

**FPGA IMPLEMENTATION OF
SECURED TRANSMISSION AND HIGH SPEED TEXT DATA COMPRESSION
USING HAMMING AND HUFFMAN CODING**

By
Mohammad Rakib

MASTER OF ENGINEERING
IN
INFORMATION AND COMMUNICATION TECHNOLOGY

Institute of Information and Communication Technology
BANGLADESH UNIVERSITY OF ENGINEERING AND TECHNOLOGY
September 2017

The project titled, “FPGA Implementation of Secured Transmission and High Speed Text Data Compression using Hamming and Huffman Coding” submitted by Mohammad Rakib, Roll No. 0412312045(P), Session: April/2012, has been accepted as satisfactory in partial fulfillment of the requirement for the degree of Master of Engineering in Information and Communication Technology on 26 September, 2017.

BOARD OF EXAMINERS



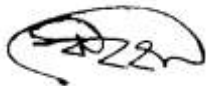
-
1. **Dr. Md. Liakot Ali**
Professor
IICT, BUET, Dhaka.
(Supervisor)

Chairman



-
2. **Dr. A. B. M. Harun-Ur-Rashid**
Professor
Department of EEE, BUET, Dhaka.

Member

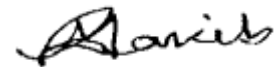


-
3. **Dr. Md. Rubaiyat Hossain Mondal**
Associate Professor
IICT, BUET, Dhaka.

Member

CANDIDATE'S DECLARATION

It is hereby declared that this project or any part of it has not been submitted elsewhere for the award of any degree or diploma.



Mohammad Rakib
Roll: 0412312045 (P)
IICT, BUET.

DEDICATION

I dedicate this project to my parents and my wife.

TABLE OF CONTENTS

List of Figures	viii
List of Tables.....	x
List of Abbreviations of Technical Symbols and Terms	xi
Acknowledgement	xii
Abstract	xiii
CHAPTER 1 Introduction.....	1
1.1 Overview	1
1.2 Motivation and Scope.....	2
1.3 Objectives of the Project	2
1.4 Project Outline.....	3
CHAPTER 2 Literature Review.....	4
2.1 Introduction	4
2.2 Huffman Algorithm.....	4
2.2.1 Construction of Huffman Tree	6
2.3 Flowchart of Huffman Encoder.....	10
2.4 Flowchart of Huffman Decoder.....	11
2.5 Hamming Code.....	12
2.6 Overview of FPGA.....	14
2.6.1 Architecture of FPGA	14
2.6.2 Design and Programming on FPGA	15
2.6.3 Programing Language for FPGA	15
2.6.4 Advantages of FPGAs.....	16
CHAPTER 3 Design and Implementation	17
3.1 Introduction	17

3.2	Architecture of the Design	17
3.2.1	Controller:	18
3.2.2	Bit-stuffing Compression Module:	18
3.2.3	Huffman Compression Module.....	19
3.2.4	Hamming Encoding Module.....	23
3.2.5	Hamming Decoding Module.....	23
3.2.6	Huffman Decompression Module.....	24
3.2.7	Bit-stuffing Decompression Module.....	24
3.3	Flow Chart of the Design	25
3.4	Algorithm of the Proposed Method.....	25
3.4.1	Transmission Unit.....	26
3.4.2	Receiver Unit	28
3.5	MATLAB Simulation.....	29
3.6	Design of the System.....	31
3.7	Software Simulation Tool—MATLAB.....	31
3.8	FPGA Simulation Tool— ModelSim.....	32
CHAPTER 4 Results and Discussions.....		34
4.1	Introduction	34
4.2	Software Simulation	34
4.3	Generated Output File of SW Simulation	35
4.4	Compilation of the FPGA Design	37
4.5	Analysis of FPGA Simulation Results	38
4.5.1	Simulation of Bit-Stuffing Compression Module.....	38
4.5.2	Simulation of Character Counter	39
4.5.3	Simulation of Frequency Sorting Module.....	40
4.5.4	Simulation of Huffman Tree Generator	40
4.5.5	Simulation of Code Generator	41

4.5.6	Simulation of Coding Module.....	42
4.5.7	Simulation of Hamming Encoding Module	42
4.5.8	Simulation of Hamming Decoding Module	43
4.5.9	Simulation of Huffman Decompression Module	45
4.5.10	Bit-stuffing Decompression Module.....	45
4.6	Error Correction Rate	46
4.7	Compression Ratio	47
4.8	Comparison with Other Compression Methods	49
CHAPTER 5 Conclusion		50
5.1	Conclusion.....	50
5.2	Future Works	51
References.....		53

