LUND UNIVERSITY

Department of Electrical and Information Technology

Master of Science Thesis

BEAST on FPGA

Design and Implementation

Author:
Ali Roman

Examiner:
Dr. Joachim Rodrigues

Lund 2012

©

The Department of Electrical and Information Technology
Lund University
Box 118, S-221 00 LUND
SWEDEN

This thesis is set in Computer Modern 10pt,
with the LaTeX Documentation System

©Ali Roman

Printed in E-huset Lund, Sweden.
Mar. 2012

Abstract

Data decoders and decoding processes in modern communication systems are of significant importance for data reliability. The key challenge for a designer is to ensure data reliability. With development of different channel coding schemes, variety of decoding algorithms are devised. The role of a hardware designer is to provide an efficient implementation of those algorithms for different hardware platforms.

This thesis focuses on the hardware implementation of the BEAST (Bidirectional Efficient Algorithm for Searching Code trees) for decoding of block codes. A complete hardware is designed to implement the strategy of the BEAST[1][3]. The design is described using VHDL language. Besides describing the implementation of the BEAST for FPGA (Field Programmable Gate Array) based platform, ASIC (Application Specific Integrated Circuits) synthesis of the design for an ASIC implementation is also discussed. Synthesis is a process of mapping the design components on basic logic gates. The results of the synthesis process can be used for implementation of the design on an ASIC platform.

The architecture of the design is synthesized by using 130nm CMOS technology, resulting in area of 0.72469 mm^2 and maximum clock frequency of 143 MH_z. BER (Bit Error Rate) simulation is performed to verify the performance of the system for different SNR (Signal to Noise Ratio) values.

Acknowledgement

Leaving Pakistan for studying in Sweden was a major step in my carrier and life that redefined my way of thinking in numerous ways. This decision had provided me an opportunity to discover myself and see my potentials in new perspective. I thank God Almighty from bottom of my heart for providing me such an opportunity and guiding me to think in this direction.

I would also take the opportunity to thank the Swedish government for their generosity by providing free education. This opportunity allowed me to experience a unique educational system along with exposure to very diverse culture and friendly educational environment.

I am very grateful to Florian Hug for his guidance, help, warm support and sharing his technical expertise. I am also thankful to Joachim Rodrigues for his guidance during my study period and for the knowledge and experience which I gained while studying VLSI design and doing IC project. I also want to thank Irina Bocharova for her guidance. I am very grateful to all my teachers and PhD students who had guided me during my study in Sweden.

I am thankful to my friends and colleagues, specifically Usman Farooq for his support and time which we spend here, Asheesh Misra for being so good friend and group mate sharing memorable experiences together, Shahid Mehmood, Yasser Sherazi, Shoaib Shazad, Raza Hussain, Farhan Khan, Karrar Rizvi, Harsh, Kaushik, Ajosh, Manivanan, Muhammad Azher and Adnan Arif for providing guidance and help in the time of need.

At the end, most importantly all my gratitude and thanks goes to my parents and my complete family. Without them I am nothing and their endless support and care had always strengthened me in my life.

Ali Roman
Lund, Feb, 2012

Contents

List of Tables

List of Figures

List of Acronyms

BEAST	Bidirectional Efficient Algorithm for Searching code Tree
FPGA	Field Programmable Gate Array
ASIC	Application-Specific Integrated Circuit
RTL	Register Transfer Level
CMOS	Complementary Metal Oxide Semiconductor
AWGN	Additive White Gaussian Noise
BPSK	Binary Phase Shift Keying
ML	Maximum Likelihood
MAP	Maximum a Posteriori
VHDL	Very high speed integrated circuits Hardware Descriptive language
RAM	Random Access Memory
FDU	Frame Distribution Unit
ASL	Add Select Logic
HFSU	Hamming Frame Selection Unit
MC	Multiplexed Comparator
SSL	Search Select Logic
FSW	Flag Status Word
NC	Node Count

BER Bit Error Rate

SNR Signal to Noise Ratio

LTE Long Term Evolution

Chapter 1

Introduction

1.1 Overview

Data communication from a source to a remote destination can be done by using a digital communication system. The performance of such a communication system is compared in terms of efficiency and data reliability. The communication channel affects the reliability of the transmitted data by adding distortion and noise. The noise and distortion affects the amplitude, phase and frequency of the transmitted signals hence corrupting the transmitted data.

To provide reliable data transmission, various error detection and correction schemes are introduced. For example the Viterbi algorithm[6], is a general maximum likelihood (ML) decoding algorithm. A ML decoder decides for a given received sequence **r** in favor of the codeword **v** such that the probability of **r** conditioned on **v** is maximized. The codeword and the related terminologies will be explained later. For larger block codes, the Viterbi algorithm is too complex and a more efficient approach to obtain a ML decision is given by the BEAST[1][3] algorithm.

Within this thesis, the BEAST decoder is implemented in hardware for deciding on a ML codeword. The hardware implementation accepts data samples representing a valid codeword from a Hamming code with added noise and will provide a ML codeword decision. The current implementation is done for the (8, 4, 4) extended Hamming code[3].

1.2 Thesis Outline

The remaining chapters in this thesis are organized as follows:

Chapter 2 introduces the basic concepts of block codes. The basic communication model is described to elaborate the concepts of data encoding, transmission and decoding process. The theoretical aspects of the BEAST are briefly discussed with a concluding example.

In chapter 3, the proposed implementation of the BEAST is described and all parts of the design and their functionality are discussed. The possible improvements in different design units are also discussed.

Chapter 4 presents the detailed implementation results for FPGA based platforms as well as the synthesis results for ASIC based structures. The comparison of BER and SNR is also provided in this chapter.

The thesis is concluded in chapter 5 with some final remarks and provides an outlook on future work that can be carried out on defined architecture and hardware.

Chapter 2

Theory of the BEAST

2.1 Background

The developments in modern communication technologies and systems with different standards and protocols used for radio, satellite and telecommunication, demand fast and efficient data transfers. Data reliability is becoming more and more an area of concerns for system designers and researchers. In order to meet the requirements of an effective communication, various decoding algorithms were developed for the purpose of error detection and correction. The BEAST[1][3] is such an example.

The BEAST was invented at LUND University and among others, can be used to efficiently decode block codes. This chapter covers important concepts about the basic communication model as well as the terms and norms used in coding theory. The strategy of the BEAST for providing ML decision is discussed with an example.

2.2 Basic communication model and block codes

A communication is said to be effective if the data generated by a source is successfully delivered to the required destination without any data loss. The generated data can be in the form of analog (voice, video or both) or in the digital form. We want to transmit the digital information from a source to its destination using digital transmission systems. In case of an analog source, the analog data is

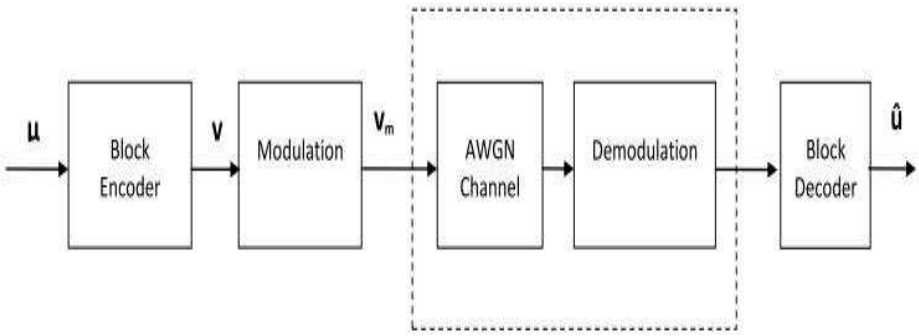

Figure 2.1: Basic communication model

first represented by digital bits using sampling, quantization and analog to digital conversion. The conversion process generates the information sequence. In case of the digital source, the information sequence is generated directly without any conversion process. The next step is to distribute the information sequence into messages, where, each message is applied to the block encoder. The block encoder encodes the message by adding redundant bits, in order to provide a codeword. The redundant bits are added for the purpose of transmission errors correction at the receiver end. The output of the block encoder is provided to the modulator, used for modulation. The modulation process maps the provided input into signaling levels for the purpose of transmission. At the receiver end, the demodulation process is carried out. At the end, the block decoder decodes the received sequence. If the output of the block decoder coincides with the transmitted codeword then the decoding process is considered successful.

2.2.1 Block Encoder

A message **u** is applied to the block encoder. The block encoder encodes the provided message according to the specific coding scheme and provides the codeword. The functionality of the block encoder is to add the redundant bits in a message in order to protect it from transmission errors. A message is the information sequence distributed into blocks of K information bits. K can be determined from the code rate R=K/N of the block code and N represents the total number of encoded bits of the block code. Consider the following information sequence and let K=4 be the number of information bits that each message contains.

Information sequence: 1 0 1 1 1 1 0 1 0 1 1 1 0 0 1 0 1 0 1 0 1 1 0 0.

The messages are denoted by $\mathbf{u}_1$, $\mathbf{u}_2$, ..., $\mathbf{u}_n$. For the given information sequence from above, we obtain the following six messages of four bits each.

$\mathbf{u}_1 = 1\ 0\ 1\ 1$. $\mathbf{u}_2 = 1\ 1\ 0\ 1$. $\mathbf{u}_3 = 0\ 1\ 1\ 1$.
$\mathbf{u}_4 = 0\ 0\ 1\ 0$. $\mathbf{u}_5 = 1\ 0\ 1\ 0$. $\mathbf{u}_6 = 1\ 1\ 0\ 0$.

Each message of length K can have $M=2^K$ bit patterns. The codeword generated by the block encoder is denoted by **v**. The difference $N\text{-}K \geq 0$ is the redundancy used to correct bit errors that occurred during transmission.

A binary (N, K) block code is said to be linear, if every codeword **v** of length N can be represented as a linear combination of K independent basic codeword, that is,

$$\mathbf{v} = \sum_{i=0}^{K-1} u_{(i)} \mathbf{g}_i \tag{2.1}$$

where $u_{(i)} \in \{0, 1\}$ is a information bit and $\mathbf{g}_i$ represents a row of the generator matrix. The generator matrix is a pre-defined matrix of size K×N represented by G, which signifies the structure of the block codes.

$$G = \begin{bmatrix} \mathbf{g}_0 \\ \mathbf{g}_1 \\ . \\ . \\ \mathbf{g}_{K-1} \end{bmatrix}$$

The parity check matrix H of size (N-K)×N having linear independent rows orthogonal to the rows of generator matrix G, can also be used to specify linear block codes[3]. Using the matrix multiplication, equation (2.1) can be equally represented as

$$\mathbf{v}_{(1,N)} = \mathbf{u}_{(1,K)} \cdot G_{(K,N)} \tag{2.2}$$

Let G be the generator matrix of (8, 4, 4) extended Hamming code[2][3] and let $\mathbf{u}$ be a message to be encoded. The G and $\mathbf{u}$ are given as

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

$$\mathbf{u} = \begin{bmatrix} 1 & 0 & 1 & 0 \end{bmatrix}$$

The corresponding codeword $\mathbf{v}$ follows from equation (2.2) as

$$\mathbf{v} = \begin{bmatrix} 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 \end{bmatrix}$$

The Hamming weight of a sequence $\mathbf{u}$ is the number of its non zero symbols and is defined as

$$w_{\mathrm{H}}(\mathbf{u}) = \sum_{i=0}^{K-1} u_{(i)} \tag{2.3}$$

where $u_{(i)}$ denotes the i^{th} symbol of the sequence $\mathbf{u}$.

The Hamming distance d_{H} is defined as the number of different information bits between two sequences. For example, using $\mathbf{u}_1$=1101 and $\mathbf{u}_2$=1011, the Hamming distance between $\mathbf{u}_1$ and $\mathbf{u}_2$ is denoted by $d_{\mathrm{H}}(\mathbf{u}_1, \mathbf{u}_2)$ and computed as

$$\begin{aligned} d_{\mathrm{H}}(\mathbf{u}_1, \mathbf{u}_2) &= \sum_{i=0}^{K-1} (\mathbf{u}_{1(i)} \oplus \mathbf{u}_{2(i)}) \\ &= \sum_{i=0}^{3} (\mathbf{u}_{1(i)} \oplus \mathbf{u}_{2(i)}) \\ &= (u_{1(0)} \oplus u_{2(0)}) + (u_{1(1)} \oplus u_{2(1)}) + (u_{1(2)} \oplus u_{2(2)}) + (u_{1(3)} \oplus u_{2(3)}) \\ &= (1 \oplus 1) + (1 \oplus 0) + (0 \oplus 1) + (1 \oplus 1) \\ &= 0 + 1 + 1 + 0 = 2 \end{aligned} \tag{2.4}$$

2.2.2 Modulation and transmission

In order to transmit the codeword $\mathbf{v}$ over a digital channel, modulation of the codeword symbols is performed. We will consider Binary Phase Shift keying (BPSK) scheme for digital modulation, where code symbols in a codeword are mapped to bipolar modulation symbols, also known as modulation alphabets M. For BPSK

scheme, $M = \left\{-\sqrt{E_\mathrm{s}}, \sqrt{E_\mathrm{s}}\right\}$. The E_s is the average energy per symbol. The modulation symbols in $\mathbf{v}_\mathrm{m}$ are defined as

$$\mathbf{v}_m = \sqrt{E_\mathrm{s}}\,(\mathbf{1} - 2\mathbf{v}) \tag{2.5}$$

If we assume a unit average energy per code symbol, then in case of previous codeword $\mathbf{v}$, defined as

$$\mathbf{v} = \begin{bmatrix}1 & 1 & 0 & 0 & 1 & 1 & 0 & 0\end{bmatrix}$$

we obtain the following modulated sequence

$$\mathbf{v}_\mathrm{m} = \begin{bmatrix}-1 & -1 & 1 & 1 & -1 & -1 & 1 & 1\end{bmatrix}$$

The modulated sequence is transmitted over a Gaussian memoryless channel. Such channel has no feedback mechanism, that is, at a time instance i, the received symbol r_i is only dependent upon current transmitted symbol. The received sequence $\mathbf{r}$ through such channel is given by

$$\mathbf{r} = \mathbf{v}_\mathrm{m} + \mathbf{n} \tag{2.6}$$

Where, n represents the disturbance caused by the channel. The disturbance is introduced in the form of Additive White Gaussian Noise (AWGN), added to the transmitted modulation symbols. An AWGN channel has constant power spectrum over the full bandwidth. An AWGN is modeled as N(μ, σ^2), representing the Gaussian distribution of the noise over channel, having mean μ=0 and variance σ^2=$N_o/2$. The variance is determined from SNR=E_s/N_o, where N_o represents the noise energy. After addition of the noise, the transmitted sequence is corrupted and the received sequence $\mathbf{r}$ for the block decoder[2][3], is given as

$$\mathbf{r} = \begin{Bmatrix}0.18 & -0.56 & 0.91 & 0.60 & -0.02 & 1.60 & 2.80 & 0.34\end{Bmatrix}$$

2.2.3 Block Decoder

The received sequence $\mathbf{r}$ from the channel is fed into the block decoder as an input, which returns a ML decision on the received sequence using the BEAST.

2.3 Understanding the BEAST

The BEAST is a ML decoding algorithm, that is, it yields the minimal bit error decoding probability assuming equiprobable information sequences. The decoding process will be considered successful if the ML codeword decision coincides with the transmitted codeword. BEAST uses code trees for providing ML decision, which we will describe in the following.

BEAST uses a forward and a backward tree, constructed from the generator matrix G. A node in such a code tree is denoted by ξ, while the root node of the forward tree will be represented by $\xi_<$ indicating the starting point. Similar, for the starting node, toor of the backward tree is denoted by $\xi_>$. The branches of the trees are labeled by codeword symbols and every path in such a tree will represent a valid codeword. The weight of the path from root (toor) node to ξ is denoted by $w_<(\xi)$ $(w_>(\xi))$.

A code tree is a graphical representation for the specific block code. This representation is also used for organizing soft decision decoding by the block decoder, depending on the algorithm used. Soft decision uses the real values from the channel and takes those probabilities into account. Soft decision ML decoding[1] is equivalent to finding the codeword $\mathbf{v}$ closest to received sequence $\mathbf{r}$ in terms of weighted Hamming distance. On other hand, the hard decision compares the signs of the data samples in received data sequence. If the data sample is positive then it will be decoded as logic zero otherwise it will be decoded as logic one.

A code tree is defined as the collection of nodes states σ_i at depth i=0, 1, ..., N, connected to the nodes states σ_{i+1} at depth $i+1$ by unidirectional branches[3]. The branches are labeled by the code symbol v_i, i=0, 1, ..., N-1. There will be at most two branches leaving from each state, and carrying opposite code symbols. The code tree for the block code is a time-variant structure where number of nodes and branches varies with the depth or level of the tree.

