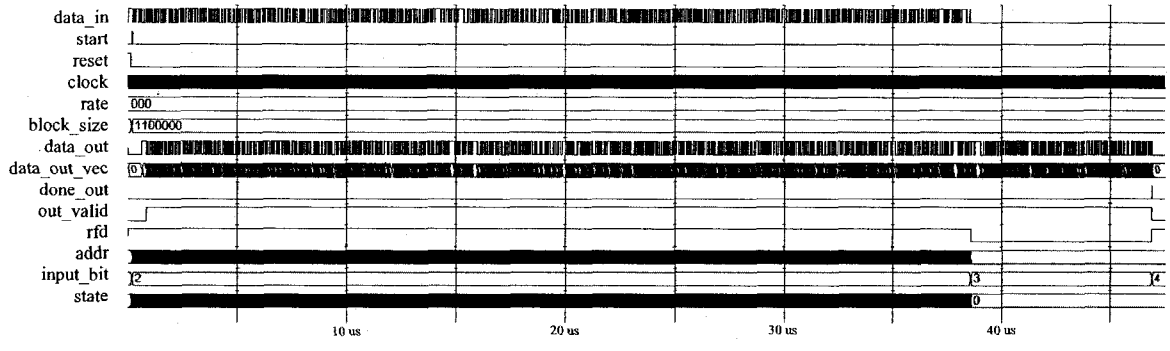


Figure 4.9: Post place-and-route simulation with code rate 5/6 and block size 96

Through analyzing the timing simulations and the various timing reports generated by the ISE suite, we determined that 80 MHz is the maximum clock frequency possible for the encoder design. Clock frequency figures and a brief discussion on the encoder's throughput are presented in the next section.

4.5 Resource Utilization and Throughput

4.5.1 Hardware Resource Utilization

The 802.16e LDPC encoder is implemented in two versions. The first implements the memory controller z_f multipliers using look-up tables (LUT), while the other version implements these multipliers using the available on-chip 18×18 dedicated multiplier units. The following two tables summarize the device utilization of the two versions on a Spartan-3 XC3S1500 FPGA chip.

Logic Utilization	**Used**	**Available**	**Utilization**
Number of Slice Flip Flops	5,266	26,624	19%
Number of 4 input LUTs	12,143	26,624	45%
Logic Distribution			
Number of occupied Slices	6,587	13,312	49%
Number of Slices containing only related logic	6,587	6,587	100%
Number of Slices containing unrelated logic	0	6,587	0%
Total Number 4 input LUTs	12,217	26,624	45%
Number of Block RAMs	6	32	18%
Number of MULT18X18s	0	32	0%
Number of GCLKs	1	8	12%
Total equivalent gate count for design	519,213		

Table 4.2: Spartan-3 XC3S1500 device utilization for LUT-based implementation

Logic Utilization	**Used**	**Available**	**Utilization**
Number of Slice Flip Flops	4,366	26,624	16%
Number of 4 input LUTs	11,169	26,624	41%
Logic Distribution			
Number of occupied Slices	6,032	13,312	45%
Number of Slices containing only related logic	6,032	6,032	100%
Number of Slices containing unrelated logic	0	6,032	0%
Total Number 4 input LUTs	11,183	26,624	42%
Number of Block RAMs	6	32	18%
Number of MULT18X18s	12	32	37%
Number of GCLKs	1	8	12%
Total equivalent gate count for design	547,221		

Table 4.3: Spartan-3 XC3S1500 device utilization for multiplier-based implementation

The multiplier-based encoder uses less device flip-flops, LUTs, and overall occupied slices. However, the total equivalent gate count is higher than the LUT-based encoder due to the use of dedicated multipliers. These optimized multipliers result in a slight speed increase as will be illustrated in the following sub-section.

4.5.2 System Throughput

The *throughput* of the encoder, can be measured in terms of bits per second or packets per second. It depends on three factors: the time to input a packet, the time to process a packet, and the minimum time to output a packet for a particular clock-rate. However, the coding algorithm dictates that the output packet is always longer than the input packet. And when the system is generating the parity bits, it cannot accept any input. Therefore, the rate at which packets can be input is the same as the output rate.

The encoder produces the output packet in two stages: first, the systematic bits are passed, unaltered, to the output after 32 latency clock cycles and, second, the parity bits are appended to the end of the systematic bits to create the output packet. The output packet is produced in one contiguous block. While the time to output a packet is equal to the length of the packet, the time to encode and output a packet is equal to the packet length plus 32 clock cycles. The latter affects the throughput of the system.

In order to calculate the maximum throughput of the system, the maximum possible clock frequency must be used. Table 4.4 presents the maximum clock frequency as reported by the various implementation stages. These figures reflect the fully routed design including input/output pins. It is expected that a higher clock frequency can be used for the encoder core when in "normal use" as an internal core of a larger design (i.e. excluding I/O pads and similar routes).

Process Name	LUT-based Encoder Clock Frequency (MHz)	Multiplier-based Encoder Clock frequency (MHz)
Synthesis	102.365	102.365
Map	82.864	87.374
Place and Route	79.962	80.263

Table 4.4: Maximum clock frequency as reported by implementation stages

Based on Table 4.4, we can average the maximum frequency to be 80 MHz. In this case the minimum time to output a maximum size packet (block size = 96) is

$$(2304 \text{ bits} + 32) / (80 \times 10^6) = 29.2 \ \mu s \tag{4.2}$$

The associated maximum throughput rate is 78.9 Mbits/s or 34246 packets/s. The throughput, however, varies with the packet length as illustrated in Table 4.5.

Output Packet Length (bits)	Time to encode and output packet (μs)	Throughput (Mbits/second)	Throughput (packets/second)
576	7.6	75.8	131578
672	8.8	76.4	113636
768	10.0	76.8	100000
864	11.2	77.1	89285
960	12.4	77.4	80645
1056	13.6	77.6	73529
1152	14.8	77.8	67567
1248	16.0	78.0	62500
1344	17.2	78.1	58139
1440	18.4	78.3	54347
1536	19.6	78.4	51020
1632	20.8	78.5	48076
1728	22.0	78.55	45454
1824	23.2	78.6	43103
1920	24.4	78.7	40983
2016	25.6	78.75	39062
2112	26.8	78.8	37313
2208	28.0	78.85	35714
2304	29.2	78.9	34246

Table 4.5: 802.16e LDPC encoder throughput for all supported packet lengths at 80MHz clock rate

4.6 Hardware on-chip Testing

The aim of this process is to confirm the operation of the system in hardware. As such, this is not a simulation process, rather a real-life on-chip test. After the timing

simulations are completed, the final circuit is generated into a *bit file* that is ready to be programmed unto an FPGA device for testing.

Our hardware testing module is the Opal Kelly XEM3010 experimentation board [87]. The board houses the Spartan-3 XC3S1500 FPGA chip along with a USB 2.0 micro-controller that allows high speed data communications with the chip. The board also features 32 MB of SDRAM and a phase lock loop (PLL) that is a multi-output clock generator. As shown in [88], Figure 4.10 depicts the block diagram for the XEM board showing the FPGA chip, the USB micro-controller, and the PLL among other components. The identifiers between brackets are the IC model numbers used.

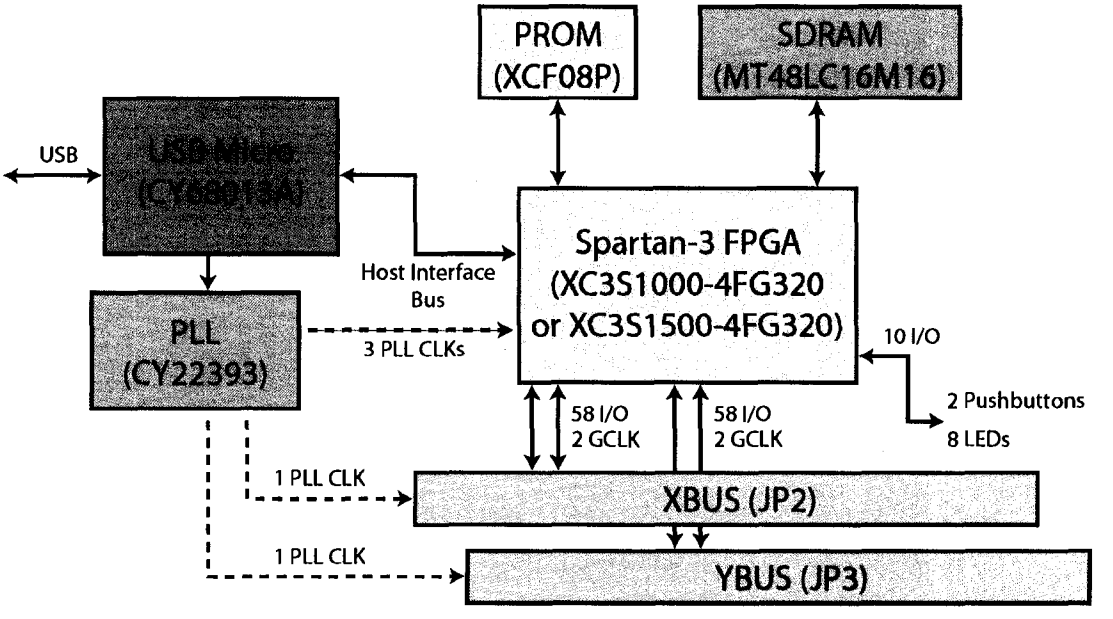


Figure 4.10: Functional block diagram of the XEM board, figure taken from [88]

In addition to the hardware, the XEM board supports a flexible software *application programmer's interface* (API) available in C++, Python, Java, and a Windows dynamic link library (DLL) versions. This API facilitates the communication between the host PC and the user's FPGA design. On the PC, one can develop a program that controls and transfers data to the FPGA design using any of the above languages. On the FPGA, Opal

Kelly provides several VHDL modules to handle the USB protocol and hide its details from the end user. Specifically, a *host interface module* is provided which communicates with the USB micro-controller at one end and a *shared bus* at the other. The bus connects to several entities named *endpoints*. Endpoints are VHDL modules that connect the shared bus to the user design's signals to be controlled or observed. According to their functionality, endpoints are classified into:

a. *Wires* that asynchronously transfer a signal's state into the design

b. *Triggers* that transfer a one-shot signal into the design synchronized to a particular clock.

c. *Pipes* that synchronously transfer multi-byte values into the design.

Figure 4.11 is a visual representation of the interface's structure. It is noted that the endpoint components are designated directions as IN or OUT depending on the user design signals they interact with.

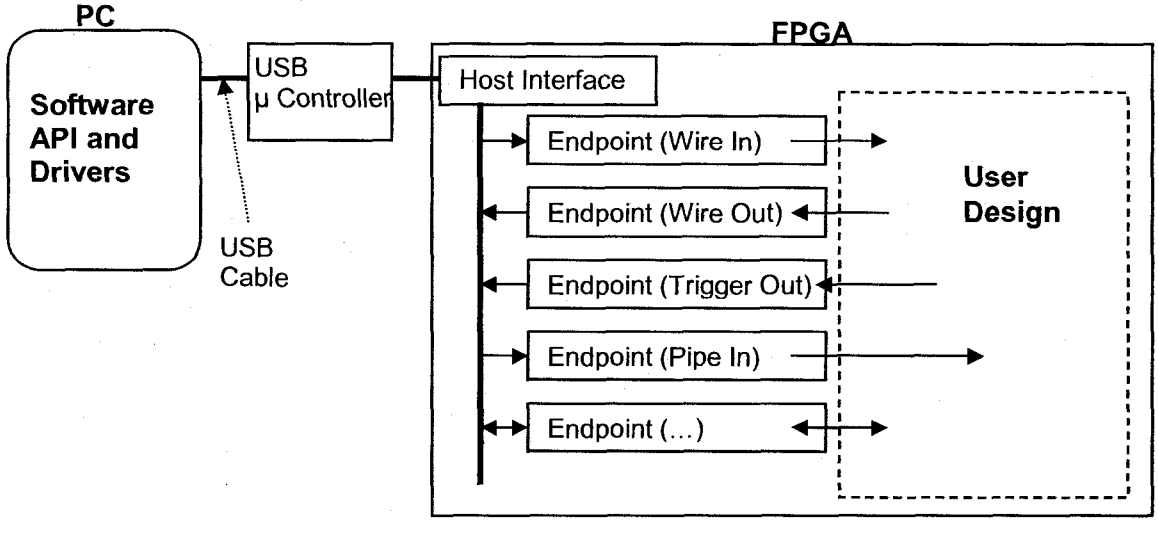


Figure 4.11: Opal Kelly interface structure

For our purposes we leveraged the XEM's flexibility to develop a software/hardware testing solution. Our testing platform can be divided into two sides:

- **On the PC side,** we developed the following:
  1. Text files holding the input test vectors (generated by MATLAB scripts). We used a similar input vector to the one used in the comprehensive behavioral simulations.
  2. A C++ controller file, provided in Appendix D, which handles the connection to the board, programming the device, supplying the test inputs, capturing the resulting encoded packets, and storing them into text files for verifications.
  3. A *Makefile* that compiles our C++ code with the rest of the API and produces the final software testing program.
- **On the FPGA side,** we developed a send/receive top-level VHDL module. The encoder does not provide buffering for input and output bits. Therefore we needed an extra VHDL module to store the input supplied by the C++ code and feeds it to the encoder, as well as capturing the serial output and buffering it before transfer back to the PC. The *TopLevel.vhd* file in Appendix B performs these tasks. Two extra block RAMs are used to host the incoming and outgoing packets.

  In addition, this top-level module connects several endpoints to the encoder. It is worth noting that Opal Kelly designed their components to use minimal resources, hence making a minimal impact on the user design's performance. Figure 4.12 presents the design of the top level module as described here.

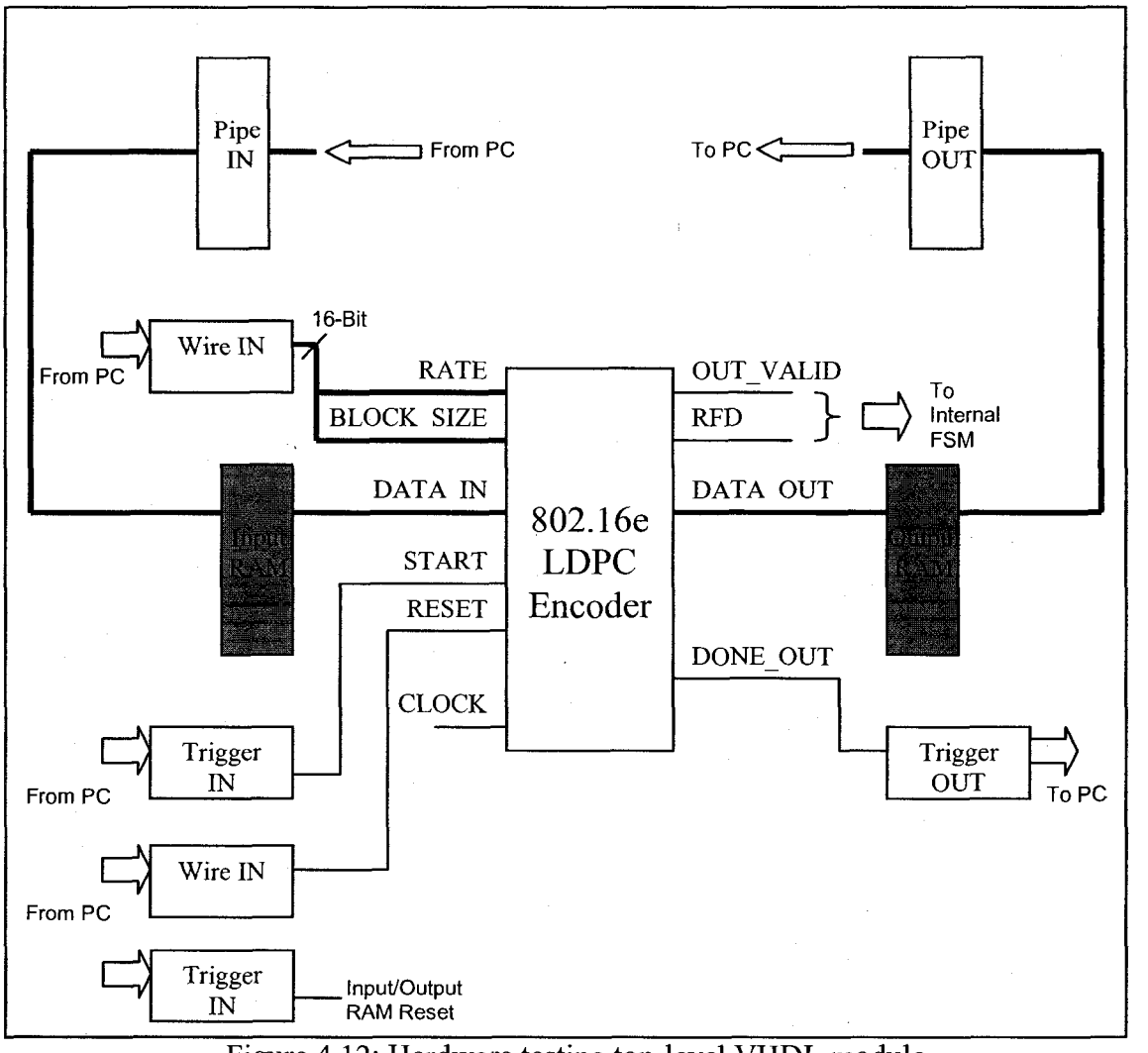


Figure 4.12: Hardware testing top-level VHDL module

The top-level module was synthesized, mapped, and routed before the final circuit is generated. The bit file was programmed unto the Spartan-3 chip via the C++ controller file. *The encoder passed all hardware tests* and perfectly matched the expected behavior of the software models. These tests concluded our testing and verification processes and provided a measure of confidence in the working encoder design.

4.7 Summary

In this chapter, we described the structure of the 802.16e LDPC encoder as an IP core. The usage of the core was explained by detailing the function and operation of the various data and control signals.

In order to verify the functionality of the encoder, we passed it through three testing processes; behavioral simulation, timing simulation, and hardware testing. Both simulation processes confirmed the expected functionality and provided insight into the encoder's resource utilization and performance. While the encoder's throughput depends on the selected code length, the maximum throughput was calculated to be 78.9 Mbps at a code length of 2304 bits and a maximum clock frequency of 80 MHz.

The Opal Kelly XEM3010 hardware prototyping platform was used to implement the design on an FPGA chip. The design passed all hardware tests and matched the operation of the software reference model.

Chapter 5

LDPC Encoders for In-Vivo RF Communications

5.1 Introduction

As introduced in Chapter 2, wireless implantable medical devices (IMDs) are part of a revolution of innovative patient treatment and monitoring solutions. One of the main challenges faced by wireless IMD designers is the low power consumption requirement. Driving this requirement is the desired lengthy operation without the need for recharging, the ability to incorporate multiple functions, and patient health considerations. This chapter explores the suitability of LDPC encoder implementations for in-body communications. We focus on our 802.16e LDPC encoder, as well as a convolutional LDPC encoder, and contrast the power results with industry leading implantable devices. Finally, options for improving the 802.16e encoder's power consumption to better fit these applications are presented.

5.2 Power Measurement Tools

To evaluate the LDPC encoders' power consumption, the following power measurement tools and methodology are utilized.

The LDPC encoders under study are implemented using Xilinx design tools and target Xilinx FPGA devices. Therefore, the integrated "Xilinx XPower – Power Analyzer" tool is a natural choice to perform power measurements. Furthermore, the

802.16e LDPC encoder utilizes specific Xilinx components which can only be evaluated with their tools.

XPower is part of the Xilinx ISE software suite and can be used to estimate the power consumption of a VHDL design post place-and-route. According to [89] the XPower tool, reports the total power as:

$$\text{Total Power} = \text{Static Power} + \text{Dynamic Power} \tag{5.1}$$

Here static power is the power consumed by the chip in *quiescent state*, that is, with *no signals toggling,* and it is due to leakage currents of the transistors on the chip. Quiescent power depends on the power supply, junction temperature, die size, and process variation, and therefore is probabilistic with a typical value [90]. Power estimation software usually reports the typical static power value for a set of device parameters, and this value remains constant for all designs on the device. Dynamic power is the additional power consumption caused by the switching activity of the signals in the design and the associated capacitive loads charging and discharging. XPower reports the static (quiescent) power as well as the dynamic power of the design.

XPower calculates power based on the concept that dynamic power consumption in CMOS circuits is primarily due to switching activity. Each element (LUT, FF, BRAM, routing segment) that can switch has a capacitance model associated with it. Clock signals and primary input signals are assigned specific frequencies by the designer. Synchronous elements are assigned *activity (or toggle) rates relative to their associated clock.* User-supplied activity rates combine with device-specific capacitance models, static power, and other data to produce a power estimate for a design.

The accuracy of the switching activity data is crucial in obtaining an accurate estimate of power consumption. XPower calculates power as a summation of the power consumed by each element in the design, which is given by:

$$P = C \times V^2 \times E \times F \tag{5.2}$$

where P is the power in Watt, C is the capacitance in Farads, V is the voltage in Volts, E is the switching activity (i.e. average number of rising transitions per clock cycle), and F is the frequency in Hz. Here the capacitance is determined for the specific design that is implemented on a specific device, and it is usually fixed during the characterization of the routing resources of the design. The voltage is a fixed value for a specific device set by default in the XPower interface. F × E is the total number of transitions for a specific element; where frequency, or the activity rate of each signal in a design, is the most variable element of the above equation. XPower incorporates the switching activity of elements in the design through using any of the following data supplied by the user:

- Global default activity rate.
- Simulation results stored in a *value change dump* (VCD) file.
- Activity rates manually entered through XPower's graphical user interface (GUI).

After the VHDL designs are synthesized and routed using the Xilinx ISE suite, place-and-route simulation models are generated. These models are then used with Mentor Graphics' "ModelSim" software to simulate the place-and-route behavior of the design. The simulations record the switching behavior of all signals and components involved in the design and in VCD files. The VCD files are then supplied to XPower yielding the design's power estimate.

In order to evaluate and compare power estimates of encoder implementations, the measurement environment was made as fixed as possible. This could be summarized with:

Input: Randomly generated input.

FPGA device: Spartan-3 FPGA package 3s1500fg320.

HDL: VHDL.

Synthesis Tool: Xilinx XST (part of the ISE suite).

Routing Tool: Xilinx ISE suite.

Simulation Tool: Mentor Graphics ModelSim.

Simulation Frequency: 50 MHz.

Power Estimation Tool: Xilinx XPower.

The XPower interface can be used to modify the clock frequency to obtain more measurement points. 100MHz and 250MHz clocks were used. In addition a single measurement with a 50% global activity rate as a default value for all unspecified signals was also recorded.

5.3 Power Measurements of the 802.16e LDPC Encoder

As presented in Chapter 3, the 802.16e LDPC encoder supports 6 code rates with 19 block sizes each. The minimum codeword length is 576 bits and corresponds to a block size of 24 bits, while the maximum codeword length is 2304 bits and corresponds to a block size of 96 bits. The largest amount of user information per packet is encoded using the largest code rate (5/6), while the least amount is encoded with the smallest supported

code rate (1/2). Therefore, code rates 1/2 and 5/6 along with block sizes 24 and 96 were used for the power estimates in order to capture the minimum and maximum.

Code Rate / Frequency (MHz)	1/2		5/6	
	Block size = 24 (n = 576)	**Block size = 96 (n = 2304)**	**Block size = 24 (n = 576)**	**Block size = 96 (n = 2304)**
50	177	175	181	180
100	204	203	212	212
250	286	285	306	307
50 MHz *with 50% default activity rate*	235	233	239	238

Table 5.1: 802.16e LDPC encoder power estimates, power in (mW)

The quiescent power reported by XPower was 142mW for this FPGA device. This value remained constant for all block sizes, rates, and clock frequencies. As observed in Table 5.1, the dynamic power increases as the clock frequency is increased. This is expected as all elements in the design toggle faster at high frequency adding to the total power consumption. However, the dynamic power varied slightly between the block sizes. This is because XPower calculates the switching activity of all elements including input and output ports as *percentages* relative to clock frequency, which results in slightly different activity rates.

An increase in power was observed with the higher code rate. This is because the larger amount of input processed by the system at a higher code rate results in higher switching activity, hence higher dynamic power consumption. Despite the fact that higher code rates require less system resources, the larger input processing effect is more visible on the overall power consumption. These trends are illustrated in Figure 5.1.