LIST OF FIGURES

Figure 2.1 Huffman Encoding Example	5
Figure 2.2 Construction of Huffman Tree (Step-1)	7
Figure 2.3 Construction of Huffman Tree (Step-2)	7
Figure 2.4 Construction of Huffman Tree (Step-3)	7
Figure 2.5 Construction of Huffman Tree (Step-4)	8
Figure 2.6 Construction of Huffman Tree (Step-5)	8
Figure 2.7 Construction of Huffman Tree (Step-6)	9
Figure 2.8 Construction of Huffman Tree (Step-7)	9
Figure 2.9 Flowchart of Huffman Encoder	10
Figure 2.10: Flowchart of Huffman Decoder.....	11
Figure 2.11 An Altera Cyclone II FPGA (DE2 Board)	14
Figure 2.12 Simplified Version of an FPGA Logic Block	15
Figure 3.1 Archietecture of the Proposed System.....	17
Figure 3.2 : Block Diagram of Controller	18
Figure 3.3 Block Diagram of Bit-stuffing Compression Module	18
Figure 3.4 Block Diagram of 7-Bit Adder	19
Figure 3.5 Block Diagram of Character Counter	20
Figure 3.6 Block Diagram of Frequency Sorting Module	20
Figure 3.7 Block Diagram of Node Sorting Module.....	21
Figure 3.8 Block Diagram of Huffman Tree Generator.....	21
Figure 3.9 Block Diagram of Code Generator	22
Figure 3.10 Block Diagram of Coding Module	22
Figure 3.11 Block Diagram of Hamming Encoder	23
Figure 3.12 Block Diagram of Hamming Decoder	23
Figure 3.13 Block Diagram of Huffman Decompression Module.....	24
Figure 3.14 Block Diagram of Bit-stuffing Decompression Module	24
Figure 3.15: Flowchart of the Proposed FPGA based System.....	25
Figure 3.16: Process Flow at Transmission Unit	26
Figure 3.17: Process Flow at Receiver Unit	28
Figure 4.1 Sample of MATLAB Simulation.....	34
Figure 4.2 A Sample Simulation Result on MATLAB.....	35
Figure 4.3: Sample Input Text File (Partial View)	36

Figure 4.4: Generated Output File in Transmission Unit. (Partial View).....	36
Figure 4.5 Result of Compilation on ModelSim.....	37
Figure 4.6 Simulation of Bit-stuffing Compression Module with 104 bits Input Data.	39
Figure 4.7 Simulation Result of Character Counter. Counted frequencies match with frequency distribution mentioned in Table 4.1.	39
Figure 4.8 Simulation Result of Frequency Sorting Module.	40
Figure 4.9 Simulation Result of Huffman Tree Generator	41
Figure 4.10 Simulation Result of Code Generator.....	41
Figure 4.11 Simulation of Coding Module	42
Figure 4.12 Simulation of Hamming Encoding Module.....	43
Figure 4.13 Simulation of Hamming Decoder Module for Error Free Input Data	44
Figure 4.14 Simulation of Hamming Decoder Module for Erroneous Input Data	44
Figure 4.15: Simulation Result of Huffman Decompression Module	45
Figure 4.16: Simulation Result of Bit-stuffing Decompression Module	46
Figure 4.17: Result of Proposed 2 Level Compressions	48

LIST OF TABLES

Table 2.1 Frequency Distribution of the Example Data	6
Table 2.2 Format of Hamming (7,4) Encoding.....	12
Table 2.3 Hamming (7, 4) code-word for data 1100.....	12
Table 3.1 Node Structure	21
Table 3.2 Functions of Matlab Simulation.....	30
Table 3.3 Modules of FPGA Implementation.....	31
Table 4.1 Frequency Distribution and ASCII values of a Test String	38
Table 4.2: Error Correction Rate of Received Data (800 bits)	46
Table 4.3: Result of Proposed Two Level Compression Method.....	48
Table 4.4: Comparison of Saving Percentage (SP) and Compression Ratio (CR) between Proposed Method and Other Compression Techniques	49

LIST OF ABBREVIATIONS

ASCII	American Standard Code for Information Interchange
ASIC	Application-Specific Integrated Circuit
CR	Compression Ratio
CLB	Configurable Logic Block
DLL	Dynamic Link Library
DSP	Digital Signal Processor
ECC	Error correction code
EEPROM	Electrically Erasable Programmable Read-Only Memory
FPGA	Field-Programmable Gate Array
GZIP	GNU's not UNIX zip
GUI	Graphical User Interface
HDL	Hardware Description Language
I/O	Input/Output
ISE	Integrated Synthesis Environment
IT	Information Technology
JTAG	Joint Test Action Group
LUT	Look-Up Table
MATLAB	MATrix LABoratory
OS	Operating System
RTL	Register-Transfer Level
RAM	Random Access Memory
ROM	Read Only Memory
SoC	System on a Chip
SP	Saving Percentage
SKS	Single Kernel Simulator
VLSI	Very-large-scale integration
VHDL	VHSIC Hardware Description Language
VHSIC	Very High Speed Integrated Circuit

Acknowledgement

First of all I must thank to the Almighty Allah (SwT.) who has the supreme authority. Without His mercy it is impossible to have any success. I would like to express my heartfelt gratitude to Dr. Md. Liakot Ali, Professor, IICT, BUET for the guidance, inspiration and constructive suggestion which helped me to complete my project work successfully. Lastly, I offer my regards and blessings to all of my friends and family members who supported me in any respect during the completion of the project.