For construction of trees, the G should be in a minimal-span form. By Definition 2.1[3], let $\mathbf{g}_i$, i=0, 1, ..., K-1, be the i^{th} row of the G. Let start($\mathbf{g}_i$) and end($\mathbf{g}_i$)denote the first and last non zero position of $\mathbf{g}_i$, respectively where $0 \leq$ start($\mathbf{g}_i$) $\leq$ end($\mathbf{g}_i$) $\leq$ N-1. The span of the row $\mathbf{g}_i$ is defined by the interval [start($\mathbf{g}_i$), end($\mathbf{g}_i$)]. The active interval for the row $\mathbf{g}_i$ is given by [start($\mathbf{g}_i$), end($\mathbf{g}_i$)-1]. For example the row [01100110] of G is active from bit position two till bit position six. The active row determines that the corresponding input bit related to that row will decide the transitions to the next states. The matrix G is said to be in the minimal-span form if start($\mathbf{g}_i$) $\neq$ start($\mathbf{g}_j$) and end($\mathbf{g}_i$) $\neq$ end($\mathbf{g}_j$), ensuring the unique starting and ending position of the active rows. The Gaussian elimination method is used to convert any given generator matrix into the minimal-span form. To ensure unique starting positions of every row, the generator matrix is reduced to row echelon form. To ensure the unique row ending positions, the "cancellation above" operation is performed, starting from the last row.

Each column of G defines a depth level of the tree. Let A_i represents the set of active rows in minimal-span generator matrix at position (column) i, i=0, 1, ..., N-1. The node states of minimal tree are defined by

$$\begin{aligned} \sigma_0 &= 0 \\ \sigma_{i+1} &= \sigma_{i+1}^{(0)}\sigma_{i+1}^{(1)}\cdots\sigma_{i+1}^{(K-1)} \end{aligned} \tag{2.7}$$

where $\sigma_{i+1}^{(j)}$ is defined as

$$\sigma_{i+1}^{(j)} = \begin{cases} u_{(j)} & \text{if } \mathbf{g}_j \in A_i \\ \text{no change} & \text{otherwise} \end{cases}$$

that is, the input bit $u_{(j)}$ associated with that active row will be the one contributing in the node state change.

The next step in construction of the tree is to determine the code symbols labeled on the unidirectional branches that are used to connect the nodes with each other. It is also known that at level i, no two simultaneous row become active at same time due to the minimal-span form of G. We will represent such active row with $\mathbf{g}^*$ and the corresponding information bit with u^*. The code symbol $v_i[3]$ is determined as

$$v_i = \begin{cases} \sigma_i\gamma_i + u^* & \text{if } \mathbf{g}^* \text{ exists} \\ \sigma_i\gamma_i & \text{otherwise} \end{cases}$$

where γ_i, i=0, 1, ..., N-1, denotes the column of G. The code symbols can also be found by bit wise multiplying the next state nodes with the considered column of G and add the resultant bits using modulo two additions. Now for construction of the forward tree, consider the first column $[1000]^T$ of previously defined G and let $u_{(0)}u_{(1)}u_{(2)}u_{(3)} \in \mathbf{u}$, represents the input bits in a message. Each input bit corresponds to the one row of G. At a time instant with received input, the respected column of the G is observed to check for an active row. If a row becomes active, then all nodes at that depth level branch into two child nodes at next depth level. If no row starts or end at particular depth level, then every node at that level will be connected to exactly one child node, having same state label. For first column of G, the first bit is one indicating that row $\mathbf{g}_0$ is active. $u_{(0)}$ associated with $\mathbf{g}_0$ is the deciding bit for transition between nodes. Consider the construction of forward tree using G as shown in Figure 2.2.

The starting point of the tree is all zero node, 0000. The next state nodes σ_{i+1} are determined as $u_{(0)}000$. If $u_{(0)}$=0 then the next state node will be 0000 otherwise the next state node will be 1000. Observing the second column $[1100]^T$, $\mathbf{g}_0$ and $\mathbf{g}_1$ are active and there will be transition to four nodes states, defined as $u_{(0)}u_{(1)}00$. The $u_{(1)}$ will be the deciding bit because $\mathbf{g}_1$ is the newly active row of G along with the first row. Meanwhile, the corresponding code symbols are also determined with the help of G and next state nodes. Let us consider the first level of the forward tree, the code symbol associated with branch, connecting the current state node

0000 with the next state node 0000, will be found by multiplying the first column, $[1000]^T$ with 0000. The result of bit wise multiplication will be 0000. Modulo two additions of the resultant bits produces the code symbol for that branch, be equal to 0. Similarly, the code symbol associated with node 1000 will be found as $[1000][1000]^T = 1$.

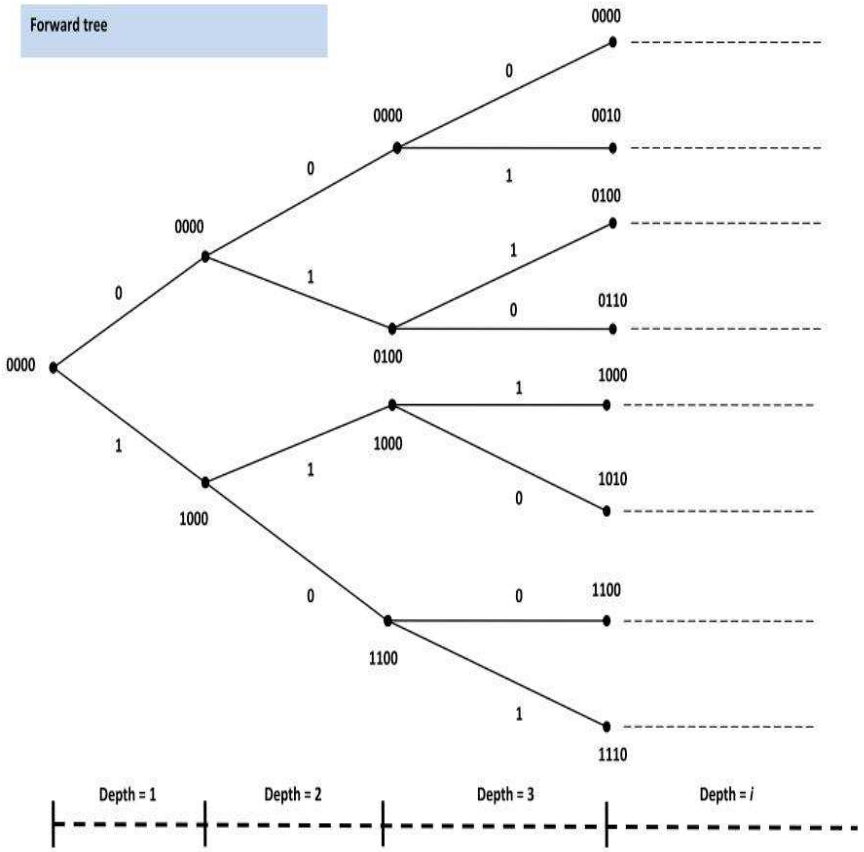

Figure 2.2: Forward tree construction

Consider the construction of backward tree using G as shown in Figure 2.3. The backward tree is constructed in same manner but G will be used in the opposite way, where, the construction is started with considering the last column of G, first.

Let us consider the last column $[0001]^T$ of G. The row $\mathbf{g}_3$ is active as start($\mathbf{g}_3$)=1, means that $u_{(3)}$ will be the deciding bit for the node change. Starting with all zero node, 0000, if $u_{(3)}$=0 then next state node will be 0000 otherwise, the next state node will be 0001. Same method for identification of code symbols regarding backward tree is used as compared to the forward tree. Considering the first level of backward tree, the code symbols for branches connecting the node 0000 with next state nodes 0000 and 0001 will be, $[0000][0001]^T = 0$ and $[0001][0001]^T = 1$, respectively.

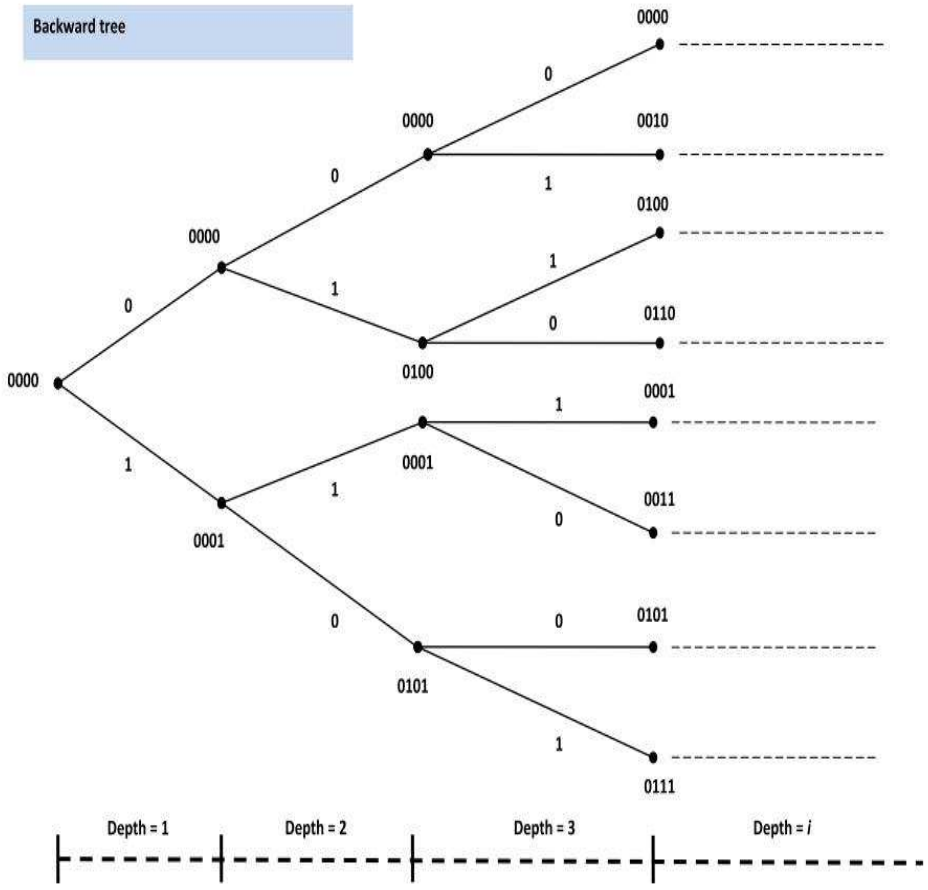

Figure 2.3: Backward tree construction

The next important step is to compute the weighted Hamming distance $\mu(\mathbf{v}, \mathbf{r})$, for the nodes in specific path of the tree. $\mu(\mathbf{v}, \mathbf{r})$ is the additive distance measure between the codeword $\mathbf{v}$ and received sequence $\mathbf{r}[1]$. Consider the given received sequence $\mathbf{r}$, the $\mu(\mathbf{v}, \mathbf{r})$ is computed as

$$\mu(\mathbf{v}, \mathbf{r}) = \sum_{i=1}^{N} \mu(v_i, r_i) \tag{2.8}$$

where $\mu(v_i, r_i)$, represents the metric weight computed by using the code symbol v_i, received symbol r_i and hard decision received sequence h_{ch}, where, $\mu(v_i, r_i)$ is defined as

$$\mu(v_i, r_i) = \begin{cases} |r_i| & \text{if } h_{ch}(r_i) \neq v_i \\ 0 & \text{otherwise} \end{cases}$$

For the forward tree according to Figure 2.2, consider the first depth level of the tree where parent node is 0000 and child nodes are 0000 and 1000. For previously defined received sequence $\mathbf{r}$, the hard decision received sequence h_{ch} is shown in Table 2.1. The $h_{ch}(r_1)$=0, at depth one of the forward tree. Let us first consider

Table 2.1: Hard decision received sequence.

$\mathbf{r}$	0.18	-0.56	0.91	0.60	-0.02	1.60	2.80	0.34
h_{ch}	0	1	0	0	1	0	0	0

the branch from node 0000, to node 0000. The $h_{ch}(r_1)$ is equal to the code symbol of the branch, hence the metric weight will be 0. Consider the branch from node 0000 to node 1000, where $h_{ch}(r_1) \neq v_1$. In this case, we will assign the absolute value of first received sample r_1=0.18 as the weight of the branch. The weighted Hamming distance regarding forward tree is shown in Figure 2.4.

Similarly, the weighted hamming distance for backward tree is computed. In this case, the last received sample 0.34 along with its hard decision received value will be processed first.

2.3.1 The strategy of BEAST

Consider that our goal for using the BEAST is to find a codeword by searching the trees, for those paths whose distance is smaller than or equal to the determined threshold. When doing ML decoding, we search for the paths with smallest distance (e.g. using the weighted Hamming distance) toward the given received sequence $\mathbf{r}$. However, the weight of this smallest distance is not known beforehand, and thus, we need to determine a suitable threshold.

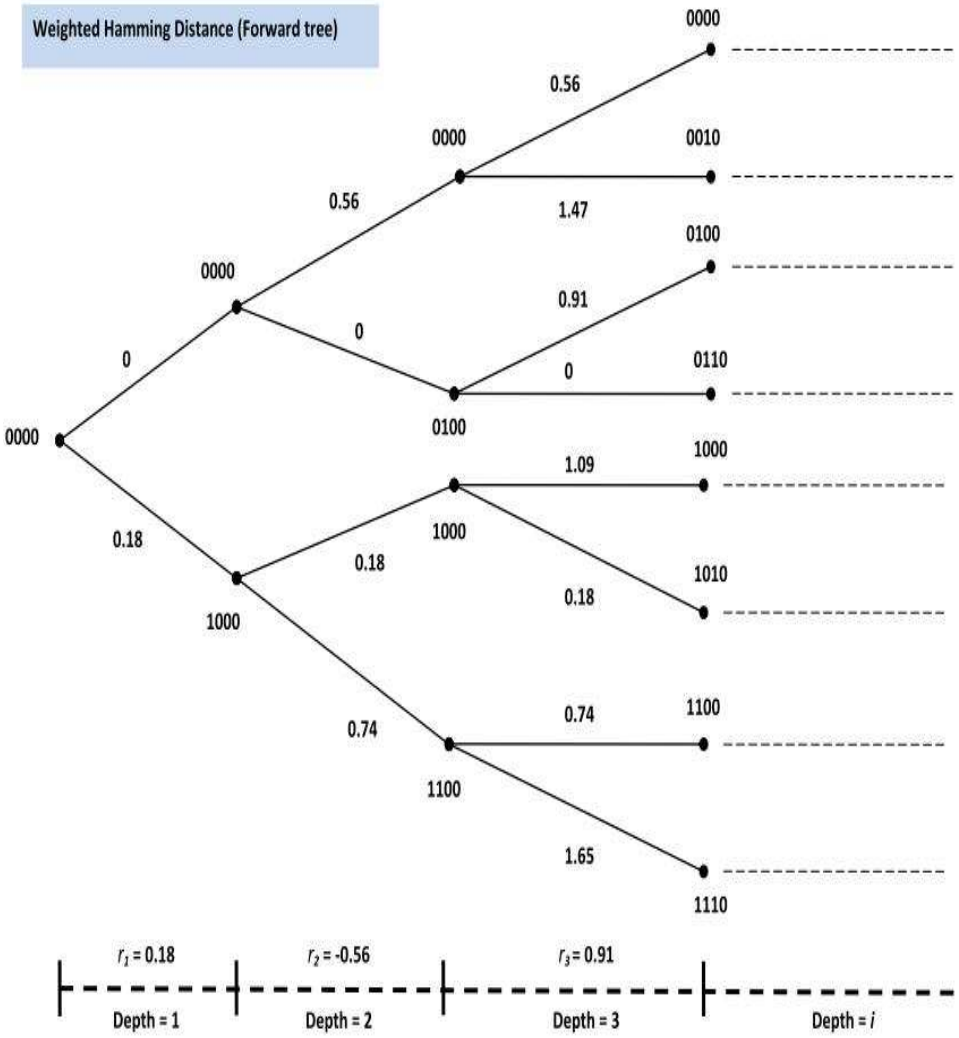

Figure 2.4: Metric weight calculation

Such a smallest path goes from $\xi_< \longrightarrow \xi_>$. Clearly there exist an intermediate node ξ. Hence it is much easier to split the search of the path with minimum Hamming weight into two searches $\xi_< \longrightarrow \xi$ and $\xi \longrightarrow \xi_>$. Due to this fact, the forward and backward search is conducted by using forward and backward tree. Therefore,

the BEAST performs the following steps.

1. Determine the threshold T.
2. Perform forward and backward search of paths.
3. Comparison of search results.

The detailed explanation of these steps is as follows.

Determine the threshold T

In BEAST, the threshold value is a very important value influencing the search complexity of the design. A good threshold value can reduce the computational time of the design as well as memory for storing search result.