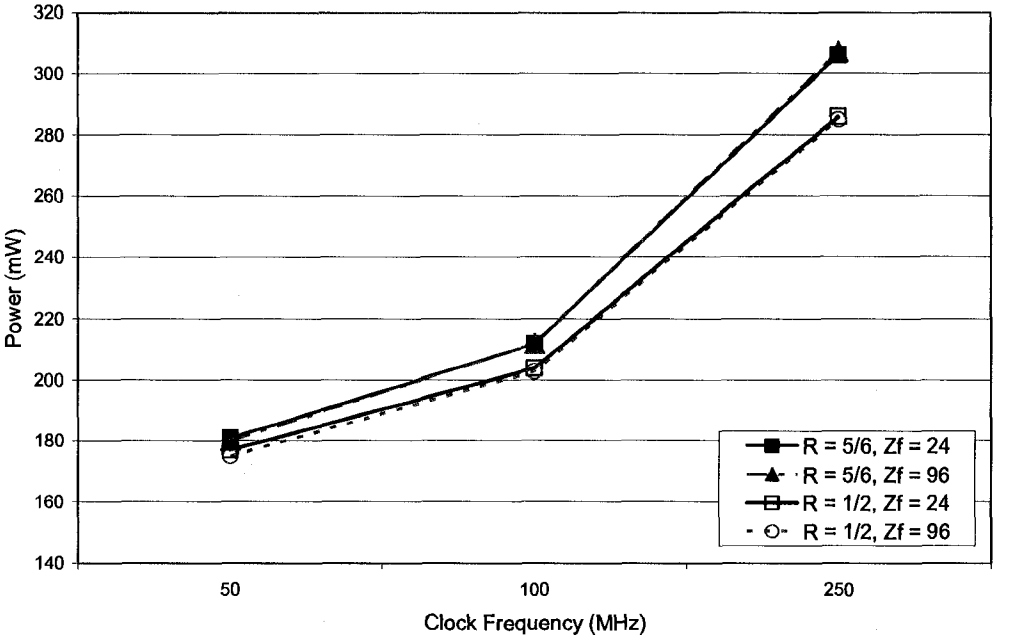


Figure 5.1: 802.16e LDPC encoder power estimates with rates 1/2 and 5/6, block sizes 24 and 96 at several clock frequencies

From the results above, it could be concluded that the system's power consumption is between the rate 1/2 values and the rate 5/6 values. For example at 50 MHz and under normal conditions the average is 178 mW. Finally, an observation is made regarding setting the global activity rate to 50% of the clock frequency. XPower uses this default value for elements not specified by the VCD file, increasing the total activity rate and power consumption of the system. This is a pessimistic rate that was observed to be higher than normal operation and is an approximated indicative of a worst case power usage at that particular frequency.

5.4 The Convolutional LDPC Encoder

5.4.1 Introduction to the Convolutional Encoder Implementation

The convolutional encoder used for this study is based on the design presented in [91]. As per a classic convolutional code, any output code bit is generated using previous

input bits and previous code bits. Hence, an important aspect of the convolutional LDPC (LDPC-CC) encoder is its memory, which indicates how many previously generated bits the encoder can store (and hence remember) to generate the current code bit. The performance and circuit complexity of the encoder increase as the memory increases. This particular design is a rate 1/2 encoder with memory M = 128. Figure 5.2 illustrates the architecture of the encoder.

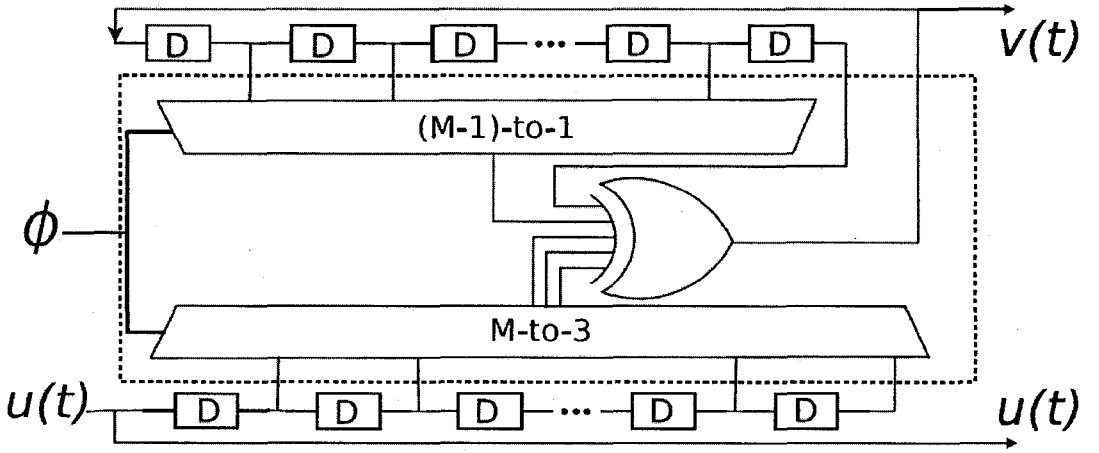


Figure 5.2: Block diagram of LDPC-CC encoder, figure taken from [91]

From [91], the LDPC-CC code-bit generation equation, for a rate 1/2 code, at time t = nM + ϕ can be written as:

$$v(nM+\phi) = v((n-1)M+\phi) + v(nM-\delta^{(v)}(\phi)) + \sum_{i=1}^{3} u(nM-\delta_i^{(u)}(\phi)) \qquad (5.3)$$

Here, n $\geq$0, ϕ is the phase and can take a value in {0, 1, 2, $\cdots$, M $-$ 1}, $\delta^{(v)}(.)$ and $\delta_i^{(u)}(.)$ are functions of the code which can take values in {1, 2, $\cdots$,M $-$ 1} and are related to its parity-check matrix. The (+) sign implies an XOR operation. u(t) represents the information bit at time t. For further details on LDPC convolutional codes see [4].

The LDPC convolutional encoder system described above, is implemented using VHDL. XOR gates are utilized to perform all parity-check operations and shift registers implement the memory functionality. The final system is synthesized for Xilinx FPGA

devices and is placed and routed targeting the Spartan-3 chip series. Behavioral simulations and post place-and-route simulations were performed to provide basis for power estimation.

LDPC-CCs have a disadvantage when comparing them to the block ones; that is the need for termination. Like traditional convolutional codes, any encoded sequence needs to be terminated, returning the encoder to the initial (known) state. This is usually done by transmitting a special termination bit sequence that does not convey any user information. The necessary termination sequence circuit adds to the circuit complexity and hardware overhead of the overall LDPC-CC encoder. The above convolutional encoder does not include the termination circuit necessary to generate a termination sequence. A solution for such circuit was described in [92] and could be implemented efficiently on FPGA devices. Figure 5.3 illustrates the proposed termination circuit around the original encoder. At the time of this study the VHDL implementation for the encoder with termination was not available, therefore power measurements are performed using the original encoder, and reported resource and area data are used to estimate the termination circuit's effect on the overall power consumption.

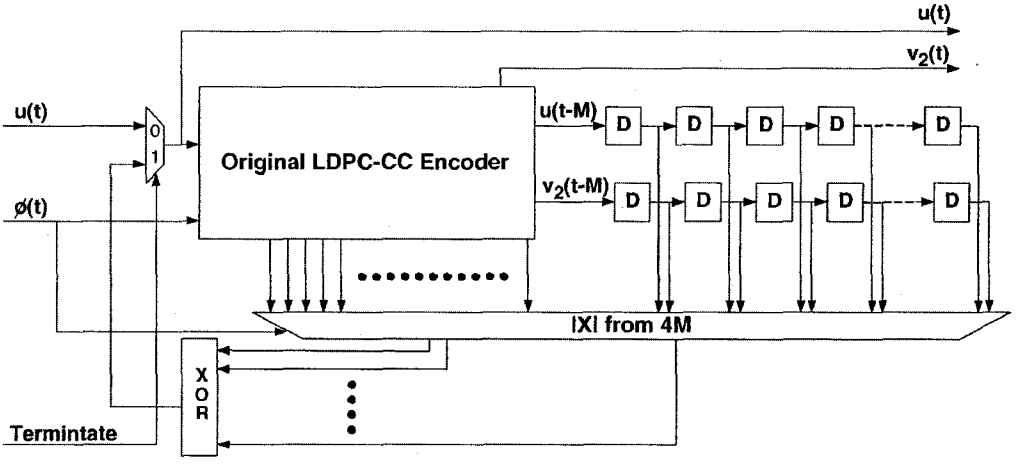


Figure 5.3: Block diagram of termination sequence generation circuit of LDPC-CC encoder, figure taken from [92]

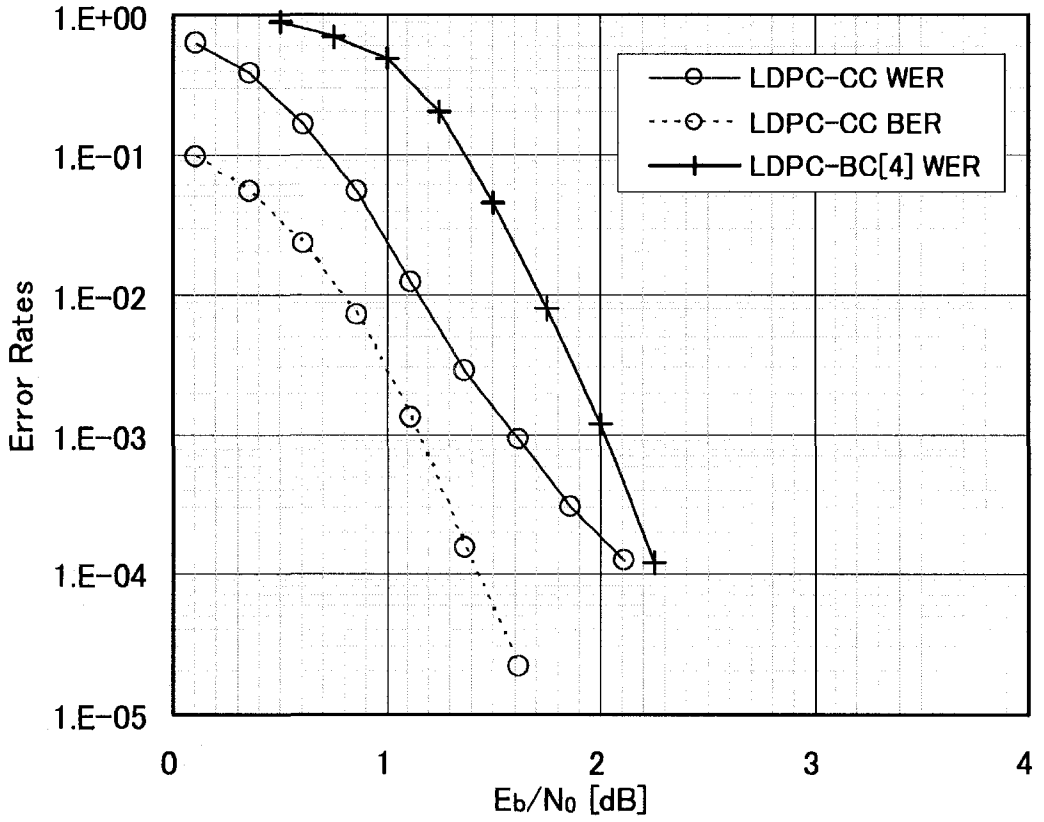


Figure 5.4: 802.16e LDPC-BC vs. LDPC-CC performance example. (Information bit length = 600 bits), figure taken from [93]

Before comparing the power consumption of the two encoders it is beneficial to briefly present the coding performance of 802.16e LDPC-BC codes compared to convolutional counterparts. The authors of [93] argue the benefits of including LDPC-CC codes into the currently under-development 802.16m standard, and they use the LDPC-BC codes supported in the 802.16e standard as examples. The performance of the 802.16e codes, as reported by [94], is compared to LDPC-CC versions derived from the block ones using the literature and method proposed by Ali Pusane, *et al.* [95]. As observed in Figure 5.4 above, The LDPC-CC codes perform better than the LDPC-BC ones for the simulated range. The authors of [93] further conclude that in addition to providing comparable performance, LDPC-CC codes hold important decoding and encoding complexity advantages and, therefore, are good 802.16 FEC candidates.

5.4.2 Power Measurements of the Convolutional LDPC Encoder

The convolutional encoder used for this study is a (128,3,6) encoder. Where 128 is the memory of the system and (3,6) indicate code rate 1/2. Therefore only rate 1/2 measurements were possible. In order to facilitate comparison with the block encoder, two input vector lengths were used, specifically 288 bits for n = 576, and 1152 bits for n = 2304. The results are summarized in Table 5.2.

Code Rate / Frequency (MHz)	1/2	
	n = 576	n = 2304
50	160	162
100	174	178
250	216	225
50 MHz *with 50% default activity rate*	160	162

Table 5.2: LDPC-CC encoder power estimates, power in (mW)

As with the block encoder, the quiescent power reported by XPower was 142 mW for this FPGA device. This value remained constant for all measurements. The dynamic power followed a similar trend as in the block encoder, increasing as the clock frequency increased and varying slightly between the code lengths.

From these results, we could characterize the system's average power consumption. For example at 50 MHz and under normal conditions the average is 161 mW. When setting the global activity rate to 50 % of the clock frequency, XPower uses this value for elements not specified by the VCD file. Unlike the block encoder which has to support many code rates, this encoder uses one code rate and an array of XOR gates to implement the parity-check operations, and these gates are all in use during regular encoding operation. Therefore, there are very few un-specified signals and elements. And setting a

default activity rate does not contribute any significant extra power consumption. The LDPC convolutional encoder's power measurements are graphed in Figure 5.5.

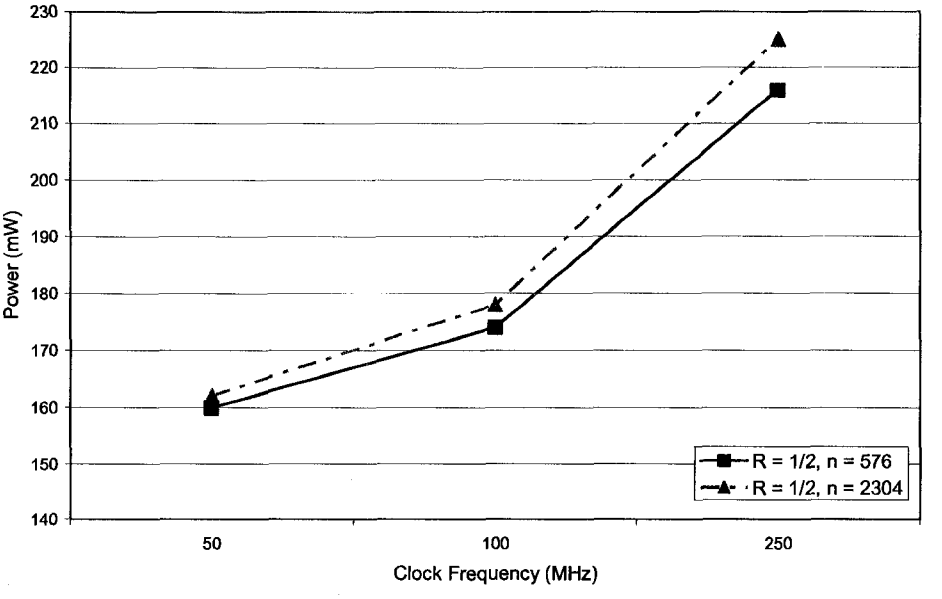


Figure 5.5: LDPC-CC encoder power estimates with rates ½ at several clock frequencies

5.5 Power Consumption Comparison of the LDPC-BC and LDPC-CC encoders

In this section, we will compare the power estimates obtained for the 802.16e LDPC encoder and the convolutional LDPC encoder. Table 5.3 below combines the power results for the two encoders at rate 1/2.

Encoder / Frequency (MHz)	LDPC Block Encoder		LDPC Convolutional Encoder	
	R = 1/2, n = 576	R = 1/2, n = 2304	R = 1/2, n = 576	R = 1/2, n = 2304
50	177	175	160	162
100	204	203	174	178
250	286	285	216	225
50 MHz *with 50% default activity rate*	235	233	160	162

Table 5.3: LDPC-CC vs. 802.16e LDPC encoder power results for rate 1/2, power in (mW)

It is evident that the power consumption of the convolutional encoder is lower than that of the block encoder at all frequencies. Figure 5.6 illustrates this power difference.

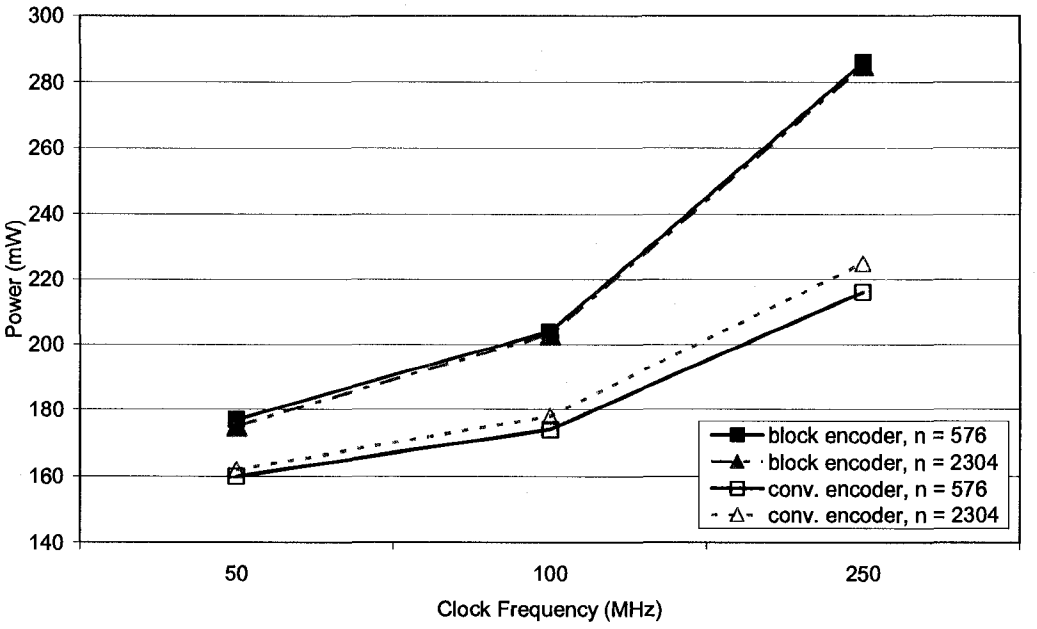


Figure 5.6: LDPC-CC vs. 802.16e LDPC Encoder power results at rate 1/2

Given that the static power component of both encoders is the same at 142 mW, power gains are achieved entirely with the dynamic component. We could define the power gain achieved with the convolutional encoder as the percentage reduction of dynamic power consumed by the convolutional encoder relative to that consumed by the block encoder. Therefore, looking only at the dynamic power, the convolutional encoder power gain can be defined by Equation 5.4:

$$C.\ E.\ Power\ Gain = [(B.\ E.\ Dyn.\ Power - C.\ E.\ Dyn.\ Power) \div B.\ E.\ Dyn.\ Power] \times 100 \quad (5.4)$$

where C.E. is the convolutional encoder and B.E. is the block one. The convolutional encoder achieves 40-49 % power gain over the 802.16e encoder as presented in Table 5.4.

Encoder / Frequency (MHz)	LDPC Block Encoder		LDPC Conv. Encoder		Conv. Encoder Power Gain	
	R = 1/2, n = 576	R = 1/2, n = 2304	R = 1/2, n = 576	R = 1/2, n = 2304	R = 1/2, n = 576	R = 1/2, n = 2304
50	35mW	33mW	18mW	20mW	49%	39%
100	62mW	61mW	32mW	36mW	48%	41%
250	144mW	143mW	74mW	83mW	49%	42%

Table 5.4: Convolutional encoder power gain (savings) based on dynamic power consumption

These results are further supported by the hardware resource utilization for both designs. As reported by the ISE tools, the block encoder uses 19% of available Slice flip-flops and 45% of 4-input lookup tables occupying 49% of available slices on the Spartan-3 device. This is a much higher hardware usage than the convolutional encoder which posts 1% for all of the above three categories. Table 5.5 is a comparison of the resource utilization for the two designs.

Encoder / Category	LDPC Block Encoder	LDPC Convolutional Encoder
Slice Flip-Flops	5,266 of 26,624 *(19%)*	268 of 26,624 *(1%)*
4-input LUTs	12,143 of 26,624 *(45%)*	220 of 26,624 *(1%)*
Occupied Slices	6,587 of 13,312 *(49%)*	252 of 13,312 *(1%)*
Block RAMs	6 of 32 *(18%)*	0 of 32 *(0%)*
Total equivalent gate count	519,213	3,663

Table 5.5: FPGA resource utilization comparison, Spartan 3 – package 3s1500fg320

The circuit complexity of the block encoder comes from its compatibility with the IEEE 802.16e standard. The full support of all code rates and block lengths specified in the standard along with the ability to switch between rates and lengths on-the-fly requires a high degree of flexibility from the design. The design is an intricate network of components such as FIFO buffers, shift registers, circular-pointers, XOR adders, memory controllers, etc. Furthermore, each code rate is associated with a corresponding parity-check matrix. These matrices are further split into sub-matrices and stored in the chip's

block RAM units which in turn require hardware resources to implement and route their bus inputs and outputs. But perhaps the most resource consuming aspect of the design is explained by the very definition of block encoders. The 802.16e LDPC encoder must receive the fixed length input and build the output code word using all the information bits. This means processing large numbers of bits during operation. The largest input packet length in this design is 1152 bits, requiring matching length shift and storage registers as well as XOR adders.

By contrast, the LDPC-CC encoder is a relatively simple design. It is a uniform bank of XOR gates with phase-controlled multiplexing. The encoded output is produced on a separate output port using the equations specified above and with no buffering. Since the encoder supports one code rate and no specific packet length, no flexibility complexity is incurred. The design implements the parity-check matrix with the multiplexed XOR gates, therefore no block RAM storage is required either.

At a first glance, the simpler convolutional encoder with low hardware usage and low power consumption is an obviously more attractive solution for low power applications than the block encoder. However, it is important to include the termination circuit discussed in Section 5.4.1 to obtain a more accurate comparison. The VHDL implementation of the LDPC-CC encoder with termination was not available for this report. However from [92] and experimental results, estimates could be extrapolated. As tabulated in [92], the LDPC-CC encoder with termination requires approximately 7 times the hardware resources required by the encoder alone. ASIC implementations of this design show similar area scaling with the terminated encoder occupying 8 times the area of the un-terminated one. However, power consumption may not scale with the same

ratio, rather from several results obtained at the VLSI lab in the University of Alberta, the terminated ASIC encoder consumes 2-3 times more power than the un-terminated one. If these ratios are applied to the dynamic power component of the above FPGA designs, the convolutional encoder's power results will approach those of the block encoder.

5.6 LDPC Encoders Analysis for Wireless Implantable Medical Devices

5.6.1 Power Efficiency of LDPC Encoders for Wireless In-Body Communications

The high attenuation and varying nature of the wireless channel through the human body presents the need for integrating error correction mechanisms to enhance the reliability of transmitted data. As mentioned in Chapter 2, it is safe to assume that a raw wireless data transmission for a typical implant may not achieve a better BER than 1 x 10^{-3}. However, a power tradeoff exists. The integration of ECC encoders/decoders adds to the circuit complexity of the implant and increases power consumption levels. While coding results in lowering the signal-to-noise ratio required at the receiver to achieve reliable transmissions. This means to achieve a similar BER, coding lowers the required transmit signal power, thus ultimately resulting in lower overall power consumption. This tradeoff presents a couple of questions:

- How power-efficient is a certain ECC code for in-vivo wireless links?
- What is the minimum transmit power required to achieve reliable in-vivo transmission with a certain ECC code?

We will attempt to answer these questions concentrating on the encoders discussed in this chapter.

A typical wireless medical implant communicates with an external monitor. MICS rules specify that implant transmitters must operate in the 402-405 MHz band while maintaining an effective isotropic power level less than 25uW (or -16dBm). This is a limit to the radiated power out of the human body, not necessarily the transmit power of the implant. In addition, this restriction suggests that the *uplink,* from the implant to the monitor, is the power consumption critical path. In section 2.4.2, we mentioned that the minimum path loss through the human body was approximately 23 dB with a maximum of 50 dB. This implies that *maximum implant transmit power* should range between 5mW to 2.5W in order not to exceed the MICS limit. However the *minimum implant transmit power* (not exceeding the MICS limit) required to achieve reliable communications is a more important factor to this study. We will derive a formula to evaluate the minimum transmit power of an implant with and without the use of ECC codes. This method is based on the studies carried out in [96] and [97].

There are several sources of signal loss that may encounter a transmission from the transmitter to the receiver:

- *Body path loss*: This is the attenuation of the signals through the human body as discussed in Chapter 2. A definitive in-body path loss model does not exist, however simulation studies could approximate the body path loss factor. As per [96], we will use 50 dB as the body path loss (BPL) factor at 403 MHz.
- *Free-space path loss*: Attenuation through free-space in a line-of-sight can be modeled with Friis' formula:

$$P_{RX} = P_{TX} G_T G_R \left(\frac{\lambda}{4\pi d} \right)^2 \text{ [W]} \tag{5.5}$$

Where P_{RX} and P_{TX} are the power at the receiver and transmitter respectively, and G_T and G_R are the transmitter and receiver antenna gains respectively. λ represents the wavelength used and d is the separation distance between the Tx and the Rx which is assumed to be larger than the far field $(\lambda/2\pi)$.

