RICE UNIVERSITY

Semi-Parallel Architectures For Real-Time LDPC Coding

by

Marjan Karkooti

A Thesis Submitted
in Partial Fulfillment of the
Requirements for the Degree

Master of Science

Approved, Thesis Committee:

Joseph R. Cavallaro, Chair
Professor of Electrical and Computer Engineering

Behnaam Aazhang
J.S. Abercrombie Professor of Electrical and Computer Engineering

---

Ashutosh Sabharwal
Faculty Fellow of Electrical and Computer Engineering

Alexandre de Baynast
Postdoctoral Research Associate, Electrical and Computer Engineering

Houston,Texas

May, 2004

ABSTRACT

Semi-Parallel Architectures
For Real-Time LDPC Coding

by

Marjan Karkooti

Error correcting codes (ECC) enable the communication systems to have a low-power, reliable transmission over noisy channels. Low Density Parity Check codes are the best known ECC code that can achieve data rates very close to Shannon limit. This thesis presents a semi-parallel architecture for decoding Low Density Parity Check (LDPC) codes. A modified version of Min-Sum algorithm has been used for the decoder, which has the advantage of simpler computations compared to Sum-Product algorithm without any loss in performance. To balance the area-time trade-off of the design, a special structure is proposed for the parity-check matrix. An efficient semi-parallel decoder for a family of $(3, 6)$ LDPC codes has been implemented in VHDL for programmable hardware. Simulation results show that our proposed decoder for a block length of 1536 bits can achieve data rates up to 127 Mbps. The design is scalable and reconfigurable for different block sizes.

Acknowledgments

This work is dedicated to the memory of my beloved father.

I would like to thank my advisor Prof. Joseph R. Cavallaro for all his guidance, patience and insightful comments throughout this project. For all the hours he spent carefully listening to me and helping me to solve my problems. I am grateful to my committee members Prof. Behnaam Aazhang, Dr. Ashutosh Sabharwal, and Dr. Alexandre de Baynast for all their advice and comments.

I can not express my appreciation to my wonderful husband, Mahmood who has always been besides me with all his love and support. He has always believed in me and inspired me. I also want to thank my Mom and my brothers who supported me emotionally throughout this time.

I am grateful to my officemate Dr. Sridhar, who has been a friend and a mentor for me. My friends, Bahar, Farbod, Amir, Vida, Lavu, Abha, Mahsa, Giti,... thanks for all your help and kindness.

I would also like to thank National Instruments for supporting this research by a fellowship. I would specially like to thank Jim Lewis from NI, for all his support and comments, for all the hours that he spend to fix the problems related to my work. I would also thank the National Science Foundation (NSF) for partially supporting this work by grant numbers ANI-9979465, EIA-0224458 and EIA-0321266.

Contents

Illustrations

Tables

Chapter 1

Introduction

1.1 Overview

In order to have a reliable communication with low power consumption over noisy channels, error correcting codes should be used. Error correcting codes insert redundancy into the transmitted data stream so that the receiver can detect and possibly correct errors that occur during transmission. Several types of codes exist. Each of which are suitable for some special applications. The encoding/decoding algorithm for each code should be modified to fit into the space of practical hardware implementation. Researchers are searching for the best codes suitable for wireless applications. There exist a large design space with trade-offs between area of the chip, speed of decoding and power consumption. In this thesis we will address this trade-offs for a particular type of error correcting codes, namely, Low Density Parity Check code (LDPC). These codes have proven to have very good performance over noisy channels.

This chapter will begin with an overview of wireless communication and coding. Then, it will talk about the error control codes and their applications. A brief description of LDPC codes and their characteristics and applications will follow. After that, we will mention some of the related work in this area and review the existing research in designing architectures for LDPC codes.

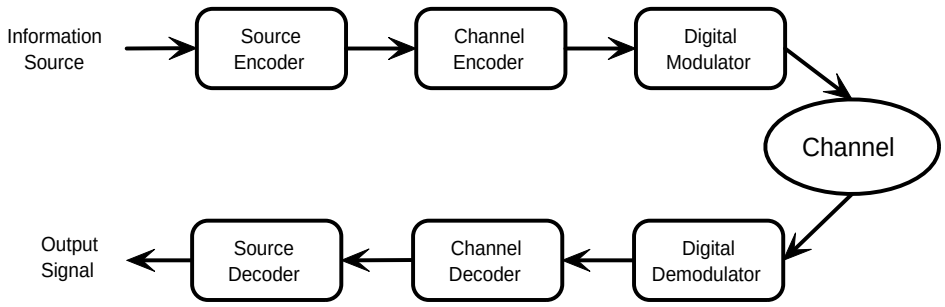

Figure 1.1 : Basic elements of a digital communication system.

1.1.1 Digital Communication System

Figure 1.1 shows a basic block diagram of a digital communication system [1]. First, information signal such as voice, video or data is sampled and quantized to form a digital sequence, then it passes through the source encoder or data compression to remove any unnecessary redundancy in the data. At this stage the information can pass through an encrypter to increase the security of the communication. Then, Channel encoder codes the information sequence so that it can recover the correct information after passing through channel. Error correcting codes such as convolutional, turbo [2] or LDPC codes are used as channel encoder. The binary sequence then is passed to the digital modulator to map the information sequence into signal waveforms. The modulator acts as an interface between the digital signal and the channel.

The communication channel is the physical medium that is used to send the signal from the transmitter to the receiver. The channel may be the atmosphere (for wireless communications), a wire line or optical fiber cable. In all of these channels, the transmitted signal is corrupted in a random manner by a variety of possible mechanisms such as additive thermal noise generated by electronic devices, man-made noise, e.g., automobile ignition noise, or atmosphere noise, e.g., lightning or

thunderstorms.

At the receiving end of the digital communications system, the digital demodulator processes the channel-corrupted transmitted waveform and reduces the waveforms to a sequence of digital values that feeds into the decrypter and channel decoder. The decoder reconstructs the original information by the knowledge of the code used by channel encoder and the redundancy contained in the received data. Channel decoders can be Viterbi [3], turbo or LDPC decoder.

Then, source decoder decompresses the data and retrieves the original information. The probability of having error in the output sequence is a function of the code characteristics, the type of modulation, channel characteristics such as noise and interference level, etc. There is a trade-off between the power of transmission and the bit error rate. Researchers are trying to minimize the power consumption while maintaining a reliable communication. This arises a need for stronger codes with more error correction abilities.

1.1.2 Coding

In 1948 Shannon published a paper which is the basis of the entire field of information theory [4]. In his work, he introduced a metric by which the information can be quantified. This metric allows one to determine the minimum possible number of symbols necessary for the error-free representation of a given message. A longer message containing the same information is said to have "redundant symbols". These can lead to the definition of three distinct types of codes [5]:

Source codes: These codes are used to remove the uncontrolled redundancy from the information symbols. Source coding reduces the symbol throughput requirement placed upon the transmitter. Source codes also include codes used

to format the data for specialized modulator/ transmitter pairs (e.g. Morse code in telegraphy).

Secrecy codes: These codes encrypt the information so that it can not be understood by anyone except the intended recipient.

Error control codes (error correcting codes or channel codes): The se codes are used to format the transmitted information so as to increase its immunity to noise. This is accomplished by inserting controlled redundancy into the transmitted information stream, allowing the receiver to detect and possibly correct errors.

As we mentioned before, in a communication system, all three types of these codes are used to increase the reliability and performance of the system.

1.1.3 Applications of error correcting codes

Since the focus of this document is on error correcting codes, here we mention some of the applications of these codes(Table 1.1).

Satellite downlinks are generally characterized as power-limited channels. Onboard batteries and solar cells are heavy and thus contribute significantly to launch costs. A communication-channel bit error rate of 10^{-5} is desired for many applications. There is thus a need for strong error control codes that operate efficiently at extremely low signal to noise ratios. Convolutional codes have been particularly successful in these applications. Turbo codes and LDPC codes are other choices for these channels. Similar principles apply to the wireless communications for cell phones, laptops and PDAs. In order to increase the battery life we need to use powerful codes like LDPC, turbo or convolutional codes.

Table 1.1 : Applications of error correcting codes.

Application	Code	Comment
Wireless communications Satellite downlink	Convolutional, Turbo, LDPC	Random noise
CD player Tape storage	Reed-Solomon + cross-interleaving	Bursty channel
Computer memory	Hamming code	-
Magnetic discs	Fire codes	-
Computer networks	CRC	-

The channel in a CD playback system consists of a transmitting laser, a recorded disc and a photo-detector. The primary contributors to errors in this channel are fingerprints and scratches of the surface. As the surface contamination affects an area that is usually quite large compared to the surface used to record a single bit, channel errors occur in bursts when the disc is played. The CD error control system handles the bursts through cross-interleaving and through the burst error-correcting capability of Reed-Solomon codes.

Various applications exist for the error control codes in computer systems, such as memory(random access and read-only memory), disk storage, tape storage and inter-processor communication. Each of these has its unique characteristics that indicates the use of certain type of codes. Hamming codes are used for the computer memories, Fire codes for magnetic discs and Reed Solomon based system is used for the tape mass storage system. Computer networks and internet use Cyclic Redundancy Code (CRC) to detect packet errors.

Table 1.2 : Performance comparison between different types of the channel codes

Code Type	Shannon Limit	LDPC	Turbo	Convolutional / Viterbi
Performance(dB) $P_{error} = 10^{-6}$	0.18	0.185	0.6	4.5

1.1.4 LDPC codes

Low Density Parity Check (LDPC) codes are a special case of error correcting codes that have recently been receiving a lot of attention because of their very high throughput and very good decoding performance. Inherent parallelism of the message passing decoding algorithm for LDPC codes, makes them very suitable for hardware implementation.

Applications of LDPC codes are not limited to digital communications. These codes can be used in any digital environment that high data rate and good error correction is important, such as optical fiber communications, satellite (digital video and audio broadcast), storage (magnetic, optical, holographic), wireless (mobile, fixed), wired line (cable modems, DSL).

Gallager [6] proposed LDPC codes in the early $1960's$, but his work received no attention until after the invention of turbo codes in 1993, which used the same concept of iterative decoding. In 1996, MacKay and Neal [7], [8] re-discovered LDPC codes. Table 1.2 shows a comparison between the best known error correcting codes. Chung et.al [9] showed that a rate 1/2 LDPC code with the block length of 10^7 in the binary input additive white Gaussian noise can achieve a threshold of just 0.0045 dB away from Shannon limit. This table shows that for very large block lengths, LDPC is the best known code in terms of performance.

Low Density Parity Check codes have several advantages over turbo codes: First, Sum-Product decoding algorithm for these codes has inherent parallelization which can be harvested to achieve a greater speed of decoding. Second, unlike turbo codes, decoding error is a detectable event which results in a more reliable system. Third, very low complexity decoders such as "Modified Min-Sum algorithm" that closely approximate Sum-Product in performance, can be designed for these codes.

While standards for Viterbi and turbo codes have emerged for communication applications, the flexibility of designing LDPC codes allows for a larger family of codes and encoder/decoder structures. Some initial proposals for LDPC codes for DVB-S2 are emerging [10].

Table 1.3 shows a comparison between the complexity of the encoder and the decoders for three different types of coding. In this table N is the code length, d is the constraint length, J is the maximum number of the decoding iterations,W_r is the row degree and W_c is the column degree of the nodes in the parity check matrix of a LDPC decoder. Comparisons show that LDPC decoding is linear with the block length, whereas in turbo, it has exponential relation with the constraint length.

In order to use LDPC codes effectively, we should design a suitable architecture for the encoder/decoder. Depending on the application, area, power or speed of decoding could be very important. Since our focus is on wireless communications, we would like to have low power architectures which are able to achieve 10 to 100 MHz data rates as it is needed for $3G$ standard or the next generation of wireless devices.

Complexity in iterative decoding has three parts. First, complexity of the computations at each node. Second, the complexity of the interconnection. And third, the number of times that local computations need to be repeated, usually referred to as the number of iterations. All of these are manageable in practice. There is a trade-off

Table 1.3 : Complexity comparison between Viterbi, turbo and LDPC encoder/decoder. In which N is the code length, d is the constraint length, J is the maximum number of the decoding iterations, W_r is the row degree and W_c is the column degree

Code Type	Encoder	Decoder
Convolutional / Viterbi	$O(Nd)$	$O(N2^d)$
Turbo	$O(N(d_1 + 1 + d_2))$	$O(JN(1 + 2^{d_1} + 2^{d_2}))$
LDPC	$O(NW_r^2)$	$O(JN(W_r + W_c))$

between the performance of the decoder, complexity and speed of decoding. We will address these trade-offs throughout this thesis in more detail.

1.2 Related Work

In the last few years some work has been done on designing architectures for LDPC coding. This subject is still very hot and researchers are looking for the best design to balance the above trade-offs. Here we mention some of the most related work in this area.

There exist different approaches on LDPC decoder implementation. Table 1.4 shows a comparison between serial, parallel and semi-parallel approaches. Serial implementation of the decoder for LDPC takes a small area for processing units, but it is very slow. This type of implementation is useful for Digital Signal Processors (DSPs) and general purpose processors. Fully parallel implementation can achieve very high data rates [11]. This approach is suitable for ASIC (Application Specific Integrated Circuit), but is infeasible for large block lengths because of the routing complexity.

Table 1.4 : Comparison between different design methodologies

Methodology	Area	Speed	Notes
Serial	Small	very low	Not useful for real-time applications
Semi-parallel	Medium	Medium	Balances the area-time trade-off
Parallel	Large	Fast	Complex routing, infeasible for large block lengths

Another approach is to have a semi-parallel decoder, in which the functional units are reused in order to decrease the area. Semi-parallel architecture takes more time to decode the codeword and the throughput is lower than fully parallel but takes smaller area.

Now, we will categorize different architectures that exist in the literature. Blanksby and Howland [11] directly mapped the Sum-Product decoding algorithm to hardware. They used the fully parallel approach and connected all the functional units with wires regarding the Tanner graph connections. Although this decoder has very good performance, the routing complexity and overhead makes this approach infeasible for larger block lengths (e.g. more than 1000 to 2000 bits). Also, implementation of all the processing units enlarges the area of the chip.

Zhang [12] offered an FPGA implementation of a $(3, 6)$ regular LDPC semi-parallel decoder which achieves up to 54 Mbps symbol decoding throughput. He used a multi-layered interconnection network to access messages from memory. Mansour [13] proposed a 1055 bit, rate 0.4, $(3, 5)$ regular semi-parallel decoder architecture which is low power. He used a fully structured parity check matrix which led to a simpler memory addressing scheme than [12]. All these architectures have used Sum-Product

or BCJR algorithms (decoding algorithm for turbo codes).

The first step in designing the LDPC encoder/decoder may seem to be designing the encoder and then design the corresponding decoder related to that particular set of LDPC code. Usually this approach leads to random-like parity check matrix, which puts a big burden on decoder design in terms of memory management, routing and interconnection of the processing units. Boutillon et al. [14] suggested reversing the conventional design sequence. Instead of trying to develop a decoder for a particular LDPC code, use an available partly parallel decoder to define a constrained random LDPC code. However, their design consisted of many random number generators, which lead to a complex hardware. The better approach is to co-design the encoder and the decoder which is used in [12] and [13].

Chen et.al. [15] designed an FPGA and ASIC Implementation of a rate 1/2 8088-b Irregular LDPC decoder. Their FPGA decoder could achieve up to 40 Mbps and the ASIC achieved 188 Mbps. Their design is one of the first implementations of the irregular LDPC codes.

There are other researchers that offered decoder architectures for different classes of LDPC codes but they have not implemented their design in hardware. For example, Kim et.al. [16] offered a parallel decoder architecture for parallel concatenated parity check codes. In these codes both parity check and generator matrices are sparse, which leads to a simpler encoding. The weak point of this approach is that the performance of the LDPC codes generated in systematic form is not as good as the codes with the same block lengths which have been constructed in random manner. Echard et.al. [17] proposed another architecture based on π-rotation parity check codes. These codes seem to have good performance but the complexity of the hardware is not obvious since they just implemented the high level design.

1.3 Thesis Contributions

The contributions of this thesis are twofold. First, we have designed a class of Low Density Parity Check codes that have good decoding performance and are suitable for hardware realization. Then, we have designed a semi-parallel decoder architecture for these codes that is flexible enough to be used for different block lengths and different code ensembles of LDPC. Modified Min-Sum algorithm has been used in this architecture which has the advantages of simpler computations with better decoding performance comparing to other decoding algorithms. The decoder has been designed and implemented using VHDL code for Xilinx FPGAs. An alternative decoder has also been designed using LabVIEW and LabVIEW FPGA. The LabVIEW version works in co-simulation and uses both the host PC and the FPGA.

1.4 Thesis Overview

The thesis is organized as follows: An introduction to linear block codes is given in chapter 2. This chapter also gives an overview of LDPC codes and their encoding/decoding algorithms. Chapter 3 discusses the code design and the proposed scalable architecture for LDPC decoder. Implementation issues, trade-offs and results are discussed in this part. An alternative architecture which has been designed using LabVIEW and LabVIEW FPGA is presented in chapter 4. Concluding remarks and future work will follow in chapter 5.