If the threshold value is chosen too small, the trees grow too small, therefore, making the system slower due to multiple search iterations. In contrast to that, if the threshold value is chosen too large, the trees grow too large and most of the nodes will be calculated. Therefore, arising memory issues for the design.

Threshold computation starts, once the complete received sequence $\mathbf{r}$ is obtained from the channel. The first step for computing the threshold is to compute the threshold increments. For BPSK modulation, the threshold increments are determined as the sorted absolute values δ_i of received symbols.

By using these sorted absolute values of received symbols, the first threshold value T_1 is computed and is given by.

$$\mathrm{T}_1 = \sum_{i=1}^{d} \delta_i \tag{2.9}$$

where $d = \lceil d_{min}/2 \rceil$. The $d_{min} = 4$, represents the minimum Hamming distance for (8, 4, 4) extended Hamming block code. The remaining threshold value will be determined as,

$$\mathrm{T}_i = \mathrm{T}_{i-1} + \delta_{i+1} \tag{2.10}$$

where i=2, 3, ..., N. The forward threshold $\mathrm{T_F}$ and the backward threshold $\mathrm{T_B}$ are computed. $\mathrm{T_F}$ and $\mathrm{T_B}$ will be used in search of paths, using the forward and the backward tree. There are two possible ways for updating and incrementing $\mathrm{T_F}$ and $\mathrm{T_B}$.

The first method uses the computed threshold values and divide them in half, that is, $\mathrm{T_F}$=$\mathrm{T_B}$=T/2. The second method[1] is a more sophisticated method, in which, only the smaller of the two trees is extended in each iteration, using $\mathrm{T_F}$ and

T_B. Set T_F=T_B=T/2 for the first iteration, observe the search results and keep a track of the number of nodes found in search. If the number of nodes obtained in forward search are more than of the number of nodes obtained in backward search, T_B will be incremented by δ_i while T_F remains the same and vise versa.

Table 2.2 compares the merits and demerits of two methods used by the BEAST for updating T_F and T_B.

Table 2.2: Comparison of methods for updating T_F and T_B.

Method	**1**	**2**
Functionality	T_F=T_B=T/2, for all iterations	Iteration 1: T_F=T_B=T/2, for other iterations, δ_i will be added to T_F and T_B, depending on the search results.
Merits	Simplified architecture for designing threshold unit in hardware as a divider or a shifter used to shift the data by factor of 2 is required to compute T_F and T_B.	Reduced computational time of design since the forward and backward trees will not be extended again every time with new threshold value.
Demerits	Increased computational time as forward and backward trees will be extended again and again every time with new threshold value.	The method will have a complex logic for hardware implementation of threshold unit and hence, will utilize more hardware.

Consider the previously defined received sequence $\mathbf{r}$ which is fed into the block decoder for computing the threshold values, T_F and T_B, respectively.

$$\mathbf{r} = \begin{pmatrix} 0.18 & -0.56 & 0.91 & 0.60 & -0.02 & 1.60 & 2.80 & 0.34 \end{pmatrix}$$

First, take the absolute value of each received symbol r_i=0, 1, 2, ..., N-1, and denote the corresponding sequence by $\mathbf{r}_{abs}$, shown as

$$\mathbf{r}_{abs} = \begin{pmatrix} 0.18 & 0.56 & 0.91 & 0.60 & 0.02 & 1.60 & 2.80 & 0.34 \end{pmatrix}$$

The $\mathbf{r}_{abs}$ is sorted in ascending order and sorted values are represented by δ

$$\delta = \begin{pmatrix} 0.02 & 0.18 & 0.34 & 0.56 & 0.60 & 0.91 & 1.60 & 2.80 \end{pmatrix}$$

Using δ, the threshold value is computed. The first threshold value T_1 is computed by using equation (2.9)

$$T_1 = 0.02 + 0.18 = 0.20 \tag{2.11}$$

To grow the trees, T_F and T_B will be computed. If we are using the first method for updating T_F and T_B then, T_F will always be equal to T_B, that is, $T/2$. Considering the initial threshold T_1, forward threshold T_F and backward threshold T_B will be 0.1.

Perform forward and backward search

After deciding T_F and T_B, the next step is to determine the nodes in the forward and backward tree, satisfying the predefined search criteria.

The forward search determines all nodes in the forward tree that satisfy

$$\mathcal{F} = \{\xi \big| w_<(\xi) \geq T_F, w_<(\xi^p) < T_F\} \tag{2.12}$$

that is, $\mathcal{F}$ is a set containing all nodes ξ in forward tree whose parent node ξ^p weight is less than T_F and their own weight is greater than or equal to T_F. The backward search similarly provides all nodes in backward tree satisfying

$$\mathcal{B} = \{\xi \big| w_>(\xi) \leq T_B, w_>(\xi^c) > T_B\} \tag{2.13}$$

where ξ^c is any of the child node of ξ in the backward tree containing weight greater than T_B and their parent node weight is less than or equal to T_B.

Comparison of search results

The nodes obtained in result of forward and backward search are compared with each other in order to find a common node ξ whose combine weight ($w_{\text{comb}}(\xi)$) satisfies.

$$w_{\text{comb}}(\xi) \leq T_F + T_B \tag{2.14}$$

where $w_{\text{comb}}(\xi) = w_>(\xi) + w_<(\xi)$. If more than one node satisfying equation (2.14) are found, the node with smallest metric weight is chosen and its corresponding path is the ML codeword decision. If no such node exists, the threshold has to be incremented and the search and comparison has to be repeated with increased values for T_F and T_B.

2.3.2 Detailed example

Consider the previously defined information sequence $\mathbf{r}$, the threshold increments δ and the code trees illustrated in Figure 2.2 and Figure 2.3. Consider the first method for calculation of the threshold T, forward threshold T_F and backward threshold T_B. By using the initial threshold T_1, the T_F will be equal to T_B, that

is, $T_1/2$. The forward and backward trees are extended. The current state nodes with respected Hamming weights and depth levels are identified using the criteria defined by equation (2.12) and equation (2.13). The results of the forward search are.

$$\mathcal{F}_\sigma = \begin{pmatrix}1000 & 0000 & 0100 & 0110\end{pmatrix}$$
$$\mathcal{F}_w = \begin{pmatrix}0.18 & 0.56 & 0.91 & 0.60\end{pmatrix}$$
$$\mathcal{F}_l = \begin{pmatrix}1 & 2 & 3 & 4\end{pmatrix}$$

Similarly consider the backward search results as

$$\mathcal{B}_\sigma = \begin{pmatrix}0000 & 0000 & 0000 & 0000 & 0000 & 0000 & 0000\end{pmatrix}$$
$$\mathcal{B}_w = \begin{pmatrix}0 & 0 & 0 & 0 & 0.02 & 0.02 & 0.02\end{pmatrix}$$
$$\mathcal{B}_l = \begin{pmatrix}0 & 1 & 2 & 3 & 4 & 5 & 6\end{pmatrix}$$

where $\mathcal{F}_l$ and $\mathcal{B}_l$, respectively specifies the level (depth) of the nodes in the forward and backward tree and $\mathcal{F}_w$ and $\mathcal{B}_w$ represents their weights in forward and backward tree satisfying the search criteria. $\mathcal{F}_\sigma$ and $\mathcal{B}_\sigma$ are the sets containing the node states which were found by search mechanism. Comparing the sets at each level gives the matching or common nodes in both trees. After completing the first iteration, the obtained results are specified in the Table 2.3, to show the comparison of obtained node states regarding each level of the forward and backward tree. The common node is found at second level of the forward tree, that is, 0000. The

Table 2.3: Node states comparison, T_F=T_B=0.1.

Level/Depth	0	1	2	3	4	5	6	7	8
Forward nodes	-	1000	0000	0100	0110	-	-	-	-
Backward nodes	-	-	0000	0000	0000	0000	0000	0000	0000

next step is to compare the combined weight of the common node $w_{\text{comb}}(\xi)$ according to equation (2.14). The $w_{\text{comb}}(\xi)$ is 0.56+0.02=0.58 which in fact is greater than T_F+T_B=0.2. Since the equation (2.14) does not hold which means that the threshold T_2 will be computed where T_F and T_B will be updated accordingly. The forward and backward search has to be re-initiated with new threshold values.

The threshold T_2=0.54 is computed using equation (2.10), T_F and T_B will be 0.27. With the new threshold, we do not have to start extending the trees from scratch as we can just extend the trees, reusing previous results. The backward

search results show no difference as compared with the previous results. If we are using method two for incrementing T_F and T_B then, T_B will be the same as previous. The numbers of backward state nodes are greater than forward state nodes. In that case, T_F will be incremented to $0.1 + \delta_3$. The initiated search with updated T_F and T_B produces the following results.

$$\mathcal{F}_\sigma = \begin{pmatrix}0000 & 1100 & 0100 & 1000 & 0110 & 1010 & 1011\end{pmatrix}$$

$$\mathcal{F}_w = \begin{pmatrix}0.56 & 0.74 & 0.91 & 1.09 & 0.60 & 1.78 & 3\end{pmatrix}$$

$$\mathcal{F}_l = \begin{pmatrix}2 & 2 & 3 & 3 & 4 & 6 & 7\end{pmatrix}$$

$$\mathcal{B}_\sigma = \begin{pmatrix}0000 & 0000 & 0000 & 0000 & 0000 & 0000 & 0000\end{pmatrix}$$

$$\mathcal{B}_w = \begin{pmatrix}0 & 0 & 0 & 0 & 0.02 & 0.02 & 0.02\end{pmatrix}$$

$$\mathcal{B}_l = \begin{pmatrix}0 & 1 & 2 & 3 & 4 & 5 & 6\end{pmatrix}$$

The node state comparison for matching nodes is shown in Table 2.4. A matching node is found at second level, that is, 0000. Comparing the new search results yields no difference as compared with the previous results. The combined weight $w_{\text{comb}}(\xi)$=0.56+0.02=0.58 is not less than or equal to T_F+T_B=0.27+0.27=0.54. Repeating the same steps by incrementing the threshold T_3=T_2+0.56=1.10. T_F

Table 2.4: Node states comparison, $T_F = T_B = 0.27$.

Level/Depth	0	1	2	3	4	5	6	7	8
Forward nodes	-	-	0000 1100	0100 1000	0110	-	1010	1011	-
Backward nodes	-	-	0000	0000	0000	0000	0000	0000	0000

and T_B will be 0.55. The forward search results are same as obtained by T_F=0.27. The forward and backward tree expansion for T_F=0.27, 0.55 and T_B=0.55 is shown in Figure 2.5 and Figure 2.6 respectively. The obtained search results are shown as following

$$\mathcal{F}_\sigma = \begin{pmatrix}0000 & 1100 & 0100 & 1000 & 0110 & 1010 & 1011\end{pmatrix}$$

$$\mathcal{F}_w = \begin{pmatrix}0.56 & 0.74 & 0.91 & 1.09 & 0.60 & 1.78 & 3\end{pmatrix}$$

$$\mathcal{F}_l = \begin{pmatrix}2 & 2 & 3 & 3 & 4 & 6 & 7\end{pmatrix}$$

$$\mathcal{B}_\sigma = \begin{pmatrix}0000 & 0000 & 0001 & 0000 & 0101 & 0000 & 0101 & 0000 & 0101 & 0000\\ 0101 & 0000 & & & & & & & & \end{pmatrix}$$

$$\mathcal{B}_w = \begin{pmatrix}0 & 0 & 0.34 & 0 & 0.34 & 0 & 0.34 & 0.02 & 0.34 & 0.02\\ 0.34 & 0.02 & & & & & & & & \end{pmatrix}$$

$$\mathcal{B}_l = \begin{pmatrix} 0 & 1 & 1 & 2 & 2 & 3 & 3 & 4 & 4 & 5 \\ 5 & 6 & & & & & & & & \end{pmatrix}$$

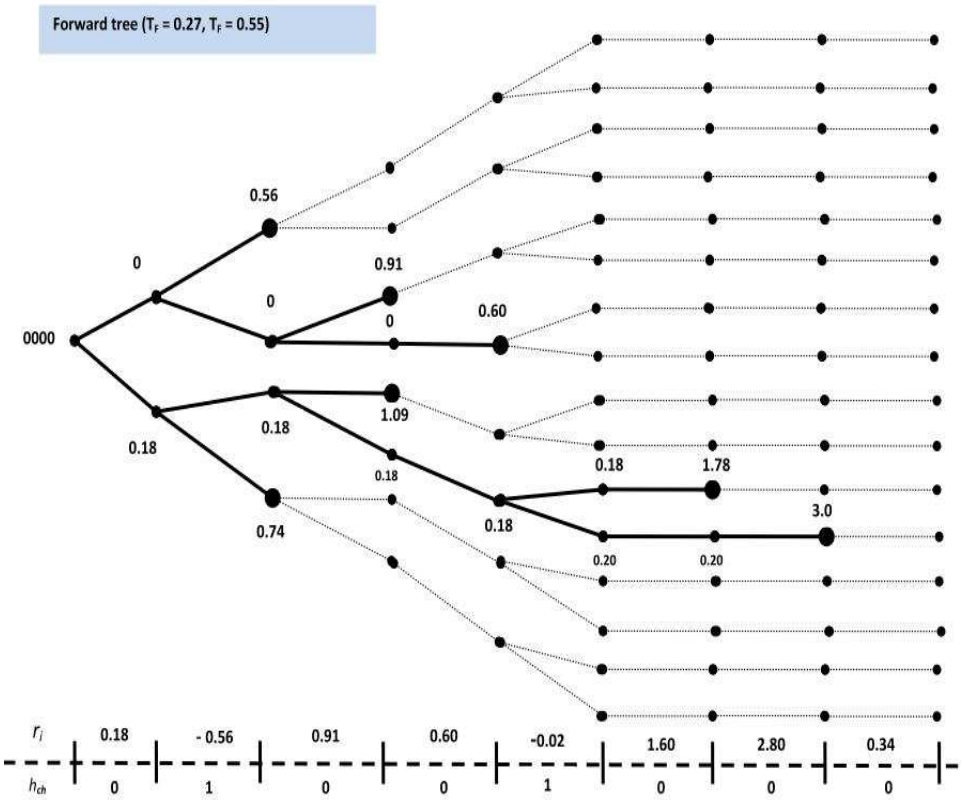

Figure 2.5: Forward tree search

Table 2.5: Node states comparison, $T_F = T_B = 0.55$.

Level/Depth	0	1	2	3	4	5	6	7	8
Forward nodes	-	-	0000 1100	0100 1000	0110	-	1010	1011	-
Backward nodes	-	-	0000	0000 0101	0000 0101	0000 0101	0000 0101	0000 0001	0000

Comparing the combined weight $w_{\text{comb}}(\xi)$=0.56+0.02=0.58, obtained from common node 0000 found at depth two as identified in Table 2.5, shows that $w_{\text{comb}}(\xi)$

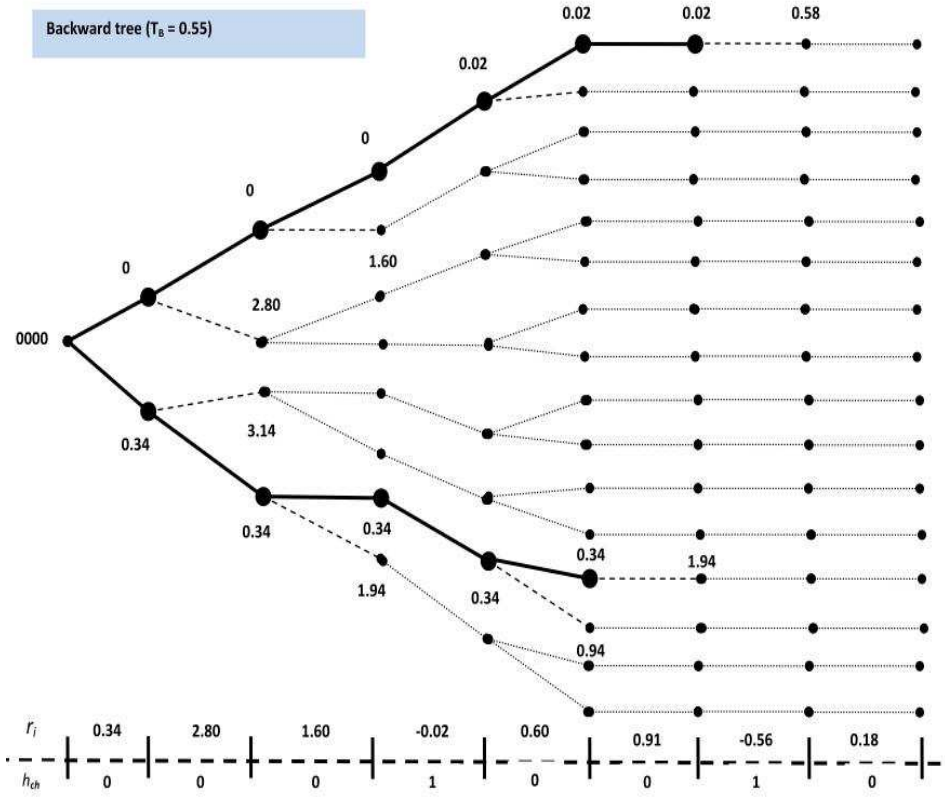

Figure 2.6: Backward tree search

is less than T_F+T_B=0.55+0.55=1.10, satisfying equation (2.14). The path containing the common node is the required ML path. Final step is to trace out all the forward nodes linked to the common node till root node in forward tree and all the backward nodes linked from the common node for determination of the codeword. The branch labels of the corresponding path determine the ML codeword decision as 00000000.