- *Reflections losses*: These are the extra losses due to reflections and non-line-of-sight transmission. These losses can be modeled through the inclusion of a *path loss exponent* to the separation distance in Friis' formula. The power at the receiver according to the modified Friis' formula becomes:

$$P_{RX} = P_{TX} G_T G_R \left(\frac{\lambda}{4\pi}\right)^2 \frac{1}{d^n} \text{ [W]} \tag{5.6}$$

- *Noise at the receiver*: This noise originates from two sources:
  - Thermal noise, which is equal to k×T×B, where k is Boltzman's constant, T is the temperature in Kelvin, and B is the signal bandwidth.
  - Receiver noise figure (RNF), which is receiver dependent.

  The total noise at the receiver (N) becomes:

$$N = k \times T \times B \times 10^{(RNF/10)} \text{ [W]} \tag{5.7}$$

After identifying the noise sources, we can define the signal power to noise power ratio (S/N) keeping in mind that S/N is different than, but related to, SNR. The later is a quantity used frequently in coding theory and is equal to $10\log_{10}(E_b/N_o)$.

$$\frac{S}{N} = \eta \frac{E_b}{N_o} = \eta 10^{SNR/10} \tag{5.8}$$

Where η is the spectral efficiency measured in bits/Hz, E_b is the energy per bit, and N_o is the noise power spectral density.

The signal power at the receiver can be used with the noise at the receiver to re-write Equation 5.8. Thus, substituting Equations 5.6 and 5.7 we get:

$$\frac{S}{N} = \frac{P_{RX}}{kTB10^{RNF/10}} = \eta 10^{SNR/10} \tag{5.9}$$

Adding the body path loss factor $10^{BPL/10}$ and re-arranging Equation 5.9 (assuming G_T and G_R to be unity), we get the minimum *transmit power that achieves SNR at the receiver*:

$$P_{TX} = \left(\frac{4\pi}{\lambda}\right)^2 d^n kTB\eta 10^{(SNR+RNF+BPL)/10} \text{ [W]} \tag{5.10}$$

To compute the energy per bit, we divide the transmit power by the data throughput R. However since η = R/B, the energy per bit equation reduces to:

$$E_{TX} = \frac{P_{TX}}{R} = \left(\frac{4\pi}{\lambda}\right)^2 d^n kT10^{(SNR+RNF+BPL)/10} \text{ [J/bit]} \tag{5.11}$$

If the transmitted data is *uncoded*, it would require an SNR of 10.4 dB to achieve a BER of 10^{-6} [96]. Therefore using this SNR value with the above equations provides a measure of the minimum required transmit power, or energy per bit, to achieve this BER level. However, if an ECC code is employed, the SNR required (SNR_c) will be lower, and the coded spectral efficiency becomes $\eta_c = R_cR/B$ where R_c is the code rate. Thus from Equation 5.10, the *minimum required transmit power for a coded system with code rate* R_c is:

$$P_{TX,C} = \left(\frac{4\pi}{\lambda}\right)^2 d^n kTB_c\eta_c 10^{(SNR_C+RNF+BPL)/10} \text{ [W]} \tag{5.12}$$

And consequently, the transmit energy per bit becomes:

$$E_{TX} = \frac{P_{TX,C}}{R_c R} = \left(\frac{4\pi}{\lambda}\right)^2 d^n kT 10^{(SNR_C + RNF + BPL)/10} \quad [\text{W}] \tag{5.13}$$

Or

$$E_{TX,C} = E_{TX,U} 10^{(SNR_C - SNR_U)/10} \quad [\text{J/bit}] \tag{5.14}$$

Where $E_{TX,U}$ is the same quantity defined through Equation 5.11, SNR_U is the uncoded SNR, and The term SNR_C-SNR_U is called the *coding gain.*

A final step to this derivation is accounting for the extra power cost of the encoder/decoder at the implant. For the purposes of our study, we will consider the scenario where only an encoder is integrated at the implant. It has been shown that including the decoder hampers the energy efficiency requiring 10-100 times more power at short distances, as the decoder typically consumes 2-3 orders of magnitude more power than the encoder [96]. Furthermore, the monitoring device rarely transmits and, unlike the implant, is not power constrained. Thus the monitor's signal power may be raised more freely to counter the effects of noise. A very simple wake-up receiver circuit may be included in the implant to accommodate for the monitor's commands. To get the *total minimum transmit energy per bit* we add the extra power cost of the encoder (P_{enc}) to Equation 5.14:

$$E_{Total} = E_{TX,U} 10^{(SNR_C - SNR_U)/10} + \frac{P_{enc}}{R_c R} \quad [\text{J/bit}] \tag{5.15}$$

Equation 5.15 can be used with Equation 5.11 to evaluate the power efficiency of ECC codes compared to uncoded systems at various distances. We performed this evaluation for the LDPC-CC and LDPC-BC encoders discussed in previous sections.

In addition to the power consumption of the encoder, we need to specify several other parameters required in Equations 5.11 and 5.15. The values are chosen similar to the ones

used in [96] and [97], and are presented in Table 5.6. It is noted that while the encoders actually run at much higher frequencies of 50MHz or 100MHz, the data throughput rate is limited by the MICS bandwidth to only 300 kbps, therefore it is assumed the encoders run for 0.6% of the time (at 50 MHz). In Table 5.6, PLE stands for the path loss exponent.

T, K°	B, kHz	R, kbps	RNF, dB	BPL, dB	*f*, MHz	PLE *n*
300	300	300	5	50	403	3

Table 5.6: Parameters used in power calculations

The SNR values at BER of 10^{-6} for both encoders are also required to reflect the coding gain achieved using the LDPC-CC or the LDPC-BC code. Without an exact hardware testing result for an encoder/decoder pair, approximation is needed. The SNR performance of a certain LDPC code depends on several factors including code rate, code length, modulation technique, channel model, decoder implementation and iteration count, etc. However for the purposes of this study, we use approximate SNR results for systems presented in literature and show relevant resemblance to our LDPC codes. For the block encoder, we approximate the SNR to be 3.8 dB for rate 5/6 and 2.1 dB for rate 1/2. These values reflect the 802.16e code performance by the systems presented in [98, 99, 100]. For the convolutional encoder, we approximate the SNR value to be 3.2 dB [96]. We initially evaluate the energy efficiency of the encoders in their current state as FPGA cores. The XPower estimates summarized in Table 5.3 are used with the appropriate SNR values to plot the minimum transmit energy for the coded system compared to the uncoded one versus varying distance from 25 cm to 10 m. Figure 5.7 compares the block encoder FPGA core, at code rate 1/2, to the uncoded system. Figure 5.8 evaluates the convolutional encoder's energy efficiency at the same code rate.

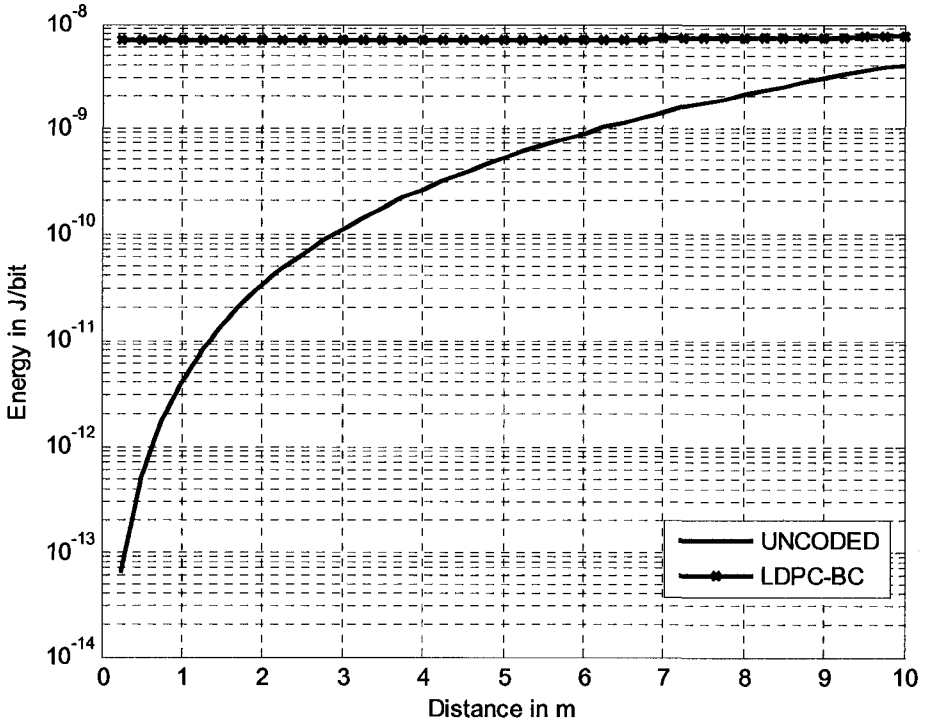


Figure 5.7: Minimum transmit energy per bit for 802.16e LDPC encoder (FPGA core) compared to an uncoded system, at rate 1/2, frequency 50 MHz, BER of 10^{-6}

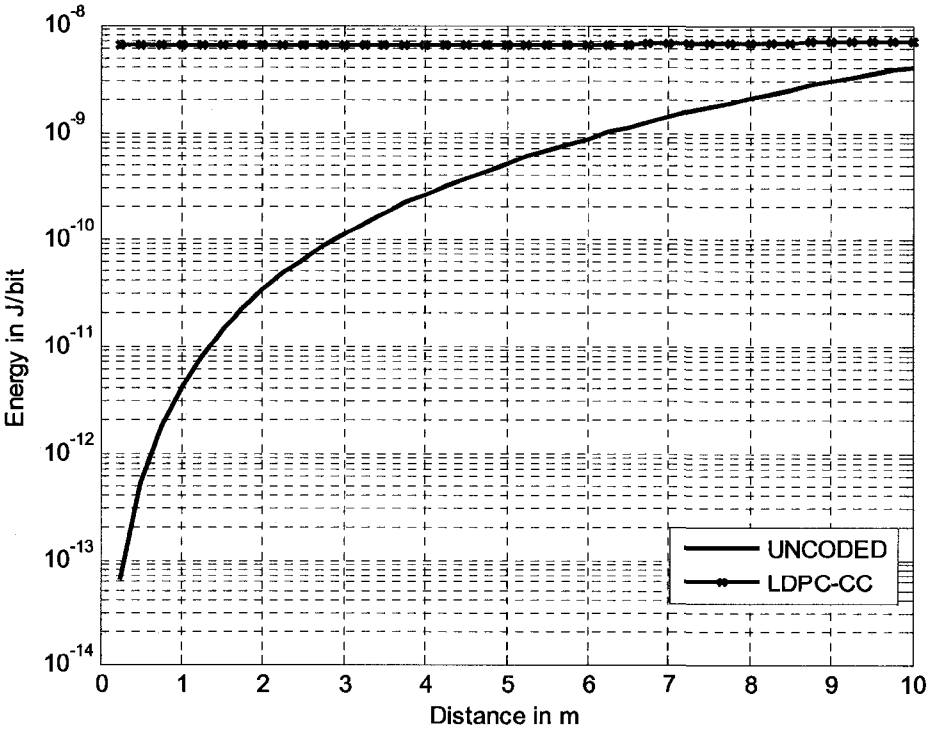


Figure 5.8: Minimum transmit energy per bit for LDPC-CC encoder (FPGA core) compared to an uncoded system, at rate 1/2, frequency 50 MHz, BER of 10^{-6}

As expected, when compared to an uncoded system, the FPGA implementations of both encoders are energy inefficient at all distances. While the encoders provide significant improvements in data reliability, their high power cost on the FPGA platform conflicts with IMD requirements rendering them as inefficient encoder choices.

ASIC devices provide significant power reductions over FPGA devices. This is due to the FPGA's *programmability*, where designs on such devices are known to consume much higher power than the same designs implemented on power-efficient ASICs. For instance, the static power component consumed in the FPGA device would be greatly reduced as ASICs can be made to use little to no power while in *wait or sleep modes*. Therefore, focusing on the dynamic power component, the figures recorded in Table 5.4 show 33 mW and 20 mW for the block and convolutional encoders at 50 MHz respectively. These figures are in the range of reasonable power consumption for wireless IMDs and warrant a further investigation of ASIC as the underlying technology for these encoders.

The gap between FPGA and ASIC technologies has been studied in literature. In [101] it is reported that the area of an FPGA design which uses LUT-only logic can be reduced 35 times when implemented on a similar process ASIC. The area gap is lowered to 18 times when the design utilizes hard blocks of memories, multipliers, and accumulators. When it comes to power, the authors of [101] report that the gap is on average 14 times for dynamic power consumption. Static power reduction is harder to quantify. Static power, which is predominantly due to transistor leakage currents, is process dependant, and other factors such as worst case leakage estimates and maturity of the process play a main role in characterizing the static power. However, the authors did

find a correlation between the area and the static power where the correlation coefficient of the static power gap to the area gap is 0.8. This is a reasonable correlation since the transistor width is normally proportional to the static power consumption, and the total area reduction partially reflects a total transistor width reduction between FPGAs and ASICs. If these scaling factors are applied to the encoder designs under study, the static power consumption of 142 mW would scale down to 9.9 mW. The block encoder's dynamic power at 50 MHz would be reduced to a mere 2.4 mW making the total power consumption 12.3 mW. The convolutional encoder would consume 11.3 mW. With these figures, the encoders would perform efficiently in wireless IMD applications. Another study shows that, considering all of the power saving options, FPGA to ASIC design conversion could save up to 20-50% on power consumption [102]. This would bring the power figures to 35.2 mW and 32.2 mW for the block and convolutional encoders respectively. These might be over estimates as a hardware implementation of the convolutional encoder based on the design discussed above was able to achieve 8.6 mW at 250 MHz frequency [91]. With the normalized power being 0.034 mW/MHz, this ASIC implementation would consume 1.7 mW during operation at 50 MHz. This measurement is much closer to the estimate obtained using the *gap technique* provided in [101].

Figure 5.9 re-evaluates the energy per bit efficiency for the 802.16e LDPC encoder using the 12.3 mW power estimate. Similarly, Figure 5.10 re-evaluates the energy per bit for the convolutional encoder of [91] using the supplied power estimate of 1.7 mW.

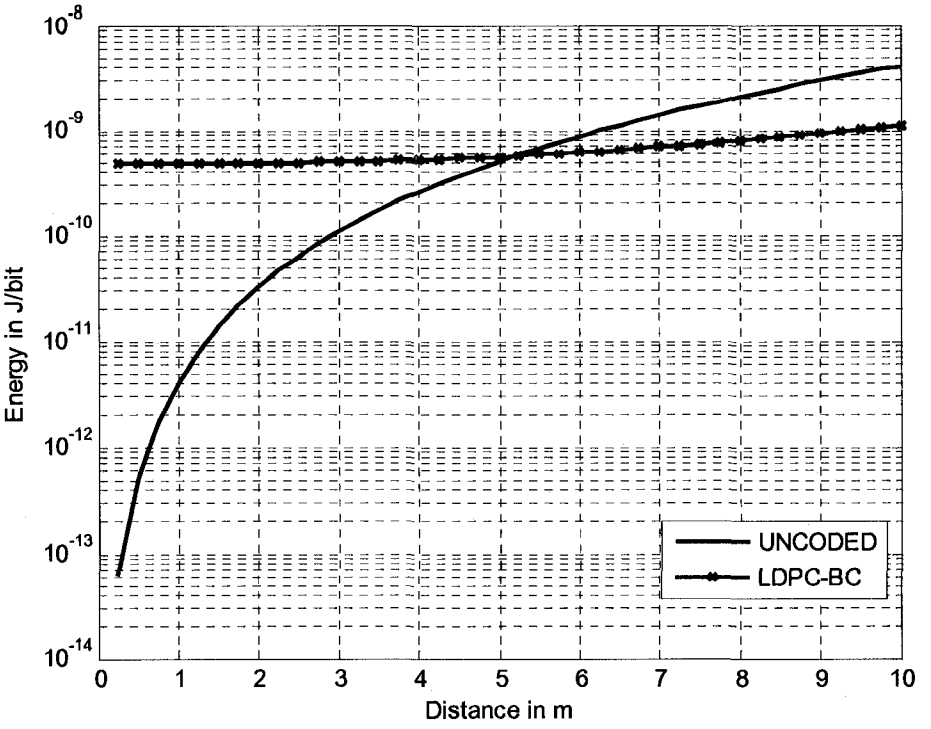


Figure 5.9: Minimum transmit energy per bit for LDPC-BC encoder (ASIC estimate) compared to an uncoded system, at rate 1/2, frequency 50 MHz, BER of 10^{-6}

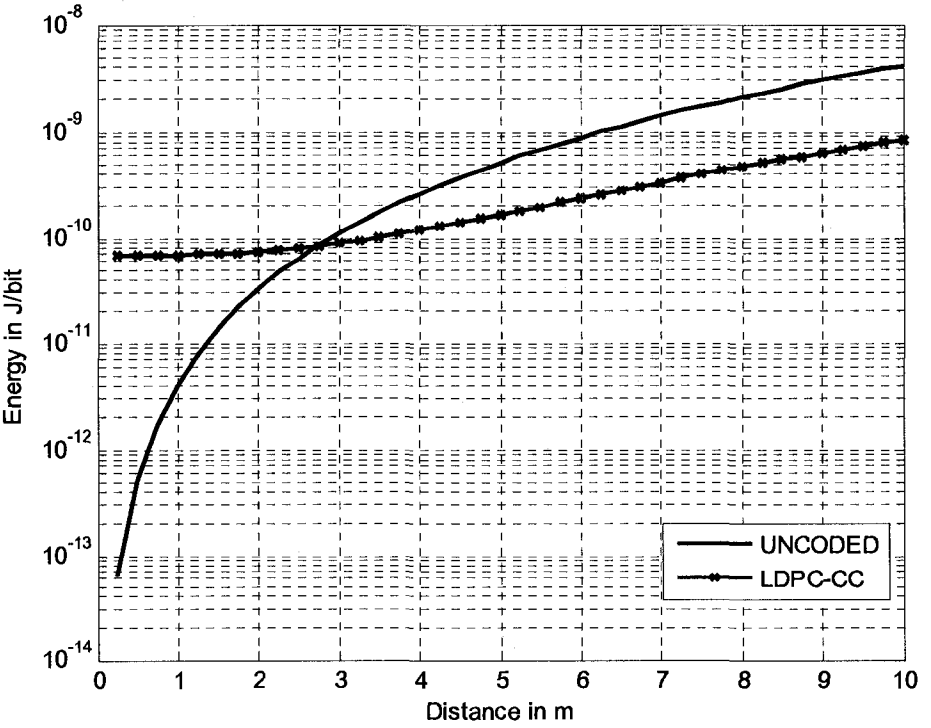


Figure 5.10: Minimum transmit energy per bit for LDPC-CC encoder (ASIC estimate) compared to an uncoded system, at rate 1/2, frequency 50 MHz, BER of 10^{-6}

It can be concluded from the figures that ASIC implementations of the encoders under study are more energy efficient options for wireless in-body communications than their FPGA counterparts. The block encoder becomes more energy efficient than the uncoded system at distances of 5 m and above. The lower power convolutional encoder becomes efficient earlier around 3 m. And in the case of a terminated LDPC-CC encoder, that consumes 2-3 times more power, the crossover point will be pushed towards the 4 m mark. These are encouraging results and show that low power construction of LDPC encoders can provide high reliability of data transmission while maintaining acceptable low power consumption.

The above results agree with the conclusions reached in [96]. In that study, several ECC coding options and their encoders were evaluated with the LDPC convolutional encoder consuming the least power for distances above 4 m.

5.6.2 Comparisons with Industry Leading IMDs

In order to gain a practical perspective into using LDPC encoders for in-body medical applications, it is important to mention industrial achievements to date and discuss their encoders' capabilities and limitations.

One of the industry leaders in medical implant communication services is Ottawa-based Zarlink Inc. In May of 2007, they introduced the world's highest performance implantable grade radio chip for in-body wireless communication systems, namely the ZL70101 chip [78].

The ZL70101 is an ASIC RF device that provides half-duplex communication with high data rates of up to 800 kbps, and operates in the 402-434 MHz frequency spectrum. Using a minimum of 2.1 Volts power supply, it needs only 5 mA of electrical current

while in continuous transmit/receive mode and down to 1 mA in low power mode. This translates into 11 mW of power consumption while in operation mode. It is expected that the encoder/decoder pair consumes even less power than the overall device. This device employs Reed-Solomon error control codes together with cyclic redundancy check (CRC) error detection for reliable wireless transmission.

In comparison, the block and convolutional LDPC encoders presented in our study are implemented on an FPGA device, and on average consume 178 mW and 161 mW at 50 MHz respectively. It is evident that, in their current form, these encoders operate at much higher power consumption levels than the industry's leading device, and are not ready for implant applications. However, as we presented in the previous section, ASIC implementations of these encoders may achieve comparable power consumption.

As per performance, LDPC codes are known to outperform Reed-Solomon codes. When the circuit complexity of the encoder and the SNR performance were considered in [96] and [97], it was shown that the LDPC-CC code consumes less energy per bit than the selected Reed-Solomon code at all distances. Our 802.16e LDPC-BC codes show comparable SNR performance to the LDPC-CC code presented in those studies, and the estimated 12 mW power consumption of the ASIC encoder compares well to industry standards.

With high throughput, superior code performance, and estimated low power consumption, ASIC implementations of the 802.16e LDPC encoder and the LDPC-CC encoder stand to make viable choices for wireless in-body transceiver devices.

5.7 Power Consumption Improvements

Aside from the substantial power gains achieved from transferring the encoder designs from FPGAs into ASICs, There are several possible approaches that may help improve the power gains further:

- The block encoder under study is designed for compatibility with the IEEE 802.16e standard. This involves high hardware overhead due to the multi-rate and multi-mode support. Compatibility with the standard is not a requirement for IMDs, and this overhead may be minimized via *tailoring the encoder for a specific rate and/or length* to a specific application. Eliminating extra code rates and the supporting control circuitry will lower the hardware usage and area, and consequently lower the overall power consumption.

  For example, if the 5/6 code rate and the 576 code length are exclusively selected, the block encoder's hardware usage may be lowered by a factor of 15 times.

- The *choice of FPGA devices* may affect the power consumption level. The Xilinx Spartan-3L devices offer 60% reduction in quiescent power consumption (99% in hibernate mode) compared to the regular Spartan-3 chips [103]. These reductions bring the FPGA-based encoders' power levels much closer to implant requirements and close to ASIC levels.

- The implementation of *sleep power states* on the chip-level can reduce the power cost of the encoder significantly. As we mentioned earlier, due to the MICS limitations on data rates the encoders will be running for 0.3-0.6 % of the time. Therefore, integrating mechanisms for shutting down the encoder when not in

operation will improve the overall power consumption of the implant and enhance battery life.

The results and estimates recorded in the previous sections provide encouraging conclusions regarding the use of LDPC encoders in wireless medical implant applications. And the improvement ideas presented here provide feasible directions for future investigation and research work.

5.8 Summary

In this chapter, we investigated the suitability of our 802.16e LDPC encoder with wireless implantable medical devices. A convolutional LDPC encoder design was also introduced. The FPGA designs' power consumption was categorized using Xilinx' XPower tool, which uses signal switching activity rates as a basis for determining the dynamic power consumption. Upon comparing the power estimates, the LDPC-CC encoder showed a 40-49 % dynamic power gain over the LDPC-BC encoder.

To evaluate an encoder's power/energy efficiency for in-vivo communications, we presented a mathematical model that defines the minimum required transmit power/energy to achieve a target BER. The FPGA designs were evaluated and showed their inefficiency for IMD applications. However, a further investigation into transferring these designs to ASIC technology proved promising. ASIC power estimates appear to be in the desired low-power range for medical implants. We conclude that LDPC codes' strong performance along with a low power encoder implementation can be a practical choice for wireless IMDs.

The final section provided several ideas that aim to improve the power consumption of the encoders and include: limiting the code rate/length support on the block encoder, utilizing low-power FPGA devices, and implementing deep sleep power states on implant transceiver chips.

Chapter 6

Conclusions and Future Work

6.1 Project Results and Analysis

The main goal of this project was to design and implement an IP core of an 802.16e compatible LDPC encoder. To accomplish this goal a number of issues had to be addressed.

First, computationally-effecient block LDPC encoding algorithms were researched. Second, the compact structure of the supported LDPC codes was utilized to achieve a low-complexity implementation. The compact representation of the parity-check matrices in the 802.16e standard provided the opportunity to minimize memory storage requirements. This was accomplished by the selection of a minimal fixed-point number representation to store the semi-scaled shift-values of the parity-check matrices. Third, the encoding algorithm was implemented using parallelism and hardware re-use in order to minimize the FPGA resource requirements. Further memory savings were achieved by implementing the specific forms of the T^{-1} and ET^{-1} sub-matrices as hardware multiplier units not requiring any memory storage.