Chapter 2

Low Density Parity Check Codes

2.1 Linear Block Codes

Since Low Density Parity check codes are a special case of linear block codes, in this chapter, we will have an overview of these class of codes to set up a ground for discussing LDPC encoding and decoding. Reader is referred to [5] for more details.

In this section we will discuss some properties of linear codes. The structure inherent in linear codes makes them particularly easy to implement and analyze.

Definition: The integers $0, 1, 2, ..., p-1$, where p is a prime, form the Galois field GF(p) under modulo p addition and multiplication.

Definition: Consider a block code C consisting of N-tuples $(c_0, c_1, ..., c_{N-1})$ of symbols from $GF(q)$. C is a q-arc linear code if and only if C forms a vector subspace over $GF(q)$. Throughout this thesis we will consider binary codes so $q = 2$.

Definition: The dimension of a linear code is the dimension of the corresponding vector space. A linear code of length N and dimension K has a total of 2^K codewords of length N. Linear codes have a number of interesting properties as follows:

Property one: The linear combination of any set of codewords is a codeword. One consequence of this is that linear codes always contain the all-zero codeword.

Property two: The minimum distance of a linear code is equal to the weight of the lowest weight nonzero codeword.

Proof: Minimum distance is defined as $d_{min} = min_{\mathbf{c},\mathbf{c}' \in C, \mathbf{c} \neq \mathbf{c}'} d(\mathbf{c}, \mathbf{c}')$, which can

be re-expressed as $d_{min} = min_{\mathbf{c},\mathbf{c}' \in C, \mathbf{c} \neq \mathbf{c}'} w(\mathbf{c} - \mathbf{c}')$. Since the codeword is linear , $\mathbf{c}'' = (\mathbf{c} - \mathbf{c}')$ is a codeword and $d_{min} = min_{\mathbf{c}'' \in C, \mathbf{c}'' \neq 0} w(\mathbf{c}'')$.

This property implies that the determination of the minimum distance (and hence the error detection and correction capabilities) of a linear code is far easier than that for a general block code.

Property three: The undetectable error patterns for a linear code are independent of the codeword transmitted and always consist of the set of all nonzero codewords.

Proof: Let $\mathbf{c}$ be a transmitted codeword and $\mathbf{c}'$ be the incorrectly received codeword. The corresponding undetectable error pattern $\mathbf{e} = \mathbf{c} - \mathbf{c}'$ must be a codeword by property one.

Let $\{g_0, g_1, ..., g_{K-1}\}$ be a basis of codewords for the (N, K) binary code C. There exist a unique representation $\mathbf{c} = a_0 g_0 + a_1 g_1 + ... + a_{K-1} g_{K-1}$ for every codeword $\mathbf{c} \in C$. Since every linear combination of the basis elements must also be a code word, there is a one-to-one mapping between the set of K-symbol blocks $(a_0, a_1, ..., a_{K-1})$ over $GF(2)$ and the codewords in C. A matrix G is constructed by taking the vectors in the basis as its rows.

$$G = \begin{bmatrix} g_0 \\ g_1 \\ \vdots \\ g_{K-1} \end{bmatrix} = \begin{bmatrix} g_{0,0} & g_{0,1} & \cdots & g_{0,N-1} \\ g_{1,0} & g_{1,1} & \cdots & g_{1,N-1} \\ \vdots & \vdots & \ddots & \vdots \\ g_{K-1,0} & g_{K-1,1} & \cdots & g_{K-1,N-1} \end{bmatrix} \tag{2.1}$$

This matrix is called **Generator Matrix** for the code $\mathbf{c}$. Generator matrix can be used to directly encode K-symbol data blocks by multiplying this matrix and the

information bits. Let $\mathbf{m} = (m_0, m_1, ..., m_{K-1})$ be a binary block of uncoded data.

$$\mathbf{m}G = (m_0, m_1, ..., m_{K-1}) \begin{bmatrix} g_0 \\ g_1 \\ \vdots \\ g_{K-1} \end{bmatrix} = m_0 g_0 + m_1 g_1 + ... + m_{K-1} g_{K-1} = \mathbf{c} \quad (2.2)$$

The dual space of a linear code C is denoted by $C^{\perp}$, which is a vector space of dimension $(N - K)$. A basis $\{h_0, h_1, ..., h_{N-K-1}\}$ for $C^{\perp}$ can be found and used to construct a **parity check matrix H**.

$$H = \begin{bmatrix} h_0 \\ h_1 \\ \vdots \\ h_{N-K-1} \end{bmatrix} = \begin{bmatrix} h_{0,0} & h_{0,1} & ... & h_{0,N-1} \\ h_{1,0} & h_{1,1} & ... & h_{1,N-1} \\ \vdots & \vdots & \ddots & \vdots \\ h_{N-K-1,0} & h_{N-K-1,1} & ... & h_{N-K-1,N-1} \end{bmatrix} \quad (2.3)$$

The parity check theorem: A vector $\mathbf{c}$ is a codeword in C if and only if $\mathbf{c}H^T = 0$. The parity check matrix for a code also offers convenient means for determining the minimum distance of the code.

Theorem: Let C have the parity check matrix H. The minimum distance of C is equal to the minimum nonzero number of columns of H for which a nontrivial linear combination sums to zero.

Proof: Let the column vectors of H be $\{d_0, d_1, ..., d_{N-1}\}$. The matrix operation $\mathbf{c}H^T$ can be expressed as follows

$$\mathbf{c}H^T = (c_0, c_1, ..., c_{N-1})[d_0 d_1 ... d_{N-1}]^T = c_0 d_0 + c_1 d_1 + ... + c_{N-1} d_{N-1} \quad (2.4)$$

If $\mathbf{c}$ is a weight-w codeword, then $\mathbf{c}H^T$ is a linear combination of w columns of H. The above expression defines a one to one mapping between weight-$w$codewords and linear combinations of w columns of H. The result follows.

The problem of recovering the data block from a codeword can be greatly simplified through the use of systematic encoding. Consider a linear code $\mathbf{c}$ with generator matrix G. Using Gaussian elimination and column reordering, it is always possible to obtain a generator matrix of the form below. This can be proved by noting that the rows of a generator matrix are linearly independent and that the column rank of the matrix is equal to the row rank.

$$G = [P|I_K] = \left[\begin{array}{cccc|ccccc} p_{0,0} & p_{0,1} & \cdots & p_{0,N-K-1} & 1 & 0 & 0 & \ldots & 0 \\ p_{1,0} & p_{1,1} & \cdots & p_{1,N-K-1} & 0 & 1 & 0 & \ldots & 0 \\ p_{2,0} & p_{2,1} & \cdots & p_{2,N-K-1} & 0 & 0 & 1 & \ldots & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ p_{K-1,0} & p_{K-1,1} & \cdots & p_{K-1,N-K-1} & 0 & 0 & 0 & \ldots & 1 \end{array}\right] \tag{2.5}$$

When a data block is encoded using a systematic generator matrix, the data block is embedded without modification in the last K coordinates of the resulting codeword.

$$\begin{aligned} \mathbf{c} &= \mathbf{m}G \\ &= \left[\begin{array}{cccc} m_0 & m_1 & \ldots & m_{K-1} \end{array}\right][P|I_K] \\ &= \left[\begin{array}{cccc|cccc} c_0 & c_1 & \ldots & c_{N-K-1} & m_0 & m_1 & \ldots & m_{K-1} \end{array}\right] \end{aligned} \tag{2.6}$$

After decoding, the last K symbols are removed from the selected codeword and passed along to the data sink. The performance of the Gaussian elimination operations on a generator matrix does not alter the codeword set for the associated code. Column reordering, on the other hand, may generate codewords that are not in the original code. If a given application requires that a particular codeword set be used and thus does not allow for column reordering, it is always possible to use some set of the coordinates other than the last k for the message positions. This can slightly complicate certain encoder/decoder designs.

Given a systematic generator matrix, the corresponding parity check matrix can be obtained as shown below:

$$
\begin{aligned}
H &= [I_{N-K} | -P^T] \\
&= \left[\begin{array}{ccccc|cccc}
1 & 0 & 0 & ... & 0 & -p_{0,0} & -p_{1,0} & ... & -p_{K-1,0} \\
0 & 1 & 0 & ... & 0 & -p_{0,1} & -p_{1,1} & ... & -p_{K-1,1} \\
0 & 0 & 1 & ... & 0 & -p_{0,2} & -p_{1,2} & ... & -p_{K-1,2} \\
\vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \ddots & \vdots \\
0 & 0 & 0 & ... & 1 & -p_{0,N-K-1} & -p_{1,N-K-1} & ... & -p_{K-1,N-K-1}
\end{array}\right]
\end{aligned} \tag{2.7}
$$

For binary codewords, $-P^T = P^T$. We should note that one can always transform the corresponding generator matrix of a given parity check matrix, to the systematic form using Gaussian eliminations.

By knowing the above definitions, we are ready to discuss the properties of LDPC codes in the next section.

2.2 Low Density Parity Check Codes

Low Density Parity Check codes are a class of linear block codes corresponding to the parity check matrix H. Parity check matrix $H_{(N-K)\times N}$ consists of only *zeros* and *ones* and is very sparse which means that the density of *ones* in this matrix is very low. Given K information bits, the set of LDPC codewords $\mathbf{c} \in C$ in the code space C of length N, spans the null space of the parity check matrix H in which: $\mathbf{c}H^T = 0$.

For a (W_c, W_r) regular LDPC code each column of the parity check matrix H has W_c *ones* and each row has W_r *ones*. If degrees per row or column are not constant, then the code is *irregular*. Some of the irregular codes have shown better performance

than regular ones. But irregularity results in more complex hardware and inefficiency in terms of re-usability of functional units. In this work we have considered regular codes to achieve full utilization of processing units. Code rate R is equal to K/N which means that $(N-K)$ redundant bits have been added to the message so as to correct the errors. Ryan [18] has a very good tutorial on LDPC codes, some of the descriptions in this work has been taken from his document.

2.3 Tanner Graph

LDPC codes can be represented effectively by a bi-partite graph called a "Tanner" graph [19], [20]. A bi-partite graph is a graph (nodes or vertices are connected by undirected edges) whose nodes may be separated into two classes, and where edges may only be connecting two nodes not residing in the same class. The two classes of nodes in a Tanner graph are "Bit Nodes" and "Check Nodes." The Tanner graph of a code is drawn according to the following rule: "Check node $f_j, j = 1, ..., N-K$ is connected to bit node $x_i, i = 1, ..., N$ whenever element h_{ji} in H (parity check matrix) is a *one*." Figure 2.1 shows a Tanner graph made for a simple parity check matrix H. In this graph each Bit node is connected to *two* check nodes (Bit degree=2) and each Check node has a degree of $four$.

Definition: Degree of a node is the number of branches that is connected to that node.

Definition: A cycle of length l in a Tanner graph is a path comprised of l edges which closes back on itself. The Tanner graph in the above figure, has a cycle of length four which has been shown by dashed lines.

Definition: The Girth of a Tanner graph is the minimum cycle length of the graph. The shortest possible cycle in a bipartite graph is clearly a length-4 cycle. Length-

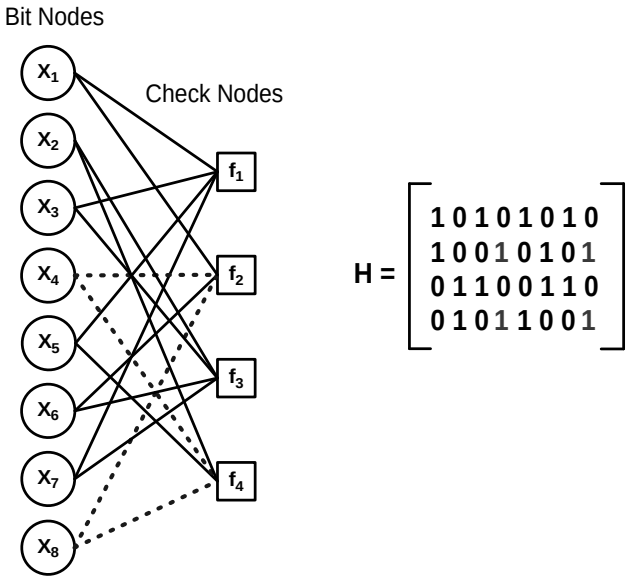

Figure 2.1 : Tanner graph of a parity check matrix.

four cycles manifest themselves in the H matrix as four 1's that lie on the corners of a sub-matrix of H.

We are interested in cycles, particularly in short cycles because they have negative impact on the decoding algorithm for LDPC codes as will be discussed later.

2.4 Designing LDPC Code

The first step in designing an LDPC code is to decide about an answer to the following questions:

1. What is the preferred *block length* of the code? It has been shown that codes with large block lengths can have very good performance. Richardson [21] showed that codes with the block length of 10^6 can achieve bit error rates that are less than 0.13 dB away from Shannon limit. The problem is that large block lengths are infeasible in practice.

2. *Regular* or *Irregular* code? For the regular code, all the Bit nodes have the same degree(d_b), and all the Check nodes have a constant degree d_c.

3. What is the *degree* of each Bit node or Check node? In other words, how many *ones* are allowed in each row or column of the parity check matrix? For the regular codes, degrees of all the Bit nodes will be the same. For irregular codes, one should decide how many different degrees is allowed to have for the Bit nodes and Check nodes. Higher degrees means that more computations should be done in each node to generate the outgoing messages. Also,nodes with higher degrees have faster convergence to their correct value.

4. What is the *Rate* of the code? Rate of the codes determines how much redundancy do we want to have in the code? For example a rate 1/2 code uses sends twice as much bits as the number of the information bits.

5. What is the maximum number of decoding iterations? We will discuss this in more detail in the decoding section.

After deciding about the above parameters, we can design the parity check matrix.

2.5 Designing the Parity Check Matrix

The Parity check matrix plays a major role in the performance The LDPC encoding/decoding. As mentioned by Gallager, this matrix should be very sparse. It also determines the complexity of the encoder/decoder. Depending on the platform who is going to do the encoding/decoding process, this matrix can be random or structured. Random matrixes are suitable for the decoders running on general purpose processors, but for dedicated hardware like FPGAs or ASICs, it is better to have a structured matrix. Structure in parity check matrix leads to a more efficient hardware representation. It also requires less memory to keep the matrix. We will discuss this

issue in more detail in the architecture design chapter. Here, we will list ways to generate a sparse matrix H. Some of these ways are more complex than the others, but they don't necessarily lead to a better code.

1. Start from all zero matrix of the size $(N - K) \times N$ and randomly invert some elements in the matrix to reach the resulting degrees for different nodes.

2. Generate H by randomly creating weight W_c columns.

3. Generate H with weight W_c columns and uniform row weights of W_r.

4. Generate H with weight W_c columns and uniform row weights of W_r with no two columns have overlap of more than one. This condition removes all the length-four cycles which results in better performance.

5. Generating H like (4) and avoiding other short cycles.

6. Generate the parity check matrix in a structured manner. For example a structure that is used in hardware design is to generate this matrix using a combination of the shifted blocks of identity matrices.

7. Generate the parity check matrix using a polynomial.

Each of the above ways have their own pros and cons, depending on the application, we can choose one of them. In this research, we have used the 6^{th} way. Since it is more suitable for the hardware design.

After designing the parity check matrix H, the generator matrix G can be derived by solving $GH^T = 0$. Performing Gaussian elimination on the resulting matrix G, will put it in systematic form $G = [I|P]$. As mentioned in the previous chapter, this results in the easy recovery of the information bits after decoding. Now, we are ready to do the encoding.

2.6 Encoding

Having the parity check matrix of a set of LDPC code, we can draw the corresponding Tanner graph. To give a general perspective about encoding of LDPC codes, we can say that one might first assign each of the information bits to a Bit node in the graph, then the values of the remaining Bit nodes are determined so that all the parity check constraints satisfy. In this way, the problem of encoding LDPC codes boils down to selecting the nodes to which assign the information bits and a strategy for calculating the values of the other Bit nodes.

In order to put encoding process in the matrix notation, to encode a message m of K bits with LDPC codes, one might compute $\mathbf{c} = \mathbf{m}G$ in which $\mathbf{c}$ is the N bit codeword and $G_{K\times N}$ is the generator matrix of the code in which $GH^T = 0$.

As an example suppose that we want to send the message $\mathbf{m} = [1011]$ over the channel. First we encode it using:

$$H = [P^T|I] = \begin{bmatrix} 1 & 1 & 1 & 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 & 1 & 0 \\ 1 & 0 & 1 & 1 & 0 & 0 & 1 \end{bmatrix} \tag{2.8}$$

and

$$G = [I|P] = \begin{bmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 0 & 1 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 1 & 1 \end{bmatrix} \tag{2.9}$$

The codeword will be equal to

$$\mathbf{c} = \mathbf{m}G = \begin{bmatrix} 1 & 0 & 1 & 1 & 0 & 0 & 1 \end{bmatrix} \tag{2.10}$$

.

At first glance, encoding might seem to be a computationally extensive task, since all the parity check equations should satisfy, which can be in quadratic relation with the code length. But in reality, encoding can be done very efficiently, and the encoding complexity can be a fraction of the decoding complexity. Several low complexity algorithms exist for the encoding of LDPC codes. Some techniques exploit the sparseness of the parity check matrix for efficient encoding [12]. Another approach is to impose some structure to Tanner graph so that encoding is transparent and simple. Repeat-Accumulate codes are an example of structured graphs . Richardson et.al. showed that transforming the Generator matrix to upper triangular form leads to reduced complexity encoding [22]. It should be noted that all the computations for the encoding are on binary values and in bit-level. So, instead of adders and multipliers, XORs and AND gates can be used which are cheaper than their counterparts.

