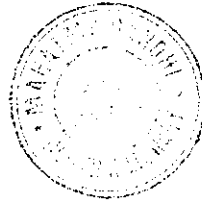


P-3544



DESIGN AND SIMULATION OF LDPC DECODER USING MODELSIM

By

V.NAVEEN

Reg No: 0710107064

S.D.SARAVANAPRIYAN

Reg No: 0710107089

R.C.SATHEESHKUMAR

Reg No: 0710107091

S.SATHISHKUMAR

Reg No: 0710107093

of

**KUMARAGURU COLLEGE OF TECHNOLOGY,
COIMBATORE - 641 049.**

(An Autonomous Institution affiliated to Anna University of Technology, Coimbatore)

A PROJECT REPORT

Submitted to the

**FACULTY OF ELECTRONICS AND COMMUNICATION
ENGINEERING**

In partial fulfillment of the requirements
for the award of the degree

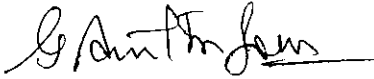
of

**BACHELOR OF ENGINEERING
in
ELECTRONICS AND COMMUNICATION ENGINEERING**

APRIL 2011

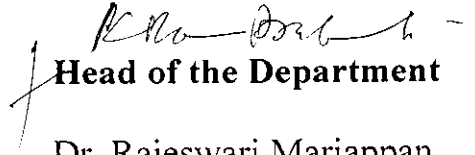
BONAFIDE CERTIFICATE

Certified that this project report entitled “**Design And Simulation of LDPC Decoder Using Modelsim**” is the bonafide work of **Mr. V.Naveen** [Reg. no. 0710107064], **Mr.S.D.SaravanaPriyan** [Reg.no.0710107089], **R.C.Satheeshkumar** [Reg. no. 0710107091] and **Mr.S.Sathishkumar** [Reg. no. 0710107093] who carried out the research under my supervision. Certified further, that to the best of my knowledge the work reported herein does not form part of any other project or dissertation on the basis of which a degree or award was conferred on an earlier occasion on this or any other candidate.



Project Guide

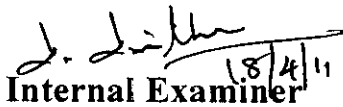
G.Amirtha Gowri



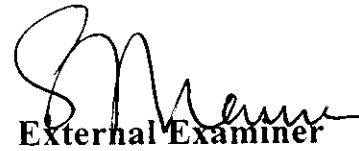
Head of the Department

Dr. Rajeswari Mariappan

The candidates with University Register No. 0710107064, 0710107089, 0710107091, 0710107093 are examined by us in the project viva-voce examination held on 18.4.2011.



Internal Examiner



External Examiner

ACKNOWLEDGEMENT

First I would like to express my praise and gratitude to the Lord, who has showered his grace and blessing enabling me to complete this project in an excellent manner.

I express my sincere thanks to our beloved Director **Dr.J.Shanmugam**, Kumaraguru College of Technology, for his kind support and for providing necessary facilities to carry out the work.

I express my sincere thanks to our beloved Principal **Dr.S.Ramachandran**, Kumaraguru College of Technology, who provided the necessary facilities for doing the project.

I would like to express my deep sense of gratitude to our ever active HOD, **Dr.Rajeswari Mariappan**, Department of Electronics and Communication Engineering, for her valuable suggestions and encouragement which paved way for the successful completion of the project work.

In particular, I wish to thank with everlasting gratitude to the project coordinator **Ms.K.Kavitha**, Assistant Professor-SRG , Department of Electronics and Communication Engineering, for her expert counseling and guidance to make this project to a great deal of success.

I am greatly privileged to express my heartfelt thanks to my project guide **Ms.G.Amirtha Gowri**, Associate Professor, Department of Electronics and Communication Engineering, without whom this project could not have been possible for her support and guidance throughout the course of this project work. I wish to convey my deep sense of gratitude to all the teaching and non-teaching staff of ECE Department for their help and cooperation.

Finally, I thank my parents and my family members for giving me the moral support and abundant blessings in all of my activities and my dear friends who helped me to endure my difficult times with their unfailing support and warm wishes.

ABSTRACT

An error correction code that efficiently utilizes the hardware and the bandwidth for high data rate communication is essential for long distance very high speed wireless systems. The LDPC codes are a forward error correction codes with performance limit well nearer to the Shannon limit. This project presents a design for the implementation of Low-Density Parity-Check (LDPC) decoders. A decoding architecture for the structured LDPC codes has been employed. Unlike many other classes of codes, LDPC codes are equipped with very fast (probabilistic) encoding and decoding algorithms. The decoding algorithms for LDPC code are available in such a manner that parallelism could be achieved with the currently available FPGAs. This advantage of LDPC codes in the face of large amounts of noise makes LDPC codes not only attractive from a theoretical point of view, but also perfect for practical applications.

The Low Density Parity Check Decoder is designed and simulated to decode the code words transmitted by the transmitter through the noisy channel. The received code words will be real valued variables. The real values are not synthesizable in FPGAs. So, the algorithms and methods to implement the real values in FPGA was developed and simulated. The appropriate methods to implement exponential, log and tanh functions needed in decoding algorithms are also developed. The hardware architecture for LDPC decoder is designed using Verilog HDL. It was functionally simulated using Modelsim 6.3f.

TABLE OF CONTENTS

CHAPTER NO.	TITLE	PAGE NO.
	ABSTRACT	
	LIST OF FIGURES	
1.	INTRODUCTION	
1.1	INTRODUCTION TO COMMUNICATION SYSTEMS	1
1.2	OBJECTIVE OF THE PROJECT	1
1.3	OVERVIEW OF THE PROJECT	2
2.	INTRODUCTION TO LDPC CODES	
2.1	INTRODUCTION	3
2.2	REGULAR AND IRREGULAR LDPC CODES	3
2.3	REPRESENTATION OF LDPC CODES	3
2.3.1	MATRIX REPRESENTATION	3
2.3.2	GRAPHICAL REPRESENTATION	4
2.4	IMPORTANT DESIGN PARAMETERS	6
2.4.1	CODE SIZE	6
2.4.2	CODE WEIGHTS AND RATE	7
2.4.3	CODE STRUCTURE	7
2.4.4	NUMBER OF ITERATIONS	8
2.5	CONSTRUCTION OF LDPC CODES	8
2.5.1	GALLAGER CODES	8
2.5.2	MACKAY CODES	9
2.5.3	REPEAT ACCUMULATE CODES	10
3.	ENCODING AND DECODING	
3.1	ENCODING	11
3.2	DECODING	12

3.2.1	BIT-FLIPPING DECODING	12
3.2.2	SUM-PRODUCT DECODING	14
4.	HARDWARE IMPLEMENTATION	
4.1	INTRODUCTION	17
4.2	ALGORITHMS FOR HARDWARE IMPLEMENTATION	17
4.2.1	IMPLEMENTATION OF REAL VALUES	17
4.2.1.1	ADDITION	18
4.2.1.2	MULTIPLICATION	18
4.2.1.3	DIVISION	19
4.2.2	IMPLEMENTATION OF LOG, TANH AND EXPONENTIAL FUNCTIONS	19
5.	SIMULATION RESULTS AND DISCUSSIONS	
5.1	SIMULATION RESULTS	22
5.2	SYNTHESIS REPORT	26
5.2.1	TANH MODULE	26
5.2.2	LOG MODULE	28
5.3	DISCUSSIONS	30
	REFERENCES	31

FIGURE	LIST OF FIGURES	PAGE NO.
1.1	Block Diagram of a Communication System	1
2.1	Graphical representation of LDPC decoder	4
2.2	Tanner graph for example code	6
2.3	Encoders for the repeat-accumulate (RA)	10
5.1	Simulated output for the code word [0000111100]	23
5.2	Simulated output for the code word [0000111100]	23
5.3	Simulated output for the code word [0100000111]	24
5.4	Simulated output for the code word [0101100001]	24
5.5	Simulated output for the code word [1011000100]	25

CHAPTER 1

INTRODUCTION

1.1 INTRODUCTION TO COMMUNICATION SYSTEMS

Communication is the process of conveying message at a distance. The need for efficient and reliable data communication systems has been rising rapidly in recent years. The following figure shows a simplified block diagram of a communication system.