The final design, as presented in Figure 3.22, was implemented in VHDL as an IP core. The system passed all software verification steps. Additionally, a real-life hardware implementation on the Opal Kelly XEM3010 hardware prototyping platform was confirmed to work correctly. The maximum clock speed of the design was determined to

be 80 MHz with a maximum throughput of 78.9 Mbps or 34246 packets per second. The design occupied 49% of the available slices on a Spartan-3 XC3S1500 FPGA chip and 18% of the block RAM resources.

Another goal for this project has been to study the power consumption of our LDPC block encoder, and to evaluate its suitability for low-power wireless implantable medical devices. While it is possible to integrate an encoder/decoder solution on an implant, It has been shown that the decoder typically consumes 2-3 orders of magnitude more power than the encoder [96]. Moreover, monitoring devices are not power-constrained and are able to adjust their transmit power levels more freely to achieve reliable transmission. Therefore, we considered the scenario where only the encoder is included at the implant. The power consumption of the 802.16e encoder, and a convolutional LDPC encoder design from [91], was estimated and compared. The 802.16e LDPC core averaged at 178 mW of total power while the relatively less-complex LDPC-CC averaged at 161 mW. The static power on the FPGA device measured at 142 mW, while the rest of the power cost was due to the dynamic power caused by the switching activity of the signals in the designs.

In order to evaluate the power efficiency of these encoders for wireless IMD applications, a mathematical model was developed to calculate the minimum transmit power required to achieve a target BER. The method used is based on the work presented in [96] and [97]. When compared to an uncoded system, it was evident that the encoder FPGA IP cores faired worse than the uncoded system over the target communication distance of 25 cm to 10 m. However, ASIC implementations are known to be more power-efficient, hence approximations were made to estimate the encoders' power

consumption if they were to be implemented on ASIC devices. The ASIC estimates were re-evaluated and the results showed that the 802.16e LDPC encoder is energy-efficient for distances of 5 m and above, while the lower power LDPC-CC encoder becomes efficient starting at 3 m. These results are based on reported approximation figures and are not as accurate as measuring the power on actual ASIC designs. However, they do provide encouraging results and illustrate that, if implemented on power-efficient ASICs, these encoders can be suitable choices for wireless implant applications.

When compared with industry-leading devices, the estimated ASIC power consumption of less than 15 mW makes both these encoders feasible choices for medical implants and warrant further investigation into the design and implementation of LDPC codes for these devices.

6.2 Future Work

The design process of the implemented encoder targeted compatibility with the 802.16e standard, low hardware utilization, and low latency. Future work may include re-visiting the design process to target more efficient hardware implementations. Ideas such as pipelining and parallelism may be investigated further to try and improve the encoders speed and throughput. The next version of this encoder core should include support for the 2/3A code rate. The scaling operation for this code rate requires a modulo function implementation instead of the divide and truncate approach used in the other rates.

As per power, it would be interesting to revisit the design with power consumption in mind and investigate power saving options. Another idea would be to investigate the power consumption of the encoder on low-power FPGA devices such as the Spartan-3L

chip series and others from competing vendors. Additionally, separating the various supported code rates and lengths into individual LDPC encoders would sacrifice the compatibility with the standard, while improving the power consumption for each encoder through eliminating extra control circuitry and resources. This would provide the opportunity to evaluate each code rate for use with wireless medical implants.

Lastly, to further this research towards a definite answer, implementing the 802.16e encoder on ASIC should be considered. An ASIC implementation would provide the means to measure the actual power consumption, and may possibly be suitable for real-life implant experimentation.

Bibliography

[1] C. E. Shannon, "A mathematical theory of communications: Parts I and II," *Bell Syst. Tech. J.*, vol. 27, pp. 379–423, 623–656, 1948

[2] R. Togneri, C. J. S. deSilva, *Fundamentals of Information Theory and Coding Design*, Chapman and Hall/CRC, 2002

[3] S. Lin and D. J. Costello, Jr., *Error Control Coding*. Englewood Cliffs, NJ: Prentice-Hall, 2nd ed., 2004

[4] C. Schlegel and L. Perez, *Trellis and Turbo Coding*, IEEE/Wiley, Piscataway, NJ, 2004

[5] R.G. Gallager, "Low-density parity-check codes," *IEEE Trans. Inform. Theory,* vol. 8, pp. 21-28, Jan. 1962.

[6] R.G. Gallager, *Low-Density Parity-Check Codes*. MIT Press, Cambridge, MA, 1963

[7] D. J. C. MacKay and R. M. Neal, "Near Shannon limit performance of low density parity check codes," *Electronics Letters*, vol. 32, pp. 1645-1646, Aug. 1996

[8] *Digital video broadcasting (DVB); Second generation framing structure, channel coding and modulation systems for broadcasting, interactive services, news gathering and other broad-band satellite applications*, EN 302 307, European Telecommunications Standards Institute (ETSI)

[9] "IEEE Standard for Information technology-Telecommunications and information exchange between systems-Local and metropolitan area networks-Specific requirements Part 3: Carrier Sense Multiple Access with Collision Detection (CSMA/CD) Access Method and Physical Layer Specifications," *IEEE Std 802.3an-2006 (Amendment to IEEE Std 802.3-2005)* , vol., no., pp.0_1-167, 2006

[10] "IEEE Standard for Local and metropolitan area networks Part 16: Air Interface for Fixed and Mobile Broadband Wireless Access Systems Amendment 2: Physical and Medium Access Control Layers for Combined Fixed and Mobile Operation in Licensed Bands and Corrigendum 1," *IEEE Std 802.16e-2005 and IEEE Std 802.16-2004/Cor 1-2005 (Amendment and Corrigendum to IEEE Std 802.16-2004)* , vol., no., pp. 0_1-822, 2006

[11] "Draft Standard for Information Technology-Telecommunications and information exchange between systems--Local and metropolitan area networks--Specific requirements-- Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) specifications: Amendment 4: Enhancements for Higher Throughput," *IEEE Unapproved Draft Std P802.11n_D3.00, Sep 2007* , vol., no., pp.-, 2007

[12] S.-Y. Chung, G. D. Forney, Jr., T. J. Richardson, and R. Urbanke, "On the design of low-density parity-check codes within 0.0045 dB of the Shannon limit," *IEEE Commun. Lett.*, vol. 5, pp. 58–60, Feb. 2001

[13] S.Bates and T.Zhang, "An introduction to low-density parity-check codes and their implementations," Semiconductor Research Corporation internal paper, 2005

[14] France Telecom Inc., "*France Telecom announces Turbo Codes Licensing Program (TCLP)*", October, 2001, Available: http://www.spectralicensing.com/pdfs/FT-TCLP.pdf, last visited on April 9, 2005

[15] Jimenez Felstrom, A.; Zigangirov, K.S., "Time-varying periodic convolutional codes with low-density parity-check matrix," *Information Theory, IEEE Transactions on* , vol.45, no.6, pp.2181-2191, Sep 1999

[16] Richardson, T.J.; Shokrollahi, M.A.; Urbanke, R.L., "Design of capacity-approaching irregular low-density parity-check codes," *Information Theory, IEEE Transactions on* , vol.47, no.2, pp.619-637, Feb 2001

[17] R. M. Tanner, D. Sridhara, A. Sridharan, T. E. Fuja, and D. J. Costello, "LDPC block and convolutional codes based on circulant matrices," *IEEE Trans. Inform Theory*, vol. 50, pp. 2966-2984, Dec. 2004

[18] Fossorier, M.P.C., "Quasicyclic low-density parity-check codes from circulant permutation matrices," *Information Theory, IEEE Transactions on* , vol.50, no.8, pp. 1788-1793, Aug. 2004

[19] Kou, Y.; Lin, S.; Fossorier, M.P.C., "Low-density parity-check codes based on finite geometries: a rediscovery and new results," *Information Theory, IEEE Transactions on*, vol.47, no.7, pp.2711-2736, Nov 2001

[20] Prabhakar, A.; Narayanan, K., "Pseudorandom construction of low-density parity-check codes using linear congruential sequences," *Communications, IEEE Transactions on* , vol.50, no.9, pp. 1389-1396, Sep 2002

[21] R. M. Tanner, "A recursive approach to low complexity codes," *IEEE Trans. Inform. Theory*, vol. IT-42, pp. 533–547, 1981

[22] Blanksby, A.J.; Howland, C.J., "A 690-mW 1-Gb/s 1024-b, rate-1/2 low-density parity-check code decoder," *Solid-State Circuits, IEEE Journal of*, vol.37, no.3, pp.404-412, Mar 2002

[23] M. M. Mansour and N. R. Shanbhag, "High-throughput LDPC decoders," *Trans. on VLSI Systems*, vol. 11, no. 6, Dec 2003

[24] Mansour, M.M.; Shanbhag, N.R., "A 640-Mb/s 2048-bit programmable LDPC decoder chip," *Solid-State Circuits, IEEE Journal of*, vol.41, no.3, pp. 684-698, March 2006

[25] Hao Zhong; Tong Zhang, "Design of VLSI implementation-oriented LDPC codes," *Vehicular Technology Conference, 2003. VTC 2003-Fall. 2003 IEEE 58th* , vol.1, no., pp. 670-673 Vol.1, 6-9 Oct. 2003

[26] Hocevar, D.E., "LDPC code construction with flexible hardware implementation," *Communications, 2003. ICC '03. IEEE International Conference on* , vol.4, no., pp. 2708-2712 vol.4, 11-15 May 2003

[27] Olcer, S., "Decoder architecture for array-code-based LDPC codes," *Global Telecommunications Conference, 2003. GLOBECOM '03. IEEE* , vol.4, no., pp. 2046-2050 vol.4, 1-5 Dec. 2003

[28] Engling Yeo; Pakzad, P.; Nikolic, B.; Anantharam, V., "VLSI architectures for iterative decoders in magnetic recording channels," *Magnetics, IEEE Transactions on* , vol.37, no.2, pp.748-755, Mar 2001

[29] G. Mehta and H. Lee. An FPGA implementation of the graph encoder-decoder for regular ldpc codes. In CRL Technical Report. Communications Research Laboratory, University of Pittsbugh, 2002

[30] Luby, M. G., Mitzenmacher, M., Shokrollahi, M. A., Spielman, D. A., and Stemann, "Practical loss-resilient codes," in Proceedings of the Twenty-Ninth Annual ACM Symposium on theory of Computing, 1997, pp. 150-159

[31] MacKay, D.J.C.; Wilson, S.T.; Davey, M.C., "Comparison of constructions of irregular Gallager codes," *Communications, IEEE Transactions on* , vol.47, no.10, pp.1449-1454, Oct 1999

[32] Li Ping; Leung, W.K.; Nam Phamdo, "Low density parity check codes with semi-random parity check matrix ," *Electronics Letters* , vol.35, no.1, pp.38-39, 7 Jan 1999

[33] Shaikh Faisal Zaheer, "Improved Rate-Compatible Low-Density Parity-Check Codes with Applications to Wireless Channels," M.Sc. Thesis, King Fahd University of Petroleum and Minerals , Dhahran, Saudi Arabia, May 2006

[34] Richardson, T.J.; Urbanke, R.L., "Efficient encoding of low-density parity-check codes," *Information Theory, IEEE Transactions on* , vol.47, no.2, pp.638-656, Feb 2001

[35] Andrews, K.; Dolinar, S.; Thorpe, J., "Encoders for block-circulant LDPC codes," *Information Theory, 2005. ISIT 2005. Proceedings. International Symposium on* , vol., no., pp. 2300-2304, 4-9 Sept. 2005

[36] Hao Zhong; Tong Zhang, "Block-LDPC: a practical LDPC coding system design approach," *Circuits and Systems I: Regular Papers, IEEE Transactions on [Circuits and Systems I: Fundamental Theory and Applications, IEEE Transactions on]* , vol.52, no.4, pp. 766-775, April 2005

[37] Lee, D.-U.; Luk, W.; Wang, C.; Jones, C., "A flexible hardware encoder for low-density parity-check codes," *Field-Programmable Custom Computing Machines, 2004. FCCM 2004. 12th Annual IEEE Symposium on* , vol., no., pp. 101-111, 20-23 April 2004

[38] Lee C., "Design of Encoder and Decoder for LDPC Codes Using Hybrid H-Matrix," *ETRI Journal* , vol.27, no.5, pp.557-562, October 2005

[39] Urard, P.; Yeo, E.; Paumier, L.; Georgelin, P.; Michel, T.; Lebars, V.; Lantreibecq, E.; Gupta, B., "A 135Mb/s DVB-S2 compliant codec based on 64800b LDPC and BCH codes," *Solid-State Circuits Conference, 2005. Digest of Technical Papers. ISSCC. 2005 IEEE International* , vol., no., pp. 446-609 Vol. 1, 6-10 Feb. 2005

[40] Hsuan-Yu Liu; Chien-Ching Lin; Yu-Wei Lin; Ching-Che Chung; Kai-Li Lin; Wei-Che Chang; Lin-Hung Chen; Hsie-Chia Chang; Chen-Yi Lee, "A 480Mb/s LDPC-COFDM-based UWB baseband transceiver," *Solid-State Circuits Conference, 2005. Digest of Technical Papers. ISSCC. 2005 IEEE International* , vol., no., pp. 444-609 Vol. 1, 6-10 Feb. 2005

[41] Yang Sun; Marjan Karkooti; Joseph R. Cavallaro, "High Throughput, Parallel, Scalable LDPC Encoder/Decoder Architecture for OFDM Systems," *Design, Applications, Integration and Software, 2006 IEEE Dallas/CAS Workshop on* , vol., no., pp.39-42, Oct. 2006

[42] Cai, Z.; Hao, J.; Tan, P.H.; Sun, S.; Chin, P.S., "Efficient encoding of IEEE 802.11n LDPC codes," *Electronics Letters* , vol.42, no.25, pp.1471-1472, December 7 2006

[43] Khan Z., and Arslan T., "*Implementation of a Real Time Programmable Encoder for Low Density Parity Check Code on a Reconfigurable Instruction Cell Architecture*", 2007, Available: http://www.aspdac.com/aspdac2007/pdf/archive/6B-3.pdf, last visited on April 9, 2005

[44] Xilinx Inc., "*802.16 LDPC Encoder v1.0 – Xilinx LogiCore Datasheet and Product Specification*", 2006, Available: http://www.xilinx.com/bvdocs/ipcenter/data_sheet/ldpc_802_16_enc_v1_0.pdf, last visited on April 9, 2005

[45] S. Haykin, *Communication Systems*. John Wiley & Sons, 4th ed., 2000

[46] A, Paulraj, R. Nabar, D. Gore, *Introduction to Space-Time Wireless Communications*. Cambridge University Press, 2005

[47] J. Proakis, *Digital Communications*. McGraw-Hill, 4th ed., 2001

[48] A.S. Tanenbaum, *Computer Networks*. Prentice Hall PTR, 4th ed., 2003

[49] "IEEE Standard for Local and Metropolitan Area Networks Part 16: Air Interface for Fixed Broadband Wireless Access Systems," *IEEE Std 802.16-2004 (Revision of IEEE Std 802.16-2001)* , vol., no., pp. 0_1-857, 2004.

[50] WiMAX Forum, "*Mobile WiMAX – Part I: A Technical Overview and Performance Evaluation*", Aug. 2006, Available: http://www.wimaxforum.org/technology/downloads/Mobile_WiMAX_Part1_Overview_and_Performance.pdf, last visited on April 9, 2005

[51] Institute of Electrical and Electronics Engineers, Inc., "*The IEEE 802.16 Working Group on Broadband Wireless Access Standards*", Available: http://www.ieee802.org/16/, last visited on April 9, 2005

[52] WiMAX Forum, "*Mobile WiMAX – Part II: A Comparative Analysis*", May 2006, Available: http://www.wimaxforum.org/technology/downloads/Mobile_WiMAX_Part2_Comparative_Analysis.pdf, last visited on April 9, 2005

[53] Jones, C.; Matache, A.; Tian, T.; Villasenor, J.; Wesel, R., "The universality of LDPC codes on wireless channels," *Military Communications Conference, 2003. MILCOM 2003. IEEE* , vol.1, no., pp. 440-445 Vol.1, 13-16 Oct. 2003

[54] Serener, A.; Natarajan, B.; Gruenbacher, D.M., "Performance of spread OFDM with LDPC coding in outdoor environments," *Vehicular Technology Conference, 2003. VTC 2003-Fall. 2003 IEEE 58th* , vol.1, no., pp. 318-321 Vol.1, 6-9 Oct. 2003

[55] Sartipi, M.; Fekri, F., "Source and channel coding in wireless sensor networks using LDPC codes," *Sensor and Ad Hoc Communications and Networks, 2004. IEEE SECON 2004. 2004 First Annual IEEE Communications Society Conference on* , vol., no., pp. 309-316, 4-7 Oct. 2004

[56] Sadeghi, N.; Howard, S.; Kasnavi, S.; Iniewski, K.; Gaudet, V.C.; Schlegel, C., "Analysis of error control code use in ultra-low-power wireless sensor networks," *Circuits and Systems, 2006. ISCAS 2006. Proceedings. 2006 IEEE International Symposium on* , vol., no., pp. 4 pp.-, 21-24 May 2006

[57] Jia Hou, Yu Yi, and Moon Ho Lee, "Multilevel LDPC Codes Design for Multimedia Communication CDMA System," *EURASIP Journal on Wireless Communications and Networking*, vol. 2004, no. 1, pp. 141-148, 2004

[58] Salmon, B.P.; Olivier, J.C., "Performance Analysis of Low Density Parity-Check Codes on a WiMAX Platform," *Wireless Communications and Networking Conference, 2007.WCNC 2007. IEEE* , vol., no., pp.569-571, 11-15 March 2007

[59] Baumgartner, B.; Reinhardt, M; Richter, G; Bossert, M, "Performance of Forward Error Correction for IEEE 802.16e," in *Proceedings of 10th International OFDM-Workshop,* Hamburg, Germany, Aug. 2005

[60] Bradley P., "*The Ultra Low-Power Wireless Medical Device Revolution*", April 2005, Available: http://news.zarlink.com/assets/Wireless_Revolution_WP_Apr05.pdf, last visited on April 9, 2005

[61] Institute of Electrical and Electronics Engineers, Inc., "*IEEE 802.15 Working Group for WPAN*", Available: http://www.ieee802.org/15/, last visited on April 9, 2005

[62] Institute of Electrical and Electronics Engineers, Inc., "*IEEE 802.15 WPAN Task Group 6 Body Area Networks (BAN)*", Available: http://www.ieee802.org/15/pub/TG6.html, last visited on April 9, 2005

[63] Federal Communications Commission, *"Medical Implant Communications"*, Available:

http://wireless.fcc.gov/services/index.htm?job=service_home&id=medical_implant, last visited on April 9, 2005

[64] Federal Communications Commission, "*Title 47--Telecommunication CHAPTER I--FEDERAL COMMUNICATIONS COMMISSION (CONTINUED) PART 95--PERSONAL RADIO SERVICES*", Available: http://www.access.gpo.gov/nara/cfr/waisidx_06/47cfr95_06.html, last visited on April 9, 2005

[65] International Telecommunication Union, "*OPINION ITU-R 98 - SPECTRUM REQUIREMENTS OF METEOROLOGICAL AIDS IN THE FREQUENCY RANGE FROM 400.15 TO 406 MHz*", 1997, Available: http://www.itu.int/dms_pub/itu-r/opb/op/R-OP-R.98-1997-PDF-E.pdf, last visited on April 9, 2005

[66] Zarlink Semiconductor Inc., "*Ultra Low-Power Wireless Medical Devices*" January 2005, Available: http://news.zarlink.com/assets/CIE_ulp_article.pdf, last visited on April 9, 2005

[67] P. Bradley; P. Chadwick; H. Higgins; A. Sivard, "*The challenge of designing in-body communications*" October 2004, Available: http://www.embedded.com/columns/technicalinsights/51200651?_requestid=280643, last visited on April 9, 2005

[68] K.Y. Yazdandoost; R. Kohno, "*Antenna for Medical Implanted Communications System*" July 2007, Available: https://mentor.ieee.org/802.15/file/07/15-07-0785-00-0ban-antenna-medical-implanted-communications-system.pdf

[69] K.Y. Yazdandoost; R. Kohno, "*Channel Modeling and Signaling of medical Implanted Communication Systems*" July 2007, Available: https://mentor.ieee.org/802.15/file/07/15-07-0787-00-0ban-channel-modeling-and-signaling-medical-implanted-communication-systems.pdf, last visited on April 9, 2005

[70] F. Kim; S. Yang; D. Shim, "*Open issues on the BAN*" January 2007, Available: https://mentor.ieee.org/802.15/file/07/15-07-0534-00-0ban-open-issues-ban.ppt, last visited on April 9, 2005

[71] Alomainy, A.; Hao, Y.; Yuan, Y.; Liu, Y., "Modelling and Characterisation of Radio Propagation from Wireless Implants at Different Frequencies," *Wireless Technology, 2006. The 9th European Conference on* , vol., no., pp.119-122, 10-12 Sept. 2006

[72] H. Lee; C. Lee; J. Kim; K. Kwak; S. Nam, "*Relationship between Power Loss and Frequency Band for Medical Implanted Communications*" November 2007, Available: https://mentor.ieee.org/802.15/file/07/15-07-0926-00-0ban-relationship-between-power-loss-and-frequency-band-medical-implanted-communications.ppt, last visited on April 9, 2005

[73] K. Yazdandoost, "*Channel Model for Body Area Networks (BAN)*" November 2007, Available: https://mentor.ieee.org/802.15/file/07/15-07-0943-00-0ban-ban-draft-channel-model.doc, last visited on April 9, 2005

[74] A.J. Johansson, "Wireless Communication with Medical Implants: Antennas and Propagation," PH.D. Thesis, Lund University, Lund, Sweden, June 2004

[75] P. Bradley, "*Implantable ultralow-power radio chip facilitates in-body communications*" June 2007, Available: http://rfdesign.com/next_generation_wireless/short_range_wireless/706RFDF1.pdf, last visited on April 9, 2005

[76] Federal Communications Commission, "*Tissue Dielectric Properties*", Available: http://www.fcc.gov/fcc-bin/dielec.sh, last visited on April 9, 2005

[77] Gupta, S.K.S.; Lalwani, S.; Prakash, Y.; Elsharawy, E.; Schwiebert, L., "Towards a propagation model for wireless biomedical applications," *Communications, 2003. ICC '03. IEEE International Conference on* , vol.3, no., pp. 1993-1997 vol.3, 11-15 May 2003

[78] Zarlink Semiconductor Inc., "*Medical Implantable RF Transceiver – Datasheet*" May 2007, Available: http://assets.zarlink.com/DS/zarlink_ZL70101_APR_07.pdf, last visited on April 9, 2005

[79] Townsend, K.A.; Haslett, J.W.; Tsang, T.K.K.; El-Gamal, M.N.; Iniewski, K., "Recent advances and future trends in low power wireless systems for medical

applications," *System-on-Chip for Real-Time Applications, 2005. Proceedings. Fifth International Workshop on* , vol., no., pp. 476-481, 20-24 July 2005

[80] B. Lewis and P. Swain, "Capsule endoscopy in the evaluation of patients with suspected small intestinal bleeding: the results of a pilot study," *Gastrointestinal Endoscopy*, vol. 56, no. 3, pp. 349–353, Sept. 2002

[81] Lorincz, K.; Malan, D.J.; Fulford-Jones, T.R.F.; Nawoj, A.; Clavel, A.; Shnayder, V.; Mainland, G.; Welsh, M.; Moulton, S., "Sensor networks for emergency response: challenges and opportunities," *Pervasive Computing, IEEE* , vol.3, no.4, pp. 16-23, Oct.-Dec. 2004

[82] AMI Semiconductor Inc., "*AMIS-53000 frequency Agile Transceiver*" January 2008, Available: http://www.amis.com/pdf/transceivers/amis53000_ds.pdf, last visited on April 9, 2005

[83] Torben Brack; Matthias Alles; Frank Kienle; Norbert Wehn, "A Synthesizable IP Core for WIMAX 802.16E LDPC Code Decoding," *Personal, Indoor and Mobile Radio Communications, 2006 IEEE 17th International Symposium on* , vol., no., pp.1-5, Sept. 2006

[84] Xilinx Inc., "*Spartan-3 Generation FPGAs*", Available: http://www.xilinx.com/products/silicon_solutions/fpgas/spartan_series/index.htm, last visited on April 9, 2005

[85] Xilinx Inc., "Spartan-3 Overview, The World's Lowest Cost Per I/O – Optimized for I/O-centric designs", Available: http://www.xilinx.com/products/silicon_solutions/fpgas/spartan_series/spartan3_fpgas/overview.htm#s3table, last visited on April 9, 2005

[86] TRLabs, (2007, May). 802.16e LDPC Encoder v1.0 – Datasheet and System Specification, Internal Document, TRLabs, Canada

[87] Opal Kelly Inc., *"XEM3010 – Xilinx Spartan-3 FPGA Integration Module"*, Available: http://opalkelly.com/products/xem3010/, last visited on April 9, 2005

[88] Opal Kelly Inc., *"XEM3010 User's Manual"*, November 2007, Available: http://opalkelly.com/library/XEM3010-UM.pdf, last visited on April 9, 2005