Abstract

In this era of big data, high speed data compression and its secured transmission are burning issues. Data compression and its reliable transmission are necessary especially for the storage and transmission related applications. This project proposes a solution in this regard and presents the design and implementation of a Field Programmable Gate Array(FPGA) based system which offers advantages over software solution in terms of higher speed, real-time performance, higher reliability, re-configurability and also ease of integration with the existing consumer electronic devices. The proposed system is a blend of bit stuffing, Huffman and Hamming algorithms. It has been simulated in the MATLAB environment to ensure the accuracy of the system and it is then designed using Verilog hardware description language (HDL) to implement into FPGA hardware. The proposed system comprises various processing modules for compressing and securing data for reliable transmission. This project is based on ASCII text data, but the proposed technique can be implemented for other data types which have been considered as future extension of this work. To secure text data for transmission, Hamming (7, 4) coding is used. It secures data by adding a layer of encryption as well as by providing facility for error detection and correction for reliable transmission. In this step, each character of sending text data is divided into groups of 4-bit data and encoded with 3 parity bits, thus resulting 7-bit code-word. This system can correct 2-bit errors in each 8-bit character. Furthermore, the system includes a two-level compression. During the first level of compression, redundant bits of 8 bit characters are removed which has been mentioned as bit-stuffing. After compressing by bit-stuffing, second level compression is done by Huffman algorithm. These two level compression processes can achieve higher saving percentage of memory. Hence, the system provides reliability in transmission and also compresses data to larger extent. The results obtained are highly promising and the system is very effective for providing high level reliability and higher saving percentage of memory which in turn reduces bandwidth and transmission time.

CHAPTER 1

Introduction

1.1 Overview

This is an era of big data where big data refers to voluminous amounts of structured or unstructured complex data sets that can be potentially mined and analyzed for understanding the real world. Many emerging applications have come out based on big data such as spy satellites, remote sensing, medical imaging, customer relationship management, corporate surveillance etc. Again applications and simulations related to seismography, oceanography, bioinformatics, cosmology etc. are coming out where billions of terabyte data known as hyper spectral data are required to store and transmit securely which can easily overwhelm limited space communication channels [1-2]. A survey report stated companies are spending 12% of their IT budget on storage and this cost is doubling every two years [2]. In this context, data compression and its secured transmission has become a demanding issue of this day [3]. The purpose of this project is to present a mechanism for high speed compression and reliable transmission of ASCII text data.

Our proposed technique is a combination of bit stuffing, Huffman and Hamming algorithms. Software implementation for the proposed system cannot fulfill the performance and real time requirements since it is inherently slow and the software designer does not have any direct control over the way with which RAM and processors interact. This imposes a limit on operating speed while a hardware designer has full control over timing operations into the RAM and direct control over the usage of expensive hardware resources. For hardware implementation, FPGA technology has brought revolution in the electronic system design and development. It is now being widely used in various applications such as telecommunications, networking, consumer, automotive and industrial applications, etc. [4]. FPGA chip adoption across all industries is driven by the fact that FPGAs have more advantages than ASICs and processor-based systems. Over the few decades, FPGA device market has been emerged as multi-billion dollar market due to its lower cost than that of ASIC, reusability, massive parallelism, higher speed than conventional computing, lower power, shorter time to market etc. [5]. For this reason, researches on FPGA based compression technique has become prominent than software based or ASIC based design.

As mentioned, one of the prime aspects of compression is to reduce bandwidth for transmission; it is a motivating topic to incorporate error correction mechanism into the compression process for reliable transmission.

1.2 Motivation and Scope

Data compression has a wide range of applications including representation of the abstract data type and file compression. A lot of researches have been conducted and also are ongoing to develop new techniques for data compression. Huffman algorithm is amongst the most popular algorithm for lossless compression. Though it was invented seven decades ago, a lot of researches [6-11] have been conducted in recent times based on this algorithm. Since software solution for data compression is inherently slow and not energy efficient, a number of FPGA based lossless data compression solutions have been proposed focusing on high throughput, low cost and energy efficiency [12-15]. FPGA based researches on data compression has added a new dimension to invent robust methodology. Among the latest FPGA based researches, Dias et al. [16] proposes a new method of code compression for embedded systems which applies two compression techniques and uses the Huffman algorithm. They have implemented the de-compressor using VHDL for FPGA and claimed that their proposed method reduces code size up to 30.6%. Rigler et. al. [17] presented hardware implementations for the LZ77 encoders and Huffman encoders that form the basis for a full hardware implementation of a GZIP encoder. The designs have been implemented as state machines in VHDL in such a way that they are suitable for implementation using either FPGA or ASIC technologies. However among all these software and FPGA based researches security and reliability issues have not much considered or lightly considered. So, there is scope of conducting research which can provide low power, high speed, real time data compression and at the same time can ensure data security and reliability for transmission.