2.7 Decoding Algorithms for LDPC Codes

In addition to presenting LDPC codes in his seminal work in 1960, Gallager also provided a decoding algorithm that is effectively optimal. Since then, other researchers have independently discovered that algorithm and related algorithms, albeit sometimes for different applications. The algorithm iteratively computes the distributions of variables in graph-based models and comes under different names, such as “Message passing algorithm”, “Sum-Product algorithm” or “belief propagation algorithm”. The iterative decoding algorithm for turbo codes is a specific instance of the Sum-Product algorithm.

In order to describe the iterative decoding, we need to use a Tanner graph for LDPC coding. Information is sent along the edges of the Tanner graph. Local

computations is done in each node of the graph. To facilitate the subsequent iterative processing, one tries to keep the graph as sparse (low density) as possible. Although that approach can be suboptimal, it is usually quite close to optimal and has an excellent complexity vs. performance tradeoff.

In order to discuss the concepts of iterative decoding, we will first introduce a simple hard-decision decoding algorithm known as "Bit flipping algorithm" that has the flavor of the more powerful algorithms. This algorithm is often of interest for very high speed applications, such as optical networking. Bit flipping algorithm has lower complexity than message passing, albeit at the cost of lower performance. This algorithm works on the hard decisions of the received signal. So, the messages are just single bits.

2.7.1 Bit Flipping Algorithm

The idea behind this algorithm is to "flip" the least number of bits until all the parity checks are satisfied. Suppose that each Bit node starts with a value of either *zero* or *one*. At each iteration the Bit node decides either to flip its value or to keep it unchanged. When a large number of the neighboring check equations are unsatisfied, the Bit node decides to flip its value. This follows from the assumption that the Bit node value which is in error, has the most number of unsatisfied check equations. This process is easier when H is low density, i.e., when only a few bits are involved in each check equation and each bit is involved in only a few check equations. We will describe this algorithm by means of an example.

Example: Consider a $(7, 4)$ Hamming code with parity check matrix:

$$H^T = \left[\begin{array}{cccc|ccc} 1 & 1 & 1 & 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 & 1 & 0 \\ 1 & 0 & 1 & 1 & 0 & 0 & 1 \end{array}\right] = \left[\begin{array}{c|c} P^T & I \end{array}\right] \tag{2.11}$$

and generator matrix $G = [I|P]$.

Suppose that transmitted codeword is: $\mathbf{c} = [1011001]$ and the received codeword with one error is:

$\mathbf{y} = \mathbf{c} + [0100000] = [1111001]$.

We may decode $\mathbf{y}$ to correct the error via the series of the parity checks implied by $\mathbf{y}H^T = 0$. For example from the columns of H^T, we can write:

$$\begin{aligned} y_1 + y_2 + y_3 + y_5 &= 0 \\ y_1 + y_2 + y_4 + y_6 &= 0 \\ y_1 + y_3 + y_4 + y_7 &= 0 \end{aligned}$$

Note that all the bits in each equation should satisfy parity with modulo-2 addition. The top two equations fail to "check", so we suspect that one of the common bits between those two equations should be in error (y_1 or y_2). Since y_1 is also used in the other equation which checks, we can conclude that y_2 was in error and should be flipped. In this way, all the parity checks will be satisfied. This example uses the assumption that it is more likely to have one bit error.

Consider again the $(7, 4)$ Hamming code discussed above, where code bits $c_k \in \{0, 1\}$ are to be transmitted over an AWGN channel as the symbols $x_k \in \{\pm 1\}$, with $x_k = (-1)^{c_k}$. We can draw the Tanner graph for this code as figure 2.2. Also, more information can be included to the graph to facilitate description of the decoding algorithm.

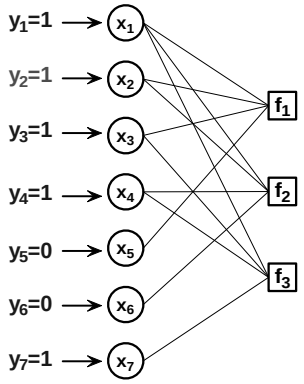

Figure 2.2 : Tanner graph of the example Hamming code.

The y_k is a received symbol from AWGN channel. $y_k = x_k + n_k$, where n_k is the noise for k^{th} bit. The graph edges can be considered as information-flow pathways to be followed in the iterative computation of various probabilistic quantities. This can be seen as a generalization of the use of the trellis branches as paths in the Viterbi algorithm implementation of maximum likelihood sequence detection/decoding.

Consider now the subgraph of the graph corresponding to the first column of the parity check matrix H (figure 2.3). In one computation of the message passing algorithm, node x_1 passes all the information that it has available to it to each of the check nodes f_j, excluding the information the receiving node already possesses. As an example the message being passed from x_1 to node f_3($x_1 \longrightarrow f_3$) is the information from the channel(via y_1) and extrinsic information node x_1 had received from nodes f_1 and f_2, on a previous half-iteration. Note that Extrinsic information are the messages to be passed between nodes. In one half iteration of the decoding algorithm, such computations ($x_i \longrightarrow f_j$) are made for all Bit-node/Check node pairs. In the other half-iteration, messages are passed in the opposite direction (from Check nodes to Bit nodes, $f_j \longrightarrow x_i$)(figure 2.4). Decoding is stopped after a maximum

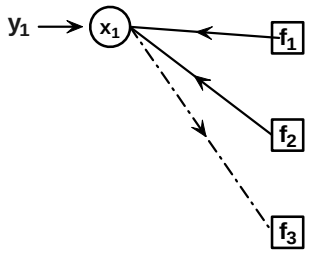

Figure 2.3 : Message passed to/from Bit nodes.

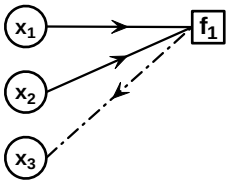

Figure 2.4 : Message passed to/from Check nodes.

number of iterations is reached or if all the parity check equations are satisfied.

Here is a summary of the decoding algorithm:

- Initialize nodes.
- Pass the messages from Bit nodes to Check nodes.
- Pass the messages from Check nodes to Bit nodes.
- Approximate the codeword from probabilistic information residing in Bit nodes. If $\mathbf{c}H^T = 0$ or maximum number of iterations reached, then stop otherwise continue iterations.

Like the optimal MAP symbol by symbol decoding of trellis codes, we are interested in computing the *a posteriori* probability (APP) that a given bit in $\mathbf{c}$ equals *one*, given the received block $\mathbf{y}$. The codeword $\mathbf{c}$ should satisfy parity check con-

straints. Without loss of generality, let us focus on the decoding of bit c_i. Thus we are interested in computing $pr(c_i = 1|\mathbf{y}, S_i)$. Where S_i is the event that the bits in $\mathbf{c}$ satisfy the W_c parity check equations involving c_i.

In order to discuss the decoding process, we need to explain some lemmas that Gallager introduced in his paper [23].

Lemma 1: Consider a sequence of m independent binary digits $\mathbf{a} = (a_0, a_1, ..., a_m)$ in which $pr(a_k) = p_k$.

Then the probability that $\mathbf{a}$ contains an even number of *ones* is :

$$1/2 + 1/2 \prod_{k=1}^{m} (1 - 2p_k) \tag{2.12}$$

And the probability that $\mathbf{a}$ contains an odd number of *ones* is one minus this value:

$$1/2 - 1/2 \prod_{k=1}^{m} (1 - 2p_k). \tag{2.13}$$

Proof: This proof follows by induction on m.

$$\begin{aligned} m = 2 : pr(even) &= pr(a_1 + a_2 = 0) \\ &= p_1 p_2 + (1 - p_2)(1 - p_2) \\ &= 1/2 + 1/2(1 - 2p_1)(1 - 2p_2) \end{aligned} \tag{2.14}$$

Assume that equation (2.12) holds for $m = L - 1$. Then with $Z_L = a_1 + ... + a_L$, we have

$$\begin{aligned} pr(Z_L = 0) &= pr(Z_{L-1} + a_L = 0) \\ &= 1/2 + 1/2(1 - 2pr(Z_{L-1} = 1))(1 - 2p_L) \\ &= 1/2 + 1/2 \prod_{k=1}^{m} (1 - 2p_k). \end{aligned} \tag{2.15}$$

Now we need to define some notation that are used in decoding algorithm:

- $R_j = \{i : h_{ji} = 1\}$: The set of column locations of the *ones* in the j^{th} row.
- $R_{j\backslash i} = \{i' : h_{ji'} = 1, i' \neq i\}$: The set of column locations of the *ones* in the j^{th} row, excluding location i.
- $C_i = \{j : h_{ji} = 1\}$: The set of row locations of *ones* in the i^{th} column.
- $C_{i\backslash j} = \{j' : h_{j'i} = 1, j' \neq j\}$: The set of row locations of *ones* in the i^{th} column, excluding location j.

Theorem (Gallager): The *a posteriori* probability (APP) ratio for c_i given the received word $\mathbf{y}$ and the event S_i is

$$\frac{Pr(c_i = 0|\mathbf{y}, S_i)}{Pr(c_i = 1|\mathbf{y}, S_i)} = \frac{(1 - P_i)}{P_i} \frac{\prod_{j \in C_i}(1 + \prod_{i' \in R_{j\backslash i}}(1 - 2P_{ji'}))}{\prod_{j \in C_i}(1 - \prod_{i' \in R_{j\backslash i}}(1 - 2P_{ji'}))}. \tag{2.16}$$

Under the assumption that the received samples in $\mathbf{y}$ are statistically independent.

Proof: By using Bayes' rule we have:

$$\frac{Pr(c_i = 0|\mathbf{y}, S_i)}{Pr(c_i = 1|\mathbf{y}, S_i)} = \frac{(1 - P_i)}{P_i} \frac{Pr(S_i|c_i = 0, \mathbf{y})/P(S_i)}{Pr(S_i|c_i = 1, \mathbf{y})/P(S_i)}. \tag{2.17}$$

Given c_{i+1}, the other $W_r - 1$ bits in a given parity check equation involving c_i must contain an odd number of ones. From lemma1, the probability of an odd number of *ones* in the other $W_r - 1$ bits of the j^{th} parity check equation is:

$$1/2 - 1/2 \prod_{i' \in R_{j\backslash i}} (1 - 2P_{ji'})$$

Similar comment holds for the $c_i = 0$ case. Because samples in y_i are statistically independent, the probability that all W_c parity checks are satisfied is the product of all such probabilities:

$$\frac{Pr(c_i = 0|\mathbf{y}, S_i)}{Pr(c_i = 1|\mathbf{y}, S_i)} = \frac{\prod_{j \in C_i}(1 + \prod_{i' \in R_{j\backslash i}}(1 - 2P_{ji'}))}{\prod_{j \in C_i}(1 - \prod_{i' \in R_{j\backslash i}}(1 - 2P_{ji'}))}. \tag{2.18}$$

Computation of the above formula is very complex, so Gallager provided an iterative algorithm which is the "Message Passing Algorithm".

Now we will combine the theorem and the lemma to get more compact result. Suppose that $r_{ji}(b)$ is the message to be passed from the Check node f_j to the Bit node x_i in which $b \in \{0,1\}$. This is the probability of the j^{th} check equation being satisfied given bit $c_i = b$ and the other bits have a separable distribution given by $\{q_{j'i}\}_{j' \neq j}$.

Also suppose that $q_{ji}(b)$ is the message to be passed from bit node x_i to check node f_j regarding the probability that $c_i = b$, $b \in \{0,1\}$. It is the probability that $c_i = b$ given extrinsic information from all check nodes, except node f_j, and channel sample y_i. Then using the lemma we can write:

$$r_{ji}(0) = 1/2 + 1/2 \prod_{i' \in R_{j \setminus i}} (1 - 2p_{ji'}) \tag{2.19}$$

$$r_{ji}(1) = 1/2 - 1/2 \prod_{i' \in R_{j \setminus i}} (1 - 2p_{ji'}). \tag{2.20}$$

Thus, the theorem may be written as:

$$\frac{Pr(c_i = 0|\mathbf{y}, S_i)}{Pr(c_i = 1|\mathbf{y}, S_i)} = \frac{(1 - p_i) \prod_{j \in C_i} r_{ji}(0)}{p_i \prod_{j \in C_i} r_{ji}(1)}. \tag{2.21}$$

Also, we can write :

$$q_{ji}(0) = (1 - p_i) \prod_{j' \in C_i} r_{j'i}(0) \tag{2.22}$$

$$q_{ji}(1) = p_i \prod_{j' \in C_i \setminus j} r_{j'i}(1). \tag{2.23}$$

The algorithm iterates back and forth to update q_{ji} and r_{ji}. To complete the loop, we need to make the assignment: $p_{ji'} \longleftarrow q_{ji'}(1)$. Before we give the iterative decoding algorithm, we need the following results:

Lemma 2: Suppose $y_i = x_i + n_i$ where $n_i \sim \mathcal{N}(0, \sigma^2)$ and $pr(x_i = +1) = pr(x_i = -1) = 1/2$. Then, for $x = \{-1, +1\}$ we can write:

$$Pr(x_i = x|y) = \frac{1}{1 + e^{-2yx/\sigma^2}} \tag{2.24}$$

Proof:

$$\begin{aligned} Pr(x_i = x|y) &= \frac{p(y|x_i = x)Pr(x_i = x)}{p(y)} \\ &= \frac{1/2e^{-(y-x)^2/2\sigma^2}}{1/2e^{-(y-1)^2/2\sigma^2} + 1/2e^{-(y+1)^2/2\sigma^2}} \\ &= \frac{e^{xy/\sigma^2}}{e^{y/\sigma^2} + e^{-y/\sigma^2}} \\ &= \frac{1}{e^{y(1-x)/\sigma^2} + e^{-y(1+x)/\sigma^2}} \\ &= \frac{1}{1 + e^{-2xy/\sigma^2}}. \end{aligned}$$

2.7.2 Sum-Product Algorithm - Probability Domain

For a $(N-K) \times N$ parity check matrix, we define $N-K$ "Check nodes" and N "Bit nodes". Check nodes represent parity check equations and Bit nodes represent the code bits. Decoding is performed iteratively. In each iteration, every Bit node passes a message to the Check nodes that are connected to it. In the next half iteration, each Check node sends a message to the Bit nodes. This message is a function of all the extrinsic information that it has received from the Bit nodes in the last part. Then, it checks if the codeword is valid or not. It does the iterations until it finds the valid code word or reaches the maximum number of the iterations.

The following steps should be done for all the is and js for which the element in the parity check matrix is a one ($hij = 1$).

Step 0: Initialize q_{ji} by:

$$q_{ji}(0) = 1 - p_i = Pr(x_i = +1|y) = \frac{1}{1 + e^{-2y_i/\sigma^2}} \tag{2.25}$$

$$q_{ji}(1) = p_i = Pr(x_i = -1|y) = \frac{1}{1 + e^{2y_i/\sigma^2}}. \tag{2.26}$$

Step 1: Horizontal stepping on r_{ji} by:

$$r_{ji}(0) = 1/2 + 1/2 \prod_{i' \in R_{j\backslash i}} (1 - 2q_{ji'}(1)) \tag{2.27}$$

$$r_{ji}(1) = 1 - r_{ji}(0). \tag{2.28}$$

Step 2: Vertical stepping on q_{ji} :

$$q_{ji}(0) = K_{ji}(1 - p_i) \prod_{j' \in C_i} r_{j'i}(0) \tag{2.29}$$

$$q_{ji}(1) = K_{ji} p_i \prod_{j' \in C_i \backslash j} r_{j'i}(1). \tag{2.30}$$

where the constants K_{ji} are chosen to ensure that $q_{ji}(0) + q_{ji}(1) = 1$

Step 3: For all the i's compute:

$$Q_i(0) = K_i(1 - p_i) \prod_{j \in C_i} r_{ji}(0) \tag{2.31}$$

$$Q_i(1) = K_i p_i \prod_{j \in C_i} r_{ji}(1). \tag{2.32}$$

where the constants K_i are chosen to ensure that $Q_i(0) + Q_i(1) = 1$.

Step 4: For every row index i:

$$\hat{c}_i = \begin{cases} 1 & if Q_i(1) > 0.5 \\ 0 & else \end{cases} \tag{2.33}$$

If $\hat{\mathbf{c}} H^T = 0$, or if maximum number of iteration is reached then stop, else continue iterations from step 1.

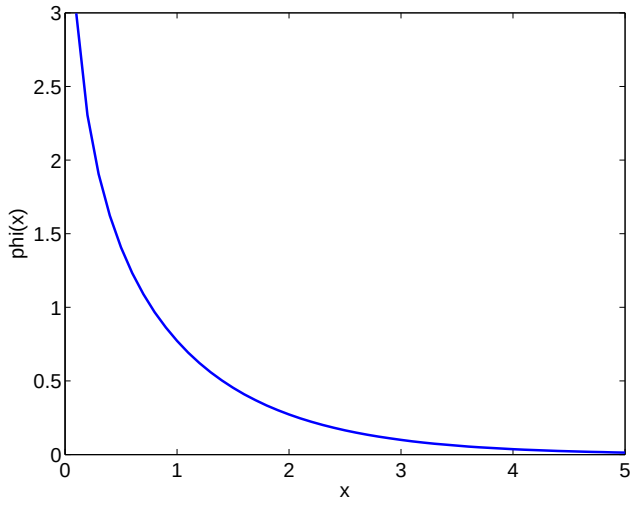

Figure 2.5 : The $\phi(x) = -\log(\tanh(x/2))$ function which is part of the Log-Sum-Product algorithm.