[89] Xilinx Inc., "*XPower Tutorial – FPGA Design*", July 2002, Available: ftp://ftp.xilinx.com/pub/documentation/tutorials/xpowerfpgatutorial.pdf, last visited on April 9, 2005

[90] Altera Inc., "*FPGA Power Management and Modeling Techniques – White Paper*", November 2007, Available: http://www.altera.com/literature/wp/wp-01044.pdf, last visited on April 9, 2005

[91] Swamy, R.; Bates, S.; Brandon, T., "Architectures for ASIC implementations of low-density parity-check convolutional encoders and decoders," *Circuits and Systems, 2005. ISCAS 2005. IEEE International Symposium on* , vol., no., pp. 4513-4516 Vol. 5, 23-26 May 2005

[92] Bates, S.; Elliott, D.G.; Swamy, R., "Termination Sequence Generation Circuits for Low-Density Parity-Check Convolutional Codes," *Circuits and Systems I: Regular Papers, IEEE Transactions on [Circuits and Systems I: Fundamental Theory and Applications, IEEE Transactions on]* , vol.53, no.9, pp. 1909-1917, Sept. 2006

[93] T. Kishigami; Y. Murakami; I. Yoshii, "*LDPC-Convolutional Codes for 802.18m FEC Scheme*", January 2008, Available: http://wirelessman.org/tgm/contrib/C80216m-08_074r1.pdf, last visited on April 9, 2005

[94] B. Classon; Y. Blankenship; K. Chung; *et al.*, "*LDPC Coding for OFDMA PHY*", January 2005, Available: http://wirelessman.org/tge/contrib/C80216e-05_066r3.pdf, last visited on April 9, 2005

[95] Pusane, A.; Smarandache, R.; Vontobel, P; Costello Jr., D. J., "On deriving good LDPC convolutional codes from QC LDPC block codes," *Information Theory, 2007. ISIT 2007. Proceedings. International Symposium on*, vol., no., pp.1221-1225, Jun. 2007

[96] Howard, S.L.; Alimohammad, A.; Chambers, G.; Iniewski, K.; Schlegel, C.; Cockburn, B.; Eikenbroek, J; Mensink, C., "Error-Correcting Codes for In-Vivo RF Wireless Links," 2007

[97] Howard, S., "Energy Efficiency of Error-Correcting Coding in Low-Power Wireless Links," *CMOS Emerging Technologies Workshop 2007*, Whistler, British Columbia, Canada, Jul. 2007

[98] Zhiyong He; Roy, S.; Fortier, P., "Powerful LDPC Codes for Broadband Wireless Networks: High-performance Code Construction and High-speed Encoder/Decoder Design," *Signals, Systems and Electronics, 2007. ISSSE '07. International Symposium on* , vol., no., pp.173-176, July 30 2007-Aug. 2 2007

[99] Xin-Yu Shih; Cheng-Zhou Zhan; Cheng-Hung Lin; An-Yeu Wu, "A 19-mode 8.29mm^2 52-mW LDPC Decoder Chip for IEEE 802.16e System," *VLSI Circuits, 2007 IEEE Symposium on* , vol., no., pp.16-17, 14-16 June 2007

[100] TurboBest Inc., "*IEEE 802.16e LDPC Encoder/Decoder Core*", Available: http://www.turbobest.com/WhitePaper80216eLDPC.pdf, last visited on April 9, 2005

[101] Kuon, I.; Rose, J., "Measuring the Gap Between FPGAs and ASICs," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on* , vol.26, no.2, pp.203-215, Feb. 2007

[102] V. Hopkin, "*FPGA-to-ASIC conversion a crucial concern*", September 2004, Available: http://www.us.design-reuse.com/articles/article8664.html, last visited on April 9, 2005

[103] Xilinx Inc., "*Spartan-3L Low Power FPGA Family*", September 2005, Available: http://www.xilinx.com/support/documentation/data_sheets/ds313.pdf, last visited on April 9, 2005

Appendix A

MATLAB Modules for Software Modeling and Processing

This Appendix presents the MATLAB code for the modules used the software modeling of the 802.16e LDPC encoder and the memory preparation.

A.1 Encoder.m

This module implements the encoding algorithm of Section 3.4.1 in a straight forward fashion.

```
% This program encodes a randomly generated user bit sequence into an
% 802.16e compliant LDPC codeword according to specified code length and
% rate. The program generates codewords for all supported code rates and
% lengths.
%

% for User Input version, un-comment the following lines and eleminate the
% loops
for x = 1:4
  if (x == 1)
    R = 1/2;
  elseif (x == 2)
    R = 2/3;
  elseif (x == 3)
    R = 3/4;
  else
    R = 5/6;
  end
  n = 576;

  for y = 1:19

    % Selecting the appropriate stored H matrix
    if (R == 0.5)
      codelength = num2str(n);
      filename = strcat('hmatrix12_',codelength);
      load (filename);
    elseif (R == 2/3)
      codelength = num2str(n);
      filename = strcat('hmatrix23_',codelength);
      load (filename);
    elseif (R == 3/4)
      codelength = num2str(n);
      filename = strcat('hmatrix34_',codelength);
      load (filename);
    else
      codelength = num2str(n);
      filename = strcat('hmatrix56_',codelength);
      load (filename);
```

```
    end

    % the following line is commented out for testing purposes
     u = round(rand([1,(n*R)])); % Radomly generated user bits

    nb = 24;      % Base-matrix size
    zf = n/nb;    % expansion factor for specified code length
    m = ceil((24*(1-R))*zf); % Number of Rows
    g = zf;       % m-g is the number of zeros in the ALT

    % Spliting the H matrix into the required sub-matrices
    A = H(1:(m-g),1:(n-m));
    B = H(1:(m-g),(n-m+1):(n-m+g));
    C = H(m-g+1:m,1:(n-m));
    D = H(m-g+1:m,(n-m+1):(n-m+g));
    E = H((m-g+1:m),n-(m-g)+1:n);
    T = H(1:(m-g),n-(m-g)+1:n);

    % Encoding

      % Step 1
      AuT = A*(u');
      CuT = C*(u');
      A_Result = mod(AuT,2); % in binary
      C_Result = mod(CuT,2); % in binary

      % Step 2
      ET = (E*inv(T))*AuT;
      E_Result = mod(ET,2); % in binary

      % Step 3
      p1T = ET+CuT;
      p1 = p1T';
      P1_Result = mod(p1,2); % in binary

      % Step 4
      p2T = -(inv(T)*(AuT+(B*p1T)));
      p2 = p2T';
      P2_Result = mod(p2,2); % in binary

      % The codeword
      v = [u P1_Result P2_Result];

      % Testing the codeword and saving the results
      a = H*(v');
      a = mod(a,2);          % in binary
      codeword_test = sum(a);
      test_mtx (x,y) = codeword_test;

      n = n + 96;
  end
end
```

A.2 finalEncoder.m

This module implements the encoding algorithm of Section 3.4.1 using the design we proposed in Chapter 3. This module serves as a reference point for the final hardware design.

```
% 802.16e LDPC Encoder Program
%
% This program encodes a randomly generated user bit sequence into an
```

```
% 802.16e compliant LDPC codeword according to specified code length and
% rate. The program generates codewords for all supported code rates and
% lengths.
%
% This program simulates a circular-pointer based architecture

% for User Input version, un-comment the following lines and eliminate the
% loops
% n = input('Enter code length(n) value: '); % user input code length value
% R = input('Enter code rate(R) value: ');   % user input code rate

% 2 loops to cover all code rates and lengths
for x = 1:4
  if (x == 1)
    R = 1/2;
    numRows = 11;
    numCols = 12;
  elseif (x == 2)
    R = 2/3;
    numRows = 7;
    numCols = 16;
  elseif (x == 3)
    R = 3/4;
    numRows = 5;
    numCols = 18;
  else
    R = 5/6;
    numRows = 3;
    numCols = 20;
  end
  n = 576;

  for y = 1:19

    % Loading the appropriate stored "BINARY" H matrix
    if (R == 0.5)
      codelength = num2str(n);
      filename = strcat('hmatrix12_',codelength);
      load (filename);
    elseif (R == 2/3)
      codelength = num2str(n);
      filename = strcat('hmatrix23_',codelength);
      load (filename);
    elseif (R == 3/4)
      codelength = num2str(n);
      filename = strcat('hmatrix34_',codelength);
      load (filename);
    else
      codelength = num2str(n);
      filename = strcat('hmatrix56_',codelength);
      load (filename);
    end

    % The following line could be commented out for testing purposes
    u = round(rand([1,(n*R)])); % Radomly generated user bits
    u_tr = u';

    % Setting universal constants and variables
    nb = 24;      % Base-matrix size
    max_z = 96;   % maximum z factor value
    zf = n/nb;    % expansion factor for specified code length
    m = ceil((24*(1-R))*zf); % total number of Rows in bits
    g = zf;       % m-g is the number of zeros in the ALT
    % clocks
    clock_pt = 0;  % pointer system clock
    dummy = 0;    % non-critical path clock

    %-------------------------------------------------------------------

    % The following section is the encoder section and each step is
```

```
% split into 2 sub-sections:
% 1- The pseudo-code
% 2- The circular-pointer encoder

        % ------- Encoding -------

% STEP ONE
  %   AuT = A*(u');
  %   CuT = C*(u');
  %   A_Result = mod(AuT,2); % in binary
  %   C_Result = mod(CuT,2); % in binary
[clock_pt reg1_pt reg1_bu clean_reg1] = A_mult(u_tr, R, zf, numRows, numCols, clock_pt);
[dummy reg2_pt clean_reg2] = C_mult(u_tr, R, zf, 1, numCols, dummy);


% STEP TWO
  %   ET = (E*inv(T))*AuT;
  %   E_Result = mod(ET,2); % in binary
[clock_pt reg3_pt clean_reg3] = stage2(reg1_bu, 11, zf, clock_pt);


% STEP THREE
  %   p1T = ET+CuT;
  %   p1 = p1T';
  %   P1_Result = mod(p1,2); % in binary
p1out_tr_pt = xor(reg2_pt,reg3_pt);
clock_pt = clock_pt + 1;
p1out_tr_pt = p1out_tr_pt((1 : zf) , 1); % cleaning and resizing for testing purposes
p1out_pt = p1out_tr_pt';


% STEP FOUR
  %   p2T = -(inv(T)*(AuT+(B*p1T)));
  %   p2 = p2T';
  %   P2_Result = mod(p2,2); % in binary
[clock_pt reg4_pt clean_reg4] = B_mult(p1out_tr_pt, R, zf, numRows, 1, clock_pt);
reg5_pt = xor(reg1_pt,reg4_pt);
clock_pt = clock_pt + 1; % Accounting for latency
[clock_pt p2out_tr_pt clean_p2] = stage6(reg5_pt, numRows, zf, clock_pt);
p2out_pt = clean_p2';
[p2_cycles, dummy] = size(clean_p2);
clock_pt = clock_pt + p2_cycles;  % Adding P2 output clock cycles

% The codeword
  %   v = [u P1_Result P2_Result];
v_pt = [u p1out_pt p2out_pt];

%---------- End of Encoding -------------------------------

%%% Storing Statistics %%%
% The following section stores the test statistics and results.
% The codeword test is performed by multiplying the codeword
% by the parity-check matrix(H). Correct encoding results
% in a zero word. Clocks and latency are also stored. The
% stored matrices are:
%     test_pt_mtx, clock_pt_mtx, latency_pt_mtx

% Testing the codeword

a_pt = H*(v_pt');
a_pt = mod(a_pt,2); % in binary
codeword_test = sum(a_pt);
test_pt_mtx(x,y) = codeword_test;
clock_pt_mtx(x,y) = clock_pt;
latency_pt_mtx(x,y) = clock_pt - n;

n = n + 96;
  end
end
```

A.3 stage2.m

This module implements the ET^{-1} matrix multiplier.

```
% This program simulates the operation of stage 2 of the circular-pointer
% based encoder. It performs the multiplication of [AuT] vector by
% [E*inv(T)] matrix.
% This architecture exploit the characteristics of the [E*inv(T)] matrix:
%     - The [E*inv(T)] matrix is always a row of zeros.
%     - Therefore the result of the multiplication is simply the
%     accumulation of all blocks of [AuT].
% The program uses a 96-bit shift register to aid in the
% accumulation/addition

function [clock, out_reg, clean_out] = stage2(input, numRows, zf, clock)

out_reg = zeros(96,1);

for shift_count = 1:numRows

   out_reg = xor(out_reg, input(1:96, 1));  % 96-bit accumulator
   input(1:96*(numRows-1),1) = input(97:96*(numRows),1); % shifting 96-bits up
   clock = clock + 1;
end
% Cleaning up the result from invalid bits
      clean_out = out_reg((1 : zf) , 1);
```

A.4 stage6.m

This module implements the T^{-1} matrix multiplier.

```
% This program simulates the operation of stage 6 of the circular-pointer
% based encoder. It performs the multiplication of [AuT+Bp1T] vector by
% [inv(T)] matrix.
% This architecture exploit the characteristics of the [inv(T)] matrix:
%     - The [inv(T)] matrix is an ALT matrix with all elements on the
%     diagonal and lower being zero identity matrices. All elements above
%     the digonal are zero matrices.
%     - Therefore the result of the multiplication is a vector of blocks,
%     where each block is a accumulation of itself with the all previous
%     blocks. This vector is also the p2 parity bits vector.
%
% The program uses 2 96-bit shift registers to aid in the
% accumulation/addition and storing of p2 blocks

function [clock, out_reg, clean_out] = stage6(input, numRows, zf, clock)

maxRows = 11; % fixed at 11 simulating hardware

temp_reg = zeros(96,1);
out_reg = zeros(maxRows*96,1);

for shift_count = 1:maxRows

   temp_reg = xor(temp_reg, input(1:96, 1));  % 96-bit accumulator
   input(1:96*(maxRows-1),1) = input(97:96*(maxRows),1); % shifting 96-bits up

   % Storing each intermidiate block result in the output register
   out_reg(96*(maxRows-1)+1:96*maxRows, 1) = temp_reg;
   if (shift_count < maxRows) % the final register is shifted only 10 times
      out_reg(1:96*(maxRows-1), 1) = out_reg(97:96*maxRows, 1); % shifting 96-bits up
   end
   clock = clock + 1;
```

```
end

clock = clock + 1; % accounting for latnecy

% Cleaning up the result from invalid bits
max_z = 96;
for i = 1:numRows
  if (i == 1)
    clean_out = out_reg((1 : zf) , 1);
  else
    clean_out = [clean_out; out_reg((((i-1)*max_z)+1) : (((i-1)*max_z)+zf) , 1)];
  end
end
```

A.5 A_mult.m

This module implements the A matrix multiplier (Only the A matrix module is shown).

```
% This program simulates the hardware implementation of a cicular-pointer
% based multiplier. It performs the (A * u) vector-matrix multiplication
% required at stage 1 of the 802.16e encoder. This architecture utilizes
% the circular-pointer concept explained in detail in the block_mult
% function

function [clock, reg1, reg2, clean_reg] = A_mult(input, R, zf, numRows, numCols, clock)

maxRows = 11; % fixed at 11 rows simulating hardware
dummy = 0;
reg1 = zeros(maxRows*96,1); % 1st output register
reg2 = zeros(maxRows*96,1); % 2nd output register (holds copy of 1st)
zero_flag = zeros(numRows,1); % flag to indicate zero identity matrix
ACLR =  zeros(numRows,1);    % Asynchronous clear for binary decoder

for col_count = 1:numCols

  % retrieving the starting-location values for all rows in a specific
  % column
  pointer = mem_retrieve ('A', R, 24, col_count, numRows, numCols, zf);

  for i = 1:numRows
    if (pointer(i) == 127)
      zero_flag(i) = 0;
      ACLR(i) = 1;
    else
      zero_flag(i) = 1;
      ACLR(i) = 0;
    end
  end

  % Selecting the right segment of [input] to be used
  input_slice = input((((col_count-1)*zf)+1) : (col_count*zf) , 1);

  [clock reg1(1:96, 1)] = block_mult(input_slice, pointer(1), zero_flag(1), ACLR(1), reg1(1:96, 1), clock, zf);
  [dummy reg1(97:192, 1)] = block_mult(input_slice, pointer(2), zero_flag(2), ACLR(2), reg1(97:192, 1), dummy, zf);
  [dummy reg1(193:288, 1)] = block_mult(input_slice, pointer(3), zero_flag(3), ACLR(3), reg1(193:288, 1), dummy, zf);

  if (R == 1/2 || R == 2/3 || R == 3/4)
    [dummy reg1(289:384, 1)] = block_mult(input_slice, pointer(4), zero_flag(4), ACLR(4), reg1(289:384, 1), dummy, zf);
    [dummy reg1(385:480, 1)] = block_mult(input_slice, pointer(5), zero_flag(5), ACLR(5), reg1(385:480, 1), dummy, zf);
  end

  if (R == 1/2 ||  R == 2/3)
    [dummy reg1(481:576, 1)] = block_mult(input_slice, pointer(6), zero_flag(6), ACLR(6), reg1(481:576, 1), dummy, zf);
    [dummy reg1(577:672, 1)] = block_mult(input_slice, pointer(7), zero_flag(7), ACLR(7), reg1(577:672, 1), dummy, zf);
```

```
  end

  if (R == 1/2)
    [dummy reg1(673:768, 1)] = block_mult(input_slice, pointer(8), zero_flag(8), ACLR(8), reg1(673:768, 1), dummy, zf);
    [dummy reg1(769:864, 1)] = block_mult(input_slice, pointer(9), zero_flag(9), ACLR(9), reg1(769:864, 1), dummy, zf);
    [dummy reg1(865:960, 1)] = block_mult(input_slice, pointer(10), zero_flag(10), ACLR(10), reg1(865:960, 1), dummy, zf);
    [dummy reg1(961:1056, 1)] = block_mult(input_slice, pointer(11), zero_flag(11), ACLR(11), reg1(961:1056, 1), dummy, zf);
  end
end

clock = clock + 1; % accounting for register latency
reg2 = reg1;
clock = clock + 1; % accounting for register latency

% Cleaning up the result from invalid bits
max_z = 96;
for i = 1:numRows
  if (i == 1)
    clean_reg = reg1((1 : zf) , 1);
  else
    clean_reg = [clean_reg; reg1((((i-1)*max_z)+1) : (((i-1)*max_z)+zf) , 1)];
  end
end
```

A.6 block_mult.m

This module implements the block multiplier.

```
% This function performs the matrix-vector multiplication at the block
% level, and adds the results of the block to the next one in colum sequence
%
% The following steps are performed to produce the shifted vector-block

function [clock, stored_block] = block_mult(input_slice, pointer, zero_flag, ACLR, stored_block, clock, zf)

for i = 1:zf
  % 1- Binary decoder translates pointer to a one-hot 96-bit value
  decoder_out = zeros(96,1);
  % if -1 flag is HIGH the ACLR signal is HIGH too forcing decoder output
  % to all zero
  if (ACLR == 0)
    decoder_out(pointer) = 1; % simulating the one-hot output of decoder
  end

  % 2- Assiging the input bit value to the one-hot output bit
  % 3- Adding the new bit to the already stored vector by XORing
  input_bit = and(input_slice(i),zero_flag);
  stored_block = xor(and(decoder_out,input_bit),stored_block);

  clock = clock + 1;

  % If pointer is at end of block, rotate back to beginning
  if (pointer == zf)
    pointer = 1;
  else
    pointer = pointer + 1;
  end
end
```

A.7 mem_retrieve.m

This module handles the retrieval of the memory contents used in the multiplications.

```
% This program simulates the hardware memory retrieval function. It returns
```

```
% the requested column of starting-locations from the requested sub-matrx
% after scaling down to the apropriate code length.
% The starting-location value is defined as follows:
%
%     [starting-location] = {0                  if  shift_value is 0
%                           zf - [shift-value]  if  shift_value is +ve
%                           127                 if  shift_value is -1
%                           }
% ** starting_location is also adjusted for matlab index purposes

function col = mem_retrieve (MTX, R, nb, colNumber, numRows, numCols, zf)

%numCols and numRows is the number of columns and rows OF THE DESIRED
%MATRIX, not the overall matrix

% Rate 1/2 Model Base-Matrix (from Standard)
H_12 = [[-1 94 73 -1 -1 -1 -1 -1 55 83 -1 -1 7 0 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1]
        [-1 27 -1 -1 -1 22 79 9 -1 -1 -1 12 -1 0 0 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1]
        [-1 -1 -1 24 22 81 -1 33 -1 -1 -1 0 -1 -1 0 0 -1 -1 -1 -1 -1 -1 -1 -1 -1]
        [61 -1 47 -1 -1 -1 -1 -1 65 25 -1 -1 -1 -1 -1 0 0 -1 -1 -1 -1 -1 -1 -1 -1]
        [-1 -1 39 -1 -1 -1 84 -1 -1 41 72 -1 -1 -1 -1 -1 -1 0 0 -1 -1 -1 -1 -1 -1]
        [-1 -1 -1 -1 46 40 -1 82 -1 -1 -1 79 0 -1 -1 -1 -1 -1 0 0 -1 -1 -1 -1 -1]
        [-1 -1 95 53 -1 -1 -1 -1 -1 14 18 -1 -1 -1 -1 -1 -1 -1 -1 0 0 -1 -1 -1 -1]
        [-1 11 73 -1 -1 -1 2 -1 -1 47 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 0 0 -1 -1 -1]
        [12 -1 -1 -1 83 24 -1 43 -1 -1 -1 51 -1 -1 -1 -1 -1 -1 -1 -1 -1 0 0 -1 -1]
        [-1 -1 -1 -1 -1 94 -1 59 -1 -1 70 72 -1 -1 -1 -1 -1 -1 -1 -1 -1 0 0 -1]
        [-1 -1 7 65 -1 -1 -1 -1 39 49 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 0 0]
        [43 -1 -1 -1 -1 66 -1 41 -1 -1 -1 26 7 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 0]];

% Rate 2/3A Model Base-Matrix (from Standard)
H_23A = [[3 0 -1 -1 2 0 -1 3 7 -1 1 1 -1 -1 -1 -1 1 0 -1 -1 -1 -1 -1 -1]
         [-1 -1 1 -1 36 -1 -1 34 10 -1 -1 18 2 -1 3 0 -1 0 0 -1 -1 -1 -1 -1]
         [-1 -1 12 2 -1 15 -1 40 -1 3 -1 15 -1 2 13 -1 -1 -1 0 0 -1 -1 -1 -1]
         [-1 -1 19 24 -1 3 0 -1 6 -1 17 -1 -1 -1 8 39 -1 -1 -1 0 0 -1 -1 -1]
         [20 -1 6 -1 -1 10 29 -1 -1 28 -1 14 -1 38 -1 -1 0 -1 -1 -1 0 0 -1 -1]
         [-1 -1 10 -1 28 20 -1 -1 8 -1 36 -1 9 -1 21 45 -1 -1 -1 -1 -1 0 0 -1]
         [35 25 -1 37 -1 21 -1 -1 5 -1 -1 0 -1 4 20 -1 -1 -1 -1 -1 -1 -1 0 0]
         [-1 6 6 -1 -1 -1 4 -1 14 30 -1 3 36 -1 14 -1 1 -1 -1 -1 -1 -1 -1 0]];

% Rate 2/3B Model Base-Matrix (from Standard)
H_23B = [[2 -1 19 -1 47 -1 48 -1 36 -1 82 -1 47 -1 15 -1 95 0 -1 -1 -1 -1 -1 -1]
         [-1 69 -1 88 -1 33 -1 3 -1 16 -1 37 -1 40 -1 48 -1 0 0 -1 -1 -1 -1 -1]
         [10 -1 86 -1 62 -1 28 -1 85 -1 16 -1 34 -1 73 -1 -1 -1 0 0 -1 -1 -1 -1]
         [-1 28 -1 32 -1 81 -1 27 -1 88 -1 5 -1 56 -1 37 -1 -1 -1 0 0 -1 -1 -1]
         [23 -1 29 -1 15 -1 30 -1 66 -1 24 -1 50 -1 62 -1 -1 -1 -1 -1 0 0 -1 -1]
         [-1 30 -1 65 -1 54 -1 14 -1 0 -1 30 -1 74 -1 0 -1 -1 -1 -1 -1 0 0 -1]
         [32 -1 0 -1 15 -1 56 -1 85 -1 5 -1 6 -1 52 -1 0 -1 -1 -1 -1 -1 0 0]
         [-1 0 -1 47 -1 13 -1 61 -1 84 -1 55 -1 78 -1 41 95 -1 -1 -1 -1 -1 -1 0]];

% Rate 3/4A Model Base-Matrix (from Standard)
H_34A = [[6 38 3 93 -1 -1 -1 30 70 -1 86 -1 37 38 4 11 -1 46 48 0 -1 -1 -1 -1]
         [62 94 19 84 -1 92 78 -1 15 -1 -1 92 -1 45 24 32 30 -1 -1 0 0 -1 -1 -1]
         [71 -1 55 -1 12 66 45 79 -1 78 -1 -1 10 -1 22 55 70 82 -1 -1 0 0 -1 -1]
         [38 61 -1 66 9 73 47 64 -1 39 61 43 -1 -1 -1 -1 95 32 0 -1 -1 0 0 -1]
         [-1 -1 -1 -1 32 52 55 80 95 22 6 51 24 90 44 20 -1 -1 -1 -1 -1 -1 0 0]
         [-1 63 31 88 20 -1 -1 -1 6 40 56 16 71 53 -1 -1 27 26 48 -1 -1 -1 -1 0]];

% Rate 3/4B Model Base-Matrix (from Standard)
H_34B = [[-1 81 -1 28 -1 -1 14 25 17 -1 -1 85 29 52 78 95 22 92 0 0 -1 -1 -1 -1]
         [42 -1 14 68 32 -1 -1 -1 -1 70 43 11 36 40 33 57 38 24 -1 0 0 -1 -1 -1]
         [-1 -1 20 -1 -1 63 39 -1 70 67 -1 38 4 72 47 29 60 5 80 -1 0 0 -1 -1]
         [64 2 -1 -1 63 -1 -1 3 51 -1 81 15 94 9 85 36 14 19 -1 -1 -1 0 0 -1]
         [-1 53 60 80 -1 26 75 -1 -1 -1 -1 86 77 1 3 72 60 25 -1 -1 -1 -1 0 0]
         [77 -1 -1 -1 15 28 -1 35 -1 72 30 68 85 84 26 64 11 89 0 -1 -1 -1 -1 0]];

% Rate 5/6 Model Base-Matrix (from Standard)
H_56 = [[1 25 55 -1 47 4 -1 91 84 8 86 52 82 33 5 0 36 20 4 77 80 0 -1 -1]
        [-1 6 -1 36 40 47 12 79 47 -1 41 21 12 71 14 72 0 44 49 0 0 0 0 -1]
        [51 81 83 4 67 -1 21 -1 31 24 91 61 81 9 86 78 60 88 67 15 -1 -1 0 0]
        [50 -1 50 15 -1 36 13 10 11 20 53 90 29 92 57 30 84 92 11 66 80 -1 -1 0]];
```

```
%-------------------------------------------------

% Selecting the appropriate "NUMERICAL" H matrix
    % g = 1;
    % n = 24;  %number of columns in the overall matrix (is constant)

if (R == 0.5)
  m=12;
  A_num = H_12(1:(m-1),1:(24-m));
  B_num = H_12(1:(m-1),(24-m+1):(24-m+1));
  C_num = H_12(m-1+1:m,1:(24-m));
elseif (R == 2/3)
  m=8;
  A_num = H_23B(1:(m-1),1:(24-m));
  B_num = H_23B(1:(m-1),(24-m+1):(24-m+1));
  C_num = H_23B(m-1+1:m,1:(24-m));
elseif (R == 3/4)
  m=6;
  A_num = H_34A(1:(m-1),1:(24-m));
  B_num = H_34A(1:(m-1),(24-m+1):(24-m+1));
  C_num = H_34A(m-1+1:m,1:(24-m));
else
  m=4;
  A_num = H_56(1:(m-1),1:(24-m));
  B_num = H_56(1:(m-1),(24-m+1):(24-m+1));
  C_num = H_56(m-1+1:m,1:(24-m));
end

if (MTX == 'A')
    col = A_num(:, colNumber);
  elseif (MTX == 'B')
    col = B_num(:, colNumber);
  elseif (MTX == 'C')
    col = C_num(:, colNumber);
end

% Performing scaling-down according to requested code length
for i = 1:numRows
  if col(i) >= 0
    col(i) = zf -(floor(col(i) * (zf/96))) + 1; %
    % The following check protects against:
    %     1- an original zero shift value
    %     2- a zero shift value appearing after scaling down.
    if (col(i) == zf+1) %
      col(i) = 1;
    end
  else
    col(i) = 127;
  end
end
```