Chapter 3

Architecture

3.1 Overview

The era of reconfigurable hardware structures gives hardware designers the liberty to design and test a complete system in an efficient manner. FPGA is a reconfigurable structure, used to provide flexible and lesser time to market designs. FPGA based platforms allow the designers to have a complete control over hardware design. This chapter mainly focuses on description of the BEAST architecture, covering from conceptual design to physical visualization of the design.

VHDL is a powerful tool used in order to provide the hardware description of the BEAST. The first step toward implementation was to observe the functional requirements and develop a behavioral model. For this purpose, a complete design is implemented in MATLAB, to check the functionality of the BEAST. After verifying the functionality, the next step was to implement the design. The block diagram describing the architecture of BEAST in hardware is shown in Figure 3.1. The architecture is divided in four main parts.

1. Input Unit.
2. Search/Select logic.
3. Compare/Output logic.
4. Control unit.

The Input Unit of the design provides a foundation for rest of the design units to work according to the BEAST. The main responsibility of this unit includes, input data reception, data storage and processing, generation of forward and backward trees and threshold calculation. The first step toward implementing the Input Unit

was the decision of an appropriate input word length, to represent input samples of received sequence **r**. A reasonable input word length is used to cover the desired data range for representing the samples. The input word length also provides an indication for overall data storage and manipulation capacity of the hardware. The basic implementation is done by using the data word length of twelve bits (covering six bits each for the integer part and fractional data), in order to represent the received data samples. The word lengths more than twelve bits may cause an increase in hardware while considering the implementation of (8, 4, 4) extended Hamming block code. On the other hand, if smaller data word length is considered then precision and range of data and result is affected, which leads to the incorrect results.

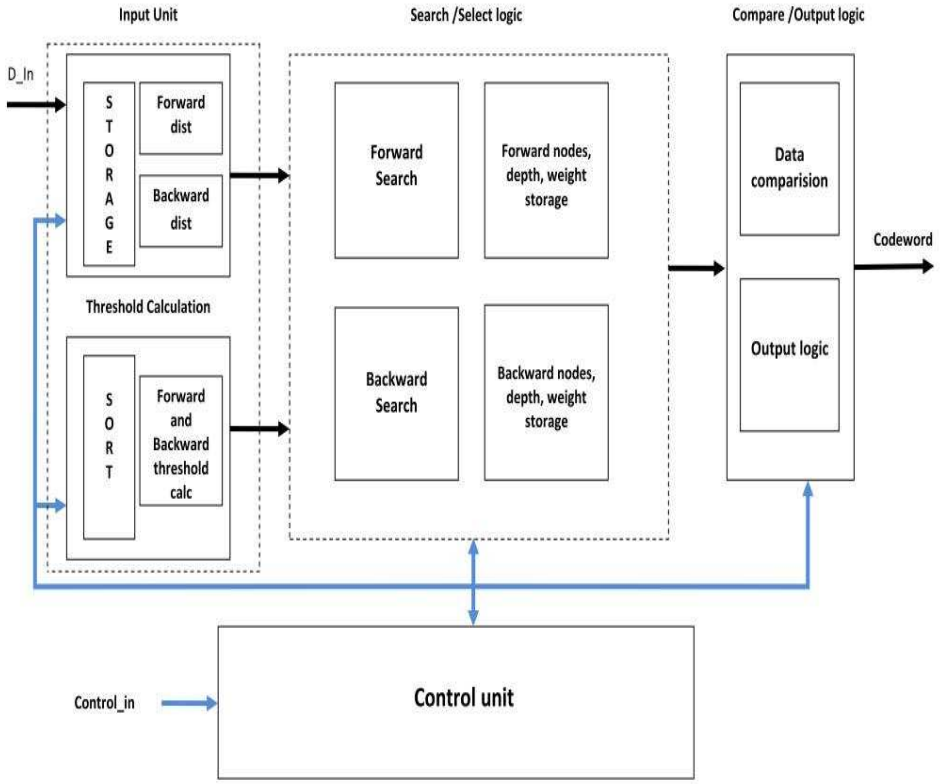

Figure 3.1: Basic architecture of the BEAST

The data sequence is received by Input Unit using the D_In interface. The absolute values $\mathbf{r}_{abs}$, of the received data samples is stored in predefined storage units.

Meanwhile, the hard decision sequence h_{ch}, regarding the received sequence is also performed and stored. Afterward, the construction of the forward and the backward tree is initiated. The data sorting process is also initiated, to compute the threshold increments δ. Data can be sorted by using any of the sorting algorithms like Bubble sort, Binary sort and Insertion sort etc. It is observed that sorting process consumes more time, therefore, making the system slower. It is also not possible to perform the complete sorting operation in a single clock cycle.

The loop unrolling process for Bubble sort algorithm is used. In result to that, the chip area is compromised to cover the synchronization problem with rest of the hardware. The T_F and T_B will be computed according to the first method for incrementing the threshold. The architecture also implements the second method for incrementing threshold values, for the purpose of future use. Both methods will be explained later in details.

Search/Select logic, initiates forward and backward search of the trees after reception of T_F and T_B. The sole purpose of establishing a search mechanism is to find out all the nodes, their depth (at level of tree where found nodes exist) with respected weight that satisfies the defined search criteria. These results will later be used by Compare/Output logic unit, to find a matched or a common node fulfilling the set criteria specified in equation (2.14). The common node satisfying the criteria will be used to select the ML path where branch labels of the path will determine the ML codeword decision. Therefore it is required to store all the results so that the results can later be used by other design units. In order to store all the information, different storage RAMs (Random Access Memory) are used.

The search results are provided to Compare/Output logic unit along with T_F and T_B. Compare/Output logic unit will compare all the provided nodes (obtained from forward and backward search by using Search/Select logic unit) at their respected depth levels in order to find the common point of interest. The comparison results will lead in deciding a codeword or a way around helps in re-initiating the Search/Select mechanism with new threshold values.

Synchronization of all the operations inside the design is the core element of the hardware. It helps in maintaining the smooth data flow from a design unit to another without any deadlocks. Synchronization also guarantees that the required tasks which are needed to be performed by the complete design will be done in correct and systematic way. A dedicated control unit is designed in order to fulfill all the set operational requirements of the design. Control unit provides, synchronization of operations inside each design unit and also among all other hardware design units. All the control signals are registered in order to meet the functional as well as timing requirements of complete hardware system.

3.2 Detailed explanation

3.2.1 Input Unit

The Input Unit is used to implement the input logic for the purpose of data reception, construction of forward and backward trees along with Threshold Calculation unit. The main reason of implementing Threshold Calculation unit along with input logic is to save memory space. By observing the Figure 3.1, it is clearly seen that the data stored in input logic storage unit can also be used for threshold calculation, with only addition of parallel data lines, fed toward the Threshold Calculation unit. Input_Samples is the design unit that defines a storage unit for storing the absolute values of the input data samples along with hard decision output. Consider the detailed architecture of Input Unit as shown in Figure 3.2.

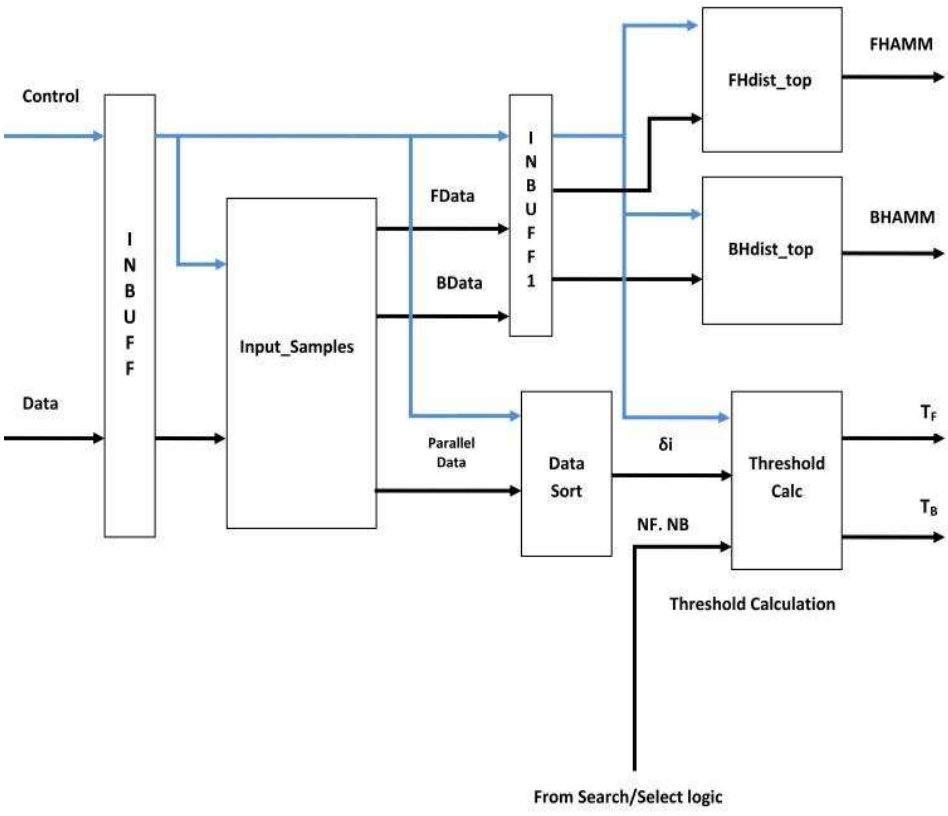

Figure 3.2: Input Unit

The received sequence **r** and control signals (internally generated by Control unit of the design) are fed through a buffer named INBUFF. INBUFF is used for the purpose of synchronization of data and control signals, with internal logic and design units of the Input Unit. Input_Samples unit receives the sequence **r** and performs the hard decision by simply comparing the sign bit of twelve bit received signed data. If sign bit is high then the decision is logic one, and vise versa. At the same time, $\mathbf{r}_{abs}$ is computed and stored in storage RAM.

Storage RAM is defined with dimensions of eight by twelve bits, that is, eight represents the number of locations used to store $\mathbf{r}_{abs}$ and each location is twelve bit wide. The widths and depths of each storage unit along with most of the architecture of the design are adjustable according to the block coding scheme, because, every storage module is generic in implementation. This feature of the design will be beneficial while implementing higher block codes and makes the design more flexible.

The data lines labeled, FData and BData (representing $\mathbf{r}_{abs}$ and h_{ch}) are applied to the FHdist_top and BHdist_top design units, through a buffer named INBUFF1. FHdist_top and BHdist_top are the pre-defined structures according to the defined generator matrix G used in Detail example section. Pre-defined structure means that the implementation of these design units is static and will subject to change as we change the generator matrix. FHdist_top and BHdist_top implements the design logic for construction of forward and backward tree according to the defined generator matrix. Both design units constitutes the basic computational logic along with storage units primarily used for storing the results, such as, weighted Hamming distance, node states and their respected levels. Consider the basic architecture used for the computational logic design of FHdist_top and BHdist_top in Figure 3.3.

The basic architecture is divided into three main sections.

1. FDU (Frame Distribution Unit).
2. ASL (Add Select Logic).
3. HFSU (Hamming Frame Selection Unit).

The FDU defines a process that is responsible for distribution of the provided input in a systematic way such that ASL unit can compute the branch metric weights for the next state nodes. The ASL unit receives the hard decision sequence h_{ch} along with absolute data samples (referred as FData or BData in Figure 3.2 and Figure 3.3) and the branch metric weights of the current state nodes from previous depth level of the tree, that is, the weight of all active nodes in previous iteration. In response to the provided inputs, ASL will compute the branch metric weights of

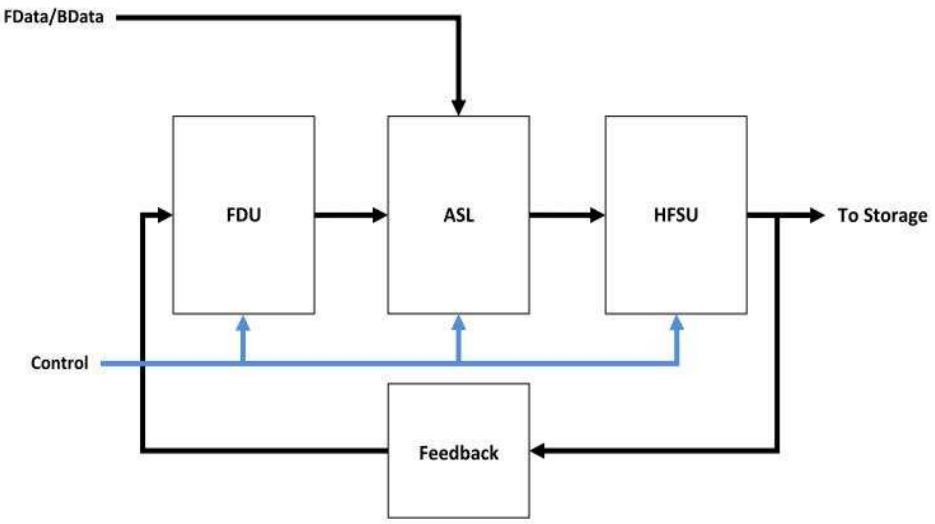

Figure 3.3: Computational logic design of FHdist_top and BHdist_top

next state nodes at current depth of the tree. The HFSU will arrange the results produced by ASL in a frame so that it would be easier for the design to store the computed weights of all active nodes at same iteration. The arranged data is also fed back to FDU via a feedback register. HFSU in technical terms is the data selector, managed and controlled by the control unit of the design.

The main computation logic regarding the construction of trees is implemented in ASL. The ASL contains the set of adders and multiplexers with some additional design logic in order to compute branch metric weights according to the provided inputs. Consider the ASL implementation, representing the second level of the forward tree as shown in Figure 3.4. At any depth of the tree, the ratio of the adders to active nodes will be half. For example, if four nodes are active at any depth level of the tree then two adders are required. Similarly, if sixteen nodes are active then eight adders will be required to perform the desired addition operation. HFtmp1 and HFtmp2 represent the parent nodes weight, of the current active nodes. HF1, HF2, HF3 and HF4 represents the weight of the child nodes, calculated with the help of h_{ch} and r_{abs} along with parent node weights.

For illustration of the concept, let us consider $r_{abs} = 0.56$, $h_{ch} = 1$, HFtmp1 and HFtmp2 be, 0 and 0.18 respectively. The next state nodes are 0000, 0100, 1000 and 1100, as shown in Figure 2.4. To start with, r_{abs} is added with HFtmp1 and HFtmp2, such that, if h_{ch} symbol is not equal to the branch code symbol then,

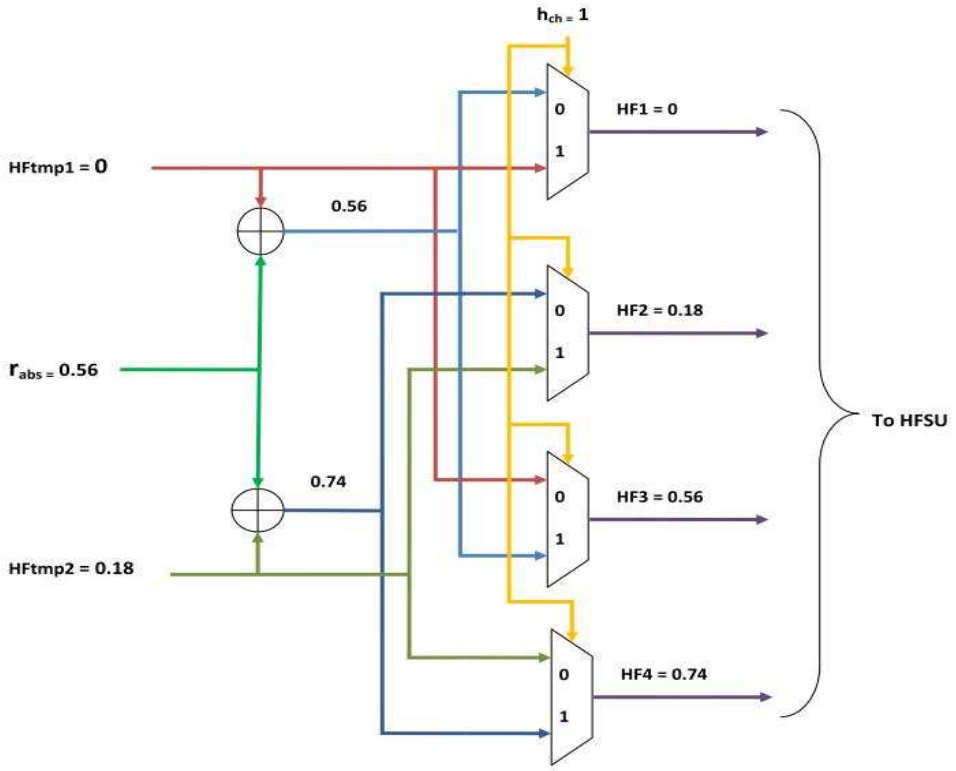

Figure 3.4: ASL structure: forward tree (second level)

the results of the additions will be used as the weighted Hamming distance for the child node. In alternative case, assign the parent node weight to the child node. The multiplexers are used to select the computed weights, which are then applied to HFSU for assignment of these computed weights to their respected child nodes.