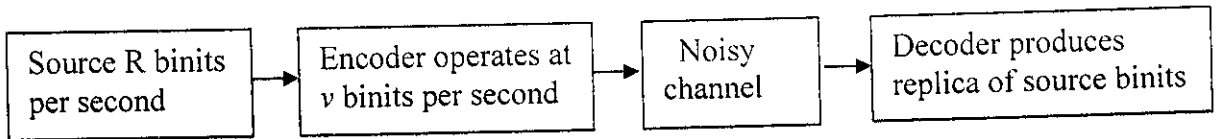


Fig 1.1 Block Diagram of a Communication System

In the figure 1.1 the noise present in the channel introduces errors in the received message. Hence, in order to retrieve back the original message transmitted by the sender some form of error correction is necessary. The forward error correction is most suited for wireless communication and for long distance communication where retransmission is very expensive.

The addition of redundancy in the coded messages for FEC implies a need for higher transmission bandwidth. Moreover, the use of error control coding adds complexity to the system. Especially the design of the decoder becomes complex. Hence the need for forward error correcting codes that effectively uses the channel bandwidth with a moderate or low complexity is necessary.

1.2 OBJECTIVE OF THE PROJECT

The objective of the project is to implement a soft decision decoder for LDPC codes in Field Programmable Gate Array (FPGA). The implemented decoder uses sum-product algorithm, which is a soft decision decoding algorithm. The necessary algorithms and methods to implement the decoder, which uses real values that are not supported by FPGA, have been developed and synthesized.

1.3 OVERVIEW OF THE PROJECT

Low-density parity-check (LDPC) codes are a class of linear block codes. They provide near-capacity performance on a large collection of data transmission and storage channels while simultaneously admitting implementable decoders. The LDPC codes were first developed by Gallager at MIT in 1962 for his PhD thesis.[1]

At the time, their incredible potential remained undiscovered due to the computational demands of simulation in an era when vacuum tubes were only just being replaced by the first transistors. These codes remained largely neglected for over 35 years. In the mean time the field of forward error correction was dominated by highly structured algebraic block and convolutional codes. Despite the enormous practical success of these codes, their performance fell well short of the theoretically achievable limits set down by Shannon. The relative quiescence of the coding field was utterly transformed by the introduction of turbo codes, proposed by Berrou, Glavieux and Thitimajshima in 1993, wherein all the key ingredients of successful error correction codes were replaced: turbo codes involve very little algebra, employ iterative, distributed algorithms, focus on average (rather than worst-case) performance, and rely on soft (or probabilistic) information extracted from the channel.

New generalizations of Gallager's LDPC codes by a number of researchers including Mackay [2], Luby, Mitzenmacher, Shokrollahi, Spielman, Richardson and Urbanke, produced new irregular LDPC codes which offer certain practical advantages and an arguably cleaner setup for theoretical results.[3] Today, design techniques for LDPC codes exist which enable the construction of codes which approach the Shannon's capacity to within hundredths of a decibel. The main research interests are low complexity encoding and efficient decoding schemes. In this project, a soft decision decoder for LDPC codes is designed, simulated and tested using Modelsim.

CHAPTER 2

INTRODUCTION TO LDPC CODES

2.1 INTRODUCTION

Low-density parity-check (LDPC) codes are a class of linear block codes. They have parity-check matrices with a very small number of non-zero entities.[4] Due to the sparseness of this matrix their decoding complexity increases only linearly with the length of the codes. LDPC codes are designed by constructing a sparse parity-check matrix first and then determining a generator matrix for the code afterwards.[5][6] The encoding is similar to that of the other block codes, the main difference of LDPC codes lies in the decoding methods. Their decoding algorithms can recover the original code words in the presence of large noise.

2.2 REGULAR AND IRREGULAR LDPC CODES

The LDPC codes are of two types: They are regular and irregular codes. The regular codes are those codes in which the row weight and column weight distributed evenly throughout the H matrix. In the case of irregular codes the row and column weights will differ throughout the matrix. Basically design of regular LDPC codes is easy. But generally it is observed that the carefully designed irregular codes will give better performance for very large code lengths. [7][8]

2.3 REPRESENTATIONS OF LPDC CODES

There are two different possibilities to represent LDPC codes.[9] Like all linear block codes they can be described via matrices. The second method is a graphical representation.

2.3.1 Matrix Representation

Let's look at an example for a low-density parity-check matrix first.

$$H = \begin{bmatrix} 1 & 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 1 & 1 \\ 0 & 1 & 1 & 1 & 0 & 1 \end{bmatrix} \quad 2.1$$

The matrix defined in equation (2.1) is a parity check matrix with dimension $n \times m$ for a (6,4) code. We can now define two numbers describing these matrixes. w_r for the number of 1's in each row and w_c for the columns. For a matrix to be called low-density the two conditions $w_c \ll n$ and $w_r \ll m$ must be satisfied. [8] The matrix is called sparse matrix since the number of ones in the matrix will be less than the number of zeros.

2.3.2 Graphical Representation

Tanner considered LDPC codes (and a generalization) and showed how they may be represented effectively by a so-called bipartite graph, now call a Tanner graph[11]. The Tanner graph of an LDPC code is analogous to the trellis of a convolutional code in that it provides a complete representation of the code and it aids in the description of the decoding algorithm. A bipartite graph is" a graph (nodes connected by edges) whose nodes may be separated into two types, and edges may only connect two nodes of different types.

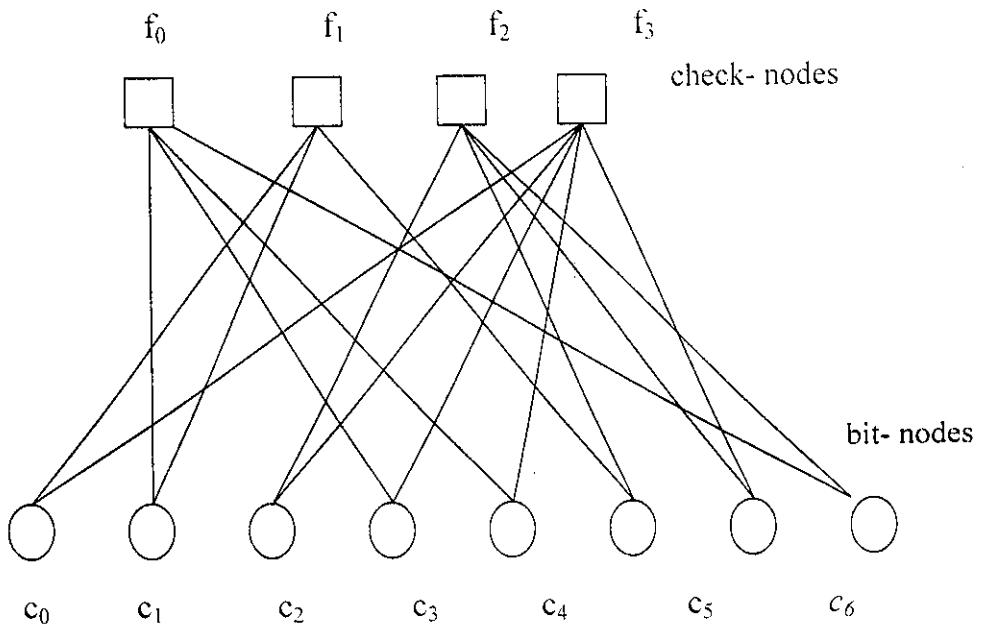


Figure 2.1

The two types of nodes in a Tanner graph are the variable nodes and the check nodes (which we shall call v-nodes and c-nodes, respectively). The Tanner graph of a code is drawn according to the following rule: check node j is connected to variable node i whenever element h_{ji} in H is a 1. One may deduce from this that there are $m = n - k$ check nodes, one for each check equation, and n variable nodes, one for each code bit c_i . Further, the m rows of

H specify the m c-node connections, and the n columns of H specify the n v-node connections.