A.8 H_generator.m

This module generates the binary versions of all the H (parity-check) matrices.

```
% This is a MATLAB script to generate the expanded binary H matrix to be
% used in the WiMAX LDPC encoder project.
% Input(s): code length, code rate
% Output(s): complete(expanded) binary H matrix in .mat format

% Model Base Matrices
%-------------------------------------------------

% Rate 1/2 Model Base-Matrix (from Standard)
H_12 = [[-1 94 73 -1 -1 -1 -1 -1 55 83 -1 -1 7 0 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1]
    [-1 27 -1 -1 -1 22 79 9 -1 -1 -1 12 -1 0 0 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1]
    [-1 -1 -1 24 22 81 -1 33 -1 -1 -1 0 -1 -1 0 0 -1 -1 -1 -1 -1 -1 -1 -1]
```

```
      [61 -1 47 -1 -1 -1 -1 -1 65 25 -1 -1 -1 -1 -1 0 0 -1 -1 -1 -1 -1 -1 -1]
      [-1 -1 39 -1 -1 -1 84 -1 -1 41 72 -1 -1 -1 -1 -1 0 0 -1 -1 -1 -1 -1 -1]
      [-1 -1 -1 -1 46 40 -1 82 -1 -1 -1 79 0 -1 -1 -1 -1 0 0 -1 -1 -1 -1 -1]
      [-1 -1 95 53 -1 -1 -1 -1 -1 14 18 -1 -1 -1 -1 -1 -1 -1 0 0 -1 -1 -1 -1]
      [-1 11 73 -1 -1 -1 2 -1 -1 47 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 0 0 -1 -1 -1]
      [12 -1 -1 -1 83 24 -1 43 -1 -1 -1 51 -1 -1 -1 -1 -1 -1 -1 -1 -1 0 0 -1 -1]
      [-1 -1 -1 -1 -1 94 -1 59 -1 -1 70 72 -1 -1 -1 -1 -1 -1 -1 -1 -1 0 0 -1]
      [-1 -1 7 65 -1 -1 -1 -1 39 49 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 0 0]
      [43 -1 -1 -1 -1 66 -1 41 -1 -1 -1 26 7 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 0]];

% Rate 2/3A Model Base-Matrix (from Standard)
H_23A = [[3 0 -1 -1 2 0 -1 3 7 -1 1 1 -1 -1 -1 -1 1 0 -1 -1 -1 -1 -1]
      [-1 -1 1 -1 36 -1 -1 34 10 -1 -1 18 2 -1 3 0 -1 0 0 -1 -1 -1 -1 -1]
      [-1 -1 12 2 -1 15 -1 40 -1 3 -1 15 -1 2 13 -1 -1 -1 0 0 -1 -1 -1 -1]
      [-1 -1 19 24 -1 3 0 -1 6 -1 17 -1 -1 -1 8 39 -1 -1 -1 0 0 -1 -1 -1]
      [20 -1 6 -1 -1 10 29 -1 -1 28 -1 14 -1 38 -1 -1 0 -1 -1 -1 0 0 -1 -1]
      [-1 -1 10 -1 28 20 -1 -1 8 -1 36 -1 9 -1 21 45 -1 -1 -1 -1 -1 0 0 -1]
      [35 25 -1 37 -1 21 -1 -1 5 -1 -1 0 -1 4 20 -1 -1 -1 -1 -1 -1 -1 0 0]
      [-1 6 6 -1 -1 -1 4 -1 14 30 -1 3 36 -1 14 -1 1 -1 -1 -1 -1 -1 -1 0]];

% Rate 2/3B Model Base-Matrix (from Standard)
H_23B = [[2 -1 19 -1 47 -1 48 -1 36 -1 82 -1 47 -1 15 -1 95 0 -1 -1 -1 -1 -1 -1]
      [-1 69 -1 88 -1 33 -1 3 -1 16 -1 37 -1 40 -1 48 -1 0 0 -1 -1 -1 -1 -1]
      [10 -1 86 -1 62 -1 28 -1 85 -1 16 -1 34 -1 73 -1 -1 -1 0 0 -1 -1 -1 -1]
      [-1 28 -1 32 -1 81 -1 27 -1 88 -1 5 -1 56 -1 37 -1 -1 -1 0 0 -1 -1 -1]
      [23 -1 29 -1 15 -1 30 -1 66 -1 24 -1 50 -1 62 -1 -1 -1 -1 -1 0 0 -1 -1]
      [-1 30 -1 65 -1 54 -1 14 -1 0 -1 30 -1 74 -1 0 -1 -1 -1 -1 -1 0 0 -1]
      [32 -1 0 -1 15 -1 56 -1 85 -1 5 -1 6 -1 52 -1 0 -1 -1 -1 -1 -1 0 0]
      [-1 0 -1 47 -1 13 -1 61 -1 84 -1 55 -1 78 -1 41 95 -1 -1 -1 -1 -1 -1 0]];

% Rate 3/4A Model Base-Matrix (from Standard)
H_34A = [[6 38 3 93 -1 -1 -1 30 70 -1 86 -1 37 38 4 11 -1 46 48 0 -1 -1 -1 -1]
      [62 94 19 84 -1 92 78 -1 15 -1 -1 92 -1 45 24 32 30 -1 -1 0 0 -1 -1 -1]
      [71 -1 55 -1 12 66 45 79 -1 78 -1 -1 10 -1 22 55 70 82 -1 -1 0 0 -1 -1]
      [38 61 -1 66 9 73 47 64 -1 39 61 43 -1 -1 -1 -1 95 32 0 -1 -1 0 0 -1]
      [-1 -1 -1 -1 32 52 55 80 95 22 6 51 24 90 44 20 -1 -1 -1 -1 -1 -1 0 0]
      [-1 63 31 88 20 -1 -1 -1 6 40 56 16 71 53 -1 -1 27 26 48 -1 -1 -1 -1 0]];

% Rate 3/4B Model Base-Matrix (from Standard)
H_34B = [[-1 81 -1 28 -1 -1 14 25 17 -1 -1 85 29 52 78 95 22 92 0 0 -1 -1 -1 -1]
      [42 -1 14 68 32 -1 -1 -1 -1 70 43 11 36 40 33 57 38 24 -1 0 0 -1 -1 -1]
      [-1 -1 20 -1 -1 63 39 -1 70 67 -1 38 4 72 47 29 60 5 80 -1 0 0 -1 -1]
      [64 2 -1 -1 63 -1 -1 3 51 -1 81 15 94 9 85 36 14 19 -1 -1 -1 0 0 -1]
      [-1 53 60 80 -1 26 75 -1 -1 -1 -1 86 77 1 3 72 60 25 -1 -1 -1 -1 0 0]
      [77 -1 -1 -1 15 28 -1 35 -1 72 30 68 85 84 26 64 11 89 0 -1 -1 -1 -1 0]];

% Rate 5/6 Model Base-Matrix (from Standard)
H_56 = [[1 25 55 -1 47 4 -1 91 84 8 86 52 82 33 5 0 36 20 4 77 80 0 -1 -1]
      [-1 6 -1 36 40 47 12 79 47 -1 41 21 12 71 14 72 0 44 49 0 0 0 0 -1]
      [51 81 83 4 67 -1 21 -1 31 24 91 61 81 9 86 78 60 88 67 15 -1 -1 0 0]
      [50 -1 50 15 -1 36 13 10 11 20 53 90 29 92 57 30 84 92 11 66 80 -1 -1 0]];
%----------------------------------------------------

% for User Input version, un-comment the following lines and eleminate the
% loops
% n = input('Enter code length(n) value: ');   % user input code length value
% R = input('Enter code rate(R) value: ');    % user input code rate


% 2 loops to cover all code rates and lengths
for x = 1:4
   if (x == 1)
      R = 1/2;
   elseif (x == 2)
      R = 2/3;
   elseif (x == 3)
      R = 3/4;
   else
      R = 5/6;
   end
```

```
for n = 576:96:2304

  w=setdiff(who,{'n', 'R', 'H_12', 'H_23A', 'H_23B', 'H_34A', 'H_34B', 'H_56'}); clear(w{:},'w');

  %constants
  nb = 24;      % Base-matrix size
  nmax = 2304;  % max code length (number of bits)

  z0 = nmax/nb; % expansion factor for nmax code length
  zf = n/nb;    % expansion factor for other code lengths
  rows = ceil(24*(1-R));

  perm = eye(zf);          % identity matrix of Zf x Zf size
  zeromatrix = zeros(zf,zf); % zeros matrix of Zf x Zf size

  %-------------------------------------------------------------------------

  % Constructing the Binary H Matrix

  % Selecting the appropriate model bade matrix
  if (R == 0.5)
    H_bm = H_12;
  elseif (R == 2/3)
    H_bm = H_23B;
  elseif (R == 3/4)
    H_bm = H_34A;
  else
    H_bm = H_56;
  end

  % scaling for code length
  scaled = floor(H_bm.*(zf/z0));   %multiplies input matrix by

  for row = 1:rows,

    % Initialize temp_row
    if (scaled(row,1) < 0)
      temp_row = zeromatrix;  %set temp_row to first input matrix for row
    else
      temp_row = circshift(perm, [0, scaled(row,1)]);
    end

    % after initializing for the first entry in the column, we con-cat the
    % rest of the columns to the row.
    for col = 2 : 24,
      if (scaled(row,col) < 0)
          temp_row = [temp_row, zeromatrix];
      else
          temp_row = [temp_row, circshift(perm, [0, scaled(row,col)])];
      end
    end

    % add rows to the H matrix.
    if (row == 1)
      H = temp_row;
    else
      H = [H; temp_row];
    end

  end

  % Saving the completed H matrix
  if (R == 0.5)
    codelength = num2str(n);
    filename = strcat('hmatrix12_',codelength);
    save (filename, 'H')
  elseif (R == 2/3)
    codelength = num2str(n);
    filename = strcat('hmatrix23_',codelength);
```

```
      save (filename, 'H')
    elseif (R == 3/4)
      codelength = num2str(n);
      filename = strcat('hmatrix34_',codelength);
      save (filename, 'H')
    else
      codelength = num2str(n);
      filename = strcat('hmatrix56_',codelength);
      save (filename, 'H')
    end
  end
end
```

A.9 Accuracy_driver.m

This module, in conjunction with *verify_accuracy.m,* evaluates the required precision for the semi-scaled matrix values.

```
for i = 1:60
  Error_present = verify_accuracy(i);
  if Error_present == 0
    NoErrors = 1
    Accuracy = i
    break;
  end
end
```

A.10 verify_accuracy.m

This module, in conjunction with *Accuracy_driver.m,* evaluates the required precision for the semi-scaled matrix values.

```
function Error_present = verify_accuracy(bitAccuracy)
Error_present = 0;
max_float_val = 0;

for H = 0:95  %All possible values in the H matrix
  zf = 24;
  for j = 1:19 %all possible zf values

    %The value that is actually stored in memory
    fraction = zf/96;

    %The floating, i.e. high-accuracy value
    float_val = zf - floor(H*fraction);

    %The binary approximation - Convert, then convert back
    binary_fraction = conv_decimal (fraction, bitAccuracy);
    approx_fraction = conv_fraction (binary_fraction);
    approx_val = zf - floor(H*approx_fraction);

    if float_val ~= approx_val
      if float_val > max_float_val
        max_float_val = float_val;
      end
      [float_val approx_val H zf zf-H];
      Error_present = 1;
    end
```

```
    zf = zf + 4;
  end
end
```

A.11 check_allvalues_script.m

This module performs a trial-and-error run through a range of precision values while looping through all possible values.

```
before_mod_errors = 0; % num of errors before correction
after_mod_errors = 0;  % num of errors after correction

% Prececion
n = 11
corr_factor = 0.0005

for H = 0:95  %All possible values in the H matrix
  errors = 0;
  %The accurate fraction to be stored in memory
  fraction = H/96;

  for zf = 24:4:96 %all possible zf values

    %The floating, i.e. high-accuracy value
    float_val = zf - floor(zf*fraction);

    %The binary approximation - Convert, then convert back
    binary_fraction = conv_decimal (fraction, n);
    approx_fraction = conv_fraction (binary_fraction);
    %The low-accuracy floating value
    approx_val = zf - floor(zf*approx_fraction);

    % Checking for errors before correction
    if float_val ~= approx_val
      errors = errors + 1;
      % if you un-comment the following line it will show that
      % the difference between the approximated and accurate values
      % is just a factor of 1.0, this happens due to lack of rounding
      % in our initial approach.
      % float_val - approx_val
    end
  end % zf loop


  if (errors ~= 0)

    % If (errors != 0) then this (input) caused an error on at least one
    % zf value
    errors;  % How many Zf values (or scaling operations cause errors)
    H;       % The H value in question
    before_mod_errors = before_mod_errors + errors; % accumulating errors

    % Correction Routine
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    fraction = H/96;               % value causing errors
    new_fraction = H/96 + corr_factor; % corrected value

    new_errors = 0;
    for zf = 24:4:96 %all possible zf values

      %The floating, i.e. high-accuracy value
      float_val = zf - floor(zf*fraction);

      %The binary approximation - Convert, then convert back
      % USING THE NEW CORRECTED FRACTION %
```

```
        binary_fraction = conv_decimal (new_fraction, n);
        approx_fraction = conv_fraction (binary_fraction);
        %The low-accuracy floating value
        approx_val = zf - floor(zf*approx_fraction);

        % Checking for errors
        if float_val ~= approx_val
          new_errors = new_errors + 1;
          zf;
          % float_val - approx_val
        end

      end % zf loop
      new_errors
      after_mod_errors = after_mod_errors + new_errors; % accumulating errors
      %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

   end % if errors exist
end % H loop

before_mod_errors
after_mod_errors
```

A.12 A_firstcol_maker.m

This module performs the full scaling operations required to prepare the first column of matrix A for storage in RAM (Only the A matrix module is shown).

```
% Model Base Matrices
%-----------------------------------------------
  * Deleted in this Appendix for brevity
%-----------------------------------------------

k=1;
rate_row = 0;

for x = 1:5
  if (x == 1)
    R = 1/2;
    numRows = 11;
    numCols = 12;
    H_num = H_12;
  elseif (x == 2)
    R = 2/3;
    numRows = 7;
    numCols = 16;
    H_num = H_23B;
  elseif (x == 3)
    R = 3/4;
    numRows = 5;
    numCols = 18;
    H_num = H_34A;
  elseif (x == 4)
    R = 3/4;
    numRows = 5;
    numCols = 18;
    H_num = H_34B;
  else
    R = 5/6;
    numRows = 3;
    numCols = 20;
    H_num = H_56;
  end
  [Hrows Hcols] = size(H_num);
  A = H_num(1:numRows, 1:numCols)';
```

```
  [Arows Acols] = size(A); %note that A was transposed

  % Work only on the first original column in A (first row after transp)!
  for i = 1:Acols
    zf = 24;
    A(1,i);
    for j = 1:19
      if A(1,i) == -1
        A_firstCol(rate_row+j,i) = -1;
      elseif A(1,i) == 0
        A_firstCol(rate_row+j,i) = 0;
      else
        A_firstCol(rate_row+j,i) = zf - floor(A(1,i)*zf/96);
      end

      % Detecting the after-scaling Zeros
      if (A_firstCol(rate_row+j,i) == zf)
        A_firstCol(rate_row+j,i) = 0;
      end

      zf = zf + 4;

    end % end of j
  end % end of i

  %-----------------------------------------------------------------
  %This section added to ensure that the 'blank', i.e. unused
  %sections of the A matrix are -1, not zero as they would be
  %otherwise.
  %-----------------------------------------------------------------
  [A_firstCol_rows A_firstCol_cols] = size(A_firstCol);
  for n = Acols+1:A_firstCol_cols
    for m = 1:19
     A_firstCol(rate_row+m,n) = -1;
    end
  end
  rate_row = rate_row + 19;

end % end for x

[A_firstCol_rows A_firstCol_cols] = size(A_firstCol);
A_firstCol_bin = [];

for n = 1:A_firstCol_rows
  A_firstCol_bin_row = [];
  for m = 1:A_firstCol_cols
    if A_firstCol(n,m) == -1
      A_firstCol_bin_insert = '1111111';
    else
      A_firstCol_bin_insert = dec2bin(A_firstCol(n,m), 7);
    end

    % Storing the resulting values in a matrix of strings (one row)
    A_firstCol_bin_row = [A_firstCol_bin_row A_firstCol_bin_insert];
    col_seperate = sprintf ('\t');% Column seperator character
    A_firstCol_bin_row = [A_firstCol_bin_row col_seperate];

  end % end m

  % Storing the results in a matrix of strings
  if (n == 1)
    A_firstCol_bin = A_firstCol_bin_row;
  else
    row_seperate = repmat(' ', 1, 88); %11cols *8-bits + 11 tabs
    A_firstCol_bin = [A_firstCol_bin; row_seperate];
    A_firstCol_bin = [A_firstCol_bin; A_firstCol_bin_row];
  end

end % end n
```

A.13 A_other_col_maker.m

This module performs all the operations required to semi-scale the matrix values in preparation for storage in RAM. All columns of the matrix except for the first are processed (Only the A matrix module is shown).

```
% Model Base Matrices
%-------------------------------------------------

  * Deleted in this Appendix for brevity
%-------------------------------------------------

k=1;
for x = 1:5
  if (x == 1)
     R = 1/2;
     numRows = 11;
     numCols = 12;
     H_num = H_12;
  elseif (x == 2)
     R = 2/3;
     numRows = 7;
     numCols = 16;
     H_num = H_23B;
  elseif (x == 3)
     R = 3/4;
     numRows = 5;
     numCols = 18;
     H_num = H_34A;
  elseif (x == 4)
     R = 3/4;
     numRows = 5;
     numCols = 18;
     H_num = H_34B;
  else
     R = 5/6;
     numRows = 3;
     numCols = 20;
     H_num = H_56;
  end
  [Hrows Hcols] = size(H_num);
  A = H_num(1:numRows, 1:numCols)';
  [Arows Acols] = size(A); %note that A was transposed

   for i = 2:Arows
        for j = 1:Acols
           A(i);
           if A(i,j) == -1
             A_Col(k,j) = -1;
           elseif A(i,j) == 0
             A_Col(k,j) = 0;
           else
             A_Col(k,j) = A(i,j); %zf - floor(B(i,j)*zf/96);
           end
        end
      %-----------------------------------------------------------------
      %This section added to ensure that the 'blank', i.e. unused
      %sections of the A matrix are -1, not zero as they would be
      %otherwise.
      %-----------------------------------------------------------------
      [A_Col_rows A_Col_cols] = size(A_Col);
      for j = Acols+1:A_Col_cols
        A_Col(k,j) = -1;
```

```
      end

      %Note: k is the counter for the row of B_Col.  It is incremented
      %once for every new B matrix, or every new column in the C/A
      %matrices.
      %j is the counter that runs along the row of B.
      %i is always one, in the case of B. It will have use in the A
      %case.

      k = k + 1;
      %end
    end
end
A_Col;

[A_Col_rows A_Col_cols] = size(A_Col);
A_Col_bin = [];
A_Col_bin_text = [];

for i = 1:A_Col_rows
  A_Col_bin_row = [];
  for j = 1:A_Col_cols
    if A_Col(i,j) == -1
      A_Col_bin_insert = [1 1 1 1 1 1 1 1 1 1 1];
    else
      A_Col_bin_insert = ConvAndRound_decimal(A_Col(i,j), 11); %11 bit precision
    end
    for k = 1:length(A_Col_bin_insert)
      A_Col_bin(i,11*(j-1)+k) = A_Col_bin_insert(k);
    end
    text_value = num2str(A_Col_bin_insert);
    % Storing the resulting values in a matrix of strings (one row)
    A_Col_bin_row = [A_Col_bin_row text_value];
    col_seperate = sprintf ('\t');% Column seperator character
    A_Col_bin_row = [A_Col_bin_row col_seperate];
  end
  % Storing the results in a matrix of strings
  if (i == 1)
    A_Col_bin_text = A_Col_bin_row;
  else
    row_seperate = repmat(' ', 1, 352);
    A_Col_bin_text = [A_Col_bin_text;  row_seperate];
    A_Col_bin_text = [A_Col_bin_text; A_Col_bin_row];
  end
end
A_Col_bin;
```