The important part of the design is the Threshold Calculation unit inside the Input Unit. In order to initiate the Search and Compare mechanism using Search/Select logic and Compare/Output logic, the threshold values T_F and T_B are required. The first method of incrementing and updating the threshold value is implemented, but on other hand, the second method is also implemented for future use. Consider the basic architecture of Threshold Calculation unit in Figure 3.5.

Absolute data samples from Input_Samples are applied to the Data Sort. Data Sort unit is responsible for sorting the data samples in ascending order, by using

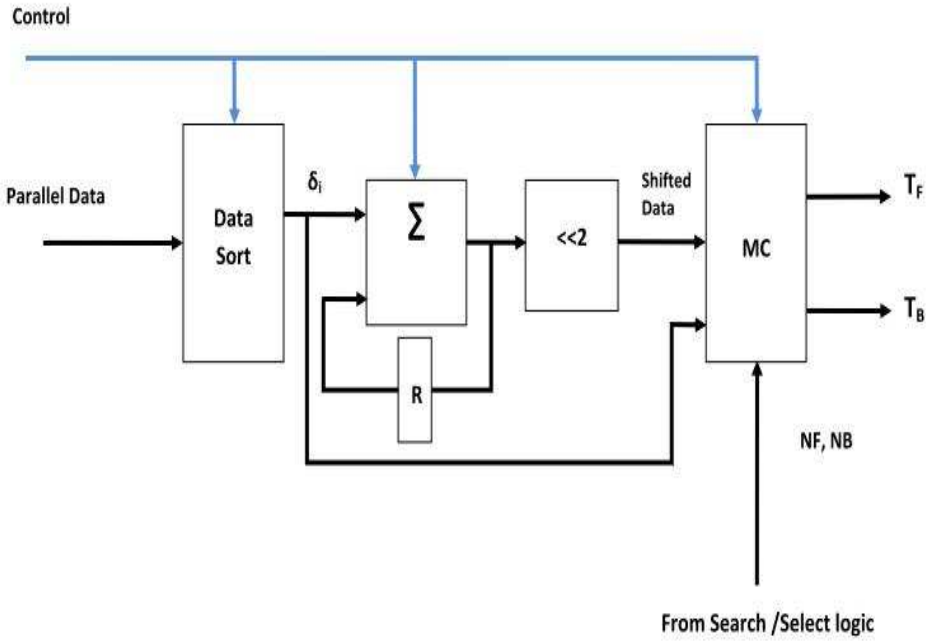

Figure 3.5: Basic architecture of Threshold Calculation unit

loop unrolled bubble sort algorithm. The results of the algorithm are known as δ. The next step is to use the sorted data for calculation of the threshold values. A systematic controlled adder unit along with data feedback is used for computation of the threshold.

The result of the adder is then left shifted by factor of two in order to perform division. The reason for performing this operation is to obtain the direct values for T_F and T_B so that whenever there is a requirement for updating T_F and T_B, then, the data is ready so that any of the value from T_F and T_B can be updated directly according to the provided control signals. MC (Multiplexed Comparator) is used for assigning the shifted threshold value to T_F and T_B. MC also devise a mechanism for updating T_F and T_B using the second method for updating the threshold. T_F and T_B are then applied to Search/Select logic for the search of the desired nodes.

Consider the second method, the difference in design logic for method two is illustrated in Figure 3.6. The decision for incrementing T_F and T_B is dependent upon NF and NB. NF and NB are the number of nodes obtained, during forward

and backward search, using Search/Select logic. NF=NB=1 by default. A comparison structure is established to check the equality condition along with iteration information. If Iteration = 1 and NF is equal to NB then, T_F=T_B=T_{1s}. T_{1s} stands for first shifted threshold, that is, T/2. In next iteration, NF will be compared with

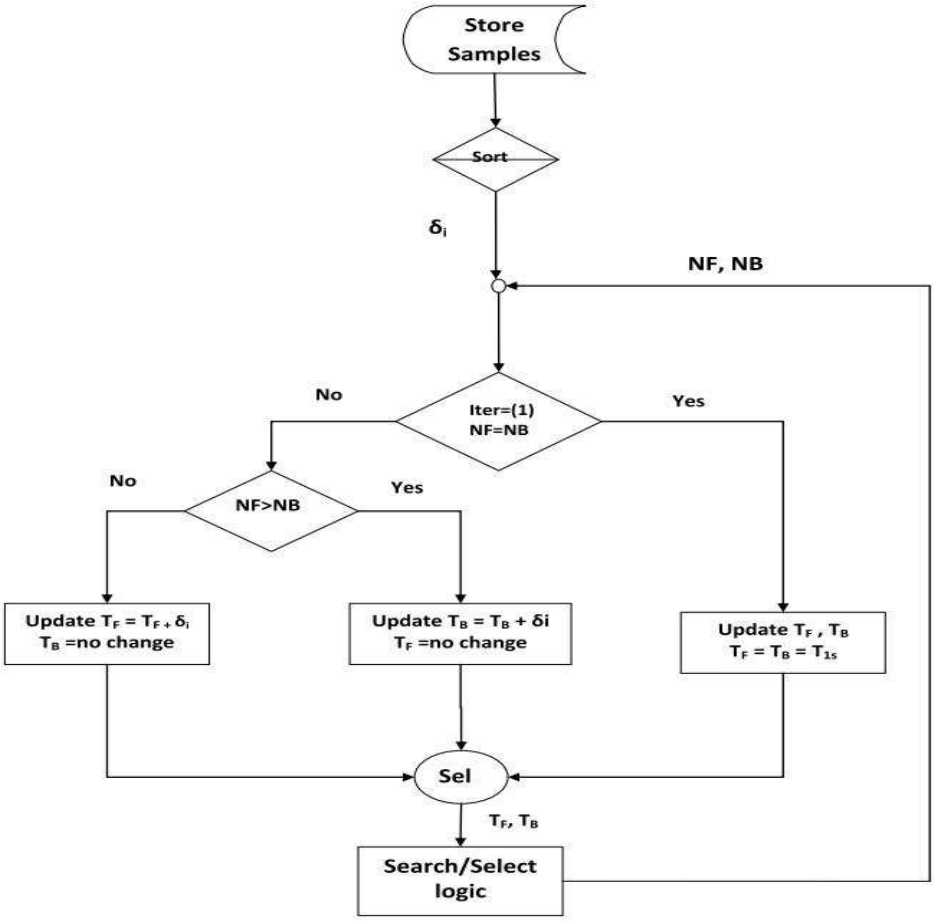

Figure 3.6: Threshold calculation

NB. If NF is greater than NB, T_B will be incremented by δ_i. If NF is less than NB, then T_F will be incremented by δ_i. This complete process is repeated until a valid codeword is found (the required situation after that there is no need for updating T_F and T_B).

3.2.2 Search/Select logic

The main motivation for implementing the Search/Select logic is to perform forward and backward search according to the defined criteria by the BEAST in equation (2.12) and equation (2.13). Consider the generic representation of Search/Select logic unit, shown in Figure 3.7. The representation is identical for both forward and backward search. For forward search, SSL (Search Select Logic) represents the design logic according to the forward search criteria, given in equation (2.12). The SSL unit accepts, T_F along with FHAMM (observe Figure 3.2 for FHAMM and BHAMM) and will select the nodes, their depth level and their corresponding weights. Similarly, for backward search specified by equation (2.13), the SSL unit considers T_B along with BHAMM and will produce the selected nodes with specified depth level and weights. The SSL units also produce the total count of the selected nodes, known as NF and NB. NF and NB will be used in threshold calculation procedure while implementing the second threshold incrementing and updating method. Later on, the produced search results are then stored at their allocated storage spaces.

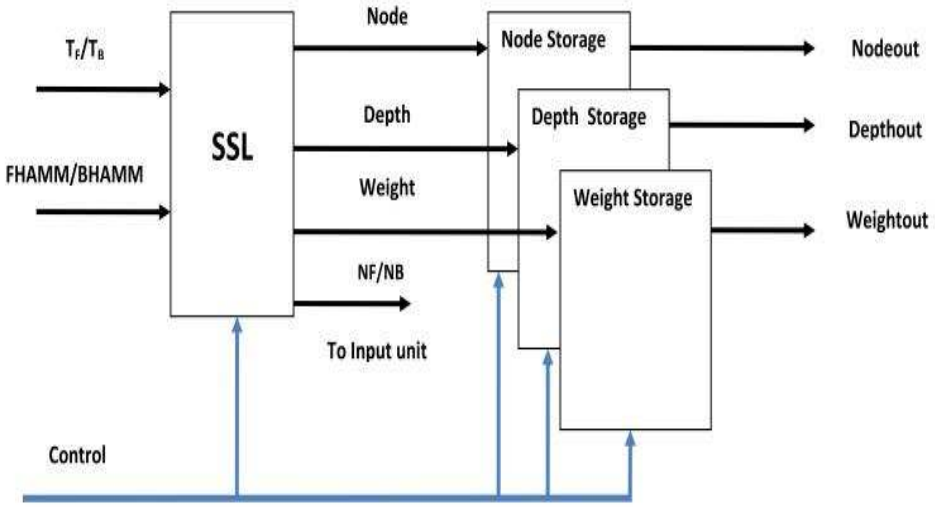

Figure 3.7: Search/Select Logic

Consider the detailed structure definition of SSL in Figure 3.8. The numbers of search units are dependent upon the maximum spread of the tree. The maximum spread refers to the total number of nodes that can be accessed by using a particular coding scheme at any depth of the tree. For example, using (8, 4, 4) extended Hamming code, the maximum spread in tree representation will be $2^{K=4}$ nodes. Therefore, sixteen search units are required to conduct search at any depth level of

the defined tree.

In hardware two different design units are constructed for forward and backward search. If forward search is considered then forward search design units are sixteen times instantiated to cover the maximum spread. Similarly, for backward search, the design units implementing backward search logic are instantiated by sixteen times. The point is, SSL do not mix up both logics (for forward and backward search) together in a single design unit. There will be two structures according to Figure 3.7 working in parallel, in which one will cover the design logic for forward search in SSL where as other SSL unit will implement the backward search logic.

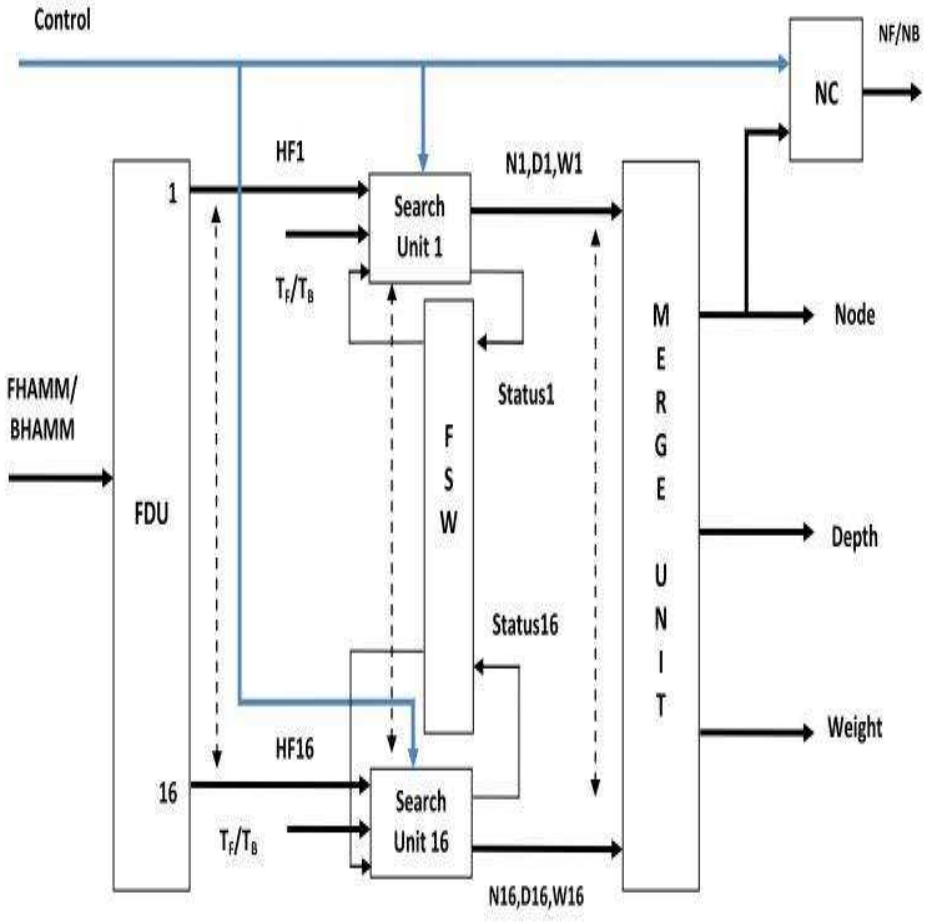

Figure 3.8: SSL Architecture

The frame containing Hamming weight of each node at specific depth (FHAMM, BHAMM from forward and backward tree) is applied to the FDU, so that, FDU will distribute the input frame into sixteen equal parts labeled HF1 to HF16. HF1 represents the weight property of the first node labeled "0000" and HF16 represents the weight property for node "1111".

Each Search unit accepts the input from FDU along with threshold information. T_F will be considered while performing the forward search, using defined SSL unit. Whereas, T_B will be considered for the backward search using another defined SSL unit. Each Search unit will also accept the control signals and Status flags (fed back as the input to all Search units).

Status flags play a very important role, in traversing the complete tree (forward and backward). They tells the design whether to extend the search till next state nodes, or just halt the search at that specific node and upcoming connected nodes with it, after the required search condition specified in equation (2.12) and equation (2.13) is met. Consider the Figure 2.5, for demonstrating the functionality of the Status flags.

Observe that at depth two of the forward tree, there exist the two nodes ("0000" with weight of 0.56 and "1100" with weight of 0.74) satisfying the desired forward search criteria. Therefore, it is required that the rest of the parts of the tree associated with these two nodes should not be traversed. Due to this reason, the concept of status flags is introduced that are managed by FSW (Flag Status Word).

The FSW represents a flags management system that is used to enable or disable any Search unit according to the Status flags. FSW keeps the track of active nodes at any depth of tree. FSW in hardware can be realized as combination of data selection switches, used to enable or disable any search unit.

Each search unit produces an indication for the node, that is, N1 to N16 (single bit each) along with depth levels of tree D1 to D16 (four bits each) and respected weights W1 to W16 (thirteen bits each). Output data containing nodes indication, depths and weights will be arranged by the Merge unit. The output labeled "Node" is a sixteen bit vector, such that, the least significant bit is N1 and most significant bit is N16. The position of the bits will be helpful in identifying the nodes as well as in comparison of the forward and backward nodes in Compare/Output logic unit.

NC (Node Count) is used to count the total number of nodes, found during forward or backward search. It accepts the Node output from Merge unit, as an input and count out the bits equal to logic one in every sixteen bit Node data output until the complete tree is traversed. Those counted bits are then added to give a total

count as NF (Forward Nodes) and NB (Backward nodes). Consider the simplified representation of search unit used for forward search in Figure 3.9.