Example: Consider a $(10, 5)$ linear block code with $w_c = 2$ and $w_r = w_c(n/m) = 4$ with the following H matrix:

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

The Tanner graph corresponding to H is depicted in Fig. 2.2. Observe that v-nodes c_0 , c_1 , c_2 , and c_3 are connected to c-node f_0 in accordance with the fact that, in the zeroth row of H , $h_{00} = h_{01} = h_{02} = h_{03} = 1$ (all others are zero). Observe that analogous situations hold for c-nodes f_1 , f_2 , f_3 , and f_4 which correspond to rows 1, 2, 3, and 4 of H , respectively. Note, as follows from the fact that $\mathbf{cH}^T = \mathbf{0}$, the bit values connected to the same check node must sum to zero. We may also proceed along columns to construct the Tanner graph. For example, note that v-node c_0 is connected to c-nodes f_0 and f_1 in accordance with the fact that, in the zeroth column of H , $h_{00} = h_{10} = 1$.

Note that the Tanner graph in this example is regular: each v-node has two edge connections and each c-node has four edge connections (that is, the degree of each v-node is 2 and the degree of each c-node is 4). This is in accordance with the fact that $w_c = 2$ and $w_r = 4$. It is also clear from this example that $mw_r = nw_c$.

For irregular LDPC codes, the parameters w_c and w_r are functions of the column and row numbers and so such notation is not generally adopted in this case

ρ_d denotes the fraction of all edges connected to degree- d c-nodes and d_c denotes the maximum c-node degree. Note for the regular code above, for which $w_c = d_v = 2$ and $w_r = d_c = 4$, we have $\lambda(x) = x$ and $\rho(x) = x^3$.

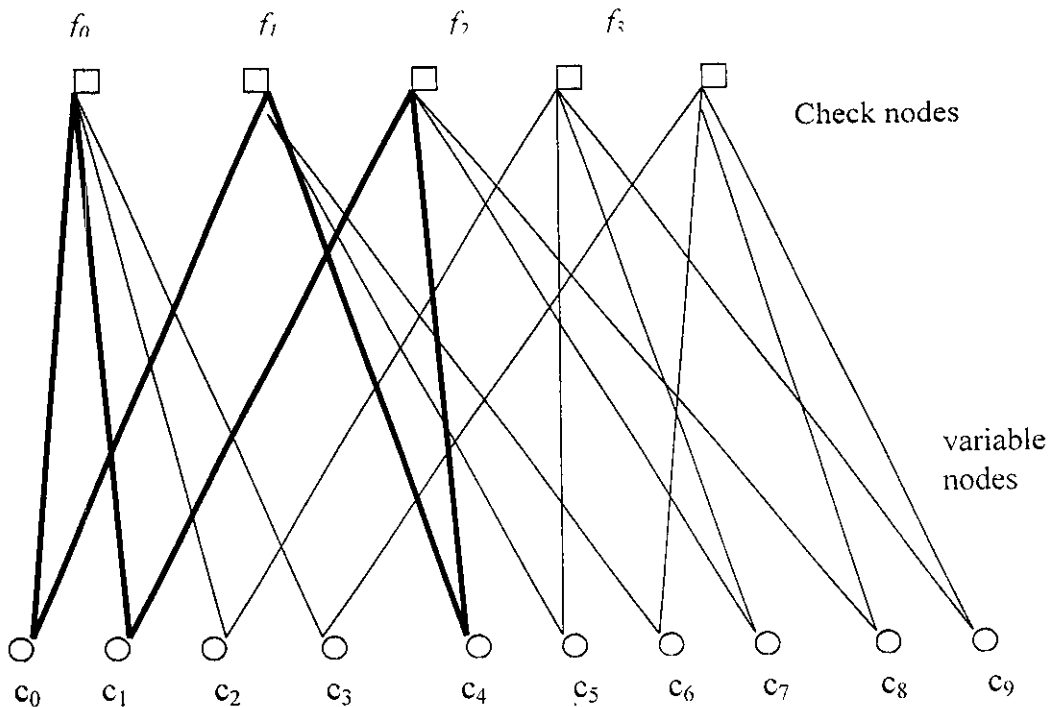


Figure 2.2: Tanner graph for example code.

A cycle (or loop) of length v in a Tanner graph is a path comprising v edges which closes back on itself. The Tanner graph in the above example possesses a length-6 cycle as exemplified by the six bold edges in the figure. The girth γ 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, and such 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 short cycles, because they degrade the performance of the iterative decoding algorithm used for LDPC codes. This fact will be made evident in the discussion of the iterative decoding algorithm.

2.4 IMPORTANT DESIGN PARAMETERS

Design of LDPC codes involves many parameters which are often determined in consideration of the target application.[11]

2.4.1 Code size

The code size specifies the dimensions of the parity check matrix ($M \times N$). Sometimes the term code length is used referring to N . Generally a code is specified using its length and

row-column weights in the form (N, j, k) . M can be deduced from the code parameters N, j and k . It has been shown that very long codes perform better than shorter ones. Long codes are therefore desirable to have good performance. However, their hardware implementation requires more resources (memory plus processing nodes).

2.4.2 Code Weights and Rate

The rate of a code, R , is the number of information bits over the total number of bits transmitted. It is expressed as $\left(\frac{N-M}{N}\right)$. Higher row and column weights result in more computations at each node because of many incoming messages. However, if many nodes contribute in estimating the probability of a bit the node reaches a consensus faster. Higher rates mean fewer redundancy bits. That is, more information data is transmitted per block resulting in high throughput. However, low redundancy means less protection of bits and therefore less decoding performance or higher error rate. Low rate codes have more redundancy with fewer throughputs. More redundancy results in more decoding performance. However, very low rates may have poor performance with a small number of connections. LDPC codes with column-weight of two have their minimum distance increasing logarithmically with code size as compared to a linear increase for codes with column weight of three or higher. As a result column-weight two codes perform poorly compared to higher column-weight codes. Column weights higher than two are usually used. Although regular codes are commonly used, carefully constructed irregular codes could have better error correcting performance.

2.4.3 Code Structure

The structure of a code is determined by the pattern of connections between rows and columns. The connection pattern determines the complexity of the communication interconnect between check and variable processing nodes in an encoder and decoder hardware implementations. Random codes do not follow any predefined or known pattern in row-column connections. Structured codes on the other hand have a known interconnection pattern. Many methods have been developed for constructing those types of codes.

2.4.4 Number of iterations

The number of iterations is the number of times the received bits are estimated before a hard decision is made by the decoding algorithm. A large number of iterations may ensure decoding algorithm convergence but will increase decoder delay and power consumption. The number of corrected errors generally decreases with an increasing number of iterations. In performance simulations a large number of iterations, (about 100 to 200), can be used. For practical applications 20 to 30 iterations are commonly used.

2.5 LDPC CODE CONSTRUCTION

Several different algorithms exist to construct suitable LDPC codes. Gallager himself introduced one. Furthermore MacKay proposed one to semi-randomly generate sparse parity check matrices. This is quite interesting since it indicates that constructing good performing LDPC codes is not a hard problem. In fact, completely randomly chosen codes are good with a high probability. The problem that will arise is that the encoding complexity of such codes is usually rather high.

Clearly, the most obvious path to the construction of an LDPC code is via the construction of p low-density parity-check matrix with prescribed properties. A large number of design techniques exist in the literature, and we introduce some of the more prominent ones in this section, albeit at a superficial level. The design approaches target different design criteria, including efficient encoding and decoding, near-capacity performance, or low-error rate floors. (Like turbo codes, LPDC codes often suffer from low-error rate floors, owing both to poor distance spectra and weaknesses in the iterative decoding algorithm.[9])

2.5.1 Gallager Codes

The original LDPC codes due to Gallager [1] are regular LDPC codes with an H matrix of the form

$$H = \begin{bmatrix} H_1 \\ H_2 \\ \cdot \\ \cdot \\ H_{w_c} \end{bmatrix} \quad (2.2)$$

where the sub matrices H_d have the following structure. For any integers μ and w_r than greater than 1, each sub matrix H_d is $\mu \times \mu w_r$ with row weight w_r and column weight 1. The sub matrix H_1 has the following specific form: for $i = 1, 2, \dots, \mu$ the i -th row contains all of its $w_r - 1$'s in columns $(i - 1) w_r + 1$ to $i w_r$. The other submatrices are simply column permutations of H_1 . It is evident that H is regular, has dimension $\mu w_c \times \mu w_r$ and has row and column weights w_r and w_c , respectively. The absence of length-4 cycles in H is not guaranteed, but they can be avoided via computer design of H . Gallager showed that the ensemble of such codes has excellent distance properties provided $w_c \geq 3$ and $w_r > w_c$. Further, such codes have low-complexity encoders since parity bits can be solved for as a function of the user bits via the parity-check matrix [10].