A.14 AB_mem_checker.m

This module verifies the output of the AB hardware memory controller against the output of its software counterpart (Only the AB matrix module is shown).

```
% Model Base Matrices
%--------------------------------------------------

  * Deleted in this Appendix for brevity
%--------------------------------------------------

R = input ('Enter the target code RATE: ', 's');

errors = 0;

if (strcmp(R,'1/2'))
  %R = 1/2;
```

```
   numRows = 11;
   numCols = 12;
   H_num = H_12;
elseif (strcmp(R,'2/3b'))
   %R = 2/3;
   numRows = 7;
   numCols = 16;
   H_num = H_23B;
elseif (strcmp(R,'3/4a'))
   %R = 3/4;
   numRows = 5;
   numCols = 18;
   H_num = H_34A;
elseif (strcmp(R,'3/4b'))
   %R = 3/4;
   numRows = 5;
   numCols = 18;
   H_num = H_34B;
else
   %R = 5/6;
   numRows = 3;
   numCols = 20;
   H_num = H_56;
end

[Hrows Hcols] = size(H_num);
AB = H_num(1:numRows, 1:numCols+1); % including B

[AB_rows AB_cols] = size (AB);

vhdl_file = fopen ('E:\PROJECTS\Memory_Work\VHDL\AB_CONTROLLER\Testing_and_Results\ab_controller_out.txt', 'r');

 for zf = 24:4:96
   for i = 1:AB_cols
      data_line = fgets(vhdl_file);
      for j = 1:AB_rows
         % Detecting -1
         if (AB(j,i) == -1)
            accurate_result = -1;
         else
            accurate_result = zf - floor((AB(j,i)/96)*zf);
         end

         % Detetcting Zeros
         if (accurate_result == zf)
            accurate_result = 0;
         end


         if (data_line( ((j*7)-6):j*7 ) == '1111111')
            vhdl_result = -1;
         else
            vhdl_result = bin2dec(data_line( ((j*7)-6):j*7 ));
         end


         if (accurate_result ~= vhdl_result)
            AB_val = AB(j,i)
            zf
            accurate_result
            vhdl_result
            errors = errors + 1;
         end
      end
   end
 end
```

A.15 input_generate.m

This module generates a user-defined random sequence of input bits.

```
num_of_bits = input ('How many input bits ? : ');

% Generate Input
u = round(rand([1,num_of_bits]));

save saved_u u;
```

A.16 packet_compare.m

This module performs the comparison between the output packets produced from the software model and the hardware implementation.

```
mat_seq = fopen ('E:\PROJECTS\WiMAX_Encoder\Matlab\Encoder\VHDL_Testing\encoded_seq_mat.txt', 'r');
vhdl_seq = fopen ('E:\PROJECTS\WiMAX_Encoder\Matlab\Encoder\VHDL_Testing\encoded_seq_mult_hw.txt', 'r');

packet_errors = 0;

% Full Version
num_of_packets = 100 * 19 * 4;

% Short version
% num_of_packets = 3 * 19 * 4;

for i = 1:num_of_packets
   mat_line = fgets(mat_seq);
   vhdl_line = fgets(vhdl_seq);
   [dummy length] = size(vhdl_line);     % removing the extra \n char

   encoding_errors = 0;
   for j = 1:(length - 2)
      if (mat_line(j) ~= vhdl_line(j))
         j;
         encoding_errors = encoding_errors + 1;
      end
   end

   if (encoding_errors ~= 0)
      i
      encoding_errors
      packet_errors = packet_errors + 1;
   end
end

closing_success = fclose('all')
```

A.17 Encoder_script.m

This module performs the comprehensive behavioral simulations looping through the required number of times while altering the encoder's parameters as needed. This module invokes the *finalEncoder.m* module presents up as a function.

```
% ******* Comprehensive Behavioral Simulation Script ***********
% Script to run through the randomly generated 100 packets of user bits
% and encode them using all possible code rates and ZFs.
% 100 codewords produced for each R and Zf

% close all;
clear all;
clc;

% Loading the randomly generated user bits
load('input_vector.mat'); % The input will be in a vector called (u)

% Output file
out_file = fopen('encoded_seq_mat.txt','w');

% loop to cover all code rates
for x = 1:4
  if (x == 1)
    R = 1/2;
  elseif (x == 2)
    R = 2/3;
  elseif (x == 3)
    R = 3/4;
  else
    R = 5/6;
  end

  % loop to cover all Z factors
  for zf = 24:4:96

    % loop to cover all 100 packets
    for i = 1:100

      input_length = zf*24*R;

      % Input packet
      u_vec = u( ((input_length*(i-1))+1) : (input_length*i) );

      % Calling the Encoder
      [codeword] = Encoder_func(R, zf, u_vec);

      % Output codeword
        x = num2str(codeword); % converting vector to a single string
        [blah, length_x] = size(x);
        % Eleminating white spaces
        for j = 1:length_x
          if (j == 1)
            codeword_str = x(j);
          else
            if (x(j) ~= ' ')
              codeword_str = [codeword_str x(j)];
            end
          end
        end

      fprintf (out_file, '%s\n', codeword_str);

    end
  end
end

closing_result = fclose ('all')
```

Appendix B

Testing and Verification VHDL Modules

This Appendix presents the VHDL code for the modules used in the behavioral and timing simulations as well as hardware testing.

B.1 Encoder_tb.vhd

This module interfaces with the encoder's top-level VHDL module, and simulates its behavior over a comprehensive set of run-time conditions and input packets.

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_unsigned.all;
USE ieee.numeric_std.ALL;
USE std.textio.all;

ENTITY Encoder_tb_vhd IS
END Encoder_tb_vhd;

ARCHITECTURE behavior OF Encoder_tb_vhd IS

        -- Component Declaration for the Unit Under Test (UUT)
        COMPONENT Encoder
        PORT(
                DATA_IN : IN std_logic;
                RATE : IN std_logic_vector(2 downto 0);
                BLOCK_SIZE : IN std_logic_vector(6 downto 0);
                START : IN std_logic;
                RESET : IN std_logic;
                CLOCK : IN std_logic;
                DATA_OUT : OUT std_logic;
                DONE_OUT : OUT std_logic;
                OUT_VALID : OUT std_logic;
                RFD : OUT std_logic
                );
        END COMPONENT;

        -- Component declaration for the input bits RAM
        component test_ram
        port (
                addr: IN std_logic_VECTOR(17 downto 0);
                clk: IN std_logic;
                dout: OUT std_logic_VECTOR(0 downto 0));
        end component;


        --Inputs
        SIGNAL DATA_IN  : std_logic := '0';
        SIGNAL START    : std_logic := '0';
```

```
	SIGNAL RESET			: std_logic := '1';
	SIGNAL CLOCK			: std_logic := '0';
	SIGNAL RATE			: std_logic_vector(2 downto 0):= "000";
	SIGNAL BLOCK_SIZE : std_logic_vector(6 downto 0):= "0000000";
	--Outputs
	SIGNAL DATA_OUT : std_logic; SIGNAL DATA_OUT_vec : std_logic_vector(0 downto 0);
	SIGNAL DONE_OUT : std_logic;
	SIGNAL OUT_VALID : std_logic;
	SIGNAL RFD : std_logic;

	--INPUT RAM
	SIGNAL addr	: std_logic_vector(17 downto 0):= (others => '0');
	SIGNAL input_bit : std_logic_vector(0 downto 0);

	-- Testing Constants and Variables
	--SIGNAL input_vector		: std_logic_vector (0 to 191999):= (others => '0');


	-----------------------------------------
	--	Flag FSM States
	-----------------------------------------
	 --constant STATE_0	: std_logic_vector ( 2 downto 0 ) := "000";
	 constant STATE_1	: std_logic_vector ( 2 downto 0 ) := "001";
	 constant STATE_2	: std_logic_vector ( 2 downto 0 ) := "010";
	 constant STATE_3	: std_logic_vector ( 2 downto 0 ) := "011";
	 constant STATE_4	: std_logic_vector ( 2 downto 0 ) := "100";
	 constant STATE_5	: std_logic_vector ( 2 downto 0 ) := "101";
	 signal STATE		: std_logic_vector ( 2 downto 0 );


	-- TEXT FILE(S)
	file OUT_DATA		: text open write_mode is
"E:\PROJECTS\WiMAX_Encoder\VHDL\WiMAX_Encoder\Testing_and_Results\encoded_seq_vhdl_short.txt";


	-- Function to change the type on a unsigned value to a bit_vector value
	function Unsigned2Bit(DataIn : unsigned) return bit_vector is
		variable Temp : bit_vector(DataIn'range);
	begin
		for k in DataIn'range loop
			if (DataIn(k) = '1') then
				Temp(k) := '1';
			else
				Temp(k) := '0';
			end if;
		end loop;
		return Temp;
	end function;



BEGIN

	-- Instantiate the Unit Under Test (UUT)
	uut: Encoder PORT MAP(
		DATA_IN => DATA_IN,
		DATA_OUT => DATA_OUT,
		RATE => RATE,
		BLOCK_SIZE => BLOCK_SIZE,
		START => START,
		RESET => RESET,
		DONE_OUT => DONE_OUT,
		OUT_VALID => OUT_VALID,
		RFD => RFD,
		CLOCK => CLOCK
	);

	-- Provides Input Data
	input_data : test_ram port map (
		addr => addr,
```

```
        clk => CLOCK,
        dout => input_bit
);

        -- CLOCK
        CLOCK <= not CLOCK after 10 ns; -- 50MHz
        RESET <= '0' after 151 ns;

        -- Hardwiring
        DATA_IN <= input_bit(0) after 1 ns;
        DATA_OUT_vec(0) <= DATA_OUT;



        Testing: process (CLOCK)
                variable OUT_LINE : line;  -- line variable
                variable RATE_ut      : integer := 1;
                variable BLOCK_SIZE_ut : integer := 24;
                variable packet_number : integer := 1;

        begin
          if (rising_edge(CLOCK)) then

                if (RESET = '1') then
                  addr       <= (others => '0');
                  BLOCK_SIZE <= BLOCK_SIZE;
                  RATE       <= RATE;
                  START      <= START;
                  STATE <= STATE_1;

                else
                  case STATE is

                        when STATE_1 =>
                          if (RFD = '1') then
                            BLOCK_SIZE <= std_logic_vector(to_unsigned(BLOCK_SIZE_ut,7)) after 1 ns;
                            if (RATE_ut = 1) then
                                  RATE <= "101" after 1 ns;
                            elsif (RATE_ut = 2) then
                                  RATE <= "011" after 1 ns;
                            elsif (RATE_ut = 3) then
                                  RATE <= "010" after 1 ns;
                            else
                                  RATE <= "000" after 1 ns;
                            end if;
                            START  <= '1' after 1 ns;
                            addr <= addr + "0000000000000000001";
                            STATE <= STATE_2;
                            else
                              BLOCK_SIZE <= BLOCK_SIZE;
                                  RATE       <= RATE;
                                  START      <= START;
                                  addr       <= addr;
                                  STATE <= STATE_1;
                            end if;


                        when STATE_2 =>
                          BLOCK_SIZE <= BLOCK_SIZE;
                          RATE       <= RATE;
                           START  <= '0' after 1 ns;

                           -- OUTPUT
                           if (OUT_VALID = '1') then
                              write(OUT_LINE, Unsigned2Bit(unsigned(DATA_OUT_vec)), left, 1);
                           end if;
                           if (RFD = '0') then
                              addr  <= addr - "0000000000000000001";
                              STATE <= STATE_3;
                           else
```

```
                    addr <= addr + "0000000000000000001";
                    STATE <= STATE_2;
                  end if;


                when STATE_3 =>
                  BLOCK_SIZE <= BLOCK_SIZE;
                  RATE       <= RATE;
                  START              <= '0';
                  -- OUTPUT
                  if (OUT_VALID = '1') then
                    write(OUT_LINE, Unsigned2Bit(unsigned(DATA_OUT_vec)), left, 1);
                  end if;
                  if (DONE_OUT = '1') then
                    writeline(OUT_DATA, OUT_LINE);
                        if (packet_number = 3) then
                          if (BlOCK_SIZE_ut = 96) then
                            if (RATE_ut = 4) then
                              RATE_ut := RATE_ut;
                              BLOCK_SIZE_ut := BLOCK_SIZE_ut;
                              packet_number := packet_number;
                              addr  <= addr;
                              STATE <= STATE_4;
                            else
                              RATE_ut := RATE_ut + 1;
                              BLOCK_SIZE_ut := 24;
                              packet_number := 1;
                              addr  <= (others => '0');
                              STATE <= STATE_1;
                            end if;
                          else
                            RATE_ut := RATE_ut;
                            BLOCK_SIZE_ut := BLOCK_SIZE_ut + 4;
                            packet_number := 1;
                            addr  <= (others => '0');
                            STATE <= STATE_1;
                          end if;
                        else
                          RATE_ut := RATE_ut;
                          BLOCK_SIZE_ut := BLOCK_SIZE_ut;
                          packet_number := packet_number + 1;
                          addr <= addr;
                          STATE <= STATE_1;
                        end if;
                  else
                    addr  <= addr;
                    STATE <= STATE_3;
                  end if;


                when STATE_4 =>
                  BLOCK_SIZE <= BLOCK_SIZE;
                  RATE       <= RATE;
                  START      <= START;
                  addr       <= addr;
                  STATE <= STATE_4;

                when others =>
                  BLOCK_SIZE <= (others => '0');
                  RATE       <= (others => '0');
                  START      <= '0';
                  addr       <= addr;
                  STATE <= STATE_1;

                end case;
              end if;
            end if;
        end process testing;

END;
```

B.2 Encoder_tb_packet.vhd

This module interfaces with the encoder's top-level VHDL module, and simulates its timing behavior while encoding one packet for a particular code rate and code length.

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_unsigned.all;
USE ieee.numeric_std.ALL;
USE std.textio.all;

ENTITY Encoder_tb_vhd IS
END Encoder_tb_vhd;

ARCHITECTURE behavior OF Encoder_tb_vhd IS

        -- Component Declaration for the Unit Under Test (UUT)
        COMPONENT Encoder
        PORT(
                DATA_IN : IN std_logic;
                RATE : IN std_logic_vector(2 downto 0);
                BLOCK_SIZE : IN std_logic_vector(6 downto 0);
                START : IN std_logic;
                RESET : IN std_logic;
                CLOCK : IN std_logic;
                DATA_OUT : OUT std_logic;
                DONE_OUT : OUT std_logic;
                OUT_VALID : OUT std_logic;
                RFD : OUT std_logic
                );
        END COMPONENT;

        --Inputs
        SIGNAL DATA_IN  : std_logic := '0';
        SIGNAL START            : std_logic := '0';
        SIGNAL RESET            : std_logic := '1';
        SIGNAL CLOCK            : std_logic := '0';
        SIGNAL RATE             : std_logic_vector(2 downto 0):= "000";
        SIGNAL BLOCK_SIZE : std_logic_vector(6 downto 0):= "0000000";

        --Outputs
        SIGNAL DATA_OUT : std_logic; SIGNAL DATA_OUT_vec : std_logic_vector(0 downto 0);
        SIGNAL DONE_OUT : std_logic;
        SIGNAL OUT_VALID : std_logic;
        SIGNAL RFD : std_logic;

        --Testing Constants
                -- Input generated for n = 1920.
        SIGNAL input_vector : std_logic_vector (0 to 1919) :=
"10101100101111001101001000010001010011011010111000010101011101111100011011011011111101001100100100011010001
11100110110100010111100010110100110110011010000001001100010011100000111010011001011000010100100000100011001 0
11110010111000001010000100011110011000111101101110110101100110011111001111001001111100101011111011011101101
01010100001101111100110000011001011101110000110101101110000001001010110110000001100000110011010110011100000
01111011101111000100110010111000110010000101010101101000100100011110000110111010110111010111110010110000101 1
10000101111110110011111100000100101110101010111001101010101011100000110001101101100000100110001101010111110 1
01110001001100000010000101001011011001000011010111111111111010011010100011111001000110111100111101001111111 1
11101000000100101110000110100010100110110111101101101110101000001100011100111100100010010011000010100111011
11110101011000110110100000101011000000110100000001010001101110000001000111000001011110001111111111011010100 1
10101001000000011011111101001100001000111101010110110000001000101000111010000010011000111111000101001100100 0
10100001111101011000110001011100000001110100010110001001010011010111011010110011011110011010001000101001010
01101101011100001010101110111110001101101101111110100110010010001101000111100110110100010111100010110100110 1
10011010000001001100010011100000111010011001011000010100100000100011001011110010111000001010000100011110011 0
00111101101110110101100110011111001111001001111100101011111101101110111010101010000110111110011000001100101 1
01110000110101101110000001001010110110000001100000110011101011001110000001111011101111000100111001011100011 00
```

```
10000101010101101000100100011110000110111010110111010111110010110000101110000101111110110011111100000100101110101011100110101010101110000011000110110110000010011000110101010111110101110001001100000010000101001011000100001101011111111111101001101010001111100100011011110011101001111111111101000000010";
	-- Modify the following two parameters for target Code Rate and Block Size.
	-- Available values:
	-- Rates - (000=5/6), (001=3/4b), (010=3/4A), (011=2/3b), (100=2/3a), (101=1/2)
	-- Sizes - (0011000=24), .... , (1100000=96)
	CONSTANT RATE_ut                    : std_logic_vector(2 downto 0):= "000";
	CONSTANT BLOCK_SIZE_ut : std_logic_vector(6 downto 0):= "0011000"; -- zf
		-- Input length under test (set according to Block Size)
	CONSTANT k          : integer:= 480;  -- input length   (k = n * R)
	SIGNAL state_counter   : integer:= 1;
	SIGNAL input_counter   : integer:= 0;

	-- TEXT FILES
	--          file IN_DATA    : text open read_mode is
"E:\PROJECTS\WiMAX_Encoder\VHDL\WiMAX_Encoder\Testing_and_Results\Encoder_in.txt";
	file OUT_DATA       : text open write_mode is
"E:\PROJECTS\WiMAX_Encoder\VHDL\WiMAX_Encoder\Testing_and_Results\Encoder_out.txt";


	-- Function to change the type on a signed value to a bit_vector value
	function Unsigned2Bit(DataIn : unsigned) return bit_vector is
		variable Temp : bit_vector(DataIn'range);
	begin
		for k in DataIn'range loop
			if (DataIn(k) = '1') then
				Temp(k) := '1';
			else
				Temp(k) := '0';
			end if;
		end loop;
		return Temp;
	end function;


BEGIN

	-- Instantiate the Unit Under Test (UUT)
	uut: Encoder PORT MAP(
		DATA_IN => DATA_IN,
		DATA_OUT => DATA_OUT,
		RATE => RATE,
		BLOCK_SIZE => BLOCK_SIZE,
		START => START,
		RESET => RESET,
		DONE_OUT => DONE_OUT,
		OUT_VALID => OUT_VALID,
		RFD => RFD,
		CLOCK => CLOCK
	);


		CLOCK <= not CLOCK after 10 ns; -- 50MHz
		RESET <= '0' after 151 ns;

		DATA_IN <= input_vector(0);
		DATA_OUT_vec(0) <= DATA_OUT;


		testing: process (CLOCK)
				variable OUT_LINE : line;  -- line variable
			begin
			  if (rising_edge(CLOCK) and (RESET = '0')) then
				if (state_counter = 1) then
					BLOCK_SIZE <= BLOCK_SIZE_ut after 1 ns;
				  RATE     <= RATE_ut after 1 ns;
					START   <= '1' after 1 ns;
					-- INPUT
					-- We should be loading input at this clock
```

```
                input_counter <= input_counter + 1;
                state_counter <= 2;

        elsif (state_counter = 2) then
                BLOCK_SIZE <= BLOCK_SIZE;
                RATE      <= RATE;
                START     <= '0' after 1 ns;
                -- INPUT
                for iii in 0 to (k-2) loop
                                input_vector(iii) <= input_vector(iii+1) after 1 ns;
                end loop;
                -- OUTPUT
                if (OUT_VALID = '1') then
                write(OUT_LINE, Unsigned2Bit(unsigned(DATA_OUT_vec)), left, 1);
                end if;
                if (input_counter = k) then -- counter = k already
                        input_counter <= 0; -- we are reading bit n and shifting it out
                        state_counter <= 3;
                else
                        input_counter <= input_counter + 1;
                        state_counter <= 2;
                end if;


        elsif (state_counter = 3) then
                BLOCK_SIZE <= BLOCK_SIZE;
                RATE      <= RATE;
                START     <= '0';
                input_vector <= input_vector;
                input_counter<= input_counter;
                -- OUTPUT
                if (OUT_VALID = '1') then
                write(OUT_LINE, Unsigned2Bit(unsigned(DATA_OUT_vec)), left, 1);
                end if;
                if (DONE_OUT = '1') then
                        writeline(OUT_DATA, OUT_LINE);
                        state_counter <= 4;
                else
                        state_counter <= 3;
                end if;


        elsif (state_counter = 4) then
                BLOCK_SIZE <= BLOCK_SIZE;
                RATE      <= RATE;
                START     <= '0';
                input_vector <= input_vector;
                input_counter<= input_counter;
                state_counter <= 4;

        end if;
    end if;
end process testing;

END;
```