1.3 Objectives of the Project

The aim of the project is to develop a FPGA based system for high speed real time text data compression with higher saving percentage of memory as well as ensuring its secured and reliable transmission.

The project work focuses on the following objectives:

- To design the Huffman algorithm based compression and decompression engine using suitable programming language then test it using sample data.
- To add another layer with the compression engine using Hamming code for secured transmission and simulate the environment using suitable programming language
- To design the system using Verilog HDL and simulate the system under FPGA environment.
- To compare the results with other data compression methods.

Outcome of this project: An HDL based design for secured transmission and high speed data compression using Hamming and Huffman coding will be the possible outcome of this project.

1.4 Project Outline

The rest of the project is organized as follows: **Chapter 2** describes the fundamentals of Huffman algorithm and Hamming coding which is required to understand different functional blocks of the implemented system of this project. It also provides an overview of FPGA technology. **Chapter 3** discusses about the design and implementation of the proposed system. **Chapter 4** covers the simulation results and discussions of the proposed system. Finally, **Chapter 5** offers suggestions for future work along with concluding remarks.

CHAPTER 2

Literature Review

2.1 Introduction

Compression is the art of representing the information in a compact form rather than its original form [18]. More concisely, data compression comprises the identification and removal of redundant elements of source data. The main objective of data compression is to reduce the size of file to store or transmit over communication channel [19]. The design of a compression algorithm involves understanding the types of redundancy present in the data and then developing strategies for removing these redundancies to obtain a compact representation of the data.

This chapter begins with the description of Huffman algorithm and Hamming coding which are used for this project. Then it provides an overview of FPGA technology, its architecture and advantages over conventional circuit design.

2.2 Huffman Algorithm

Huffman algorithm is a lossless statistical data compression scheme. It takes advantage of the disparity between frequencies of the symbols or characters in the content. It uses less memory for the frequently occurring characters and more memory for the more rare characters. Huffman is an example of a *variable-length* encoding—some characters may only require 2 or 3 bits and other characters may require 7, 10, or 12 bits. The savings from not having to use a full 8 bits for the most common characters makes up for having to use more than 8 bits for the rare characters and the overall effect is that the file almost always requires less space. It was developed by David A. Huffman while he was a Sc.D. student at MIT, and published in the 1952 paper "A Method for the Construction of Minimum-Redundancy Codes" [20].

There are two types of Huffman coding—static and dynamic. In the static method the frequencies of each character in the alphabet are assigned before the program begins and are stored in a look-up table. In the dynamic method, on the other hand, one pass through the

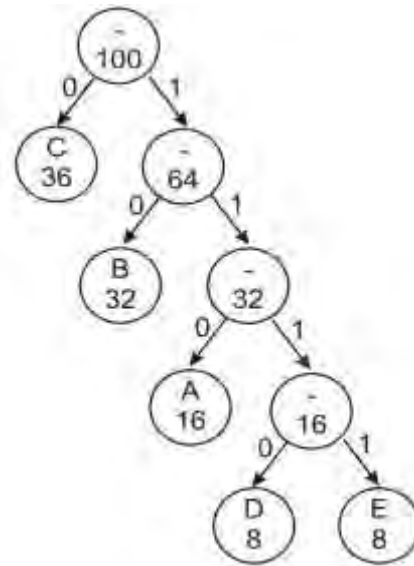
content has to be made to determine the frequency of each character. Once the histogram has either been calculated or provided, the two algorithms are identical. Elements are selected two at a time, based on frequency; lowest frequency elements are chosen. The two elements are made to be leaf nodes of a node with two branches. The frequencies of the two elements selected are then added together and this value becomes the frequency for the new node. The algorithm continues selecting two elements at a time until a Huffman tree is complete with the root node having a frequency equal to the total number of characters of the content. An example of Huffman encoding is given in Figure 2.1.

Character	Frequency
A	16
B	32
C	36
D	8
E	8

Histogram

Character	Frequency
A	110
B	10
C	0
D	1110
E	1111

Huffman Codes



Huffman Tree

Figure 2.1 Huffman Encoding Example

The branches of the tree represent the binary values 0 and 1 according to the rules for common prefix-free code trees. The path from the root tree to the corresponding leaf node defines the particular code-word.

2.2.1 Construction of Huffman Tree

The Huffman algorithm generates the most efficient binary code tree at given frequency distribution. Prerequisite is a table with all symbols and their frequency. Any symbol represents a leaf node within the generated tree.

The following general procedure has to be applied:

1. Creating a collection of singleton trees, one for each character, with weight equal to the character frequency.
2. From the collection, picking out the two trees with the smallest weights and removing them.
3. Combining them into a new tree whose root has a weight equal to the sum of the weights of the two removed trees and with the two trees as its left and right subtrees.
4. Adding the new combined tree back into the collection.
5. Repeating steps 2 to 4 until there is only one tree left.
6. The remaining node is the root of the optimal encoding tree.