Gallager codes were generalized by Tanner in 1981 [11] and were studied for application to code-division multiple-access communication channel in. Gallager codes were extended by MacKay and others.

2.5.2 MacKay Codes

MacKay had independently discovered the benefits of designing binary codes with sparse H matrices and was the first to show the ability of these codes to perform near capacity limits. MacKay has archived on a web page a large number of LPDC codes he has designed for application to data communication and storage, most of which are regular. He provided the algorithms to semi-randomly generate sparse H matrices. A few of these are listed below in order of increasing algorithm complexity (but not necessarily improved performance).

1. H is created by randomly generating weight- w_c columns and (as near as possible) uniform row weight.
2. H is created by randomly generating weight- w_c columns, while ensuring weight- w_r rows, and no two columns having overlap greater than one.
3. H is generated as in 2, plus short cycles are avoided.
4. H is generated as in 3, plus $H = [H_1 \ H_2]$ is constrained so that H_2 is invertible (or at least H is full rank)

One drawback of MacKay codes is that they lack sufficient structure to enable low-complexity encoding. Encoding is performed by putting H in the form $[P^T \ I]$ via Gauss-Jordan elimination of P , which then allows the user bits to be put in the systematic form

$G = [I \ P]$. The problem with encoding via G is that the sub matrix P is generally not sparse so that, for codes of length $n = 1000$ or more, encoding complexity is high.

2.5.3 Repeat-Accumulate Codes

A type of code, called a repeat-accumulate (RA) code, which has the characteristics of both serial turbo codes and LDPC codes, was proposed in. The encoder for an RA code is shown in Fig. 2.3, where it is seen that user bits are repeated (2 or 3 times is typical), permuted, and then sent through an accumulator (differential encoder). These codes have been shown to be capable of operation near capacity limits, but they have the drawback that they are naturally low rate (rate 1/2 or lower). [13]

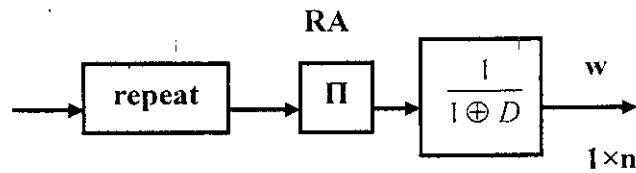


Fig.2.3 : Encoders for the repeat-accumulate (RA)

CHAPTER 3

ENCODING AND DECODING

P-3544



3.1 ENCODING

The LDPC encoder transforms each input message block 'u' into a distinct N-tuple (N-bit sequence) code word 'c'. The codeword length N, where $N > K$, is then referred to as the block-length. And, there are 2^K distinct code words corresponding to the 2^K message blocks. This set of the 2^K code words is termed as a $C(N, K)$ linear block code. The word linear signifies that the modulo-2 sum of any two or more code words in the code $C(N, K)$ is another valid codeword. The number of non-zero symbols of a codeword 'c' is called the weight, while the number of bit-positions in which two code words differ is termed as the distance. The minimum distance of a linear code is denoted by d_{\min} , and determined by the weight of that codeword in the code $C(N, K)$, which has the minimum weight.[14]

The unique and distinctive nature of the code words implies that there is a one-to-one mapping between a K-bit information sequence 'u' and the corresponding N-bit codeword 'c' described by the set of rules of the encoder.

A generator matrix 'G' is determined by performing Gauss-Jordan elimination on 'H' to obtain it in the form:

$$[H' = [A, I_{N-K}]] \quad 3.1$$

Where 'A' is a $(N-K) \times K$ binary matrix and I_{N-K} is the size N-K identity matrix. The generator matrix is then:

$$G = [A, I_{N-K}] \quad 3.2$$

Since LDPC codes are linear block codes, a codeword is generated by multiplying the input vector with the generator matrix,

$$c = uG \quad 3.3$$

Where 'c' is the code word and 'u' is the input vector bits. Since 'G' matrix is not sparse, the matrix multiplication at the encoder will have complexity in the order of n^2

3.2 DECODING

The class of decoding algorithms used to decode LDPC codes is collectively termed message-passing algorithms since their operation can be explained by the passing of messages along the edges of a Tanner graph. Each Tanner graph node works in isolation, only having access to the information contained in the messages on the edges connected to it. The message-passing algorithms are also known as iterative decoding algorithms as the messages pass back and forward between the bit and check nodes iteratively until a result is achieved (or the process halted). Different message-passing algorithms are named for the type of messages passed or for the type of operation performed at the nodes. In some algorithms, such as bit-flipping decoding, the messages are binary and in others, such as belief propagation decoding, the messages are probabilities which represent a level of belief about the value of the codeword bits. It is often convenient to represent probability values as log likelihood ratios, and when this is done belief propagation decoding is often called sum-product decoding since the use of log likelihood ratios allows the calculations at the bit and check nodes to be computed using sum and product operations.[15]

There are 2 methods that can be used for decoding of LDPC codes

- Bit-flipping decoding
- Sum-product decoding

3.2.1 Bit-Flipping Decoding

The bit-flipping algorithm is a hard-decision message-passing algorithm for LDPC codes. A binary (hard) decision about each received bit is made by the detector and this is passed to the decoder. For the bit-flipping algorithm the messages passed along the Tanner graph edges are also binary: a bit node sends its value as 0 or 1 to the check node and each check node sends a message to each connected bit node, declaring what value the bit is based on the information available to the check node. The check node determines that its parity-check equation is satisfied if the modulo-2 sum of the incoming bit values is zero. If the majority of the messages received by a bit node are different from its received value the bit node changes (flips) its current value. This process is repeated until all of the parity-check equations are satisfied.

The bit-flipping decoder can be immediately terminated whenever a valid codeword has been found by checking if all of the parity check equations are satisfied. This is true of all

message-passing decoding of LDPC codes and has two important benefits; firstly additional iterations are avoided once a solution has been found, and secondly a failure to converge to a codeword is always detected.

The bit-flipping algorithm is based on the principal that a codeword bit involved in a large number of incorrect check equations is likely to be incorrect itself. The sparseness of H helps spread out the bits into checks so that parity-check equations are unlikely to contain the same set of codeword bits.

Algorithm:

```

1: procedure DECODE(y)
2:
3:  $I = 0$  Initialization
4: for  $i = 1 : n$  do
5:    $M_i = y_i$ 
6: end for
7:
8: repeat
9:   for  $j = 1 : m$  do Step 1: Check messages
10:    for  $i = 1 : n$  do
11:       $E_{j,i} = \sum_{i' \in B_{j,i}} (M_{i'} \bmod 2)$ 
12:    end for
13:   end for
14:
15:   for  $i = 1 : n$  do Step 2: Bit messages
16:     if the messages  $E_{j,i}$  disagree with  $y_i$  then
17:        $M_i = (r_i + 1 \bmod 2)$ 
18:     end if
19:   end for
20:
21:   for  $j = 1 : m$  do Test: are the parity-check
22:      $L_j = \sum_{i' \in B_j} (M_{i'} \bmod 2)$  equations satisfied
23:   end for
24:   if all  $L_j = 0$  or  $I = I_{max}$  then
25:     Finished
26:   else
27:      $I = I + 1$ 
28:   end if
29: until Finished
30: end procedure