B.3 TopLevel.vhd

This is a send/receive module that interfaces with the encoder and buffers inputs and outputs during the hardware testing process. It also interacts with the Opal Kelly Bus modules.

```
----------------------------------------------------------------------------
-- WiMAX_Encoder, An LDPC Encoder for 802.16e wireless standard
--
-- Engineer:      Samer Chomery
-- Create Date:   25th/06/07
-- Design Name:   802.16e Encoder
-- Module Name:   TopLevel
-- Project Name:  WiMAX_Encoder
-- Target Device: Spartan III
-- Tool versions: ISE 8.1.03i
-- Description:          This is the top-level module used to interface with the Opal Kelly
--               boards. It connects both the interface with the Encoder system and
--               facilitates communication with the PC
--
--
-- Dependencies:  None - This is the Top-Level module
--
-- Sub_modules:   Encoder.vhd
--               okLibrary.vhd
--               various .ngc files suuplied by OK
--
--
-- Revision:
-- Revision 0.01 - File Created
-- Additional Comments:
--
----------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

---- Uncomment the following library declaration if instantiating
---- any Xilinx primitives in this code.
--library UNISIM;
--use UNISIM.VComponents.all;

entity TopLevel is
  Port ( hi_in          : in  STD_LOGIC_VECTOR (7 downto 0);
         hi_out         : out STD_LOGIC_VECTOR (1 downto 0);
         hi_inout       : inout STD_LOGIC_VECTOR (15 downto 0);
         clk1           : in  STD_LOGIC;
         i2c_sda     : out  std_logic;
         i2c_scl     : out  std_logic;
         hi_muxsel  : out  std_logic);
end TopLevel;


architecture Behavioral of TopLevel is

        -- Host Interface Component
        component okHostInterface is
                port (
                        -- USB links
                        hi_in       : in std_logic_vector(7 downto 0);
                        hi_out      : out std_logic_vector(1 downto 0);
                        hi_inout    : inout std_logic_vector(15 downto 0);
                        -- Internal (host interface) Bus links
                        ti_clk      : out std_logic;
                        ok1         : out std_logic_vector(30 downto 0);
                        ok2         : in std_logic_vector(16 downto 0)
                );
        end component;

        -- User Design
        component Encoder is
          Port ( DATA_IN  : in  STD_LOGIC;
                 DATA_OUT : out  STD_LOGIC;
                 RATE     : in  STD_LOGIC_VECTOR (2 downto 0);
                 BLOCK_SIZE : in  STD_LOGIC_VECTOR (6 downto 0);
```

```
        START : in  STD_LOGIC;
         RESET : in  STD_LOGIC;
        DONE_OUT : out  STD_LOGIC;
         OUT_VALID: out  STD_LOGIC;
        RFD : out  STD_LOGIC;  -- See documentation regarding this signal
        CLOCK : in  STD_LOGIC);
end component;

-- RAM to receive from PipeIn
component input_ram
            Port ( addra: IN std_logic_VECTOR(10 downto 0);
                   addrb: IN std_logic_VECTOR(10 downto 0);
                   clka: IN std_logic;
                   clkb: IN std_logic;
                   dina: IN std_logic_VECTOR(0 downto 0);
                   doutb: OUT std_logic_VECTOR(0 downto 0);
                  wea: IN std_logic);
end component;

-- RAM to send to PipeOut
component output_ram
            Port ( addra: IN std_logic_VECTOR(11 downto 0);
                   addrb: IN std_logic_VECTOR(11 downto 0);
                   clka: IN std_logic;
                   clkb: IN std_logic;
                         dina: IN std_logic_VECTOR(0 downto 0);
                         doutb: OUT std_logic_VECTOR(0 downto 0);
                         wea: IN std_logic);
end component;

---------- Endpoint Components used ----------
component okWireIn is
            port (
                        ok1                 : in std_logic_vector(30 downto 0);
                        ok2                 : out std_logic_vector(16 downto 0);
                        ep_addr             : in std_logic_vector(7 downto 0);
                        ep_dataout: out std_logic_vector(15 downto 0));
end component;

component okTriggerIn is
            port (
                        ok1                 : in std_logic_vector(30 downto 0);
                        ok2                 : out std_logic_vector(16 downto 0);
                        ep_addr             : in std_logic_vector(7 downto 0);
                        ep_clk              : in std_logic;
                        ep_trigger : out std_logic_vector(15 downto 0));
end component;

component okTriggerOut is
            port (
                        ok1                 : in std_logic_vector(30 downto 0);
                        ok2                 : out std_logic_vector(16 downto 0);
                        ep_addr             : in std_logic_vector(7 downto 0);
                        ep_clk              : in std_logic;
                        ep_trigger : in std_logic_vector(15 downto 0));
end component;

component okPipeIn is
            port (
                        ok1                 : in std_logic_vector(30 downto 0);
                        ok2                 : out std_logic_vector(16 downto 0);
                        ep_addr             : in std_logic_vector(7 downto 0);
                        ep_write            : out std_logic;
                        ep_dataout: out std_logic_vector(15 downto 0));
end component;

component okPipeOut port (
ok1         : in  std_logic_vector(30 downto 0);
ok2         : out std_logic_vector(16 downto 0);
ep_addr      : in  std_logic_vector(7 downto 0);
```

```
        ep_read      : out std_logic;
        ep_datain    : in  std_logic_vector(15 downto 0));
        end component;

        ---------- ENCODER FSM STATES ----------------
        constant S_IDLE         : std_logic_vector ( 1 downto 0 ) := "01";
        constant INPUT_LOAD             : std_logic_vector ( 1 downto 0 ) := "11";
        constant FINISH_OUTPUT  : std_logic_vector ( 1 downto 0 ) := "10";
        signal STATE            : std_logic_vector ( 1 downto 0 );

        ---------- SIGNALS ----------------

        signal wire01data  : std_logic_vector(15 downto 0);
        signal trigger_01  : std_logic_vector(15 downto 0);
        signal trigger_02  : std_logic_vector(15 downto 0);
        signal trigger_03  : std_logic_vector(15 downto 0);
        signal PipeInWrite : std_logic;
        signal PipeInData  : std_logic_vector(15 downto 0);
        signal PipeOutRead : std_logic;
        signal PipeOutData : std_logic_vector(15 downto 0);
        signal ti_clk      : std_logic;
        signal ok1         : std_logic_vector(30 downto 0);
        signal ok2         : std_logic_vector(16 downto 0);

        signal ramI_addra  : std_logic_vector(10 downto 0);
        signal ramI_addrb  : std_logic_vector(10 downto 0);
        signal ramO_addra  : std_logic_vector(11 downto 0);
        signal ramO_addrb  : std_logic_vector(11 downto 0);
        signal ramI_out    : std_logic_vector(0 downto 0);
        signal ramO_in     : std_logic_vector(0 downto 0);
        signal ramO_wea    : std_logic;

        signal encoder_start : std_logic;
        signal out_valid     : std_logic;
        signal rfd           : std_logic;

begin

  -- Hardwiring (must be done according to documentation)
  -- These are pins specified in the .ucf file
  hi_muxsel <= '0';
  i2c_scl  <= 'Z';
  i2c_sda  <= 'Z';


        -- RAM_FSM
        RAM_FSM: process (ti_clk)  -- clocked by the host interface closk (48 MHz)
        begin
                if (rising_edge(ti_clk)) then
                        if (trigger_02(0) = '1') then -- RAM reset
                                ramI_addra <= (others => '0');
                                ramO_addrb <= (others => '0');
                        else
                                if (PipeInWrite = '1') then
                                        ramI_addra <= ramI_addra + "00000000001";
                                end if;

                                if (PipeOutRead = '1') then
                                        ramO_addrb <= ramO_addrb + "000000000001";
                                end if;
                        end if;
                end if;
        end process;


        -- Encoder_FSM
        Encoder_Communication: process (clk1)
        begin
                if (rising_edge(clk1)) then
                        if (wire01data(0) = '1') then
```

```
                encoder_start <= '0';
                ramO_addra <= (others => '0');
                raml_addrB <= (others => '0');
                ramO_wea  <= '0';
                STATE <= S_IDLE;
        else

                case STATE is

                        when S_IDLE =>
                                ramO_addra <= (others => '0');
                                ramO_wea  <= '0';
                                if (trigger_01(0) = '1') then -- start
                                        encoder_start <= '1';
                                        raml_addrb <= raml_addrb + "00000000001";
                                        STATE <= INPUT_LOAD;
                                else
                                        encoder_start <= '0';
                                        raml_addrb <= (others => '0');
                                        STATE <= S_IDLE;
                                end if;


                        when INPUT_LOAD =>
                                encoder_start <= '0';
                                ramO_wea  <= '1';
                                if (rfd = '1') then
                                        raml_addrb <= raml_addrb + "00000000001";
                                        STATE <= INPUT_LOAD;
                                else
                                        raml_addrb <= (others => '0');
                                        STATE <= FINISH_OUTPUT;
                                end if;
                                if (out_valid = '1') then
                                        ramO_addra <= ramO_addra + "000000000001";
                                else
                                        ramO_addra <= (others => '0');
                                end if;


                        when FINISH_OUTPUT =>
                                encoder_start <= '0';
                                raml_addrb <= (others => '0');
                                if (out_valid = '1') then
                                        ramO_wea  <= '1';
                                        ramO_addra <= ramO_addra + "000000000001";
                                        STATE <= FINISH_OUTPUT;
                                else
                                        ramO_wea  <= '0';
                                        ramO_addra <= (others => '0');
                                        STATE <= S_IDLE;
                                end if;


                        when others =>
                                encoder_start <= '0';
                                ramO_addra <= (others => '0');
                                raml_addrB <= (others => '0');
                                ramO_wea  <= '0';
                                STATE <= S_IDLE;

                        end case;
                end if;
        end if;
    end process;

-- Host Interface Instance
HostInterface: okHostInterface
        port map(
                hi_in => hi_in,
```

```
                hi_out => hi_out,
                hi_inout => hi_inout,
                ti_clk => ti_clk,
                ok1 => ok1,
                ok2 => ok2);

        -- My design (Encoder) instance
        WiMAX_Encoder: Encoder
                port map(
                        DATA_IN  => ramI_out(0),
                        DATA_OUT => ramO_in(0),
                        RATE => wire01data(15 downto 13),
                        BLOCK_SIZE => wire01data(7 downto 1),

                        START => encoder_start, -- trigger in
                        RESET => wire01data(0),
                        DONE_OUT => trigger_03(0), -- trigger out
                        OUT_VALID => out_valid,
                        RFD => rfd,
                        CLOCK => clk1);

        -- Input RAM instance
        input_block_ram : input_ram
                port map (
    addra => ramI_addra,
    addrb => ramI_addrb,
    clka => ti_clk,
    clkb => clk1,
    dina => PipeInData(0 downto 0),
    doutb => ramI_out,
    wea => PipeInWrite);

        -- Output RAM instance
        output_block_ram : output_ram
port map (
    addra => ramO_addra,
    addrb => ramO_addrb,
    clka => clk1,
    clkb => ti_clk,
    dina => ramO_in,
    doutb => PipeOutData(0 downto 0),
    wea => ramO_wea);


        ------ Endpoint Instances ------
        WireIn_01 : okWireIn
port map (
    ok1         => ok1,
    ok2         => ok2,
    ep_addr     => x"00",
    ep_dataout  => wire01data);

        TriggerIn_01 : okTriggerIn
port map (
    ok1         => ok1,
    ok2         => ok2,
    ep_addr     => x"40",
    ep_clk      => clk1,
    ep_trigger  => trigger_01);

        TriggerIn_02 : okTriggerIn
port map (
    ok1         => ok1,
    ok2         => ok2,
    ep_addr     => x"41",
    ep_clk      => ti_clk,
    ep_trigger  => trigger_02);

        TriggerOut : okTriggerOut
port map (
```

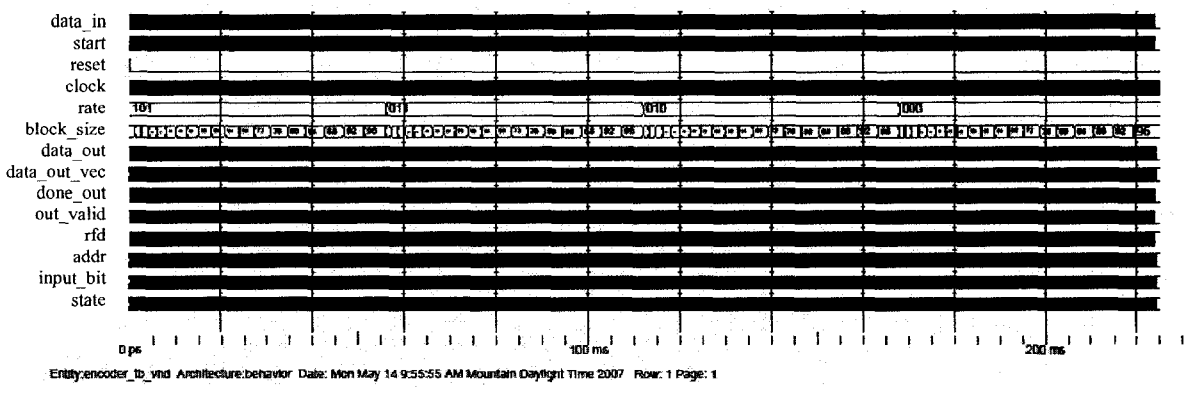
```
        ok1         => ok1,
        ok2         => ok2,
        ep_addr     => x"60",
        ep_clk      => clk1,
        ep_trigger  => trigger_03);

            PipeIn : okPipeIn
    port map (
        ok1         => ok1,
        ok2         => ok2,
        ep_addr     => x"80",
        ep_write    => PipeInWrite,
        ep_dataout  => PipeInData);

  PipeOut : okPipeOut
    port map (
                        ok1         => ok1,
        ok2         => ok2,
        ep_addr     => x"A0",
        ep_read     => PipeOutRead,
        ep_datain   => PipeOutData);


end Behavioral;
```

Appendix C

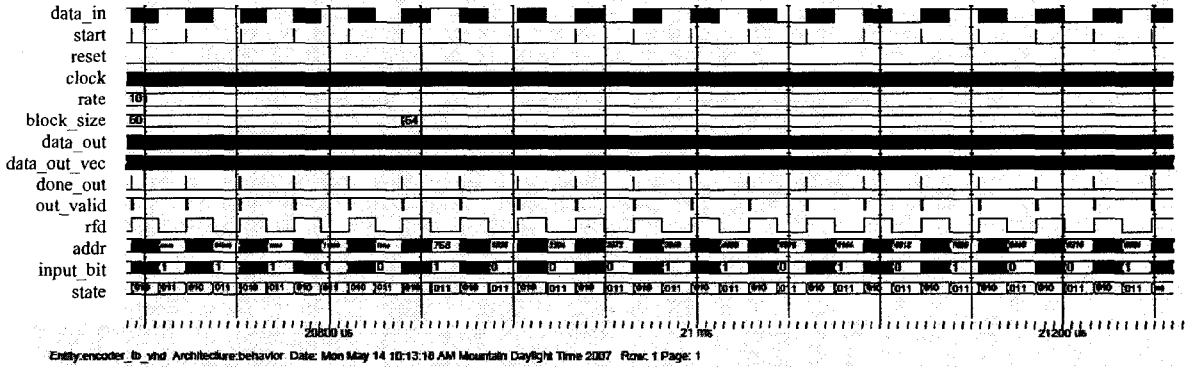
Comprehensive Behavioral Simulation Waveforms

This Appendix presents several waveforms captured during comprehensive behavioral simulations of the 802.16e LDPC encoder.

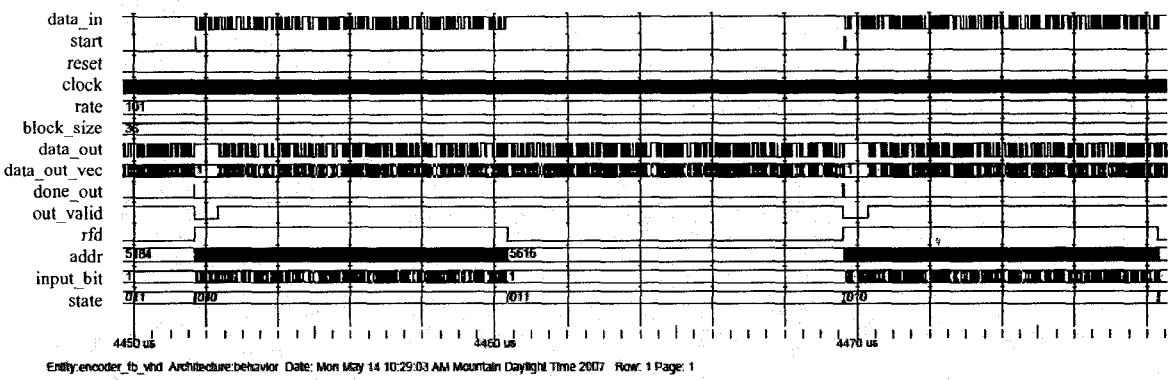
C.1 Waveform of the full behavioral simulation test



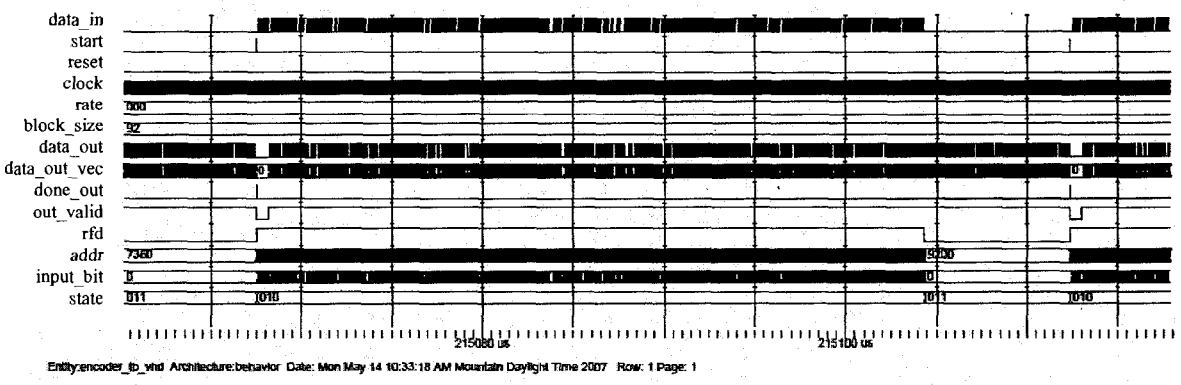
C.2 Waveform illustrating the encoder working with rate 1/2



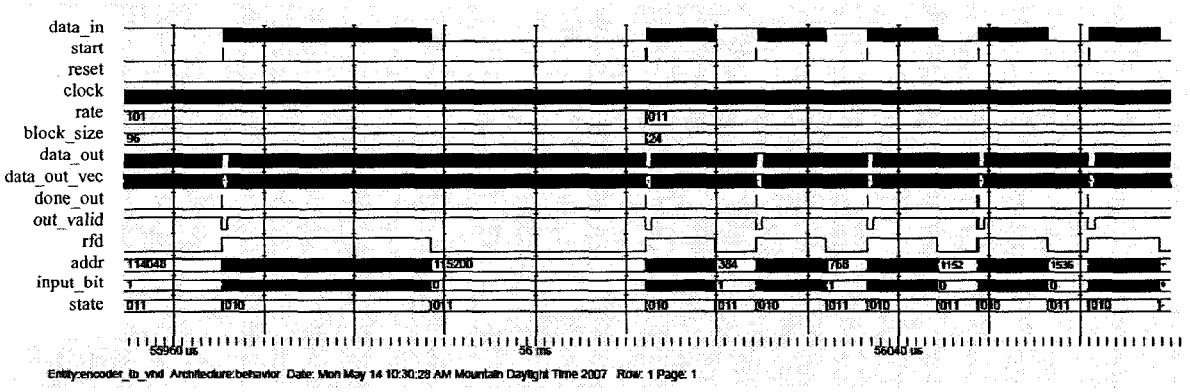
C.3 Waveform of full packet encoding at code rate 1/2 and block size 36



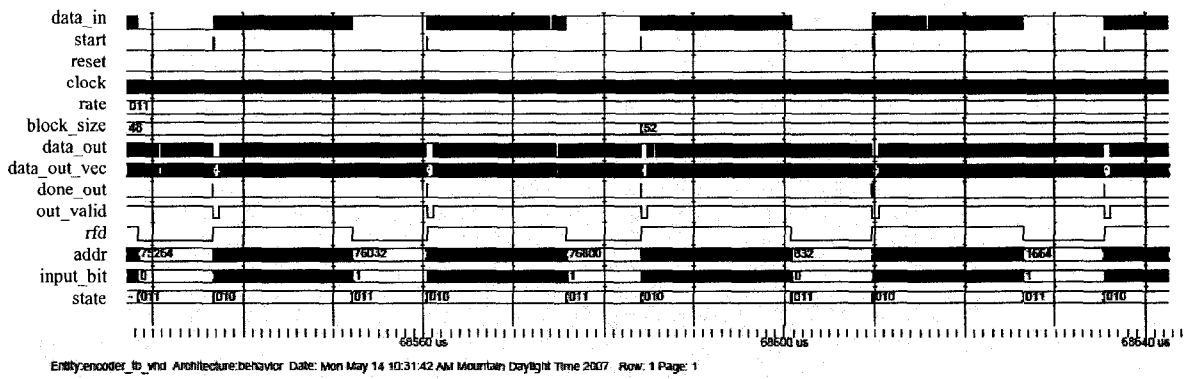
C.4 Waveform of full packet encoding at code rate 5/6 and block size 92



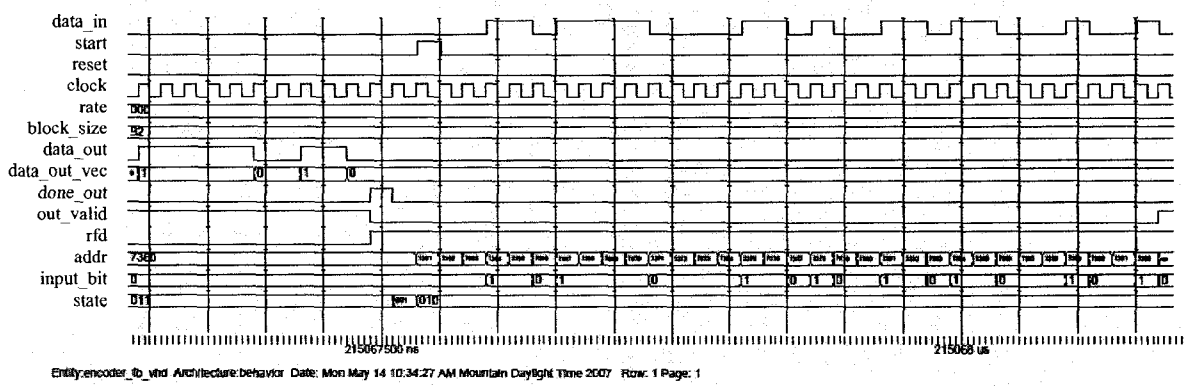
C.5 Waveform illustrating the dynamic change of encoding rate



C.6 Waveform illustrating the dynamic change of block size



C.7 Waveform illustrating a packet-to-packet transition



Appendix D

C++ Hardware Test and Control File

This Appendix presents the C++ code that was used to connect, control, and program the FPGA device on the Opal Kelly XEM3010 board. The code also runs a test of the encoder design.

```
#include <iostream>
#include <fstream>
#include <okCUsbFrontPanel.h>
#include <string>
#include <stdlib.h>
using namespace std;

#define WIREIN_RESET_MASK  0x0001
#define WIREIN_RATE_MASK   0xE000
#define WIREIN_ZF_MASK     0x00FE
#define RATE_12_BITS  0xA000
#define RATE_23_BITS  0x6000
#define RATE_34_BITS  0x4000
#define RATE_56_BITS  0x0000

// Function to convert the bit value stored in a byte into an ascii character
char bitTochar (unsigned char x);

int main ()
{
  // ****************** Variables ********************

  // XEM Device Variable
  int device_count = 0;
  string serial;
  int major_ver = 0;  //firmware major version
  int minor_ver = 0;  //firmware minor version
  int device_model = 0;
  double freq = 0;      // PLL output frequency
  long errorCode;

  // File Variables
  char in_char;
  char out_char;
  unsigned char in_buf[3840];  // 1920 * 2
  unsigned char out_buf[4608]; // 2304 * 2

  // LDPC Code related Variables
  short int zf;     // zf as loop index
  short int zf_bits;
  int rate ;        // rate as loop index
  double code_rate;  // rate in double format
  short int rate_bits;
  int code_length ;  // code length (output length)
  int input_length ; // input length
  int packet ;      // packet number as loop index
```

```
// ***************** Input/Output Files ******************
ifstream in_file;
in_file.open("input_vector.txt");
ofstream out_file;
out_file.open("encoded_seq_hw.txt");

// ***************** FPGA Communication ******************
// ---- Creating a pointer to a FrontPanel object

okCUsbFrontPanel *xem;
xem = new okCUsbFrontPanel();

// ---- Counting the devices attached to the USB bus
device_count = xem->GetDeviceCount();
cout << "\nThere are (" << device_count << ") devices attached to USB bus" << endl;

// ---- Opening USB communication with Board
if (xem->OpenBySerial() != 0) {
   printf("\nError opening device \n");
   return -1;
}
else {
        serial = xem->GetSerialNumber();
        cout << "\nDevice with serial number (" << serial << ") Opened Successfully !!" <<endl;
}

// ---- Version Checks
if (!(xem->IsFrontPanel3Supported())) {
        cout << "FrontPanel 3 is Not Supported on this board !!"<< endl;
        if (!(xem->IsFrontPanelEnabled()))      // Is any Front Panel running on board ?
           cout << " But FrontPanel is enabled!!" << endl;
}
else {
        major_ver = xem->GetDeviceMajorVersion();
        minor_ver = xem->GetDeviceMinorVersion();
        cout << "   The firmware's version number is: " << major_ver << "." << minor_ver << endl;
        device_model = xem->GetBoardModel();
        if (device_model == 3) // enumerated type
           cout << "   Device Model : XEM 3010 " << endl;
}

// ---- Setting up Frequency of PLL
okCPLL22393 *pll = new okCPLL22393;
pll->SetReference(48.0f);
pll->SetPLLParameters(0, 400, 48);
pll->SetOutputSource(0, okCPLL22393::ClkSrc_PLL0_0);
pll->SetOutputDivider(0, 8);
pll->SetOutputEnable(0, true);

xem->SetPLL22393Configuration(*pll);

freq = pll->GetOutputFrequency(0);
cout << "   PLL output freq. = " << freq << " MHz" << endl;

// ---- Loading design file to FPGA
errorCode = xem->ConfigureFPGA("wimax_encoder_mult.bit");
if (errorCode < 0)
        cout << "Configuring Device Failed!!, Error Code: "<<errorCode<<endl;

// ---- Reseting FPGA Design
xem->SetWireInValue(0x00, 0xffff, WIREIN_RESET_MASK);
xem->UpdateWireIns();
xem->SetWireInValue(0x00, 0x0000, WIREIN_RESET_MASK);
xem->UpdateWireIns();
// ---- The Design-Test loop ----
for (rate = 1; rate <= 4; rate++) {
        if (rate == 1){
           code_rate = 1/2.0;
           rate_bits = RATE_12_BITS;}
        else if (rate == 2){
```

```
            code_rate = 2/3.0;
            rate_bits = RATE_23_BITS;}
          else if (rate == 3){
            code_rate = 3/4.0;
            rate_bits = RATE_34_BITS;}
          else{
            code_rate = 5/6.0;
            rate_bits = RATE_56_BITS;}

          for (zf = 24; zf <= 96; zf = zf + 4) {
            code_length = int(zf * 24);
            input_length = int(code_length * code_rate);
            zf_bits = zf<<1;

            for (packet = 1; packet <= 100; packet++) {
              // -----> Sending Data
              // Setup rate and block length
                  xem->SetWireInValue(0x00, rate_bits, WIREIN_RATE_MASK);
                  xem->UpdateWireIns();
                  xem->SetWireInValue(0x00, zf_bits, WIREIN_ZF_MASK);
                  xem->UpdateWireIns();
                  // Read bits from in file
                  for (int i = 0; i < (input_length*2); i+=2){
                    in_file.get(in_char);
                    in_buf[i] = atoi(&in_char);
                    in_buf[i+1] = 0;
                  }
                  // Reset RAM address pointers
                  xem->ActivateTriggerIn(0x41, 0);
                  // Send bits to PipeIn
                  errorCode = xem->WriteToPipeIn(0x80, (input_length*2), in_buf);
                  if (errorCode < 0)
                    cout << "Writing data to PipeIn failed!!, Error Code: "<<errorCode<<endl;
                  // start FSM
                  xem->ActivateTriggerIn(0x40, 0);

                  // -----> Waiting for DONE_OUT signal from design
                  xem->UpdateTriggerOuts();
                  while (!(xem->IsTriggered(0x60, 0x0001)))
                  {
                    xem->UpdateTriggerOuts();
                  }
                  //cout << "Encoding Packet Done!!!\n";

        // -----> Receiving Data
                  // Receive bits from PipeOut
                  xem->ReadFromPipeOut(0xA0, (code_length*2), out_buf);
                  // Write bits to out file
                  for (int i = 0; i < (code_length*2); i+=2){
                    out_char = bitTochar(out_buf[i]);
                    out_file.put(out_char);
                  }
                  out_file.put('\n');
              }
              // returning the reading cursor to beginning of input file stream
              in_file.seekg(0, ios::beg);
            }
        }
        in_file.close();
        out_file.close();
        return 0;
}

char bitTochar (unsigned char x)
{
        if (x == 1)
                return '1';
        else
                return '0';
}
```