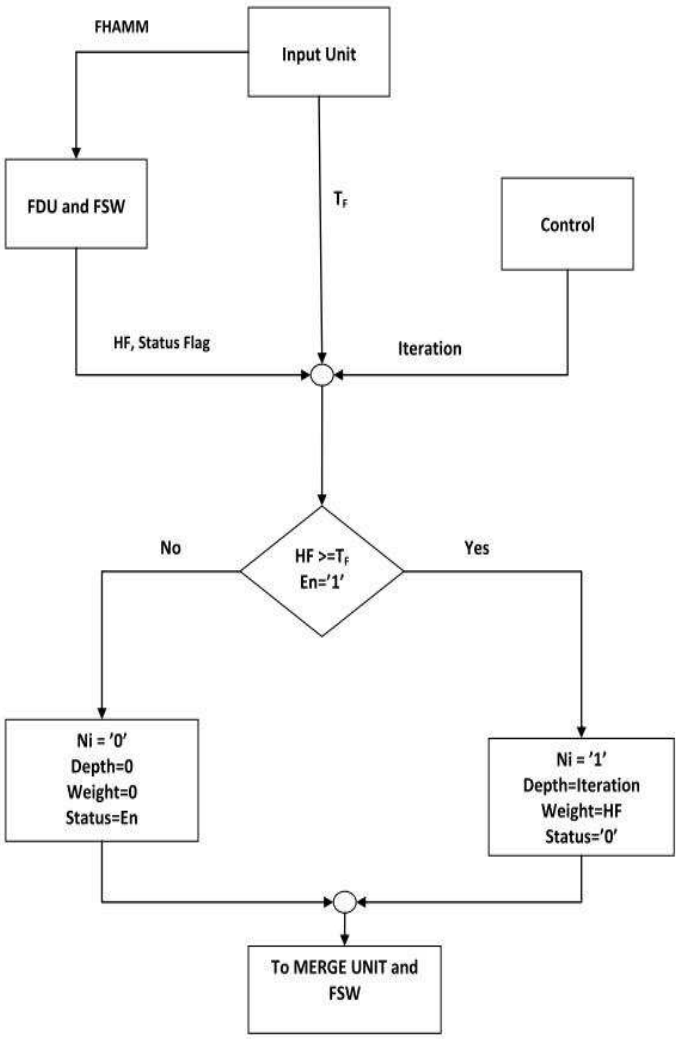

Figure 3.9: Structure of basic Search unit

FDU in forward search unit accepts FHAMM as an input and distribute the data into HF1 to HF16, by equal proportions. At the same time FSW will set the Status

flags and map them with En inputs of the Search units. The weight input is being compared with threshold T_F and En input is also observed. If the required conditions are met then Node output Ni will be set to logic one otherwise it will be set to zero, Depth output will be assigned the iteration number coming from Control unit and weight output will be assigned to HF.

Status flag will be cleared, to indicate that, rest of the parts of the tree associated with that node will not be compared with required criteria. On the other hand, if any of the specified condition does not met then all the output will be assigned to their default values whereas the En input is assigned to the Status flag.

Similar kind of implementation can be seen for the backward search logic, except that, the comparison condition and mechanism for usage of the status flags will be reversed. The results of the search (forward and backward) are stored in allocated storage areas.

Observing the structure of Search/Select logic unit, it can be concluded that the Search/Select logic unit is more area intensive, in terms of storage. The stored results regarding found forward and backward nodes will then be provided to Compare/Output logic unit, for final comparison.

3.2.3 Compare/Output logic

The Compare/Output logic unit is mainly divided into two sections.

1. Compare logic.
2. Output logic.

Compare logic

The crucial point for the codeword determination is to conduct a comprehensive comparison mechanism to find a common node. The common node will lead us toward determination of a valid ML path. According to the equation (2.14), it is required to find a common node in forward and backward tree whose combined weight is less than or equal to the sum of T_F and T_B. Consider the Compare logic implementation in the Figure 3.10.

The Comp_Logic unit will accept T_F, T_B, data from forward search logic (Fnode, FDepth and FWeight) as well as data from backward search logic (Bnode, BDepth and BWeight), generated by Search/Select logic units and find the common node. The Comp_Logic unit is managed by the Control unit, and will produce Node_out, LF_out, LB_out, En and some important Status flags. Node_out represents a common node. LF_out and LB_out represents the depth levels of the

forward and backward tree where the common node is found. En will be used as an input to COMBUFF and will be used to enable or disable COMBUFF. Status flags are used for indicating the completion of comparison mechanism. COMBUFF is a sequential buffer used to link the results obtained via Compare Logic to output pins of the design unit. The Comp_Logic unit is a pipelined implementation and for simplicity, let us assume that at each level of the tree, only four nodes exists, whereas in hardware, the design is implemented to cover the spread of sixteen nodes. The Comp_Logic unit implementation covering four nodes is shown in Figure 3.11.

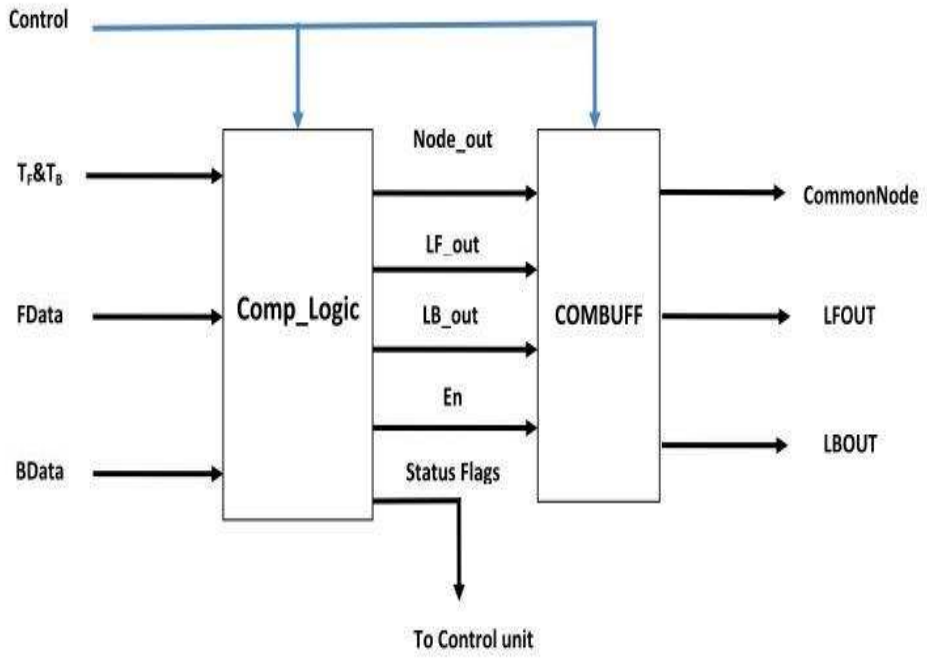

Figure 3.10: Compare logic architecture

FDATA (representing Fnode, FDepth and FWeight) and BDATA (representing Bnode, BDepth and BWeight) from the Search/Select unit is distributed by FDU. COMREG design unit intelligently compares the forward nodes with backward nodes to find a common node, and compares the combined weight of the found nodes with the threshold, according to equation (2.14).

COMPCL are the combinational modules, used to select a common node in that case, where multiple common nodes are satisfying equation (2.14). COMPOP is the design unit with similar functionality as COMPCL, used for selecting a final common node while comparing all the found common nodes in all iterations rather

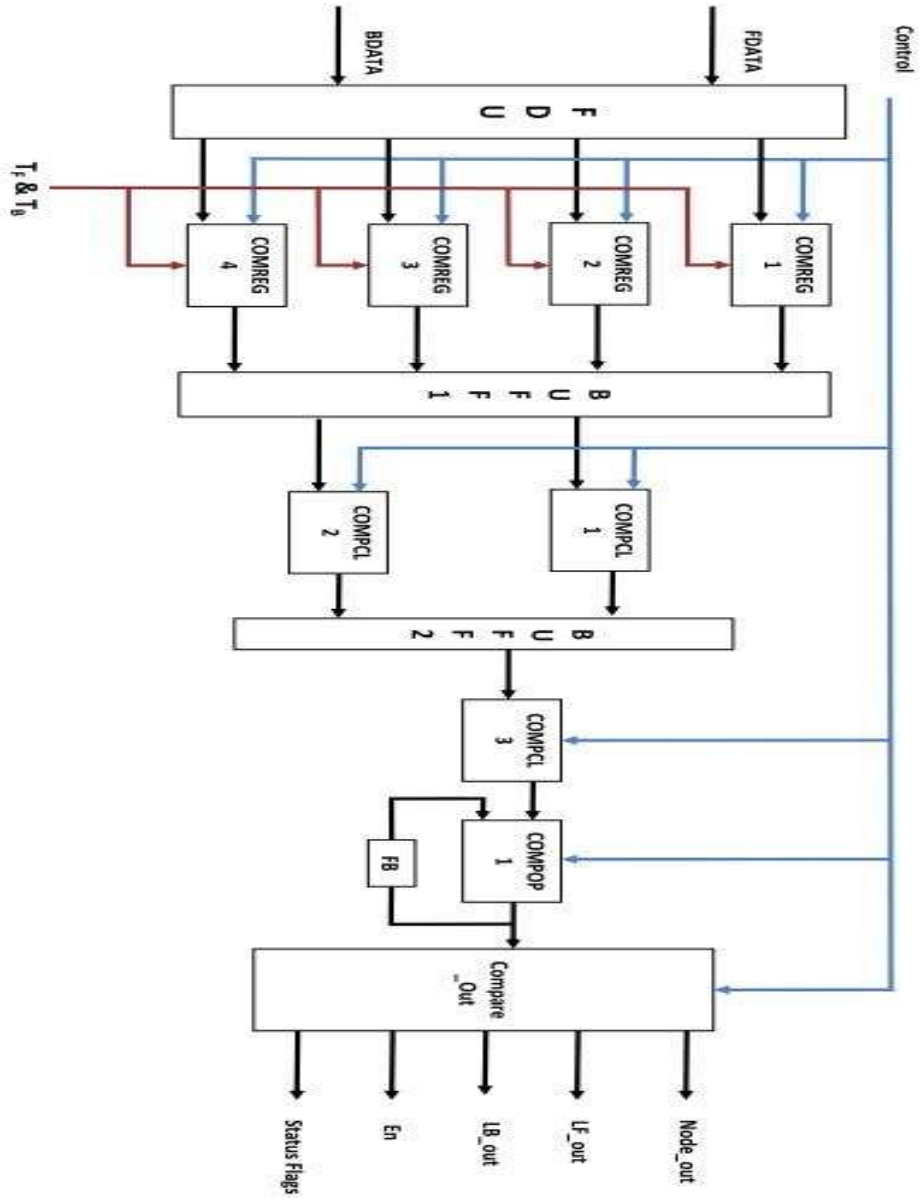

Figure 3.11: Comp_Logic architecture

than in a single iteration. Our main aim for setting the current arrangements of the design units in Comp_Logic unit is that we want to converge the comparison of the nodes to a single common node such that all the nodes within single iteration are compared. At the same time we also want to cover that scenario where multiple common nodes are found at different iteration levels of the design.

The basic architecture of COMREG is shown in Figure 3.12. The number of COMREG instances depends upon the total spread of the tree in terms of nodes. For four nodes, we require four COMREG instances, similarly for sixteen nodes, there will be sixteen COMREG instances. COMREG utilizes two adders for computing sum of T_F and T_B as well as WeightF (weight of the node in forward tree) and WeightB (weight of the node in backward tree). The comparator logic in COMREG consists on set of comparators used to check the desired conditions specified in equation (2.14). The corresponding bits of obtained node vectors during the forward and

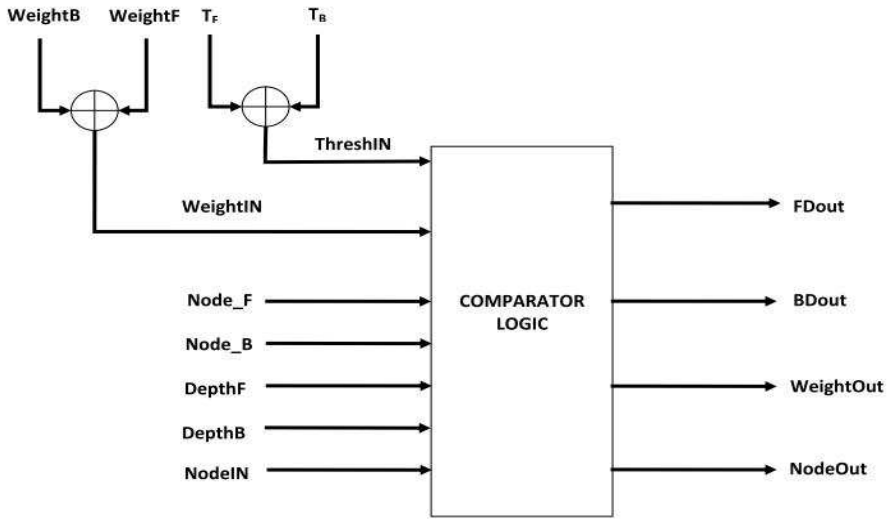

Figure 3.12: COMREG architecture

backward search are compared, to check for the common node. If such node is found then the next step is to check out the combined weight named WeightIN of the node, with sum of T_F and T_B according to equation (2.14). If the desired condition is satisfied, then, we store the node and its corresponding depth levels (DepthF, DepthB for forward and backward tree respectively) for further use by COMPCL units. The design flow for comparator unit in COMREG is illustrated in Figure 3.13, shown as

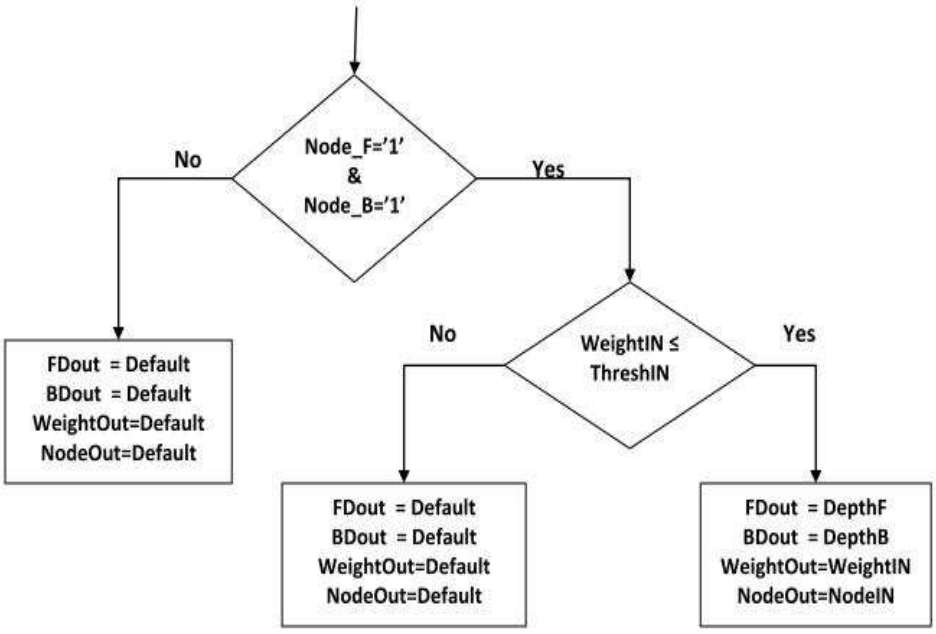

Figure 3.13: Comparator logic in COMREG

The Comparator is arranged to work in such a way that same positioned node bits of the forward and backward node vectors provided by FDU are compared by COMREG unit. The node input bits at each COMREG instance are labeled as Node_F and Node_B. In first step, if Node_F=Node_B='1' then we will proceed to the next step otherwise we will assign the COMREG outputs to their default values. The next step is to compare the nodes according to equation (2.14). This complete process will help us in identifying a common node at single iteration. There is also a possibility that instead of having a single common node, there are multiple common nodes at single iteration level. It is also possible that on other iteration levels, we may also have multiple common nodes.

The arrangement of COMPCL design units are used to compare all the common nodes corresponding to single depth level, in order to find the common node, having minimum target weight among other found common nodes at that depth level. The number of COMPCL instances will be half after each pipelined level until it will reduce to one instance. The first pipelined level had two COMPCL instances as there are only four nodes at each level, compared with the help of four COMREG instances in the scenario shown in Figure 3.11. The next pipelined level will have

only one COMPCL instance. For the spread of sixteen nodes, there will be eight COMPCL instances at first level, four at second level and two at third pipelined level. Ultimately, we are left with one COMPCL instance along with one COMPOP instance to provide the convergence to a single common node with minimum target weight. The design flow for COMPCL is shown in Figure 3.14.