```

Where,

$E_{j,i}$ - message from checknode to bit node

m_i - message from bit node i to the check node

y - received message

3.2.2 Sum-product decoding

The sum-product algorithm is a soft decision message-passing algorithm. It is similar to the bit-flipping algorithm. But with the messages representing each decision (check met, or bit value equal to 1) now probabilities. The sum-product algorithm is a soft decision algorithm which accepts the probability of each received bit as input. The input bit probabilities are called the a priori probabilities for the received bits because they were known in advance before running the LDPC decoder. The bit probabilities returned by the decoder are called the posterior probabilities. In the case of sum-product decoding these probabilities are expressed as log-likelihood ratios.[16]

For a binary variable x it is easy to find $p(x = 1)$ given $p(x = 0)$, since $p(x = 1) = 1 - p(x = 0)$ and so we only need to store one probability value for x . Log likelihood ratios are used to represent the metrics for a binary variable by a single value

$$p(x = 0) = \frac{p(x = 1)/p(x = 0)}{1 + p(x = 1)/p(x = 0)} = \frac{e^{-L(x)}}{1 + e^{-L(x)}} \quad 3.4$$

and

$$p(x = 1) = \frac{p(x = 0)/p(x = 1)}{1 + p(x = 0)/p(x = 1)} = \frac{e^{L(x)}}{1 + e^{L(x)}} \quad 3.5$$

The benefit of the logarithmic representation of probabilities is that when probabilities need to be multiplied log-likelihood ratios need only be added, reducing implementation complexity. The sum-product algorithm iteratively computes an approximation of the MAP value for each code bit. Input is the log likelihood ratios for the a priori message probabilities. The a priori probabilities for the BSC are

$$r_i = \log p/(1-p) \text{ if } y_i = 1$$

Or

$$r_i = \log(1-p)/(p) \text{ if } y_i = 0 \quad 3.6$$

In sum-product decoding the extrinsic message from check node j to bit node i , $E_{j,i}$, is the LLR of the probability that bit i causes parity-check j to be satisfied.

$$E_{j,i} = \log \left(\frac{1 + \prod_{i' \in B_j, i' \neq i} \tanh(M_{j,i'}/2)}{1 - \prod_{i' \in B_j, i' \neq i} \tanh(M_{j,i'}/2)} \right) \quad 3.7$$

The intrinsic message from check node j to bit node i , $M_{j,i}$, is given by,

$$M_{j,i} = \sum_{j' \in A_i, j' \neq j} E_{j',i} + r_i \quad 3.8$$

The total LLR of the bit stream is

$$L_i = \sum_{j \in A_i} E_{j,i} + r_i \quad 3.9$$

The total LLR can be either positive or negative number. The hard decision is taken. When total LLR is positive the decision is '0' else '1'. The code word is z . Then syndrome calculation is done by $s=zH'$. When s is zero then z is a valid codeword, and the decoding stops, returning z as the decoded word

For an AWGN channel the a priori LLRs are given by

$$r_i = 4y_i(E_s/N_o) \quad 3.10$$

The extrinsic LLR and the total LLR calculation are done to find the codeword .

Algorithm:

procedure DECODE(r)

$I = 0$

for $i = 1 : n$ do

 for $j = 1 : m$ do

```

    end for
end for
repeat
for j = 1 : m do
    for i ∈ Bj do

$$E_{j,i} = \left( \prod_r \alpha_{j,r} \right) \phi \left( \sum_r \phi(\beta_{j,r}) \right)$$

    end for
end for
for i = 1 : n do
    
$$L_i = \sum_{j \in A_i} E_{j,i} + r_i$$


$$z_i = \begin{cases} 1, & L_i \leq 0 \\ 0, & L_i > 0 \end{cases}$$

end for
for i = 1 : n do
    for j ∈ Ai do

$$M_{j,i} = \sum_{j' \in A_i, j' \neq j} E_{j',i} + r_i$$

    end for
end for
I = I + 1
end if
until Finished
end procedure

```

Where,

$E_{j,i}$ - extrinsic information from check node to bit node

$M_{i,j}$ - information from bit node to check node

CHAPTER 4

HARDWARE IMPLEMENTATION

4.1 INTRODUCTION

The LDPC codes can be decoded by using tanner graph method. In this method, as explained in the sum-product decoding, the check nodes can be used to manipulate the extrinsic information concurrently with that of the other check nodes. Hence parallelism can be achieved using the current FPGAs.[17] In this project the decoder for LDPC codes have been implemented by designing the necessary algorithms. These algorithms are used for implementing the real values in FPGAs.

Due to their low complexity and parallel manipulation for decoding, the LDPC codes are more suited for higher data rate systems. Such an implementation is done in this project.

4.2 ALGORITHMS FOR HARDWARE IMPLEMENTATION

The real values are not supported by the synthesis tools for implementing in FPGAs. So, necessary algorithms for synthesizing and implementing real values in FPGAs were developed. Also, the log and tanh functions needed for calculating the extrinsic information are not readily available for simulation in Modelsim, so methods to implement log and tanh functions are also developed.

4.2.1 Implementation of Real Values

The real values are implemented by expressing them as mantissa and exponent form as explained below,

Let x be a real valued number, $x= 1.7394$. Then this number can be represented as mantissa and exponent as

$$X= 17394 E -4$$

Here mantissa of $x = 17394$ and exponent of x is -4 .

The normal arithmetic operations such as addition, multiplication and division for real values are implemented as follows,

4.2.1.1 Addition

Let a and b be the two real values whose sum is to be found. Then using the exponent and mantissa form the addition is done as follows,

$$a=1.2345 \quad b=3.456$$

Mantissa of a, $x= 12345$ and exponent of x, $x_e= -4$

Mantissa of b, $y= 3456$ and exponent of y, $y_e= -3$

1. Normalizing the exponents

$$x= 12345, x_e= -4$$

$$y= 34560, y_e= -4$$

2. $sum= (x+y) E(x_e \text{ or } y_e)$

3. Answer = 46905 E -4

4.2.1.2 Multiplication

In order to multiply two real values, we must at first normalize the exponent part of the numbers and then multiply the mantissa part together.

Eg.: $a= 7.623 \quad b= 1.5012$

$$x= 7623, x_e= -3$$

$$y= 15012, y_e= -4$$

1. $mul= x*y;$

2. $mule= x_e+y_e$

3. Answer = 114436476 E -7

Normalizing the exponents

$$= 114436 E -4$$

4.2.1.3 Division

Before trying to divide two real values, their exponents should be normalized. The obtained mantissa parts can now be divided using the below algorithm:

- 1) Let x & y be the mantissa parts of the two numbers.
- 2) Assign temp=0.
- 3) Check if x is greater than or equal to y.
- 4) If true, subtract y from x and increment a temporary variable temp.
- 5) Repeat from step 2 until the condition is false.
- 6) Assign a[0]=temp.
- 7) Then multiply x by 10 and repeat from step 2.
- 8) Do the above steps again to obtain the values of a[1], a[2], a[3] & a[4].
- 9) The result of the division is a[0]*10000+a[1]*1000+a[2]*100+a[3]*10+a[4],
with an exponent of -4.

4.2.2 Implementation of Log, Tanh And Exponential Functions

The log and tanh function are essential to calculate the extrinsic information in a check node as given in the equation 3.7 which is reproduced here,

$$E_{j,i} = \log \left(\frac{1 + \prod_{i' \in B_j, i' \neq i} \tanh(M_{j,i'}/2)}{1 - \prod_{i' \in B_j, i' \neq i} \tanh(M_{j,i'}/2)} \right) \quad 4.1$$

The log and tanh function are implemented using the exponential function.

4.2.2.1 Exponential Function

The exponential function is implemented using the power series,

The exponential function is given as,

$$e^x = \sum_{n=0}^{\infty} \frac{x^n}{n!} = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \frac{x^4}{4!} + \dots \quad 4.2$$

where x is a real value.

The number of terms taken here is 10, which gives a precision of 5 digits, sufficient for implementing log and tanh functions.

4.2.2.2 Tanh Function

The tanh function for a real value x is given as,

$$\tanh x = \frac{\sinh x}{\cosh x} = \frac{e^x - e^{-x}}{e^x + e^{-x}} = \frac{e^{2x} - 1}{e^{2x} + 1} \quad 4.3$$

It is required to find the extrinsic information for a particular bit node from a check node in sum-product algorithm. Thus if exponential function for a real value could be found then the tanh function for that value could be calculated.