Sum-Product algorithm used in decoding of LDPC codes requires a large number of multiplications of probabilities which makes the algorithm numerically unstable, specially for very long codes. Thus as with the Viterbi and BCJR algorithms, a log-domain version of the algorithm is preferred.

Now, we define the following log likelihood ratios as part of the decoding algorithm:

$$Lc_i = \log \frac{P_r(x_i = +1|y_i)}{Pr(x_i = -1|y_i)} \tag{2.34}$$

$$Lr_{ji} = \log \frac{r_{ji}(0)}{r_{ji}(1)} \tag{2.35}$$

$$Lq_{ji} = \log \frac{q_{ji}(0)}{q_{ji}(1)} \tag{2.36}$$

$$LQ_i = \log \frac{Q_i(0)}{Q_i(1)}. \tag{2.37}$$

2.7.3 Sum-Product Algorithm - Log Domain

This algorithm iterates over columns and rows of parity check matrix H, and operates on nonzero entries by performing the following steps:

Step 0: initialize Lq_{ji} by:

$$Lq_{ji} = Lc_i = 2y_i/\sigma^2 \tag{2.38}$$

Step 1: Evaluate Lr_{ji} by:

$$Lr_{ji} = (\Pi_{i' \in R_{j\setminus i}} \alpha_{ji'}).\phi(\Sigma_{i' \in R_{j\setminus i}} \phi(\beta_{ji'})) \tag{2.39}$$

where,

$$\begin{aligned} \alpha_{ji} &= sign(Lq_{ji}) \\ \beta_{ji} &= \|Lq_{ji}\| \\ \phi(x) &= -\log(\tanh(x/2)) \\ &= \log(\frac{e^x+1}{e^x-1}), \end{aligned}$$

Step 2:

$$Lq_{ji} = Lc_i + \Sigma_{j' \in C_i \setminus j} Lr_{j'i} \tag{2.40}$$

Step 3:

$$LQ_i = Lc_i + \Sigma_{j \in C_i} Lr_{ji} \tag{2.41}$$

Step 4: For every row index i:

$$\hat{c}_i = \begin{cases} 1 & if LQ_i < 0 \\ 0 & else \end{cases} \tag{2.42}$$

If $\hat{\mathbf{c}}H^T = 0$ or if maximum number of iteration is reached then stop, else continue iterations from step 1.

2.7.4 Min-Sum Algorithm

Consider the update equation for Lr_{ji} in the Sum-Product algorithm:

$$Lr_{ji} = (\Pi_{i' \in R_{j \setminus i}} \alpha_{ji'}).\phi(\Sigma_{i' \in R_{j \setminus i}} \phi(\beta_{ji'})) \tag{2.43}$$

The $\phi(x)$ is a function which is decreasing for the values of $x > 0$. Figure (2.5) shows a plot of this function. It is intuitive that the term corresponding to the smallest β_{ji} in the above summation dominates, so that:

$$\phi(\Sigma_{i' \in R_{j \setminus i}} \phi(\beta_{ji'})) = \phi(\phi(min_{i'} \beta_{ji'})) = min_{i'} \beta_{ji'} \tag{2.44}$$

Notice that the second equality follows from $\phi(\phi(x)) = x$. Thus the Min-Sum algorithm is the same as Sum-Product algorithm in which step (1) is replaced by this equation:

Step 1':

$$Lr_{ji} = (\Pi_{i' \in R_{j \setminus i}} \alpha_{ji'}).min_{i' \in R_{j \setminus i}} \beta_{ji'}) \tag{2.45}$$

Because of the approximation in this equation, there is a degradation in the performance of Min-Sum comparing to Sum-Product algorithm.

2.7.5 Modified Min-Sum Algorithm

In the literature, it has been experimentally shown that scaling the soft information during the decoding using min-sum algorithm, results in better performance. Scaling slows down the convergence of iterative decoding and reduces the overestimation error comparing to Sum-Product algorithm. Heo [24] showed that density evolution techniques can be used to determine the optimal scaling factor. He also showed that for a (3,6) LDPC code scaling factor of 0.8 is optimal. In this algorithm, it is enough

to change the step 2 in Min-Sum algorithm with :

$$[Step2' :]Lq_{ji} = (Lc_i + \Sigma_{j' \in C_i \backslash j} Lr_{j'i}) * \gamma \tag{2.46}$$

in which γ is the scaling factor.

We will discuss the important parameters and simulation results of our design in the next chapter. Implementation diagrams and statistics of the designed architecture will follow.

Chapter 3

LDPC Decoder Design

3.1 Algorithmic Parameters of the Design

The structure of the parity check matrix has a major role in the performance of the decoder. Finding a good matrix is an essential part of the decoder design. As mentioned earlier, parity check matrix determines the connections between different processing nodes in the decoder according to the Tanner graph. Also, degree of each node is proportional to the amount of computations that should be done in that node. For example a $(3, 12)$ LDPC has twice as many connections as a $(3, 6)$ code, which results in twice as many messages to be passed across the nodes and the memory needed to store those messages is twice the memory required for a $(3, 6)$ code. Chung et.al. [25] showed that $(3, 6)$ is the best choice for rate 1/2 LDPC code. We have used a $(3, 6)$ code in our design.

In each iteration of the decoding, first all the Check nodes receive and update their messages and then, in the next half-iteration all the Bit nodes update their messages. If we choose to have a one-to-one relation between processing units in the hardware and Bit and Check nodes in the Tanner graph, then the design will be fully parallel. Obviously, a fully parallel approach takes a large area; but is very fast. There is also no need for central memory blocks to store the messages. They can be latched close to the processing units [11]. With this approach, the hardware design can be fixed to relate to a special case of the parity check matrix.

Table 3.1 : LDPC decoder hardware resource comparison.

Design Parameters	Fully Parallel	Semi Parallel	Fully Serial
Code Length	N	N	N
Message Length	K	K	K
Code Rate	K/N	K/N	K/N
No. of BFUs	N	N/S	1
No. of CFUs	$N-K$	$(N-K)/S$	1
Memory Bit	$(W_c+1)Nb$	$(W_c+1)Nb$	$(W_c+1)Nb$
Wire	$2(W_c+1)Nb$	$(W_c+1)Nb/S$	$2(W_c+W_r)b$
Time Per Iteration	T	ST	$T/2(2N-K)$
Counter (Address Generator)	0	$W_r(W_c+1)$	1
Address Decoder (for Memories)	0	$W_r(W_c+1)$	1
Memory Type	Scattered Latches	Several Memory Blocks	One Memory Block

Table 3.1 shows a comparison between the resources for a parallel, semi-parallel or serial implementation of the decoder. In this table , W_c is the degree of Bit nodes, W_r is the degree of the Check nodes, b is the number of the bits per message and S is the folding factor for the semi-parallel design.

Implementing LDPC decoding algorithm in fully-serial architecture has the smallest area since it is sufficient to have just one Bit Functional Unit (BFU) and one Check Functional Unit (CFU). The fully-serial approach is suitable for Digital Signal Processors (DSPs) that have only a few functional units. However, speed of the decoding is very low in a serial decoder.

To balance the trade-off between area and time, the best strategy is to have a semi-parallel design. This involves the creation of "l_c" CFUs and "l_b" BFUs, in which $l_c << N - K$ and $l_b << N$ and then the reuse of these units throughout decoding time. For semi-parallel design, the parity check matrix should be structured in order to enable re-usability of units. Also, in order to design a fast architecture for LDPC decoding, we should first design a good H matrix which results in good performance. Following the block-structured design similar to [13], we have designed H matrices for $(3, 6)$ LDPC codes.

3.1.1 Design of Parity Check Matrix

Figure 3.1 shows the structured parity check matrix that has been used in this thesis. The matrix consists of $(3 \times 6 = 18)$ blocks of size s in which s is a power of *two*. Each $s \times s$ block is an identity matrix that has been shifted to the right a_{mn} times, $m = 1, ..., 3, n = 1, ..., 6$. The shift values can be any value between 0 and $s - 1$ [26], [27], and have been determined with a heuristic search for the best performance in the codes of the same structure. Our approach is different from [13] since the sub-

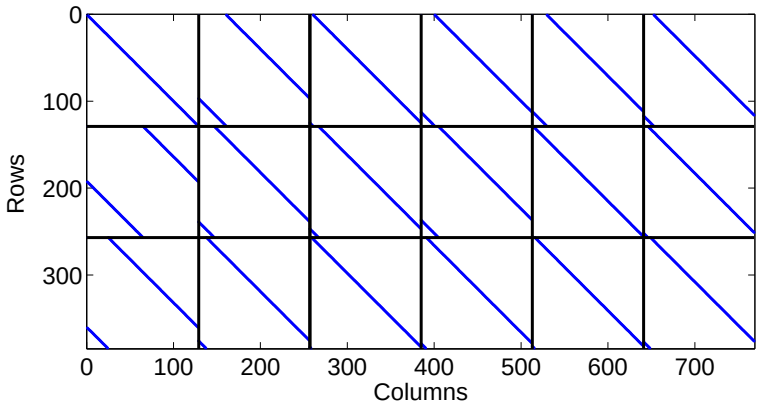

Figure 3.1 : Parity Check Matrix of a (3,6) LDPC code.

block length is not a prime number. Also, shifts are determined by simulations and searching for the best matrix that satisfies our constraints (with the highest girth). Mao *etal.* [28] performed a heuristic search to find good LDPC codes at short block lengths. They introduced an algorithm to determine the average girth of a graph and showed that the *girth distribution* is an important entity associated with the Tanner graph of a code which relates the performance of the iterative belief propagation algorithm to the structure of the graph. This means that the graphs with the highest average girth have the best performance comparing to the other graphs of the similar block length.

3.1.2 Average girth calculation algorithm

Suppose that girth at node u is the length of the shortest cycle that passes through that node. Girth distribution, $g(l), l = 4, 6, ..l_{max}$ of a Tanner graph is the fraction of the symbol nodes with girth l, where l_{max} is the maximum girth in the graph. The average girth of the Tanner graph is

$$\sum_{k=2}^{l_{max}/2} g(2k).2k. \tag{3.1}$$

To compute the girth at a given node u, a tree is "grown" step by step starting from the "root" u. At step k, all the nodes at distance k from u are included into the tree. This procedure is repeated until, at step k, a node connected to at least two nodes included at step $k-1$ is included. This introduces the formation of the first cycle. Integer $2k$ is then the girth at node u. The complexity of this algorithm is low and quite manageable for the short block lengths. The complexity of computing the girth distribution is $O(n^2)$ where n is the block length.

In order to design a good decoder, we have to decide about some parameters such as *type of the decoding algorithm, block length, maximum number of iterations, number of bits in each message.*

3.1.3 Choosing the suitable decoding algorithm

Figure 3.2 shows the result of some simulations based on the designed LDPC code. Simulations are done for the 768 bits block of the rate 1/2 LDPC code which is sent through additive white Gaussian noise (AWGN) channel. Figure shows that Min-Sum algorithm which is an approximation of the Sum-Product, suffers from some performance loss because of the approximations. On the other hand, Modified Min-Sum shows even better performance than Sum-Product in some SNR ranges. For this simulations, maximum number of iterations is set to 20.

Table 3.2 shows a comparison between the number of calculations needed for each of the decoding algorithms for a $(3, 6)$ LDPC code in each iteration of decoding. From the table it is clear that Modified Min-Sum algorithm substitutes the costly function evaluations with addition and shift. Although Modified Min-Sum has a few

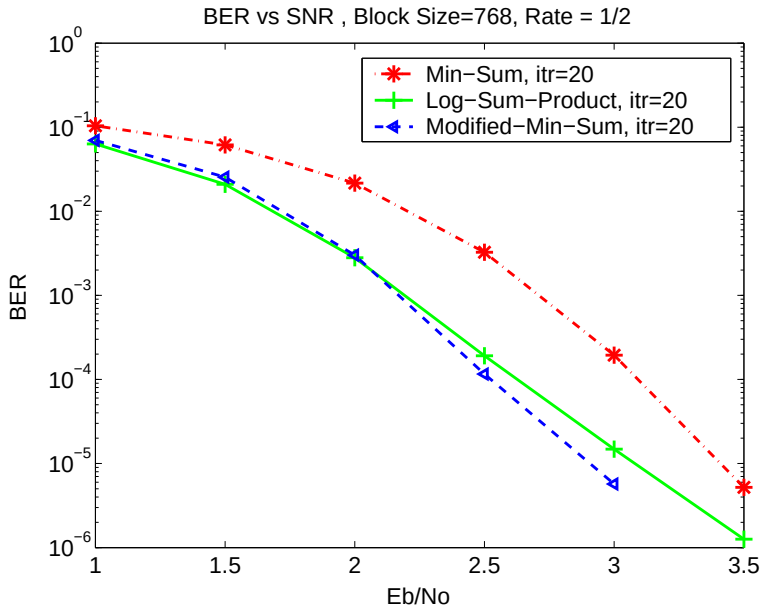

Figure 3.2 : Simulation results for the decoding performance of different algorithms.

more additions than other algorithms, it is still preferred since nonlinear function evaluations are omitted.

Figure 2.5 shows $\phi(x) = -\log(\tanh(x/2))$, the nonlinear function in the Log-Sum-Product algorithm. Because of the exponential decline for the values of $x \in [0, 1]$, $\phi(x)$ is very prone to quantization error which results in loss of the decoder performance. There exist two approaches to determine values of this function. One is direct implementation of log and tanh in hardware as [11] which is costly for hardware.

The other approach is to use look-up tables (LUT) as [12] which is very sensitive to the number of quantization bits and number of LUT values. The other issue is the amount of memory needed to store these LUTs. For example, in a $(3, 6)$ LDPC code, each Check Functional Unit (CFU) needs to do *six* LUT reads at the same time which means either using *one* LUT and spending 6 cycles evaluating the values or having 6 LUTs and spending *one* cycle. The former approach is very slow while the

Table 3.2 : Complexity comparison between decoding algorithms per iteration.

Algorithm	**Addition**	**Function Evaluation** $f(x) = -\log(\tanh(x/2))$	**Shift**
Log-Sum-Product	$24 \times (N-K) + 7 \times N$	$12 \times (N-K)$	-
Min-Sum	$24 \times (N-K) + 7 \times N$	-	-
Modified Min-Sum	$24 \times (N-K) + 10 \times N$	-	$6 \times N$

latter takes a large area. Since in the decoding process there is need to store all the messages that pass between nodes, any decrease in the amount of required memory is greatly desired. This makes Modified Min-Sum algorithm a better approach for the hardware.

3.1.4 Block Length

Figure 3.3 shows a comparison between the performances of two sets of $(3, 6)$ LDPC codes of rate $1/2$ and block lengths of 768 and 1536 designed with above structure and also with random generated parity check matrix. Increasing the block length improves the performance, but at the same time it increases the amount of the computations linearly (assuming that other parameters are fixed). From the figure, it can be seen that this structure has a minor effect on the performance of the decoder.

3.1.5 Number of the Quantization Bits

Since this decoders work on soft information, the messages that are sent between nodes are real values. In order to represent these values in fixed point arithmetic, we need to quantize them. There is some performance loss because of the quantization.

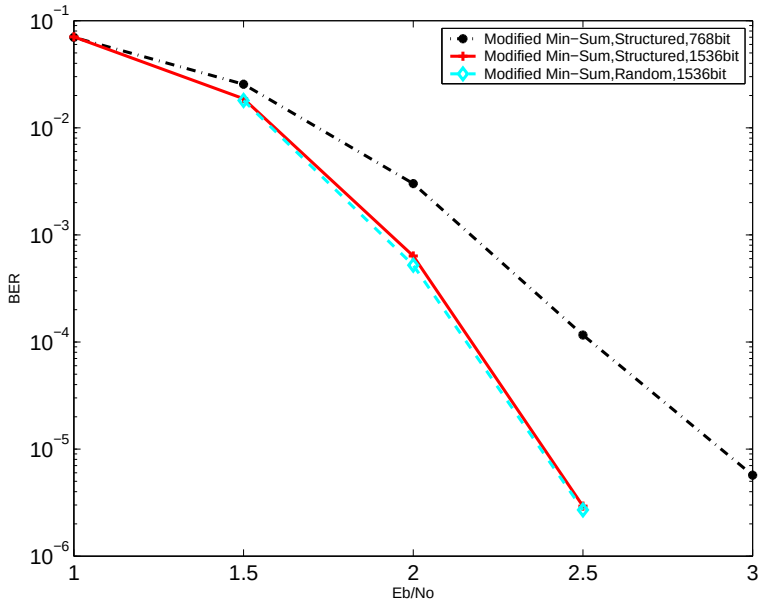

Figure 3.3 : Simulation results for the decoding performance of different block lengths.