Let us understand the algorithm with the string “**hip hop happy**”. There are total 13 characters in the string with the frequency distribution mentioned in Table 2.1.

Table 2.1 Frequency Distribution of the Example Data

Character	Frequency
p	4
h	3
_ (space)	2
a	1
i	1
o	1
y	1

With above frequency distribution, Huffman tree can be constructed by following the steps mentioned next.

Step—1. The process begins with a collection of singleton tree; the weight (frequency) of each node is mentioned along with them in Figure 2.2.



Figure 2.2 Construction of Huffman Tree (Step-1)

Step—2. Two smallest nodes will be selected. There are four nodes with the minimal weight of 1 and any two of them can be picked. We choose 'o' and 'y' and combine them into a new tree whose root is the sum of the weights chosen. Those two nodes will be replaced with the combined tree. The nodes remaining in the collection are shown in the light gray box at in Figure 2.3.

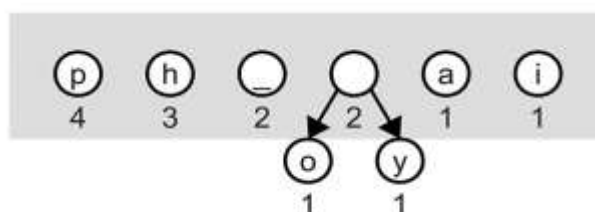


Figure 2.3 Construction of Huffman Tree (Step-2)

Step—3. Now step-2 will be repeated, this time there is no choice for the minimal nodes, it must be 'a' and 'i' (Figure 2.3). So those will be combined into a tree of weight 2 as shown in Figure 2.4.

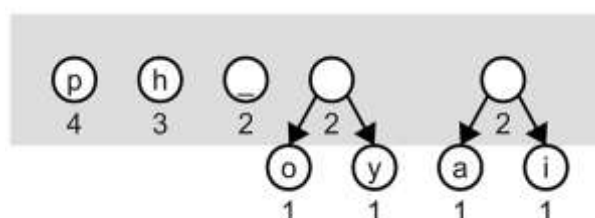


Figure 2.4 Construction of Huffman Tree (Step-3)

Step—4. Again, two smallest nodes will be combined and built a tree of weight 4 as shown in Figure 2.5.

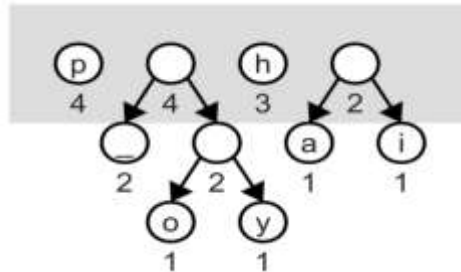


Figure 2.5 Construction of Huffman Tree (Step-4)

It is noticeable that when a combined node is built, it doesn't represent a character like the leaf nodes do. These interior nodes are used along the paths that eventually lead to valid encodings, but the prefix itself does not encode a character.

Step—5. Another iteration combines the weight 3 and 2 into a combined tree of weight 5 as shown in Figure 2.6.

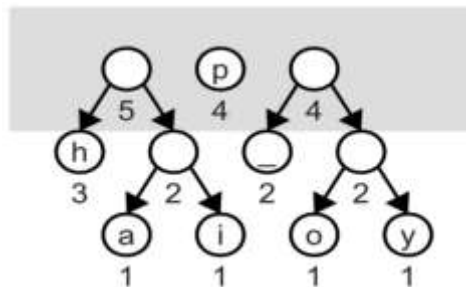


Figure 2.6 Construction of Huffman Tree (Step-5)

Step—6. Combining the two 4s in Figure 2.6 gets a tree of weight 8 as shown in Figure 2.7.

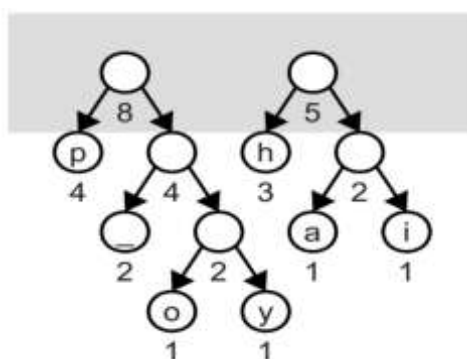


Figure 2.7 Construction of Huffman Tree (Step-6)

Step—7. Finally, the last two trees in Figure 2.7 will be combined to get final tree in Figure 2.8. The root node of the final tree will always have a weight equal to the number of characters in the source data. In this example it is 13 as shown in Table 2.1.