4.2.2.3 Log Function

The ln function for a real value z is given as,

$$\ln z = 2 \left(\frac{z-1}{z+1} + \frac{1}{3} \left(\frac{z-1}{z+1} \right)^3 + \frac{1}{5} \left(\frac{z-1}{z+1} \right)^5 + \dots \right); \quad 4.4$$

This series can be derived from the above Taylor series. It converges more quickly than the Taylor series, especially if z is close to 1. For example, for z = 1.5, the first three terms of the second series approximate ln(1.5) with an error of about 3×10^{-6} . The quick convergence for z close to 1 can be taken advantage of in the following way: given a low-accuracy approximation $y \approx \ln(z)$ and putting,

$$A = \frac{z}{\exp(y)}; \quad 4.5$$

the logarithm of z is

$$\ln z = y + \ln A. \quad 4.6$$

It is also used to determine the extrinsic information for a bit node from a check node.

CHAPTER 5

SIMULATION RESULTS AND DISCUSSIONS

The simulation is done using Modelsim 6.3f. The message bits are encoded using MATLAB 7.6.0.324 (R2008a). The code words are then passed through the Gaussian channel and the final code word are obtained from the channel using Matlab. Then these code words are decoded, using the algorithm developed, using Modelsim 6.3f.

5.1 Simulation Results

The simulation result for the soft decision decoder is given as follows. The input data is first encoded using Matlab and it is transmitted through the AWGN channel. The parity check matrix used for encoding is

$$H = \begin{bmatrix} 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 1 & 0 & 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 \\ 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \quad 5.1$$

1. For the input data [00001] the codeword generated using Matlab is

$$C = [0000111100]$$

The received data for the transmitted codeword is

$$[-1.0934 \ -0.6371 \ -1.2942 \ 0.0916 \ 0.9318 \ 1.0570 \ 1.5334 \ 1.0296 \ -1.0478 \ -1.4162]$$

The decoder output for the received data is given below:

In the above example the fourth bit is received with error and after 10 iteration of decoding algorithm the output is decoded.

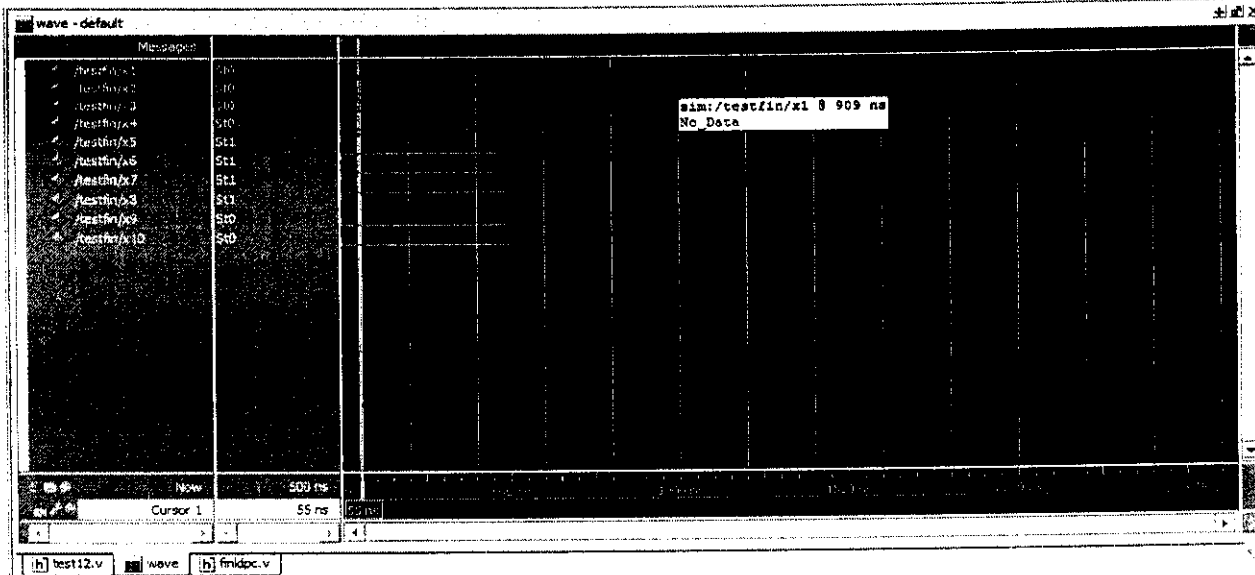


Figure 5.1: Simulated output for the code word [0000111100]

2. For the input data [00110] the codeword generated using Matlab is

$$C = [0011001111]$$

The received data for the transmitted codeword is

$$[-1.1867 \ -0.2742 \ 0.4117 \ 3.1832 \ -1.1364 \ -0.8861 \ 2.0668 \ 1.0593 \ 0.9044 \ 0.1677]$$

The decoder output for the received data is given below:

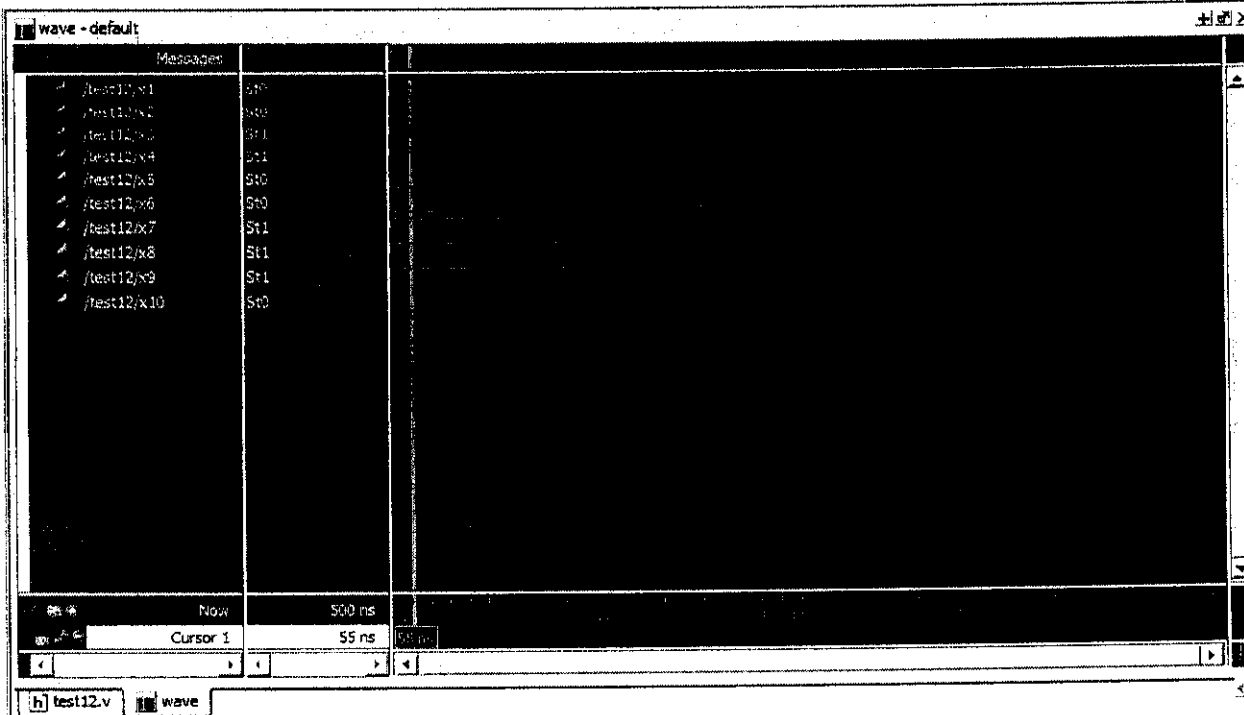


Figure 5.2: Simulated output for the code word [0000111100]

3. For the input data [01000] the codeword generated using Matlab is

$$C = [0100000111]$$

The received data for the transmitted codeword is

$$[-1.8020 \ 1.1287 \ -1.5282 \ -0.2924 \ -1.4025 \ -0.7356 \ -0.8903 \ 0.5390 \ -0.0853 \ 0.9704]$$