Number of the bits used in the messages is compared in 3.4. This figure compares the performance of the Min-Sum algorithm using 4,5,6 bits for the messages for a code with the block length of 768 bits. We assume that each $i : f$ message has a sign bit plus i bits for the integer and f bits for the fractional part. So, the total number of the bits used in each message is $1 + i + f$. For example a $2 : 4$ message uses $1 + 2 + 4 = 7$ bits. Figure 3.4 shows a comparison between $1 : 4, 2 : 2, 2 : 3, 2 : 4$ bit messages. It is obvious that using two bits for the integer part is necessary and 2:2 outperforms 1:4 even with less number of bits. Also, figure shows that increasing the number of bits from 2:2 to 2:3 or 2:4 gives a small improvement to the decoding performance with 20% to 40% increase in the area. We have used the 5 bits in our design which is related to the 2:2 case.

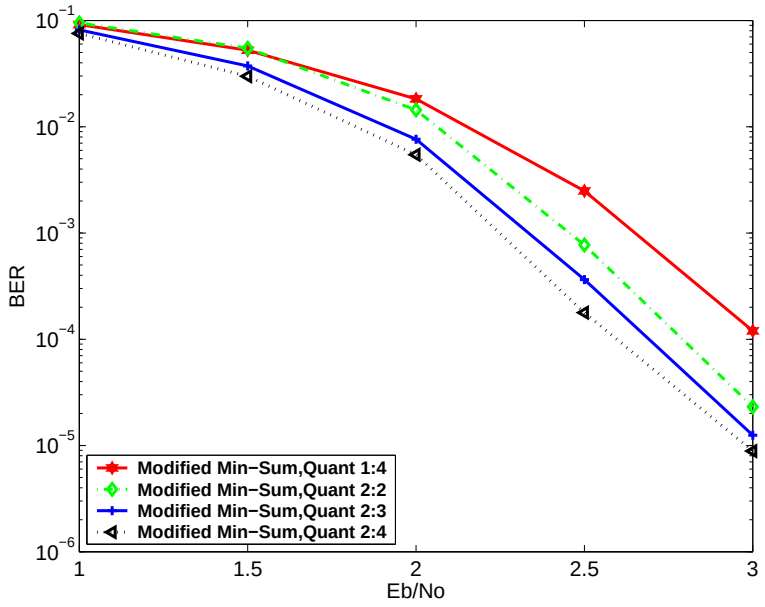

Figure 3.4 : Comparison between the performance of the LDPC decoder using different number of bits for the messages for a code with the block length of 768 bits.

3.1.6 Maximum Number of the Iterations

A comparison between three curves with different stopping criteria is shown in figure 3.5. It is obvious that increasing the number of the iterations, increases the performance. The drawback is that it takes more time to decode. Increasing the maximum number of iterations from 5 to 10, doubles the decoding time (In the worst case, since some of the iterations can be skipped of the valid codeword is found earlier).

Next section talks about the proposed architecture that has been designed using the above parameters.

3.2 Reconfigurable Architecture Design

For LDPC codes, increasing the block length results in a performance increase. That is because the Bit and Check nodes receive some extrinsic information from the nodes

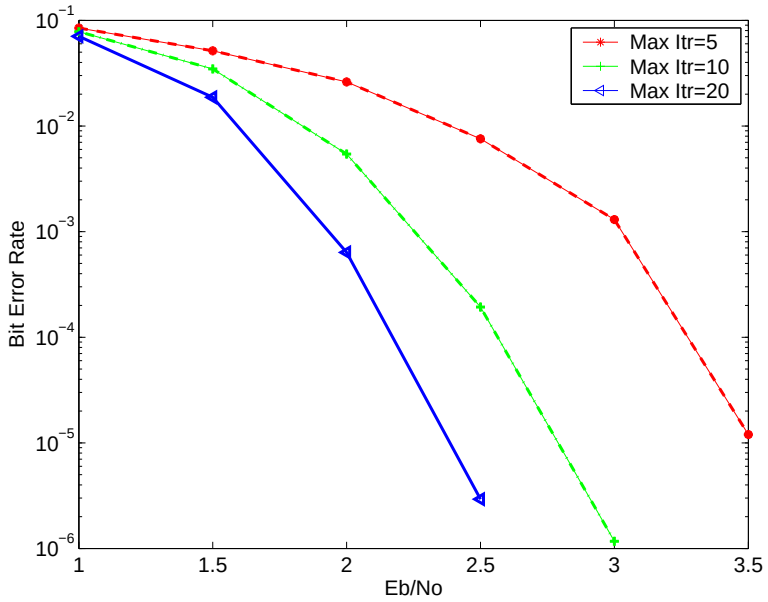

Figure 3.5 : Comparison between the performance of the LDPC decoder with different stopping criteria or a code with the block length of 1536 bits

that are very far from them in the block. This increases the error correction ability of the code. Having a scalable architecture which can be scaled for different block lengths enables us to choose a suitable block length N for different applications. Usually N is in the order of $500 \sim 5000$ for practical uses. Our design is flexible for block lengths of $N = 6 \times 2^{\theta}$ for a (3,6) LDPC code. As an example for $\theta = 8$, N is equal to 1536. By choosing different values for θ we can get different values for the block length. We will discuss the statistics and design of the architecture for block length 1536 bits. The proposed LDPC decoder can be scaled for the other lengths such as 768. It should be noted that changing the block length is an off-line process, since a new bitstream file should be compiled to download to an FPGA.

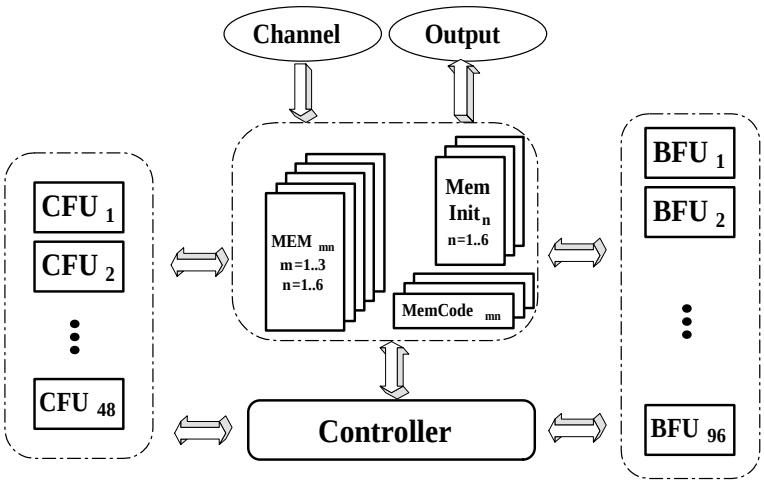

Figure 3.6 : Overall architecture of an LDPC decoder.

3.2.1 Overall Architecture for LDPC Decoder

The overall architecture for a $(3, 6)$ LDPC decoder is shown in figure 3.6. This semi-parallel architecture consists of $W_c \times W_r = 3 \times 6 = 18$ memory units ($MEM_{mn}, m = 1, ..., W_c, n = 1, ..., W_r$) to store the values passed between Bit nodes and Check nodes and Wr memories ($MemInit_n$) to store the initial values read from the channel. $MemCode_{mn}$ stores the code bits resulted from each iteration of the decoding. This architecture has several Bit Functional Units and Check Functional Units that can be reused in each iteration. Since the code rate is $1/2$, there are twice as many columns in the parity check matrix as rows, which means that number of BFUs should be two times the number of CFUs to balance the time spent on each half-iteration. For the block length of 1536, we have chosen the parallelism factor of 48 for CFUs and 96 for BFUs. Each of these units will be used $16 = 1536/96$ times in each iteration. These units will perform computations on different input sets that will be synchronized by the controller unit.

3.2.2 Control Unit

Control unit supervises the whole process of the decoding. When a block is ready at the input, the information bits are read from the FPGA I/O pins. In each clock cycle, P of these messages are read and stored in $Mem - Inits$. When the whole block is stored in the memories, the CFUs start reading from memories and processing the information. After all the CFUs update their messages, BFUs start reading from memories and updating the values in the MEM_{mn}. In the meanwhile, it writes the threshold bits in the $MEM - Code_{mn}$ (the decoded codeword). When all the values are updated, the first iteration ends and the next iteration starts. At the same time as next set of CFUs, values of $MEM - Code_{mn}$ is checked to see if all the parity check equations are satisfied or not. By the time that CFUs are done, the result of the validity check of the codeword found at the end of the previous iteration is ready. If the code is valid, then decoder starts sending out the resulting codeword and inputting the new block.

3.2.3 Check Functional Unit

Figure 3.7 shows the interconnection between memories, address generators and CFUs that are used in the first half of iterations. In each cycle $ADGC_{mn}$ generate addresses of the messages for the CFUs. Split/Merge (S/M) units pack/unpack messages to be stored/read to/from memories. To increase the parallelism factor, it is possible to pack more messages (i.e. δ) to put to a single memory location. This poses a constraint on the design of H matrix, since the shift values should all be multiples of δ. The finite state machine "control unit" supervises the flow of messages in/out of memories and functional units.

Figure 3.8 shows the Architecture for Check Functional Units (CFUs). This ar-

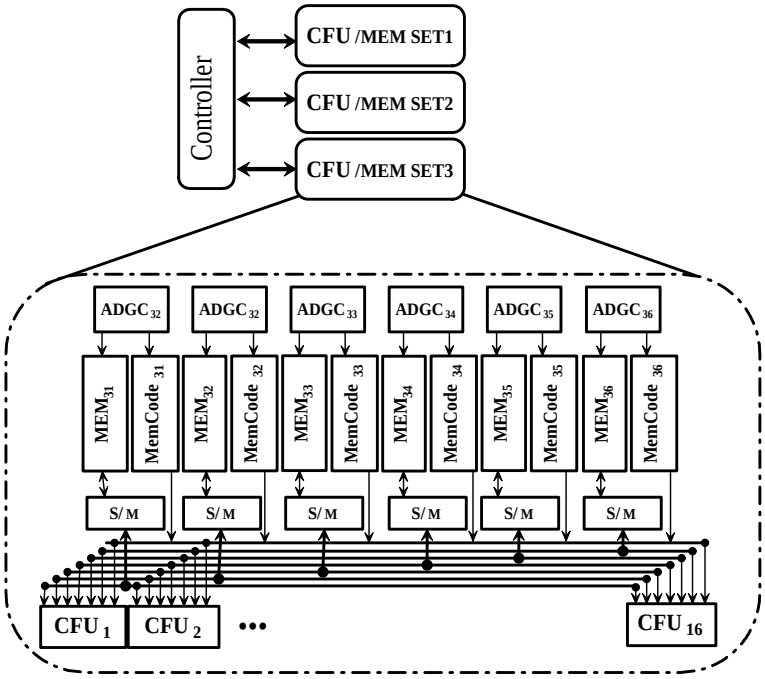

Figure 3.7 : Connections between memories, CFUs and address generators.

chitecture calculates the messages based on equation 2.45. Since we are using the Modified Min-Sum algorithm, the computations inside CFUs are less complex compared to the Sum-Product algorithm. Each CFU has $Wr = 6$ inputs and 6 outputs. This unit computes the minimum among different choices of five out of six inputs. CFU outputs the result to output ports corresponding to each input which is not included in the set. For example $out1$ is the result of:

$$out1 = \prod_{i=2}^{6} sign(in_i).\min(abs(in2), abs(in3), ..., abs(in6)) \tag{3.2}$$

in which $abs(.)$ is the absolute value function.

Also, during the computations of the current iteration, CFU checks the code bits resulting from the previous iteration to check if the code bits satisfy the corresponding parity check equation (step 5 of the decoding algorithm). After the first half of the

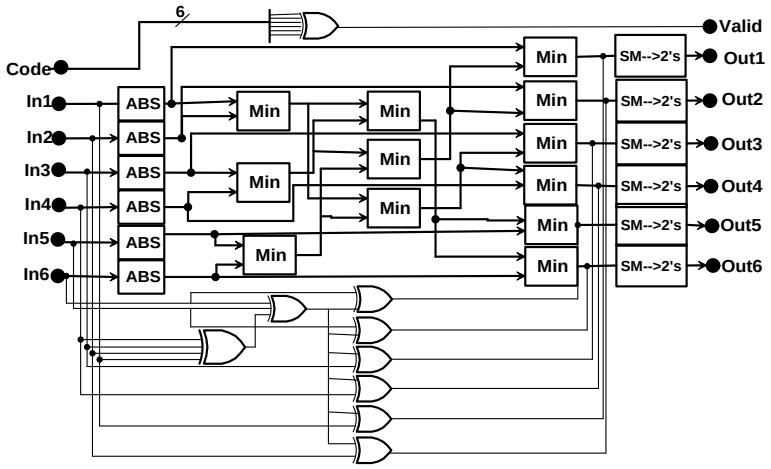

Figure 3.8 : Check Functional Unit (CFU) architecture

iteration is complete, the result of all parity checks on the codeword will be ready too. With this strategy, computations in Check nodes and Bit nodes can be done continuously without the need to wait for checking the codeword resulting from the previous iteration. This increases the speed of the decoding.

3.2.4 Bit Functional Unit

The interconnection between BFUs and memory units and address generators *ADGB* is shown in figure 3.9. Locations of the messages in the memories are such that a single address generator can service all the BFUs. Controller makes sure that all the units are synchronized.

The architecture of a Bit Functional Unit is shown in the figure 3.10. This unit computes the messages based on equation 2.46. BFU scales the messages with a scaling factor of γ. Heo [24] shows that scaling factors of $0.75 \sim 0.85$ are all good with 0.8 to be optimal. Since scaling of 0.75 can be done with two shifts and one addition, instead of multiplication, we have chosen this scaling factor for our design.

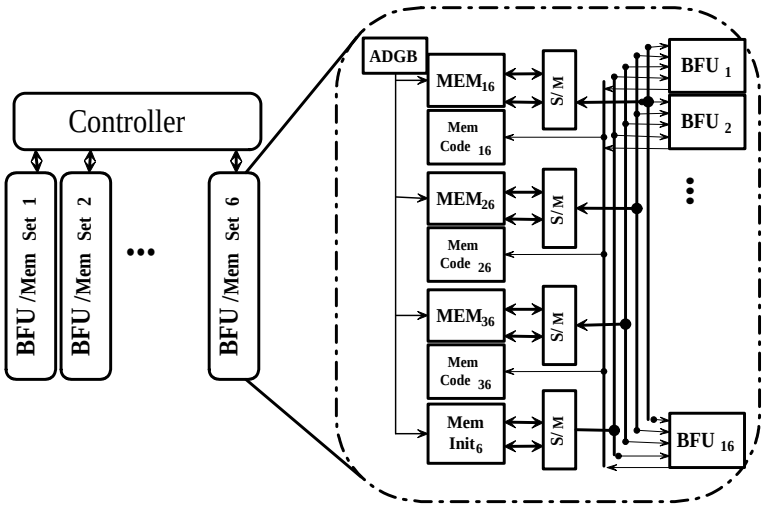

Figure 3.9 : Connections between memories, BFUs and Address generators.

This architecture can also be used for the structured irregular codes with some minor modifications. For example, assume that the parity check matrix of the irregular code is similar to figure 3.1, but it has 4 block rows and 7 block columns in which some of the blocks are full of zeros, then we can have an irregular code with row degrees of $6, 7$ and column degrees of $3, 4$. We should add some circuitry so that for the blocks full of zero in the parity check matrix, it sends a zero message to the corresponding inputs of the BFU/CFUs. In this case the BFUs will have 5 input/outputs and CFUs will have 8 input/outputs.

3.3 FPGA Architecture

For real-time hardware, fixed-point computations are less costly than floating point [29], [30]. A fixed-point decoder uses quantized values of the soft information. There is a trade-off between the number of quantization bits, area of the design, power consumption and performance. Using more bits decreases the bit error rate, but increases

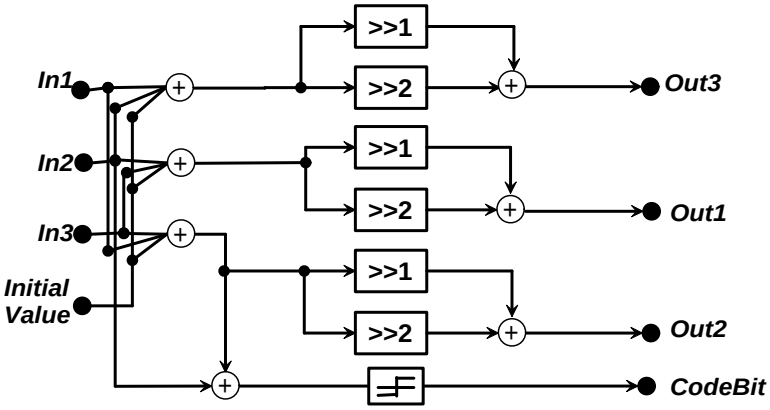

Figure 3.10 : Bit Functional Unit (BFU) architecture

the area and power consumption of the chip. Also, depending on the nature of the messages, the number of bits used for integer or fractional part of the representation is important. Our simulations show that using 5 bits for the messages is enough for good performance. These messages will be divided into one sign bit, two integer bits and two fractional bits. Figure 3.4 shows the performance of the decoder using $4, 5, 6$ bits and the floating point version.