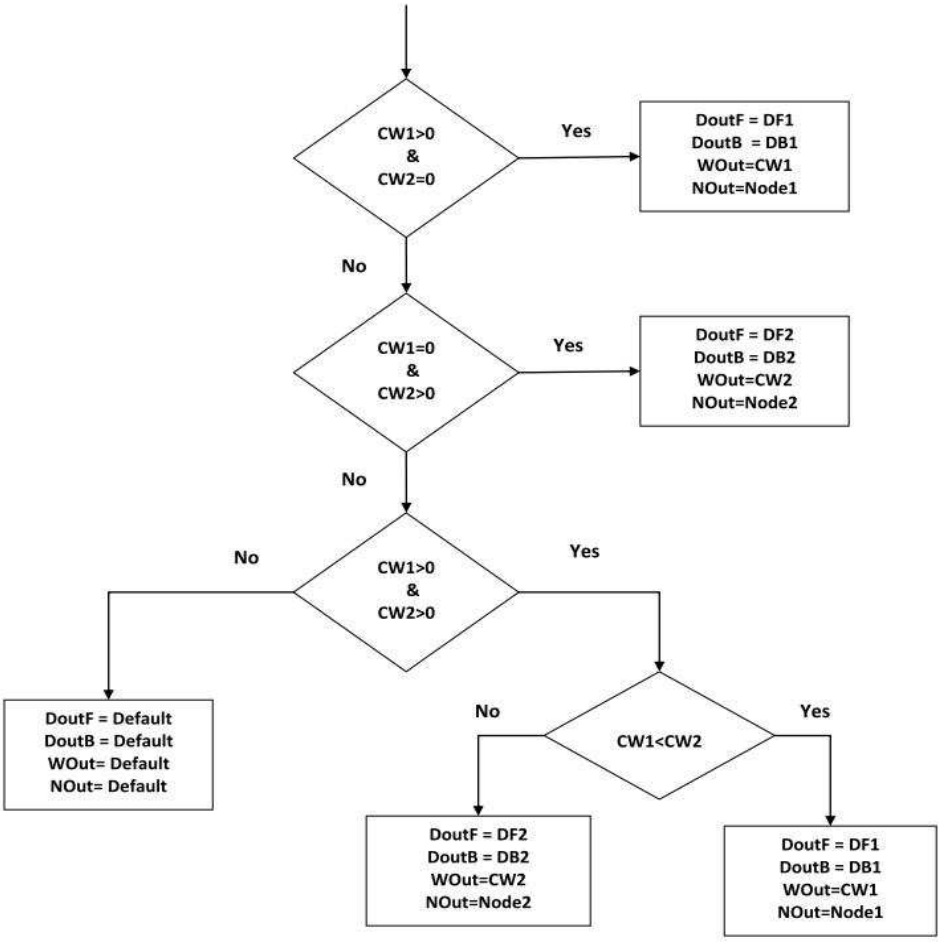

Figure 3.14: COMPCL Logic

Each COMPCL will compare the results of two COMREG units at a specific time. This process will continue until all the nodes are compared at specific depth of tree. Assume that, CW1 and CW2 are the weight output of resultant nodes from COMREG instance one and two respectively. These inputs to COMPCL unit are

compared and outputs of COMPCL unit are set according to specified conditions. The COMPOP also implements a similar strategy for comparing the outcome of the results of COMPCL instance 3, where the output of COMPOP is been feed back as an input by using FB (Feedback register).

Compare_out is a decisive design unit that not only maps the output of Compare unit to COMBUFF but also sets the status flags. Status flags are required by the control unit, in order to decide that the result is acceptable, or there is a need to restart the whole mechanism, from search till comparison and updating threshold values for finding a ML codeword.

Output logic

In case, where a common node is found, the final step is to decide a ML path containing a valid codeword by using the common node along with respected tree depth levels. The Output logic in the BEAST architecture is responsible for deciding a ML codeword path and code symbols are retrieved by using that path. The Output logic architecture implementation is specified in Figure 3.15.

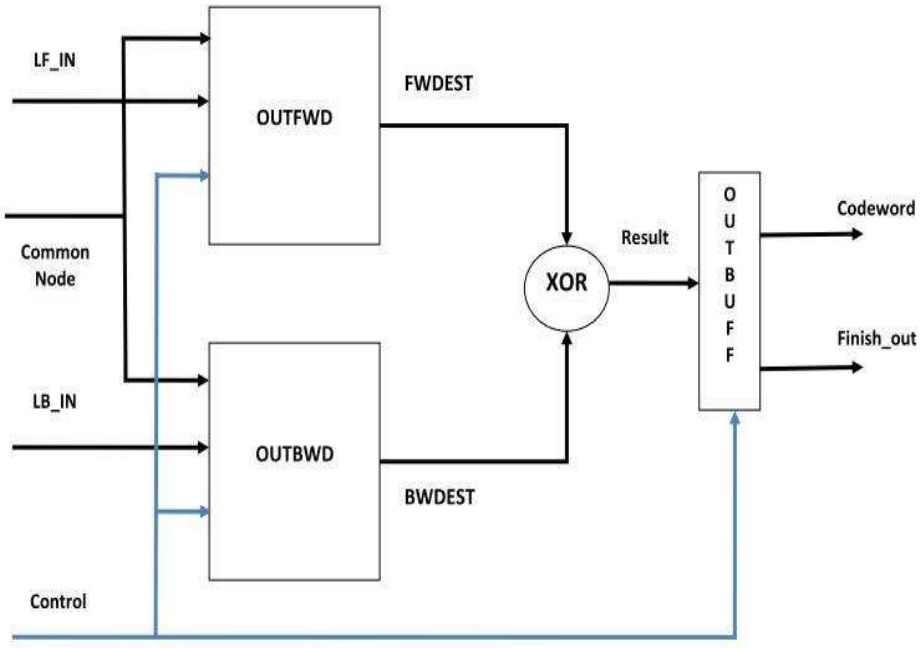

Figure 3.15: Output Logic

The OUTFWD design unit accepts the common node along with LF_IN and determine the code symbols, using the forward tree. Similarly, the OUTBWD produces the code symbols from the backward tree, based on the common node and LB_IN. To combine the obtained code symbols in order to provide a ML codeword, the exclusive OR operation is performed. The FWDEST and BWDEST are eight bits vectors upon which the exclusive OR operation is performed to provide a complete codeword, labeled Result. OUTBUFF defines a process, which is used to map the desired output to the main output interface of the design and also sets the Finish_out flag, for indicating that the decoding process is finished with provided received sequence set.

Consider a very basic example for elaboration of the concept regarding Output logic, where LF_IN = 3, LB_IN = 5 and CommonNode input is "1000". Assume that FWDEST="10100000" and BWDEST="00011011" (FWDEST and BWDEST are set to "00000000" by default). The resultant codeword after an exclusive OR operation will be "10111011". Consider the OUTFWD architecture as shown in Figure 3.16. The Level Decoder unit is used to decode the LF_IN into Nen and Enout, where, Nen and Enout are the eight bit signals. The OUTGEN design unit accepts both of these vectors along with provided common node input labeled CNode and will extract code symbols using forward tree, labeled FWDEST. The OUTGEN unit consists on series of design units named Codesymb_est. The structure of a single Codesymb_est unit is shown in Figure 3.17. The number of Codesymb_est units are dependent upon the total number of code symbols that a codeword contain. Each Codesymb_est unit is mapped with single Nen and ENout bit. The interconnections between two adjacent Codesymb_est units are done in such a way that the Nxtnode output of first module will be connected to Prevnode input of the next module. The main purpose of Nen is to select a common node from Compare logic or a node coming from previous adjacent Codesymb_est unit. The high bit in Nen vector will indicate that the Codesymb_est unit connected to that pin will pass CNode to the next unit.

The bits in En_out at the same time, are used for enabling or disabling OUTCELL design unit on the basis of LF_OUT. The OutSym output of each OUTCELL unit will be mapped with FWDEST output of OUTGEN design unit. Suppose that LF_IN = 3, then Nen(3) will be set to high so that the common node generated by the Compare logic unit is used by Codesymb_est unit connected with that pin. The rest of the other bits of Nen vector will be zero. The En_out vector will also set to "00001111" for enabling the design, to traverse from LF_IN till starting 0000 node. OUTCELL units contains simple case statements according to the generator matrix G. A similar logic is implemented in OUTBWD unit, but capitalizing on backward tree rather than forward tree.

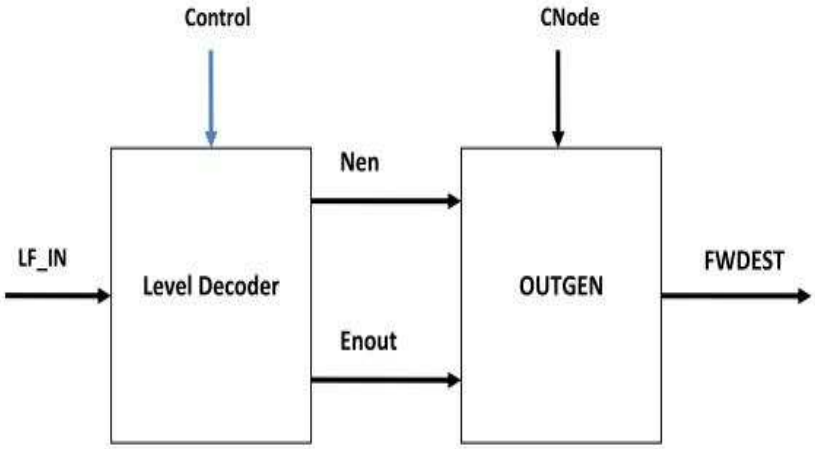

Figure 3.16: OUTFWD architecture

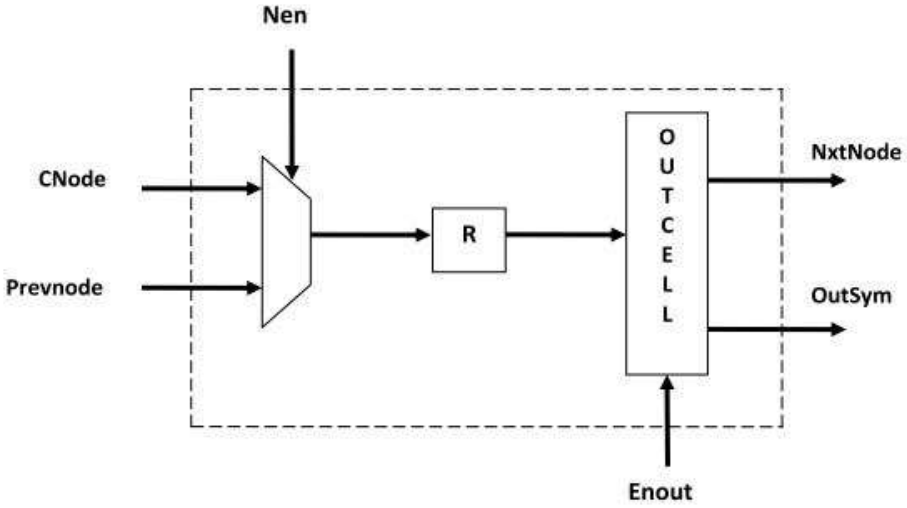

Figure 3.17: CodeSymb_est cell structure

3.2.4 Control unit

The core part of hardware that enables the design units to work smoothly without any bottlenecks and deadlocks is the Control unit. Control unit not only synchronizes all the modules inside the design but also ensure that the proper operation will be performed by each individual component of the design while working as a single unit. For achieving this purpose, a comprehensive Control unit is designed and implemented for the BEAST.

The state description of the control unit is shown in Figure 3.18. The complete hardware is assured to work with eight well defined states of operations which are triggered on occurrences of some external and internal events generated by different hardware modules inside the BEAST implementation. The operation performed in each state is briefly described as follows.

Idle State

The Idle state is a default state of the controller. In idle state all the designed units are initiated at their default operation. All the control signals are hold to their default values. The state transition depends upon a trigger of an external event generated by Start input of the design. Start input tell the design whether to start the decoding process or not. If Start input pin is set high then next state will be Load_IN and hardware initiates the decoding process.

Load_IN State

After receiving a trigger from start input pin, the Input Unit starts receiving the data samples, using its input interface. The Control unit's responsibility is to provide addresses for the storage RAMs so that these samples can be stored. In this state EN_InBUFF signal is set to high in order to ensure write operation by Input_Samples unit inside Input Unit for storing the data samples.

The En_InBUFF is tied with Wen(Write enable of the storage RAM). Whenever Wen is set to high then the hardware will store input samples into the RAM. The addresses to the RAM are provided by FADDR and BADDR signals, linked with address lines of RAM units used by Input Unit. Whenever the write operation is completed then Input Unit enables a signal linked to Sample_WriteIN input of the controller. The controller observes the signal and if it is found high then the controller switches to the next state labeled Hdist_IN. At the same time all the other control lines linked with hardware other than Input Unit are kept at their default states.

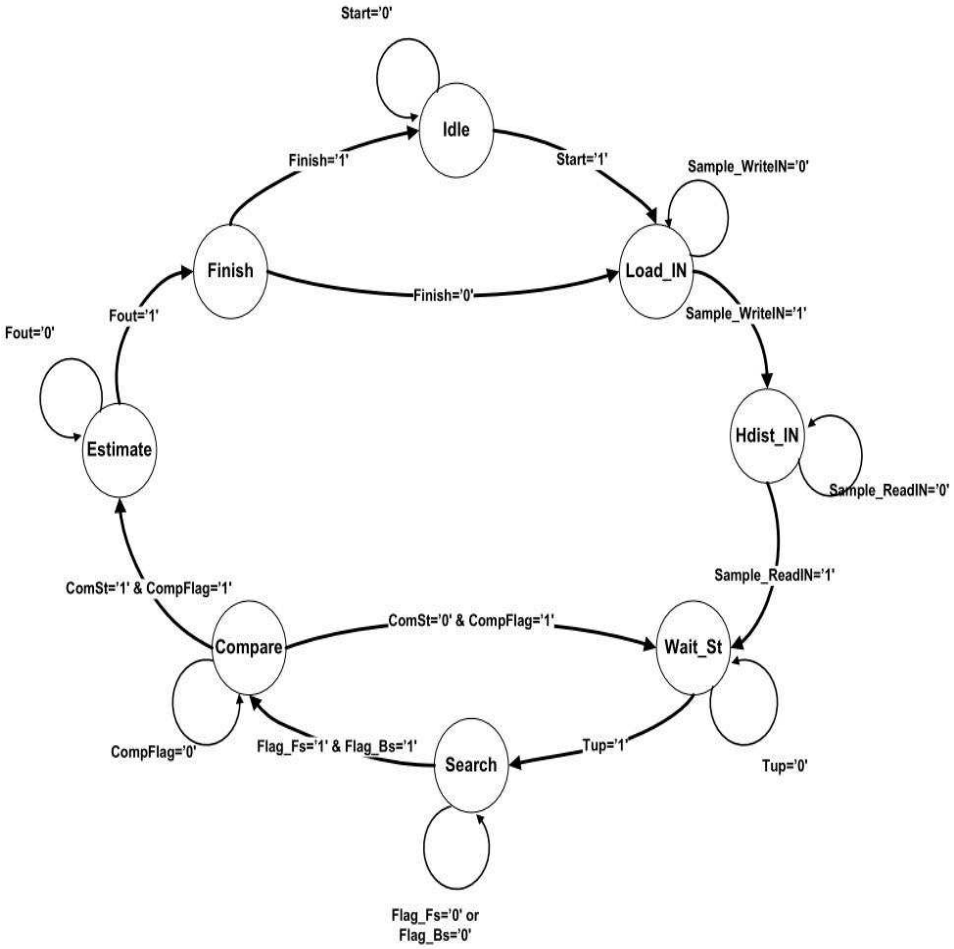

Figure 3.18: Control unit architecture

Hdist_IN State

Hdist_IN as the name suggests, that in this state, the architecture of the BEAST will cover the construction of forward and backward trees and perform calculation of the weighted Hamming distance. For that purpose, it is required by the Control unit to provide the addresses to Input Unit, for the purpose of data reading and set up those control signals, required by the Input Unit to perform read operation.

The En_inBUFF is set to low logic level for indicating that the Input Unit has to perform read operation, while, FADDR and BADDR are incremented to provide

the read addresses. Addr_HIN is the address line used by storage RAM so that the results of operations performed by FHdist_top and BHdist_top can be stored. Addr_HIN is linked with conventional address bus used by Input Unit for providing addresses to FHdist_top and BHdist_top.

At the same time, the Input Unit starts computing the threshold values. Whenever the read operation is finished, the Input Unit sets an indicator flag, linked with Sample_ReadIN input of the controller. When the Sample_ReadIN pin receives logic one, the controller enables the Wait State to wait for the completion of threshold calculation operation, which was started along at that time.

Wait State

The threshold calculation procedure involves data sorting process that take more time for completion than weighted Hamming distance calculation. Without the threshold values, the Search/Select logic design unit is unable to initiate the search mechanism, therefore, it is required to introduce a wait state. Tup is an input signal to the controller, which receives a high (logic 1) to indicate that the threshold values are ready to be used by Search/Select logic. The next state will be Search State.