The decoder output for the received data is given below:

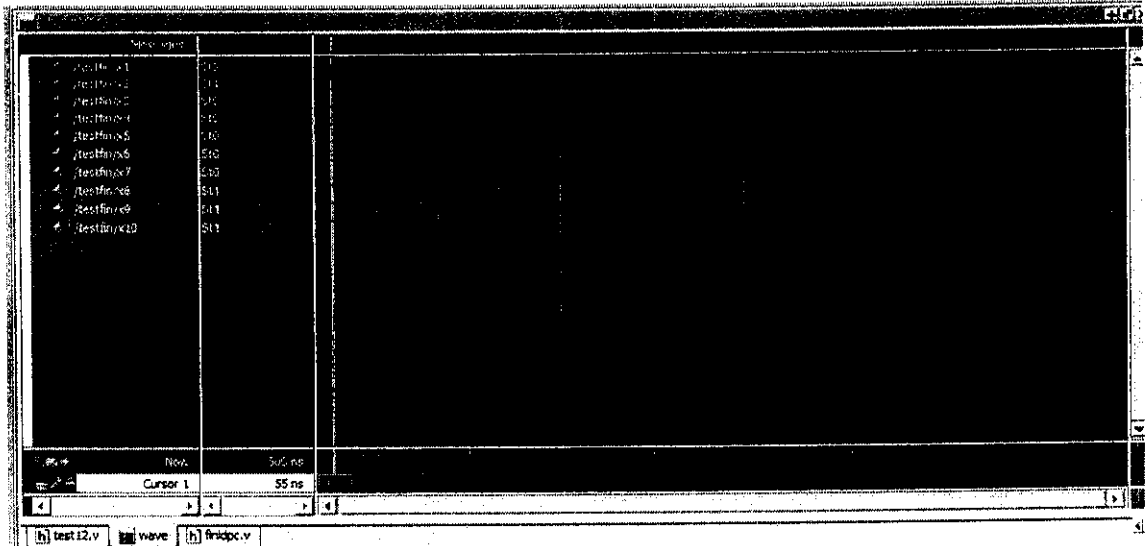


Figure 5.3: Simulated output for the code word [0100000111]

4. For the input data [01011] the codeword generated using Matlab is

$$C = [0101100001]$$

The received data for the transmitted codeword is

$$[-1.3244 \ -0.2492 \ -0.9060 \ 1.2158 \ 0.1401 \ -0.1068 \ -0.1081 \ -1.0282 \ -0.7545 \ 1.1310]$$

The decoder output for the received data is given below:

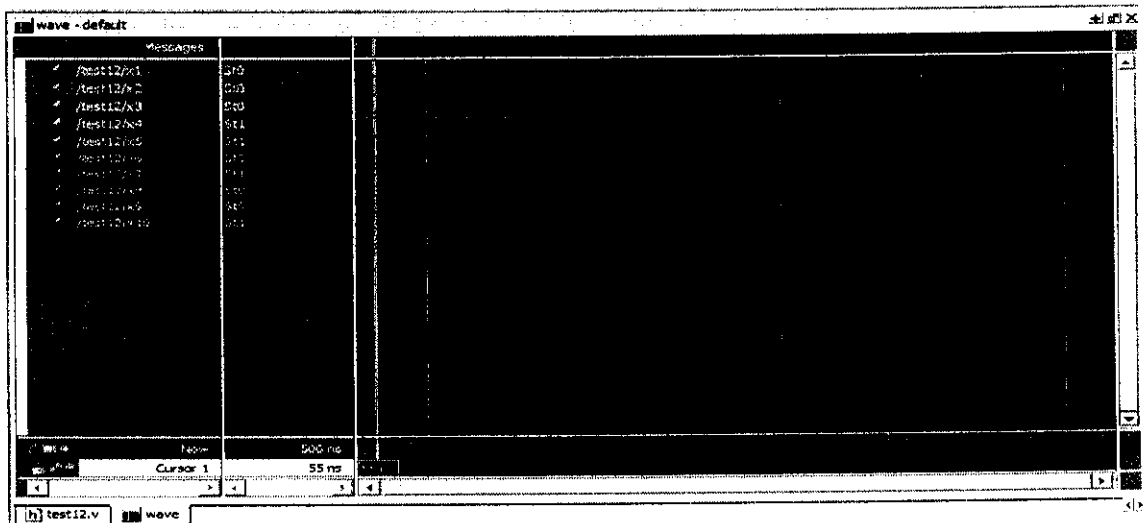


Figure 5.4: Simulated output for the code word [0101100001]

5. For the input data [10110] the codeword generated using Matlab is

$$C = [1011000100]$$

The received data for the transmitted codeword is

[0.7837 -1.8328 1.0627 1.1438 -1.5732 -0.4045 -0.4054 0.9812 -0.8364 -0.9127]

The decoder output for the received data is given below:

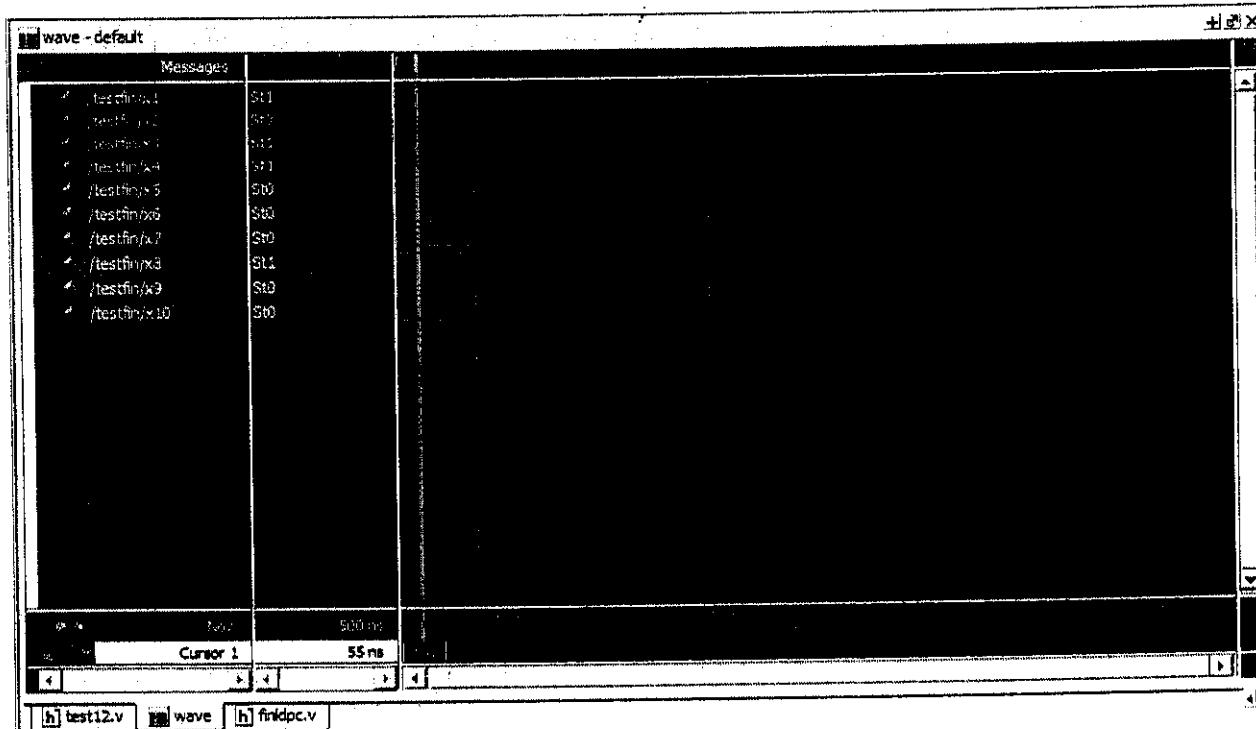


Figure 5.5: Simulated output for the code word [1011000100]

5.2 Synthesis Report Of Tanh Module And Log Module

The Synthesis report of tanh module and log module is given. Various resources and their utilization is shown.