In general, ports are the expensive parts of the memory blocks. As a result, the memory blocks in the FPGA have no more than two ports. In order to increase the number of the message read/writes in each clock cycle in the dual-port memories, we pack eight message values and store them in a single memory address. This enables us to read $2 \times 8 = 16$ messages per memory per cycle.

A prototype architecture has been implemented by writing VHDL (Hardware Description Language) code [31]and targeted to a Xilinx VirtexII-3000 FPGA. Table 3.3 shows the utilization statistics of the FPGA. Based on the Leonardo Spectrum synthesis tool report, the maximum Clock frequency of this decoder is 121 MHz. Considering the parameters of our design, it takes 96 cycles to initialize the memories

Table 3.3 : Xilinx VirtexII-3000 FPGA utilization statistics.

Resource	Used	Utilization rate
Slices	11,352	79%
4 input LUTs	20,374	71%
Bonded IOBs	100	14 %
Block RAMs	66	68 %

with the values read from the channel, 32 cycles for each CFU and BFU half-iterations, and 48 cycles to send out the resulting codeword. Assuming that the decoder does μ iterations to finish the decoding, the data rate can be calculated with the following equation:

$$Datarate = \frac{(blocklength \times decoderfrequency)}{cycles}$$

and,

$$\begin{aligned} cycles &= \frac{N}{2\lambda} + \mu(\frac{2(N-K)}{l_c} + \frac{2N}{l_b}) + \frac{N-K}{l_c} \\ &+ \frac{(N-K)}{2\lambda} = (96 + \mu \times (32+32) + 32 + 48) \end{aligned}$$

In which N is the block length, K is number of the information bits, λ is the packing ratio for the messages in the memories, l_b is number of BFUs, and l_c is the number of CFUs. With maximum number of iterations, $\mu = 20$ (worst case), the data rate can be 127 Mbps. This architecture is suitable for a family of codes with similar structure as described earlier and different block lengths, parallelism ratios and message lengths.

Table 3.4 demonstrates a comparison between some of the architectures for LDPC decode that are currently available and our design.

In order to reconfigure the decoder for other block lengths, we should note that changing the block-size of the codeword changes the sizes of the memory blocks. If

Table 3.4 : Summary for some of the available architectures for LDPC decoder

Reference No.	Block Length	Code Type	Arch. Type	Data Rate	Dec. Alg.
[12]	9216	Regular	Semi-parallel	54 Mbps	Sum-prod
[13]	305,1055	Regular	Semi-parallel	-	BCJR
[11]	1024	Irregular	Parallel	1Gbps	Sum-prod
[15]	8088	Irregular	Semi-parallel	40,188 Mbps	Sum-prod
Proposed Architecture	768,1536 $6 * 2^q$	Regular	Semi-parallel	127 Mbps	Modified Min-Sum

we assume that the codes are still $(3, 6)$ and have a parity check matrix similar to figure 3.1, then all the CFUs, BFUs and address generators can be used for the new architecture. The size of the memories changes and there will be a slight modification in the address generator units because they should address a different number of memory words. This can be done by changing the size of the counters used in the address generators. Since the counters are parametric in the VHDL code, this can be done with a new compilation of the code using these new values.

Next section talks about the design of the LDPC encoder/decoder using LabVIEW and LabVIEW FPGA. A similar architecture to the VHDL version is designed using LabVIEW.

Chapter 4

Implementation of the LDPC Encoder / Decoder in LabVIEW

4.1 Implementation in LabVIEW Host

An end-to-end communication link has been implemented using LabVIEW from National Instruments. A block diagram for this design is presented in figure 4.1. First, information bits are feed to the LDPC encoder. Then, it modulates the encoded signal using BPSK modulation. The modulated codewords are sent across additive white gaussian (AWGN) channel. The received bits enter the decoder to correct all the errors that have occurred through transmission and find the original data.

The decoder uses Sum-Product algorithm and is quite general. It can work with any class of parity check matrix and LDPC code. This decoder does the computations in different processing nodes in serial. The result is a more abstract design but it takes a fair amount of time to do the decoding. Different parameters of the decoder can be changed during the process. For example, signal to noise ratio(SNR) of the communication, maximum number of the iterations for decoding,etc. Figures 4.3, 4.4, 4.5 show block diagrams of the Virtual Instruments (VIs) of the LDPC encoder / decoder implemented in LabVIEW.

Table 4.1 shows the hierarchy of the LabVIEW VIs that are used for this implementation. It should be noted that this decoder works in fully simulation mode, which means that the whole model runs on the PC. Another approach is to use co-

Figure 4.1 : Block diagram of the implementation of end to end communication link in LabVIEW

Table 4.1 : Hierarchy of the LabVIEW Implementation-Simulation only mode .

Figure Number	Description	Parent	Children
4.3	Communication system	-	4.4
4.4	LDPC Decoder-Simulation mode	4.3	4.5
4.5	Phi Function	4.4	-

simulation in the sense that encoding takes place on the host PC and the decoding on the FPGA. This implementation is discussed in the next section.

4.2 LDPC Decoder Implementation in LabVIEW FPGA

In this section we will discuss the design parameters and strategies for the implementation of an LDPC decoder using LabVIEW FPGA. This decoder is designed for a rate 1/2, (3,6) LDPC code with a block length of 768 bits. The block diagram of the design is shown in figure 4.2. As shown in the figure, the Host computer interacts with the channel and the FPGA. Only the LDPC decoder runs on the FPGA.

Figure 4.6 shows the Host version of the LDPC decoder which runs on the PC and controls the inputs/ outputs to the decoder that runs on the FPGA (figure 4.7).

Here is a description of the decoder that runs on the FPGA. During the initializa-

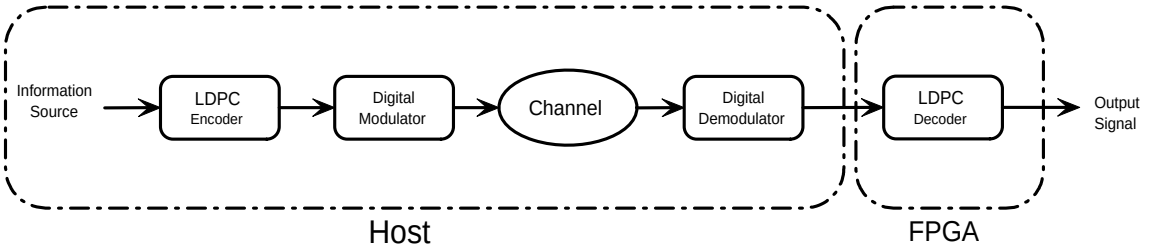

Figure 4.2 : Block diagram of the implementation of end to end communication link in LabVIEW

tion step, it reads the soft information from channel and stores them in the memories MEM_{ij}, $i \in \{1, ..., W_c\}$ and $j \in \{1, ..., W_r\}$. Also, it keeps a copy of the initial input values in the memories $MEMI_j$, $j \in \{1, ..., W_r\}$. In the next step, each CFU reads the values from memories and computes the messages to be passed to BFUs using Modified Min-Sum algorithm and stores them back in the memories. Then, BFUs read the values from memories and compute the messages to pass to the Check nodes. They also threshold the resulting values to find a codeword. Next step is to check if the resulting codeword is valid or not. To increase the throughput, this step is combined with the CFU calculations to avoid computing a new set of address and redundant computations. In this step it checks to see if all the parity check equations satisfy or not. If the codeword is valid, then it stops decoding of this block and starts the next block. Otherwise, it continues the iterations until it reaches the maximum number of the iterations.

Since the smallest integer in LabVIEW is 8 bit, the decoder uses 8 bit values for the messages. If we could change the values to 5 bits, we could save some area without any major performance loss. Table 4.2 shows the resource utilization statistics of the designed decoder using LabVIEW FPGA. We have compiled the design for the Xilinx

Table 4.2 : Device utilization statistics for the architecture designed in LabVIEW FPGA using Xilinx VirtexII-3000 FPGA

Resource	Used	Utilization rate
Slices	8443	58%
MULT18X18s	2	2%
External IOBs	93	19%
Block RAMs	24	25 %

VirtexII-3000 FPGA which is on the PXI-7833 board from National Instruments. The decoder is able to run on the 3M gate FPGA board and the Host computer controls it. This decoder is designed for the block length of 768 bits. For larger block lengths, we basically need to change the amount of memory used for the design.

The graphical view of the LabVIEW FPGA implementation of the LDPC decoder is shown in the following figures. For an detailed description of LabVIEW features, reader should refer to LabVIEW user manual [32].

Table 4.3 shows a hierarchy of the LabVIEW FPGA implementation, which describes the relation between different figures that follow.

Table 4.3 : Hierarchy of the LabVIEW Implementation Co-simulation mode.

Figure Number	Description	Parent	Children
4.6	LDPC decoder co-simulation (Host)	-	4.7
4.7	LDPC decoder co-simulation (FPGA)	4.6	4.8,4.9, 4.12,4.14
4.8	Initializing the memories	4.7	-
4.9	Connection of the CFU units and memories	4.7	4.9
4.10	Four CFUs connected to split/ merge units	4.9	4.11
4.11	Check functional unit implementation	4.10	-
4.12	Connections between BFUs and memories	4.7	4.13
4.13	Bit Functional Unit calculations	4.12	-
4.14	Sending out the decoded information bits	4.7	-