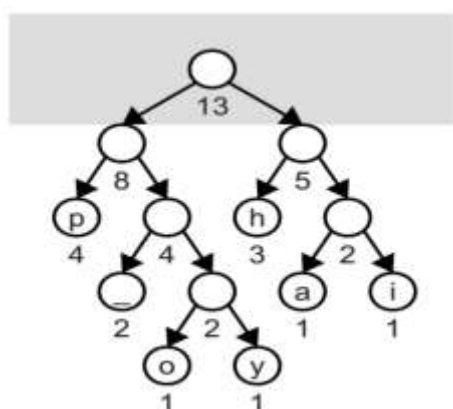


Figure 2.8 Construction of Huffman Tree (Step-7)

Formation of the Huffman tree can be different. When there are choices among equally weighted nodes (Figure 2.2 and Figure 2.4), picking a different two nodes will result in a different, but still optimal prefix codes. Similarly when combining two sub-trees, it is as equally valid to put one of the trees on the left and the other on the right as it is to reverse them (Figure 2.5).

2.3 Flowchart of Huffman Encoder

Figure 2.9 shows a simple flowchart of Huffman encoder.

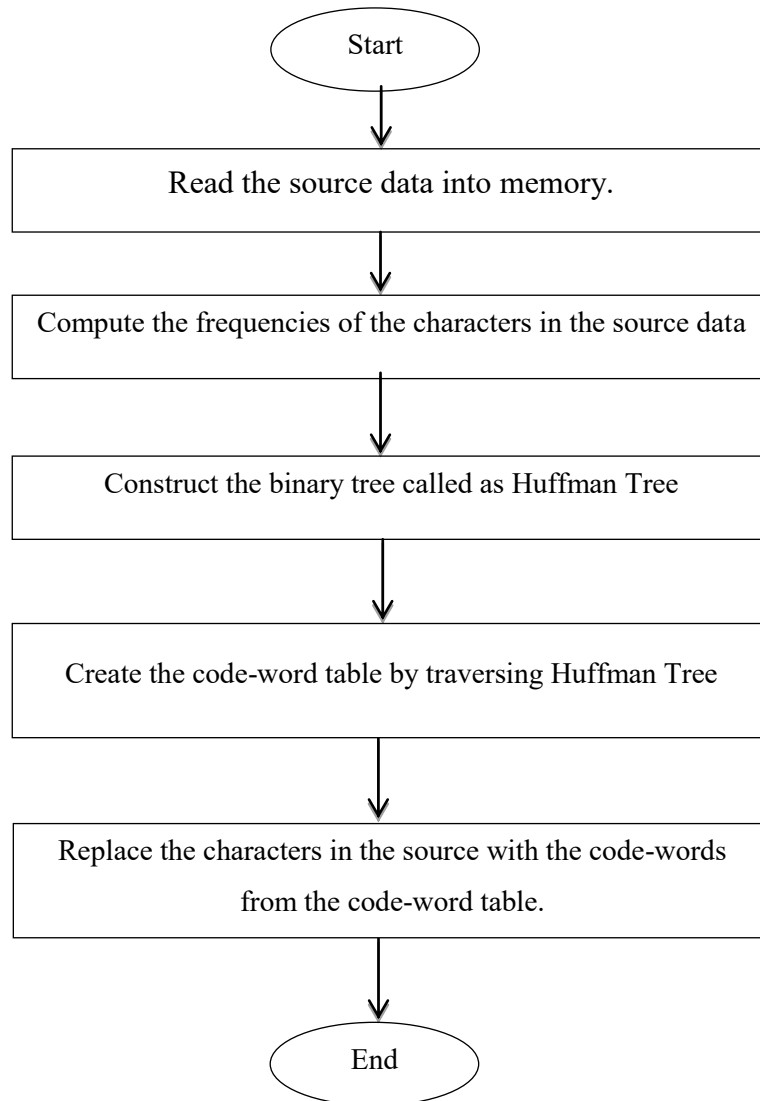


Figure 2.9 Flowchart of Huffman Encoder

2.4 Flowchart of Huffman Decoder

Figure 2.10 shows a flowchart of Huffman decoder.