5.2.1 Tanh module

HDL Synthesis Report

Macro Statistics

# Multipliers	: 319
10x5-bit multiplier	: 1
11x5-bit multiplier	: 2
13x5-bit multiplier	: 2
14x5-bit multiplier	: 1
15x5-bit multiplier	: 1
16x5-bit multiplier	: 3
18x5-bit multiplier	: 2
19x5-bit multiplier	: 2
20x5-bit multiplier	: 1
21x5-bit multiplier	: 3
22x5-bit multiplier	: 1
23x5-bit multiplier	: 2
24x5-bit multiplier	: 2
25x5-bit multiplier	: 1
26x5-bit multiplier	: 3
27x5-bit multiplier	: 1
28x5-bit multiplier	: 2
31x5-bit multiplier	: 2
32x32-bit multiplier	: 16
32x5-bit multiplier	: 190
4x11-bit multiplier	: 19
4x15-bit multiplier	: 18
4x5-bit multiplier	: 20
4x8-bit multiplier	: 19
5x5-bit multiplier	: 1

8x5-bit multiplier	: 1
9x5-bit multiplier	: 1
# Adders/Subtractors	: 1978
1-bit adder carry out	: 93
2-bit adder	: 93
2-bit adder carry out	: 93
2-bit subtractor	: 9
3-bit adder	: 279
3-bit adder carry out	: 93
3-bit subtractor	: 9
32-bit adder	: 138
32-bit subtractor	: 967
4-bit adder	: 186
4-bit subtractor	: 9
5-bit subtractor	: 9
# Comparators	: 1070
32-bit comparator great equal	: 940
32-bit comparator greater	: 45
32-bit comparator less	: 45
33-bit comparator greater	: 11
33-bit comparator less	: 29
# Multiplexers	: 14
32-bit 4-to-1 multiplexer	: 14

Design Summary Report:

Number of External IOBs	: 16 out of 190	8%
Number of External Input IOBs	: 16	
Number of External Input IBUFs	: 16	
Number of LOCed External Input IBUFs	: 16 out of 16	100%
Additional JTAG gate count for IOBs	: 768	
Peak Memory Usage	: 151 MB	
Total REAL time to MAP completion	: 3 secs	
Total CPU time to MAP completion	: 1 secs	

5.2.2 Log module

HDL Synthesis Report

Macro Statistics

# Multipliers	: 356
10x5-bit multiplier	: 1
11x5-bit multiplier	: 2
13x5-bit multiplier	: 2
14x5-bit multiplier	: 3
15x5-bit multiplier	: 3
16x5-bit multiplier	: 5
18x5-bit multiplier	: 2
22x5-bit multiplier	: 1
23x5-bit multiplier	: 2
24x5-bit multiplier	: 2
25x5-bit multiplier	: 1
28x5-bit multiplier	: 2
29x5-bit multiplier	: 8
31x5-bit multiplier	: 2
32x32-bit multiplier	: 27
32x5-bit multiplier	: 188
4x11-bit multiplier	: 25
4x15-bit multiplier	: 18
4x5-bit multiplier	: 28
4x8-bit multiplier	: 29
5x5-bit multiplier	: 1
6x5-bit multiplier	: 1
7x5-bit multiplier	: 1
8x5-bit multiplier	: 1
9x5-bit multiplier	: 1
# Adders/Subtractors	: 2166
1-bit adder carry out	: 87
2-bit adder	: 87

2-bit adder carry out	: 101
2-bit subtractor	: 39
3-bit adder	: 279
3-bit adder carry out	: 101
3-bit subtractor	: 29
32-bit adder	: 138
32-bit subtractor	: 967
4-bit adder	: 150
4-bit subtractor	: 9
5-bit adder	: 170
5-bit subtractor	: 9

Comparators : 1350

32-bit comparator great equal	: 1160
32-bit comparator greater	: 60
32-bit comparator less	: 60
33-bit comparator greater	: 21
33-bit comparator less	: 49

Multiplexers : 18

32-bit 4-to-1 multiplexer	: 18
---------------------------	------

Design Summary Report:

Number of External IOBs	: 29 out of 190	15%
Number of External Input IOBs	: 29	
Number of External Input IBUFs	: 29	
Number of LOCed External Input IBUFs	: 29 out of 29	100%
Additional JTAG gate count for IOBs	: 1168	
Peak Memory Usage	: 251 MB	
Total REAL time to MAP completion	: 5 secs	
Total CPU time to MAP completion	: 2 secs	

5.3 Discussion

From the above simulation we analyzed the for every 1000 message bits transmitted through a noisy Gaussian channel, the LDPC decoder corrects up to 960 bits.

$$\text{Bit Error Rate} = 40/1000$$

$$= 0.04$$

This is an appreciable performance for this small number of codes. In real time practical systems the message bits transmitted will be in thousands of bits, and under those conditions this LDPC decoder performance is well nearer to the Shannon limit. Hence the developed algorithms and methods for designing and implementing the LDPC decoder are found to perform well in noisy channels. Though the complete synthesis of this LDPC decoder is beyond the scope of this project, the synthesis for tanh and log functions are also presented.

REFERENCES

- [1] Gallager, "Low-density parity-check codes," IRE Transactions on Information Theory, vol. IT-8, no.1, pp. 21–28, January 1962.
- [2] David J.C. MacKay and Christopher P. Hesketh, "Performance of Low Density Parity Check Codes as a Function of Actual and Assumed Noise Levels", Electronic Notes in Theoretical Computer Science 74 (2003).
- [3] D. J. C. MacKay, "Good error-correcting codes based on very sparse matrices," IEEE Trans. Inform. Theory, vol. 45, no. 2, pp. 399–431, March 1999.
- [4] S.-Y. Chung, G. D. Forney, Jr., T. J. Richardson, and R. L. Urbanke, "On the design of low-density parity-check codes within 0.0045 dB of the Shannon limit," IEEE Commun. Letters, vol. 5, no. 2, pp. 58–60, February 2001.
- [5] Seo Sangwon; Mudge Trevor; Zhu Yuming; Chakrabarti, Chaitali, "Design and Analysis of LDPC Decoders for Software Defined Radio", Signal Processing Systems, 2007 IEEE Workshop on Digital Object Identifier:10.1109/SIPS.2007.4387546 Publication Year: 2007, Page(s): 210 – 215.
- [6] M. Chiani and A. Ventura, "Design and performance evaluation of some high-rate irregular low-density parity-check codes," in IEEE GLOBECOM, 2001, vol. 2, pp. 990-994.
- [7] T. Richardson and R. Urbanke, "An introduction to the analysis of iterative coding systems", Codes, Systems, and Graphical Models, IMA Volume in Mathematics and Its Applications, pages 1-37. Springer, 2001.
- [8] Todd K. Moon, "Error Correction Coding, Mathematical Methods and Algorithms", A John Wiley & Sons, Inc, Publication, New Delhi, India, Ed.2006.
- [9] R. M. Tanner, "A recursive approach to low complexity codes," IEEE.
- [10] T. Richardson, A. Shokrollahi, and R. Urbanke, "Design of capacity approaching low-density parity-check codes", IEEE Trans. Inf. Theory, vol. 47, no. 2, pp. 619–637, Feb. 2001.
- [11] "Design of repeat-accumulate codes for iterative detection and decoding," IEEE Trans. Signal Processing, vol. 51, pp. 2764–2772, Nov.2003.
- [12] R. Narayanaswami, "Coded modulation with low-density parity-check codes," Master's thesis, Dept. Elect. Eng., Texas A&M Univ., College Station, TX, 2001.
- [13] Sarah J. Johnson, "Introducing Low-Density Parity-Check Codes", School of

- [14] F.R. Kschischang, B.J. Frey and H.-A. Loeliger, "Factor graphs and the sum-product algorithm", IEEE Trans. Inform. Theory, vol 47, pp. 498-519, Feb. 2001.
- [15] Jin sha,"Multi Gb/s LDPC Code design and implementation" vol.17,no 2 February 2009
- [16] T. Richardson and R. Urbanke, "The renaissance of Gallager's low density parity-check codes," IEEE Commun. Mag., vol. 41, no. 8, pp.126–131, Aug. 2003.