Search State

In Search state, the controller is responsible of coordinating mainly with Search/Select logic unit to ensure that the search and select operation is performed in systematic way as described by the algorithm. The Search/Select logic design unit establishes the forward and backward search and set the status flags which are mapped with Flag_Fs and Flag_Bs inputs to the controller. Whenever both inputs are found high then controller moves to Compare State of operation.

Compare State

In Compare state, read addresses are provided to the Search/Select logic so that the search operation results stored at respected storage places can be provided to Compare/Output logic design unit. The Compare State plays a very crucial role in design because this is the state that defines the criteria, for whether to go for a ML codeword decision or restart the whole process from search of nodes till comparison and identification of common node.

The controller defined by control unit will accept the status flags generated by Compare/Output logic. ComSt and ComFlag are the two input signals that help the controller to take decision. If both the inputs are set to high (the common node exists) then the next state will be Estimate state. If ComSt is zero and ComFlag is set to one (Key criteria for the common node is not satisfied) then the next state will be Wait State. The new Search will be conducted with updated threshold

values and remaining process will be repeated again and again unless ComSt and ComFlag are set to logic one.

Estimate State

Using the Estimate state's defined signals, the Output logic unit will decide the ML codeword. After codeword decision, the Output logic unit will set a flag linked with Fout input of the control unit. If Fout is high then controller switches to Finish State otherwise it will remain in Estimate state and wait for Fout flag to get high.

Finish State

Finish signal in this state is an external event which will be triggered if all the data samples which are required to be decoded are finished. If Finish input is high the system will be to the default state otherwise it will jump to Load_IN state for processing of new data sample set of a block code.

3.3 Possible improvements

In any hardware design implementation, there are always the chances of improvement that a designer may consider. In BEAST implementation, a different implementation strategy for some parts of the design may optimize the overall design in terms of area utilization and speed.

The major improvement that can reduce the overall area of the design is to introduce a single frame strategy instead of having different frames for storage of Nodes, depths and their respected weights during Search/Select operation. The current design utilizes sixteen bit Node frame for representing the status of sixteen nodes at any iteration level, sixty four bit frame to represent depth levels at any level of tree (one vector each for forward and backward tree) and two hundred and eight bit frame for containing the weight information of nodes.

It means that each information frame has to be stored separately. But if a single frame mechanism is utilized then sixteen bits are required for node representation, four bits instead of 64 bits are required for depth level and two hundred and eight bits are required for weight representation. Total number of bits (TOT) required to be stored by Search/Select logic unit are computed as

$$TOT = 2 \times [(16 \times 8) + (64 \times 8) + (208 \times 8)] \tag{3.1}$$

$$TOT = 4608(bits) \tag{3.2}$$

By using a single frame strategy, that is, instead of assigning four bits each for every node for depth representation which leads to the total sum of 64 bits in each

frame(16 nodes $\times$ 4 bits each), only four bits are required for representing depth of every node in single frame which leads to.

$$TOT(SingleFrame) = 2 \times [(16 \times 8) + (4 \times 8) + (208 \times 8)] \tag{3.3}$$

$$TOT(SingleFrame) = 3648\,(bits) \tag{3.4}$$

Hence lesser bits are required to represent the same information which leads to chip area improvement. This improvement will be more effective while implementing higher block codes. Second alternative for saving chip area will be to implement time multiplexed architectures rather than parallel logic architectures utilized at different levels of the design.

Chapter 4

Implementation of BEAST

4.1 Overview

The major decision that a hardware designer would have to take after verifying the functional and timing requirements of the design is to implement the design on a specific platform. There are the two distinct options that a designer may consider for its hardware implementation. The first option is to implement the design using ASIC design flow and the other option is to use the reconfigurable hardware platforms (like FPGA's) for implementation of the design. The BEAST is primarily implemented on FPGA but also, the design has been synthesized by using ASIC design flow.

This chapter mainly covers both the implementation strategies regarding the BEAST. The detailed area and timing analysis has been performed for showing the trends regarding area utilization and timing limitations of the design. After verifying the design functionality, some of the performance parameters are observed like analysis of BER (Bit Error Rate) and SNR (Signal to Noise Ratio).

4.2 ASIC Synthesis

In order to synthesize the design using ASIC synthesis techniques, Design Vision tool is used. The design can be synthesized by using different synthesis constraints. To check the maximum speed under which the design can operate properly, the timing constraints can be utilized. To observe the area on silicon that can be covered by the design, the area constraints can be utilized.

ASIC synthesis will provide the detail description of design in the form of a Gate netlist along with timing information which is used to perform timing simulations for post synthesis analysis. BEAST decoder is synthesized by using two synthesis constraints in order to provide the maximum speed as well as the minimum area of the design.

4.2.1 Maximum speed

The maximum speed of the design in terms of frequency is 142.8572 MH_Z while having clock speed of 7 nano seconds. The design is been synthesized by setting clock constraints so that no negative slack exists. The design had a critical path which exists in Input Unit instance UUT2 of the design in sorting mechanism. The critical path is UUT2/UUT3T/L1inst/out1_Reg[0] to UUT2/UUT3T/Tout7_Reg[12]. The timing analysis of the design is specified in Table 4.1.

Table 4.1: Timing report.

Point	**Incr (nS)**	**Paths (nS)**
Clock clk (rise edge)	7.00	7.00
Clock network delay (ideal)	0.00	7.00
Clock uncertainty	-1.00	6.00
UUT2/UUT3/Tout7_reg[12]/ck(QDFFRBELD)	-1.00	6.00 r
Library setup time	-0.13	5.87
data required time		5.87
data arrival time		-5.87
Slack (MET)		0

4.2.2 Minimum area

Using the minimum area constraint in order to constrain the design for achieving the area near to zero, the design was constrained to get minimum area that can be used to implement the hardware. Since the area on chip is directly related

to the overall cost of the design hence reducing the area means the reduction in cost of implementation of the design. The chip area of the design is reduced from 744085.752750 um^2 to 724609.273184 um^2. Consider the area report shown as Table 4.2. The design is utilizing more sequential logic area as compared to combinational logic because of the fact that majority of the structure is purely sequential and also for synchronization of different design units, register units are used.

Table 4.2: Area report.

Number of Ports	24
Number of Nets	1113
Number of Cells	8
Number of References	8
Combinational Area	243409.918968 um^2
Non Combinational Area	481199.354216 um^2
Net Interconnect Area	Undefined (Wire load has zero net area)
Total Cell Area	724609.273184 um^2
Total Area	Undefined

4.2.3 Comparison of Maximum Speed vs. Minimum Area

The cell report produced by the tool is used to extract area information of each individual design units inside the complete hardware. The respective area consumed by each module is shown in Table 4.3. It can be observed from the statistics that Input Unit of the design is consuming approximately forty seven percent of the total design area. This is because of the fact that the trees (forward and backward) are also constructed and maintained in this design unit along with threshold calculation.

The maximum area is utilized by the Input Unit for storing input samples along with weighted Hamming distance for respected forward and backward trees. Similar is the case with Fsearch_top and BSearch_top inside Search/Select logic unit. Search/Select unit also consumes more storage spaces for storing search results.

4.3 FPGA Implementation

The final step after verifying the functional as well as post synthesis simulation was to implement the design on reconfigurable hardware structure like FPGA based platforms. The design was implemented by using XUP Virtex_II Pro Development System. This system is an advanced hardware platform that contains a high performance Xilinx Virtex_II Pro Platform FPGA.

Table 4.3: Comparison of Maximum Speed vs. Minimum Area.

Cell Name	**Maximum Speed** (um^2)	**Percentage** (%)	**Minimum Area** (um^2)	**Percentage** (%)
EstimateCW_top	6882.56003	0.9241	6850.56103	0.9454
QDFFRBELD	28.1600	0.0038	28.1600	0.0039
INVDLD	7.6802	0.0002	3.8400	0.0006
Control_unit	1661.4399	0.2238	1651.9999	0.2278
Input_unit	343210.2354	46.1250	336385.2755	46.4229
FSearch_top	160305.9184	21.5440	154812.1585	21.3649
BSearch_top	159635.1985	21.4538	154085.1186	21.2645
Compare_top	72354.5605	9.7239	70792.9605	9.7698
	744085.7527	100	724609.2731	100

The complete development system consists of numerous categories of peripheral components which can be used together in order to implement a complex system. The first step toward the implementation of the design was to synthesize the design for FPGA platforms which was done by using Xilinx ISE tool. For this purpose a user constraints file was constructed to map the design inputs and outputs to the provided interfaces on development system.

The device utilization summary obtained from FPGA implementation is shown in Table 4.4. It can be seen that majority of the device resources are fully used due to huge design size. This can be seen by observing the number of slices that the design occupied on FPGA platform. The Advance HDL synthesis report generated

Table 4.4: Device Utilization Summary (xc2vp30-7ff896).

Logic Utilization	**Used**	**Available**	**Utilization**
Number of Slices	12485	13696	91%
Number of Slice Flip Flops	14438	27392	52%
Number of 4 input LUTs	13617	27392	49%
Number of bonded IOBs	12	556	2%
Number of GCLKs	1	16	6%

by Xilinx ISE tool categorizes the number of generated design components during implementation process. The Advance HDL synthesis report is specified in Table

4.5. Registers and flip flops are the main components that are the cause of excessive area usage on FPGA. For implementing higher Block codes the focus should be on the amount of registers which will be used in order to save some area.

Table 4.5: Advance HDL synthesis report.

Components	**Amount**
Adders/Subtractors	121
Counters	9
Registers	14840
Comparators	150
Multiplexers	17
Xors	1

4.4 Comparison of BER and SNR

After implementing the design, Bit Error Rate (BER) analysis is performed for different values of Signal to Noise Ratio (SNR). For that purpose, the first step was to generate the 1250000 Blocks representing a same codeword for analysis. The reference codeword of 00000000 is used. The AWGN noise under the contribution of different SNR values are added to the reference codeword and 1250000 vectors are generated with the help of Matlab.

After generation of the random vectors which serves as an input to the BEAST implementation, the vectors are stored in set of files which are applied to the RTL design using Modelsim. The output is also stored in output files generated through Modelsim. The final step was to import the Modelsim generated output files in Matlab and compare the result with Matlab generated reference files containing 1250000 reference vectors without any bit error. The error bits are added together and are divided by total number of bits to give the BER values.

The detailed results of the analysis are shown in Figure 4.1. By observing the obtained results, it is concluded that the response of the system for (8, 4, 4) extended Hamming codes is within the desired theoretical bounds. With increase in SNR values, the BER is getting smaller. In start with low SNR, there is more noise corrupting the data signals and the probability of errors is higher. The design will estimate a different codeword in some cases. By increasing the SNR values, the results are becoming more and more accurate. At higher values of SNR, BER is so small and ultimately lesser error bits are found which can be observed at SNR=7 and onwards.

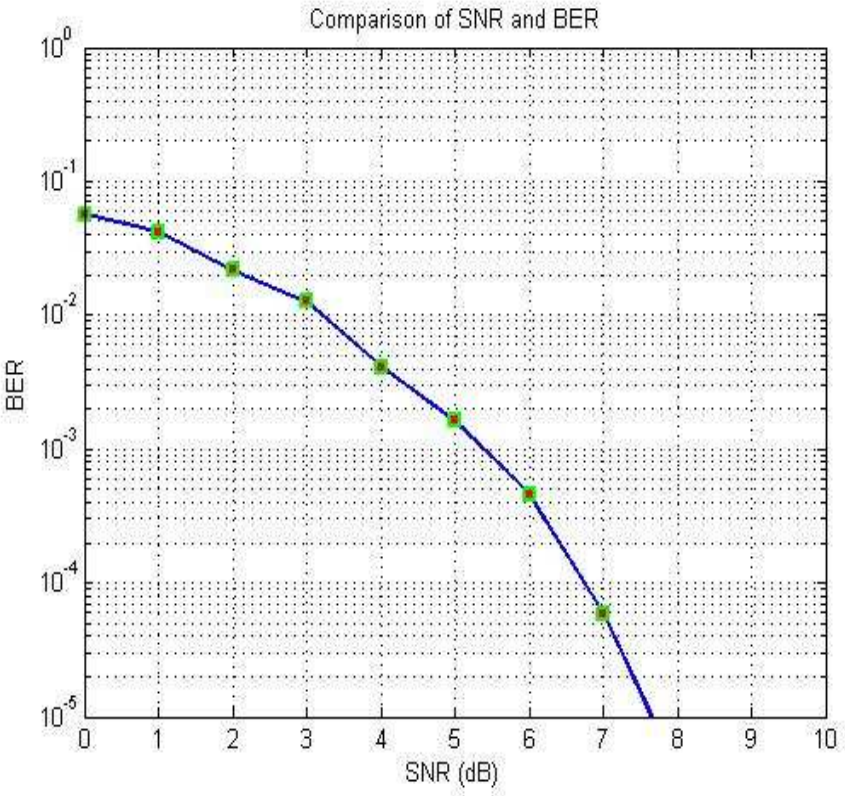

Figure 4.1: Comparison of BER and SNR

Chapter 5

Conclusion and Future work

5.1 Conclusion

The main focus of this work was to implement a well defined structure that can be used to decide a codeword using the BEAST. The functionality and concepts regarding BEAST are verified by the implementation. The first step toward the implementation was to visualize a structure from algorithm and develop a conceptual design.

The next step was to implement the behavioral model using Matlab. After testing the functionality by using a behavioral model, the next step was to develop a synthesizable model using VHDL. The implementation was coded in VHDL and simulated by modelsim, to provide functional and timing simulations of the design.

Afterwards the design was implemented on FPGA as well as synthesized by using an ASIC design flow. It can be concluded from the implementation that the BEAST is an efficient approach to obtain a ML decision using block codes. Majority of the design structure is generic with exception to some of the design units, which can also be improved.

The verification work was done by the use of Xilinx Virtex_II FPGA while ASIC synthesis was done by using UMC130 high speed standard cell library. The design covers 91 percent of the slices on FPGA. By performing ASIC synthesis it was observed that the design can work with maximum clock frequency of 143 MH_z while having a chip area of 0.72469 mm^2.

5.2 Future Work

There are some design optimizations and suggestions that can be considered while implementing the hardware for higher block codes and these are.

- Fully generic structure
  The contribution of the generator matrix G in the design implementation can be made generic to cover different block codes. For that purpose, the tree generation mechanism and calculation of the weighted Hamming distance within the Input Unit along with decision of the codeword mechanism in Compare/Output logic can be changed while keeping the remaining design as same.

- Storage space reduction
  Introduction of the single frame strategy for storage of nodes with their depths and weights in Search/Select logic unit can be implemented in order to save the storage space. The current design is storing the depth levels separately for each node but by using single frame strategy only one entry is required to store the depth level for entire frame rather than every single node.

The throughput of the hardware can be observed and compared under different communication standards like LTE (Long Term Evolution) and others. These analysis can be helpful for future research and use.

The power analysis can also be performed to check the power requirements while implementing the design by using ASIC system design flow.

Bibliography

[1] I. Bocharova, R. Johannesson, B. D. Kudryashov, and M. Lončar, "BEAST decoding for block codes", *In Proc. Int. ITG Conf. Source and Channel Coding*

[2] I. Bocharova, R. Johannesson, and B. D. Kudryashov, "BEAST decoding of block codes obtained via Convolutional codes", *IEEE transactions on Information Theory, 51(5):1880-1891, May 2005*

[3] M. Lončar, " Tamming of the BEAST, " *PhD Thesis, LUND University, October 2007*

[4] I. Bocharova, R. Johannesson, and B. D. Kudryashov, "Trellis Complexity of short linear codes", *IEEE transactions on Information Theory, 361-368, 2007*

[5] I. Bocharova, M. Handlery, R. Johannesson, and B. D. Kudryashov, "A BEAST for prowling in trees", *IEEE transactions on Information Theory, Vol 50, 1295-1302, June 2004*

[6] A. J. Viterbi, "Error bounds for Convolutional codes and an asymptotically optimal decoding algorithm", *IEEE transactions on Information Theory, Vol IT-13, 260-269, April 1967*

[7] F. Hug, "On Graph-Based Convolutional Codes", *Master Thesis, Lund University, 2009*

[8] Peter J. Ashenden, "The designers Guide to VHDL", 3^{rd} *edition, ISBN 978-0-12-088785-9, 2008*

[9] Pong P. Chu, "RTL Hardware Design using VHDL", *John Wiley and Sons Inc, ISBN 9780471720928*

[10] Trellis Structures, www.academicearth.org/lectures/trellis-representations-binary-linear-block-codes-1

[11] Decoder Structures, www.eit.lth.se/fileadmin/eit/courses/eti180/Slides2011/Kamuf-2011.pdf

[12] XUP Virtex_II Pro Development System User Guide.