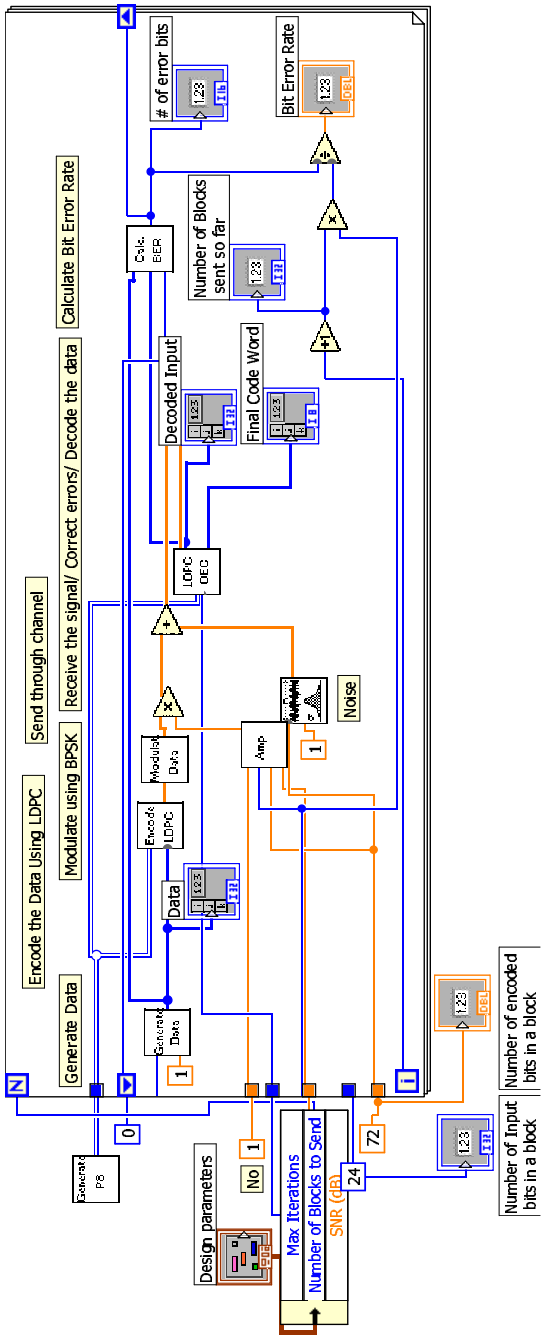

Figure 4.3 : Implementation of end to end communication link in LabVIEW

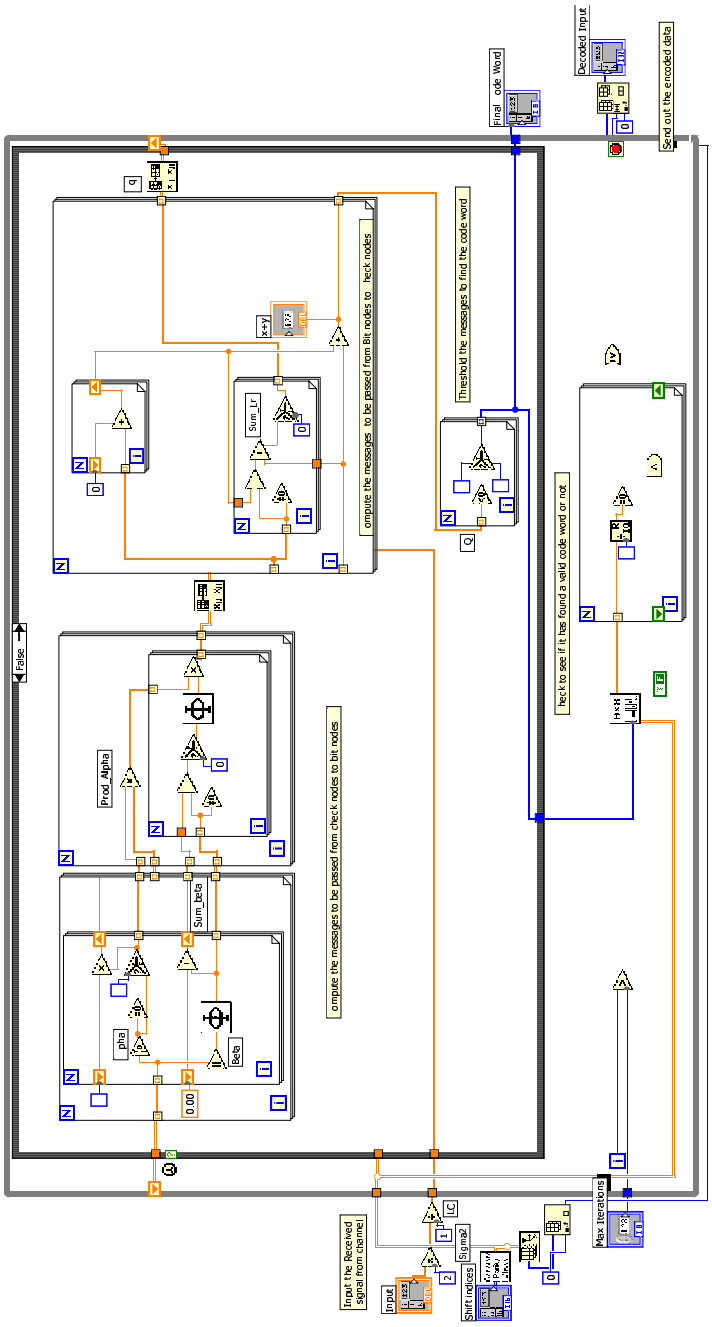

Figure 4.4 : Implementation of the LDPC decoder in LabVIEW

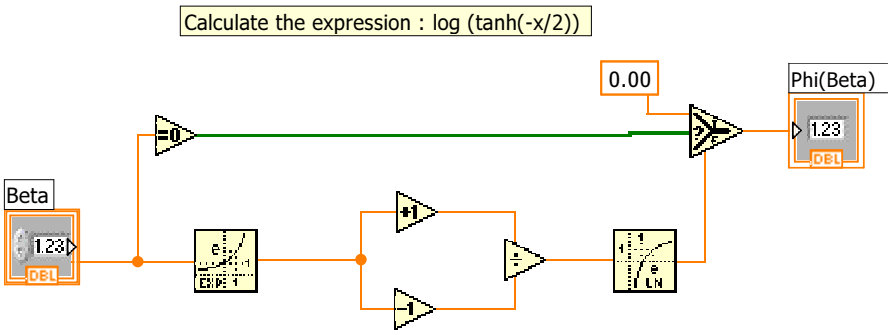

Figure 4.5 : Implementation of $\phi(x) = -log(tanh(x/2))$ in LabVIEW

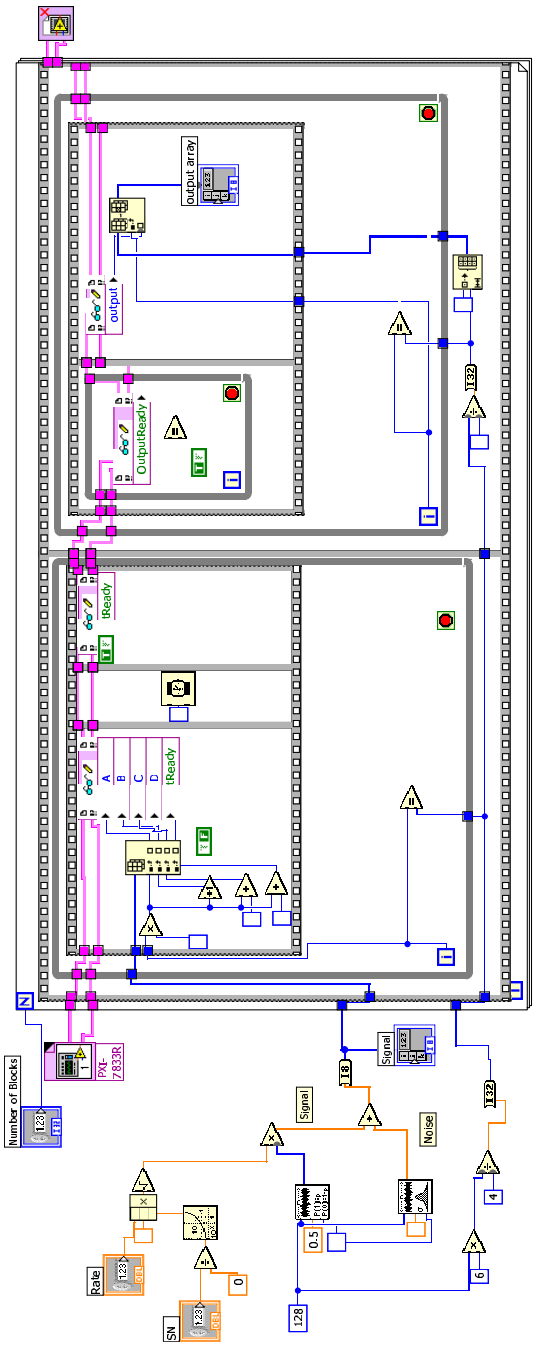

Figure 4.6 : The Host version of the LDPC decoder

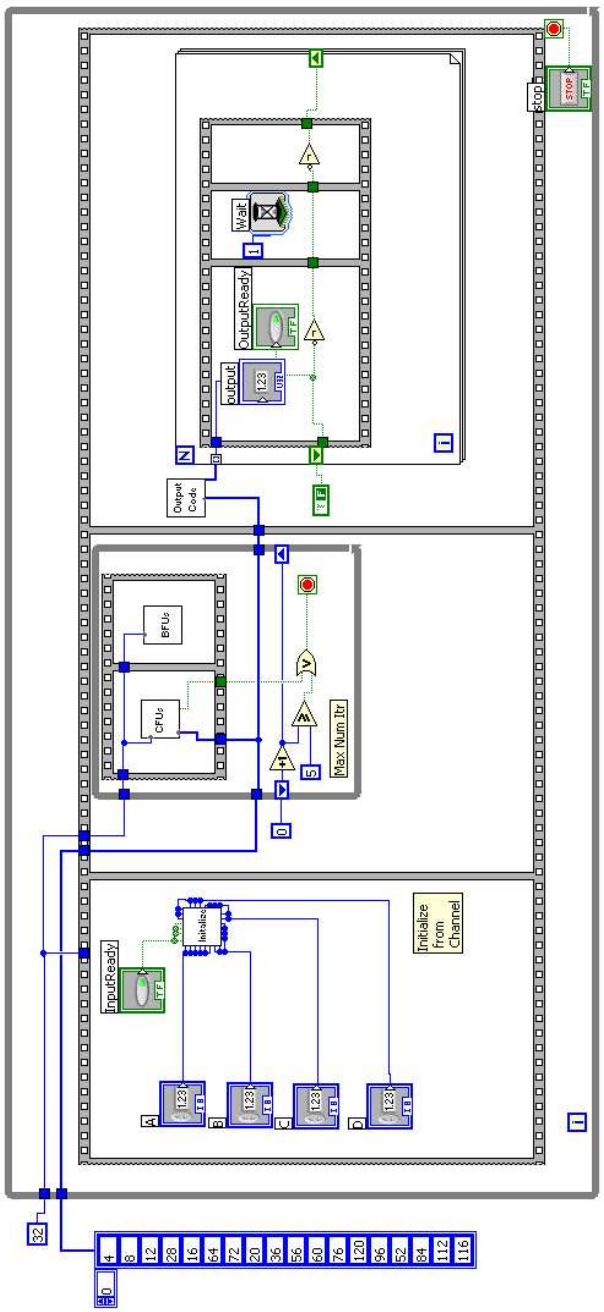

Figure 4.7 : Implementation of LDPC decoder in LabVIEW FPGA

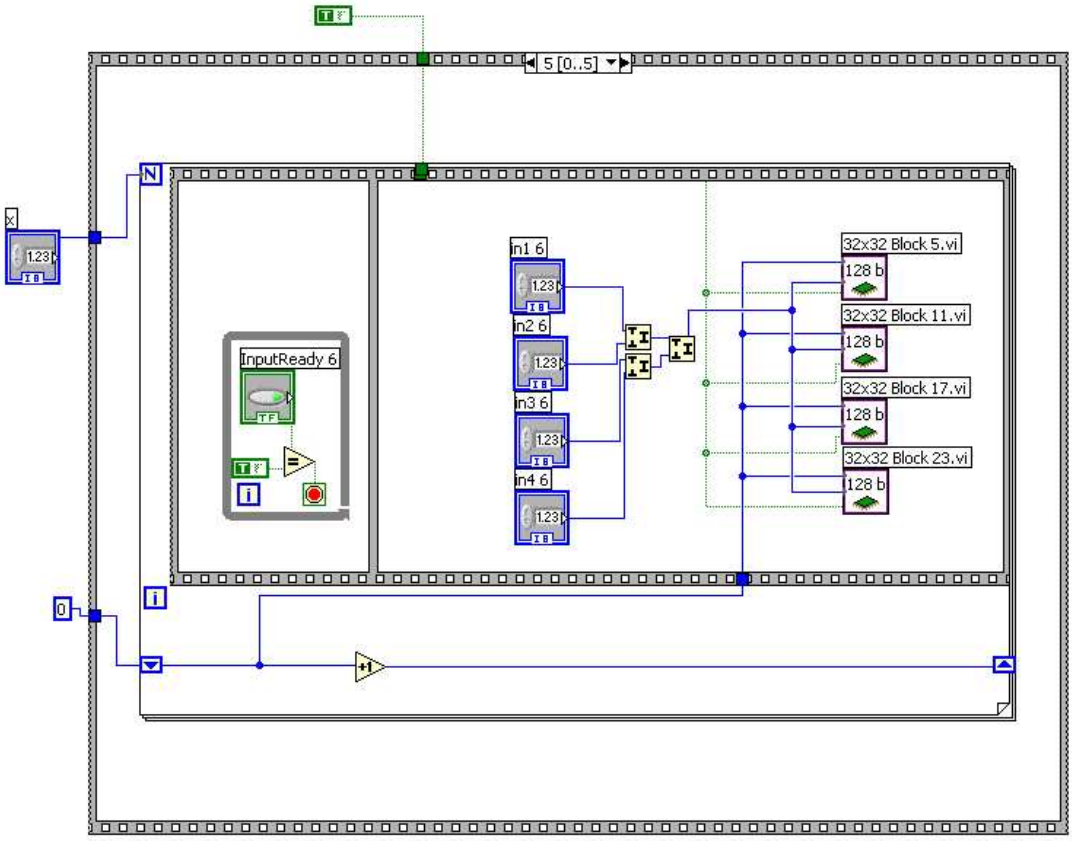

Figure 4.8 : Initializing the memories by reading from channel

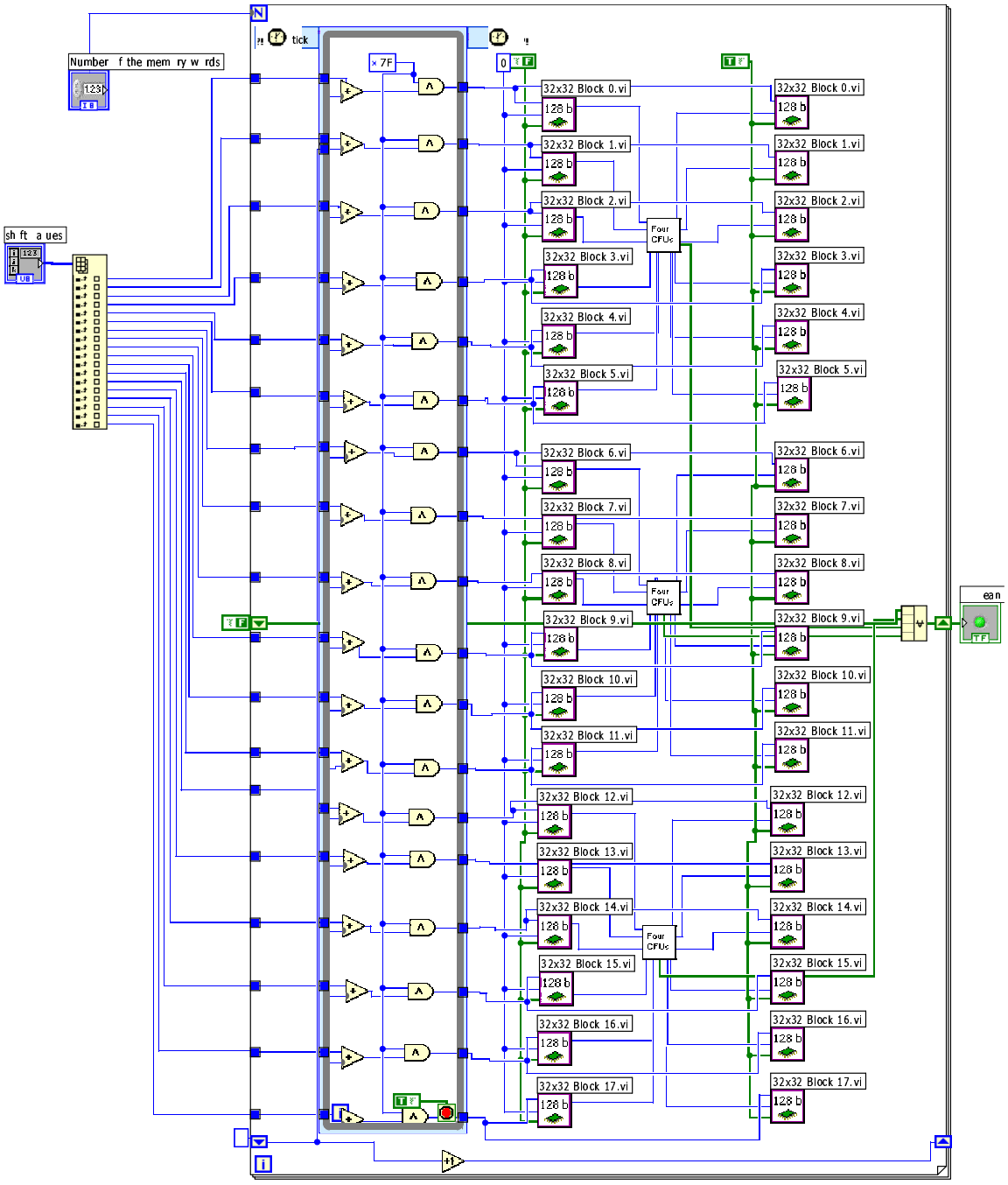

Figure 4.9 : Connection of the CFU units and memories

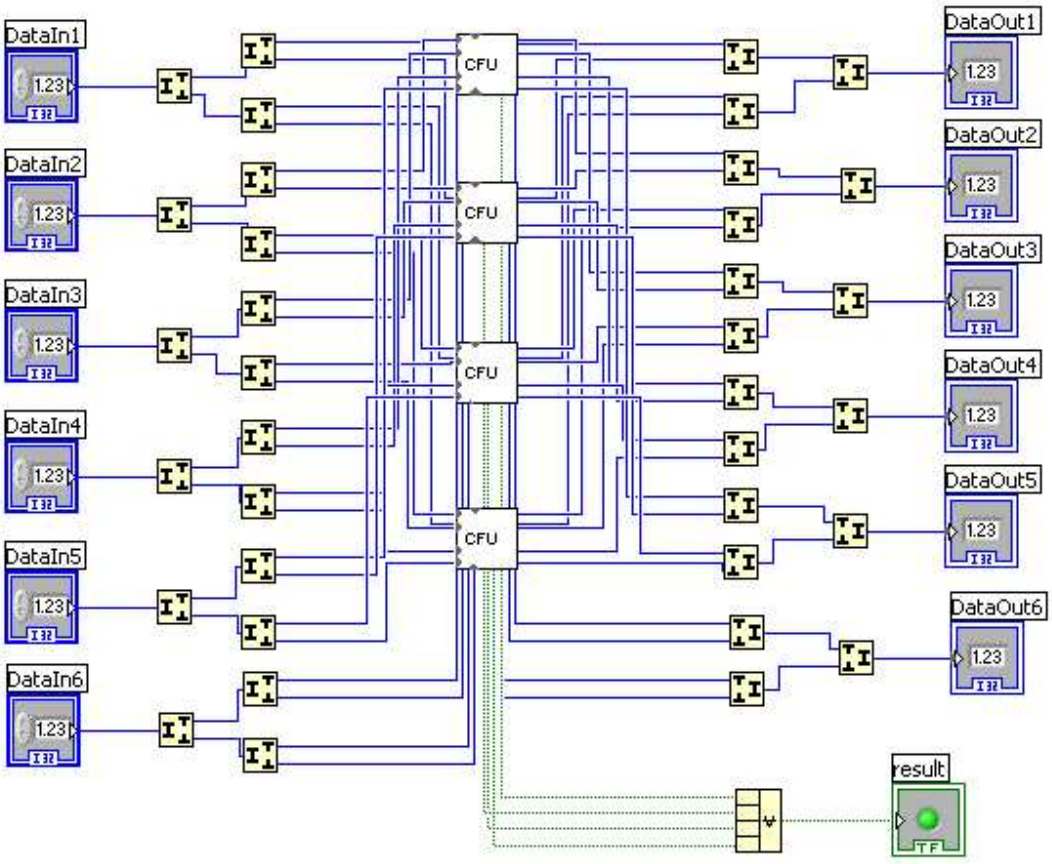

Figure 4.10 : Four CFUs connected to split/ merge units

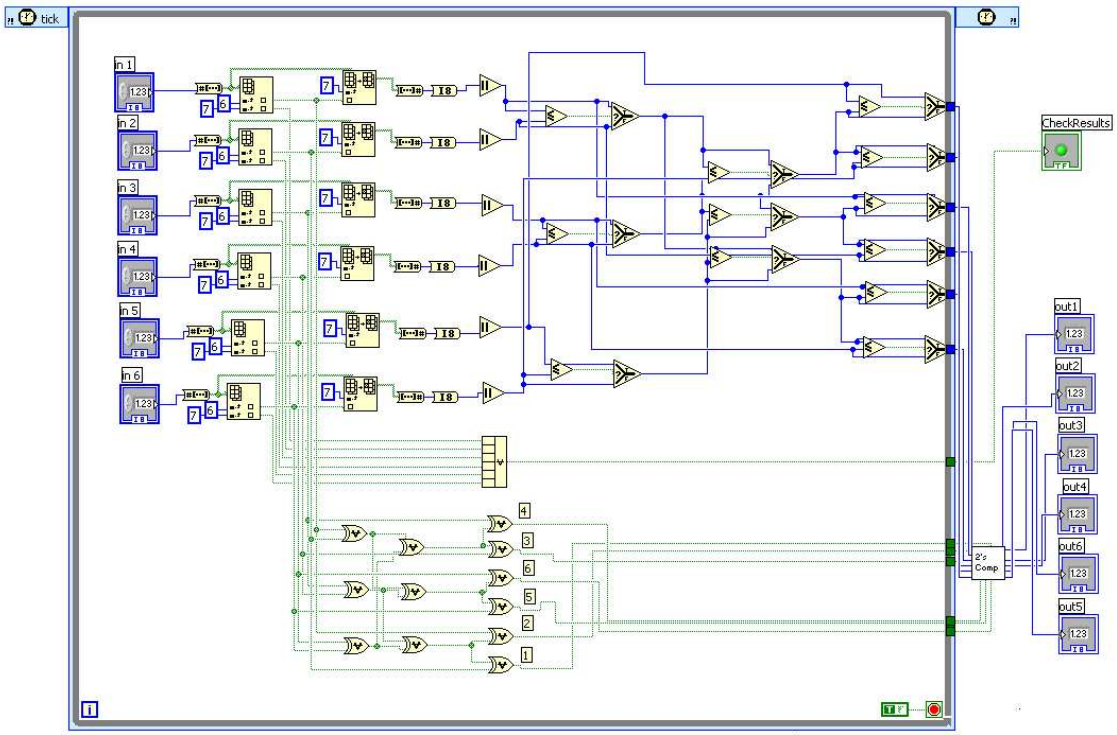

Figure 4.11 : Check functional unit implementation

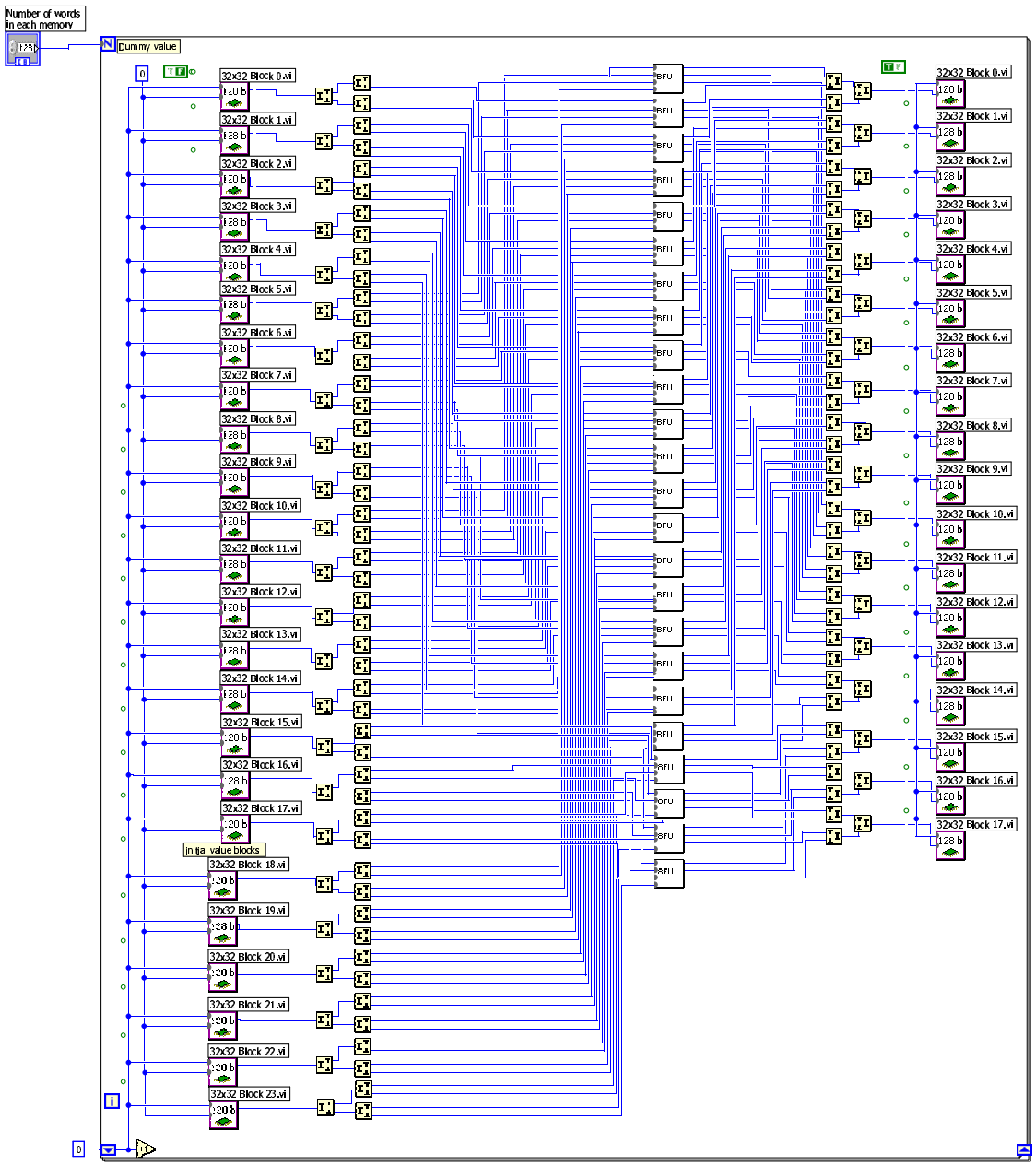

Figure 4.12 : Connections between BFUs and memories

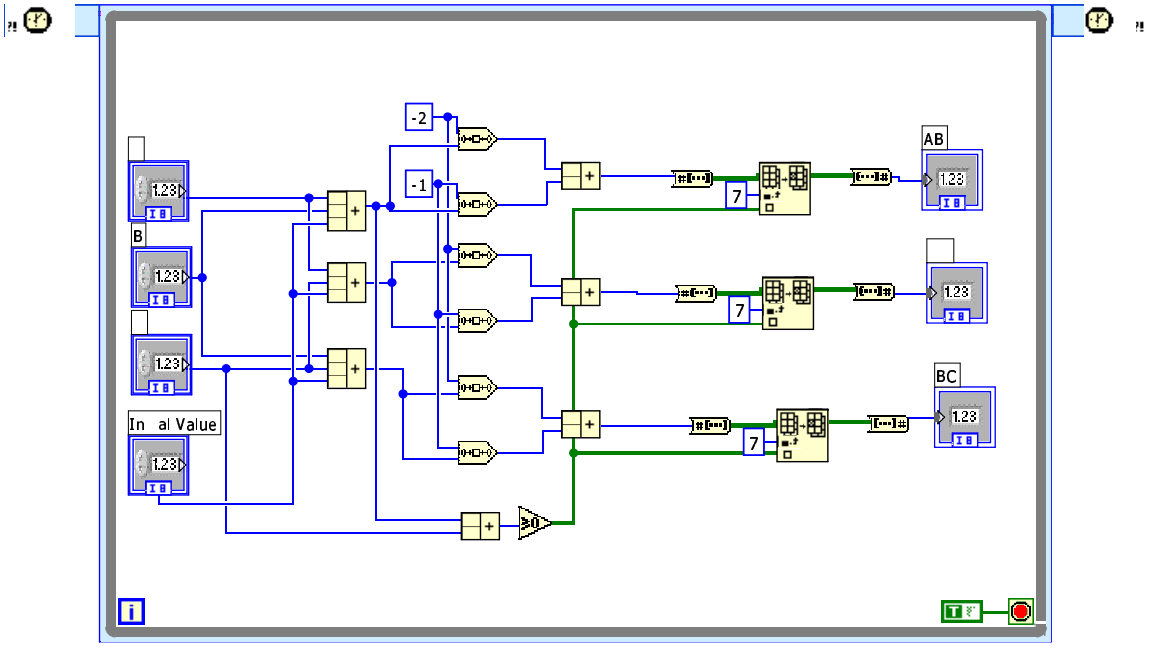

Figure 4.13 : Bit Functional Unit calculations

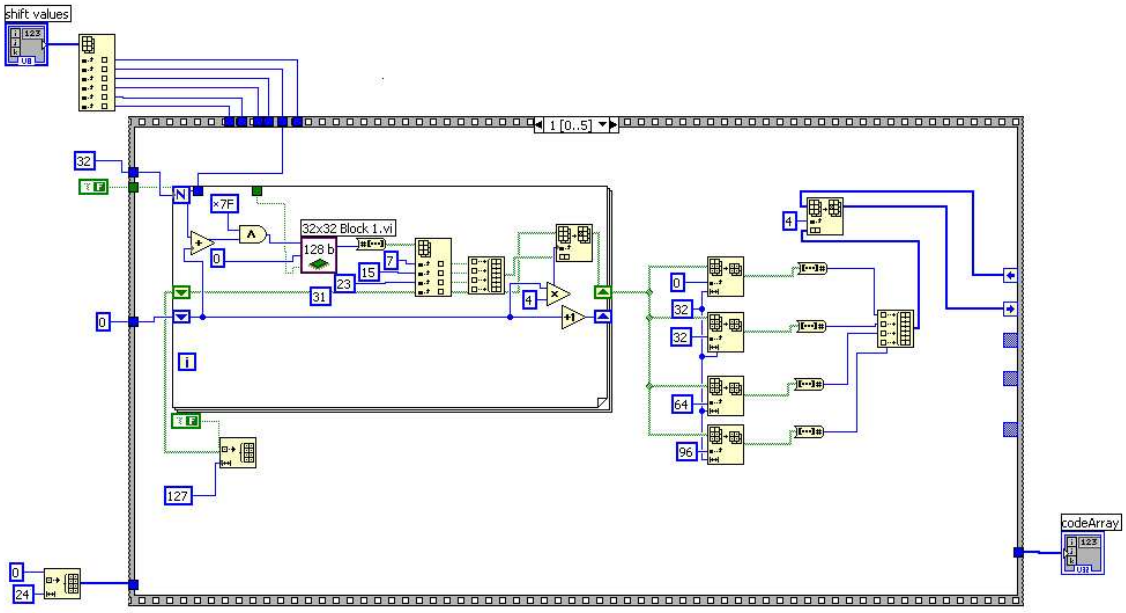

Figure 4.14 : Sending out the decoded information bits.

Chapter 5

Conclusions and Future Work

5.1 Conclusions

Throughout this thesis document, we discussed the design issues and parameters of designing a semi-parallel decoder for LDPC codes. We showed how structure of the parity check matrix affects the performance of LDPC decoding. Also, the relation between the type of the decoder, block length, code rate, maximum number of the decoding iterations, number of bits used for the messages passed between nodes, etc. have effect on the achievable bit error rates in different signal to noise ratios.

In this research, a decoder for LDPC coding has been designed and implemented using VHDL for Xilinx VirtexII FPGA. Modified Min-Sum algorithm has been used as the decoding algorithm, since it is very suitable for hardware design and has very good performance. The structure of the design and semi-parallel architecture balances the trade-off between area and speed for decoding. For a block length of 1536 bits, it achieves a data rate of 127 Mbps. Also, this reconfigurable architecture can be easily scaled to larger block lengths.

5.2 Future Work

In order to be able to use this decoder in realistic applications, we need to design an encoder for the LDPC codes. This enables us to integrate the encoder/decoder pair to the testbed for wireless communication which has been designed in center for

multimedia communications (CMC) at Rice university.

The designed decoder is suitable for $(3, 6)$ regular codes, the next step will be some modifications to support irregular codes too. Suitably designed irregular LDPC codes have shown better performance than regular codes.

Since we have implemented and verified the decoder on FPGA, the next step will be migration to ASIC (application specific integrated circuit) and fabricate a chip for the decoder.

Appendix A

Appendix : Some Notes on Software Settings

A.1 ModelSim

In this appendix we will explain some of the notes that should be considered when setting up the software.

1. After installing ModelSim, it is necessary to compile the libraries necessary for analyzing VHDL. These are the commands that should be used:

-Run "compxlib -s mti_se -f all -l all -o $c : \backslash modeltech\_5.7a \backslash xilinx\_libs$" from the Windows command line.

-Run the following in ModelSim:

- vmap simprims $C : \backslash Modeltech\_5.7a \backslash xilinx\_libs \backslash simprim$
- vmap unisims $C : \backslash Modeltech\_5.7a \backslash xilinx\_libs \backslash unisim$
- vmap XilinxCoreLib $C : \backslash Modeltech\_5.7a \backslash xilinx\_libs \backslash XilinxCoreLib$
- vmap simprims_ver $C : \backslash Modeltech\_5.7a \backslash xilinx\_libs \backslash simprim\_ver$
- vmap unisims_ver $C : \backslash Modeltech\_5.7a \backslash xilinx\_libs \backslash unisim\_ver$
- vmap XilinxCoreLib_ver $C : \backslash Modeltech\_5.7a \backslash xilinx\_libs \backslash XilinxCoreLib\_ver$

2. In order to be able to simulate post place and route design, ModelSim needs to know the link to Simprim library. The following commands need to be executed on the command line to compile each of the SIMPRIM source files. The SIMPRIM VHDL libraries are written using VITAL libraries, the packages defined by the standard are

fully accelerated by ModelSim. These commands can be saved in a do to run as a script.

- cd < $directorystoragelocation$ >
- vlib simprim
- vmap simprim simprim
- vcom -work simprim {$XILINX\vhdl\src\simprims\simprim\_Vcomponents.vhd$}
- vcom -work simprim {$XILINX\vhdl\src\simprims\simprim\_Vpackage.vhd$}
- vcom -work simprim {$XILINX\vhdl\src\simprims\simprim\_VITAL.vhd$}

If the SIMPRIM library exists already compiled in a shared area then only the following line needs to be executed, the vmap command maps the logical library to the physical directory.

vmap simprim $F : \some\_directory\vendors\xilinx\simprim$

3. Performing Behavioral Simulation [33]: Before Model Technologys simulation tools can be used to simulate the design, the parent design and the testbench need to be analyzed. These design files are analyzed with the vcom command, into a local, default, work library, created using the vlib command.