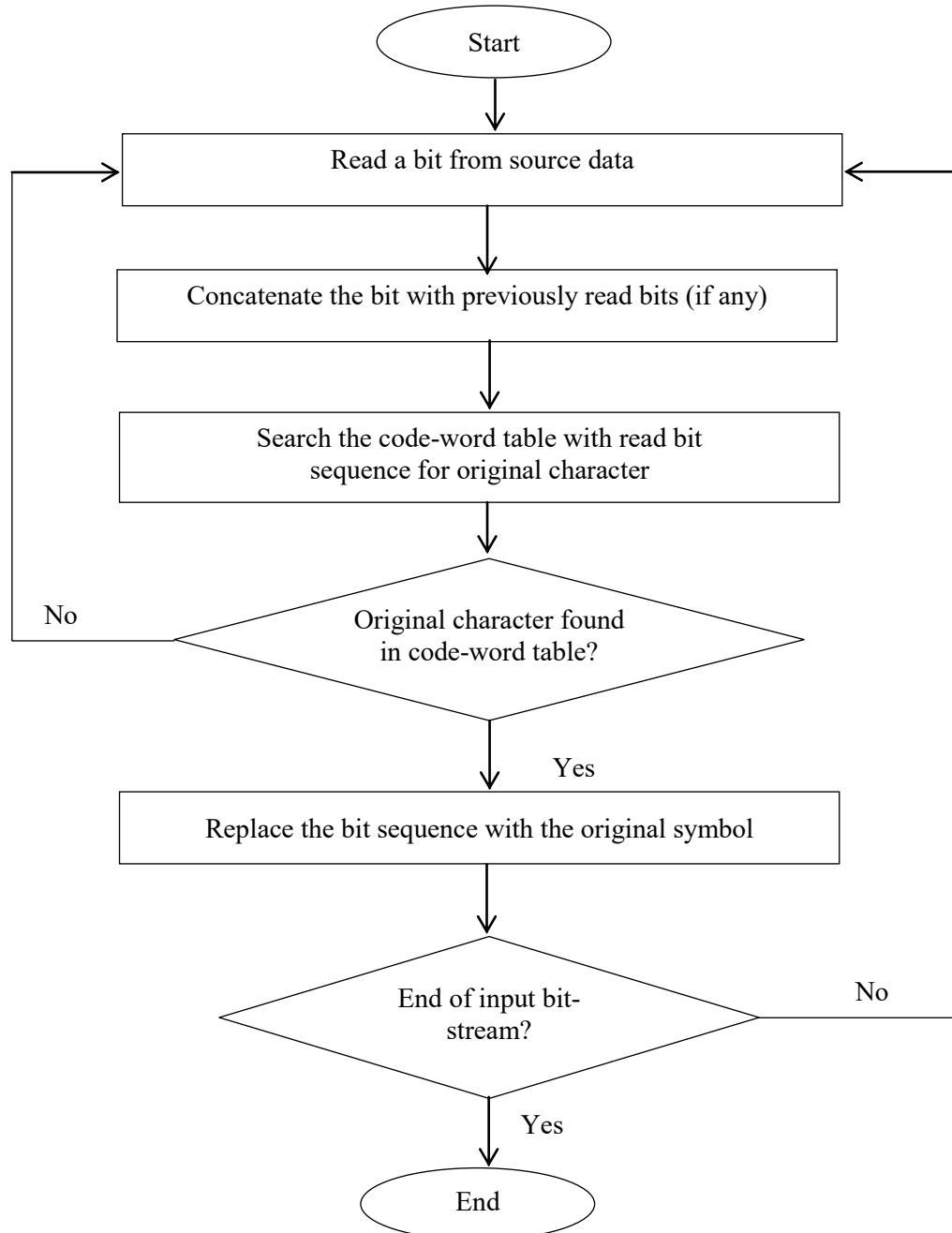


Figure 2.10: Flowchart of Huffman Decoder

2.5 Hamming Code

When digital data is transmitted or stored in nonvolatile memory, it is crucial to have a mechanism that can detect and correct a certain number of errors. Error correction code (ECC) encodes data in such a way that a decoder can identify and correct errors in the data. The most common types of ECC used in RAM are based on the codes proposed by R. W. Hamming [21]. Hamming codes are capable of correcting single bit error. In the Hamming code, k parity bits are added to an n -bit data word, forming a new word of $n + k$ bits. The bit positions are numbered in sequence from 1 to $n + k$. Those positions numbered with powers of two are reserved for the parity bits. The remaining bits are the data bits. The code can be used with data of any length.

For example, let us consider the 4-bit data word is 1100. To generate Hamming (7, 4) code, 3 parity bits are included with this data word and the resultant 7 bits are arranged as mentioned in Table 2.2.

Table 2.2 Format of Hamming (7,4) Encoding

Bit Position	7	6	5	4	3	2	1
Bit Value	1	1	0	P_4	0	P_2	P_1

Each parity bit is calculated as follows:

- Parity bit $P_1 = \text{XOR of bits } (3,5,7) = 0 \oplus 0 \oplus 1 = 1$
- Parity bit $P_2 = \text{XOR of bits } (3,6,7) = 0 \oplus 1 \oplus 1 = 0$
- Parity bit $P_4 = \text{XOR of bits } (5,6,7) = 0 \oplus 1 \oplus 1 = 0$

As per Hamming encoding rules, each parity bit is set so that the total number of 1's in the checked positions, including the parity bit, is always even. The 4-bit data word is written into the memory together with the 3 parity bits as a 7-bit composite word. By substituting the 3 parity bits in their proper positions, Hamming (7, 4) code-word is obtained as shown in Table 2.3.

Table 2.3 Hamming (7, 4) code-word for data 1100

Bit Position	7	6	5	4	3	2	1
Bit Value	1	1	0	0	0	0	1

When Hamming (7, 4) codes are received, they are checked again for errors. The parity of the word is checked over the same groups of bits, including their parity bits. The 3 check bits are evaluated as follows:

- $C_1 = \text{XOR of bits (1, 3, 5, 7)}$
- $C_2 = \text{XOR of bits (2, 3, 6, 7)}$
- $C_4 = \text{XOR of bits (4, 5, 6, 7)}$

A 0 check bit designates an even parity over the checked bits, and a 1 designates an odd parity. Since the bits were written with even parity, the result, $C = C_4C_2C_1 = 000$, indicates that no error has occurred. However, if the 3-bit binary number formed by the check bits gives the position of the erroneous bit if only a single bit is in error.