Analyze the parent design and testbench file. Select the MTI ModelSim, and go to the project-directory and type the following:

- vlib work
- vcom mycore.vhd
- vcom testbench.vhd

Invoke the simulator. The simulator may now be invoked by typing in the following command: vsim cfg_testbench The cfg_testbench needs to correspond to the name

of the VHDL configuration declared in the testbench. This loads the testbench, the parent design, and the simulation model of the mycore core, stored in the location referenced by xilinxcorelib.

Bibliography

[1] J. G. Proakis. *Digital Communications*, volume 3. McGRAW Hill, 1995.

[2] A. Glavieux C. Berrou and P. Thitimajshima. Near Shannon Limit Error-Correcting Coding and Decoding: Turbo-Codes. *Proceedings of International Communications Conference (ICC'93)*, pages 1064–1070, 1993.

[3] A. Viterbi. Orthogonal tree codes for communication in the presence of white gaussian noise. *IEEE Transactions on Communications*, 15:238 –242, Apr 1967.

[4] Joy A. Thomas Thomas M. Cover. *Elements of Information theory*. John Wiley and Sons, 1991.

[5] S.B. Wicker. *Error Control Systems for Digital Communication and Storage*. Prentice Hall, New Jersey, 1995.

[6] R.G. Gallager. Low-Density Parity-Check Codes. *IRE Transactions on Information Theory*, 8:21–28, Jan 1962.

[7] D.J.C. MacKay and R.M. Neal. Near Shannon Limit Performace of Low Density Parity Check codes. In *Electronics Letters*, volume 32, pages 1645–1646, Aug 1996.

[8] D.J.C. MacKay. Good Error-Correcting Codes Based on Very Sparse Matrices. *IEEE Transaction on Information Theory*, 45(2):399–431, Mar 1999.

[9] S. Chung, Jr. G. D. Forney, T. Richardson, and R. Urbanke. On the design of low-density parity-check codes within 0.0045 db of the shannon limit. *IEEE Communications Letters*, 5:58–60, February 2001.

[10] L.N. Lee. LDPC Code, Application to the Next Generation Wireless Communication Systems, 2003. Fall VTC, Panel Presentation by Hughes Network.

[11] A.J. Blanksby and C.J. Howland. A 690-mW 1-Gbps 1024-b, Rate-1/2 Low-Density Parity-Check Code Decoder . *Journal of Solid State Circuits*, 37(3):404–412, Mar 2002.

[12] T. Zhang. *Efficient VLSI Architectures for Error-Correcting Coding*. PhD thesis, University of Minnesota, Minneapolis, Jul 2002.

[13] M.M. Mansour and N.Shanbhag. Low Power VLSI Decoder Architectures for LDPC Codes. *Proceedings of the International Symposium on Low Power Electronics and Design. ISPLED '02*, pages 284–289, 2002.

[14] J. Castura E. Boutillon and F.R. Kschischang. Decoder First Code Design. In *Proceedings of the 2nd International Symposium on Turbo codes and Related Topics*, pages 459–462, Brest,France, Sept 2000.

[15] Y. Chen and D. Hocevar. A FPGA and ASIC Implementation of Rate 1/2 8088-b Irregular Low Density Parity Check Decoder. *GLOBECOM*, 2003.

[16] G. Sobelman S. Kim and J. Moon. Parallel VLSI Architectures for a Class of LDPC Codes. *IEEE International Symposium on Circuits and Systems*, 2:II–93 – II–96, May 2002.

[17] R. Echard and S. Chang. The p-rotation low-density parity check codes. *IEEE Global Telecommunications Conference, GLOBECOM '01*, 2:980 – 984, Nov 2001.

[18] W. Ryan. An Introduction to Low-Density Parity-Check Codes. http://www.ece.arizona.edu/ ryan/NewApr 2001.

[19] R.M. Tanner. A recursive approach to low complexity codes. *IEEE Transactions on Information Theory*, 27(5):533–547, Sep 1981.

[20] B.J. Frey F.R. Kschischang and H.A. Loeliger. Factor graphs and the sum-product algorithm. *IEEE Transactions on Information Theory*, 47(2):498–419, Feb 2001.

[21] A. Shokrollahi T.J. Richardson and R. Urbanke. Design of Capacity-Approaching Irregular Low-Density Parity-Check Codes. *IEEE Transactions on Information Theory*, 47(2):619–637, Feb 2001.

[22] T. Richardson R. Urbanke. Efficient Encoding of Low-Density Parity Check Codes. *IEEE Trans. on Information Theory*, 47(2):638–656, Feb 2001.

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

[24] J. Heo. Analysis of Scaling Soft Information on Low Density Parity Check Codes. *Electronics Letters*, 39(2):219 –221, Jan 2003.

[25] S.Y. Chung, T.J. Richardson, and R.L. Urbanke. Analysis of Sum-Product Decoding of Low-Density Parity-Check Codes Using a Gaussian Approximation. *IEEE Transactions on Information Theory*, 47(2):657–670, Feb 2001.

[26] J. Rosenthal and P.O. Vontobel. Constructions of regular and irregular LDPC codes using Ramanujan graphs and ideas from Margulis. *Proc. of 38th Allerton Conference on Communication, Control and Computing*, pages 248–257, Oct. 2000.

[27] G.A.Margulis. Explicit constructions of garaphs without short cycles ans low density codes. *Combinatorica*, 2:71–78, 1982.

[28] Y. Mao and A.H. Banihashemi. A Heuristic Search for Good Low-Density Parity-Check Codes at Short Block Lengths. *IEEE International Conference on Communications, ICC*, 1(11-14):41 –44, Jun 2001.

[29] David A. Patterson John L. Hennessy. *Computer architecture, a quantitative approach.* Morgan Kaufmann publishers, third edition, 2003.

[30] John F. Wakerly. *Digital design principles and practices.* Prentice-Hall, 1990.

[31] Peter J. Ashenden. *The designer's guide to VHDL.* Morgan Kaufmann publishers, second edition, 1996.

[32] National Instruments. *LabVIEW Basics I,II Course Manual*, Dec 2001. Ver 6.1, www.ni.com.

[33] Xilinx. *CORE Generator Guide-ISE 5.* www.xilinx.com.