Consider following three cases-

- $C_4C_2C_1 = 000$ means no error
- $C_4C_2C_1 = 001$ means error in first bit.
- $C_4C_2C_1 = 101$ means error in 5th bit because 101 is binary of 5.

The error can then be corrected by complementing the corresponding bit.

Hamming (7, 4) code can detect and correct a single bit error. A modified Hamming code to generate and check parity bits for a single error-correction and double-error-detection scheme is most often used in real systems. The modified code uses a different parity check bit scheme that balances the number of inputs to the logic for each check. The balancing minimizes the delay through the error correction and detection circuits. These circuits can be used in a RAM subsystem to add check bits during write operations and to correct single errors and detect double errors during read operations.

2.6 Overview of FPGA

FPGA is a semiconductor device comprising programmable logic components and programmable interconnects. It contains up to thousands of gates. The programmable logic components can be programmed to replicate the functionality of basic logic gates such as AND, OR, XOR, NOT or more complex combinational functions such as decoders or simple math functions. In most FPGAs, this programmable logic also includes memory elements such as flip-flops or more complete blocks of memories. The FPGA configuration is generally specified using a HDL, similar to that used for an ASIC. FPGAs can be used to implement any logical function that an ASIC could perform. Figure 2.11 shows an Altera Cyclone II FPGA known as DE2 Board.



Figure 2.11 An Altera Cyclone II FPGA (DE2 Board)

2.6.1 Architecture of FPGA

The most common FPGA architecture consists of an array of logic blocks known as Configurable Logic Block (CLB), I/O pads, and routing channels. Generally, all the routing channels have the same width. Multiple I/O pads may fit into the height of one row or the width of one column in the array. An application circuit must be map into an FPGA with adequate resources. The typical FPGA logic block consists of a 4-input lookup table (LUT), and a flip-flop as shown in Figure 2.12. There is only one output, which can be either the

registered or the unregistered LUT output. The logic block has four inputs for the LUT and a clock input. Since clock signals are normally routed via special-purpose dedicated routing networks in commercial FPGAs, they and other signals are separately managed.

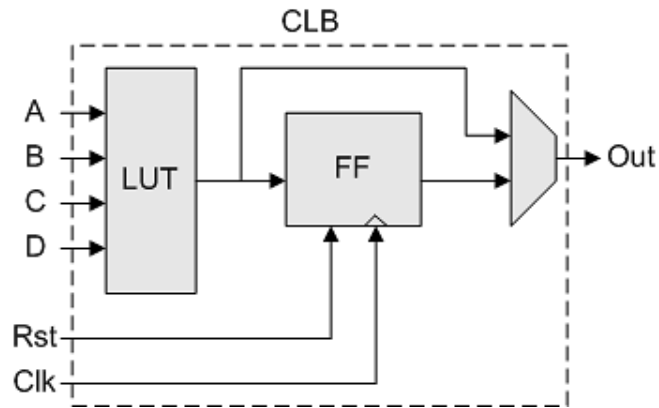


Figure 2.12 Simplified Version of an FPGA Logic Block

2.6.2 Design and Programming on FPGA

To define the behavior of the FPGA, the user provides an HDL or a schematic design. In a typical design flow, the RTL description in VHDL or Verilog is simulated by creating test benches to simulate the system and observe results. After that, a technology-mapped *netlist* is generated using an electronic design automation tool. The *netlist* can then be fitted to the actual FPGA architecture using a process called *place-and-route*, usually performed by the FPGA company's proprietary *place-and-route* software. Then the *netlist* is translated to a gate level description where simulation is repeated to confirm the synthesis proceeded without errors. The user will validate the map, *place-and-route* results via timing analysis, simulation, and other verification methodologies. Once the design and validation process is complete, the binary file generated is used to reconfigure the FPGA. The file is then transferred to the FPGA via a serial interface known as JTAG or to an external memory device like an EEPROM.

2.6.3 Programming Language for FPGA

The most common HDLs are VHDL and Verilog. Although, complexities of using HDLs has been compared to the equivalent of assembly languages. There are attempts ongoing to raise the abstraction level through the introduction of alternative languages. National Instrument introduced LabVIEW which is a graphical programming, has an FPGA add-in

module available to target and program FPGA hardware. The LabVIEW approach drastically simplifies the FPGA programming process. To simplify the design of complex systems in FPGAs, there exist libraries of predefined complex functions and circuits that have been tested and optimized to speed up the design process.

2.6.4 Advantages of FPGAs

- **High Performance**—By taking advantage of hardware parallelism, FPGAs exceeds the computing power of digital signal processors (DSPs) by accomplishing more instructions per clock cycle. By controlling I/O at the hardware level FPGA provides faster response times.
- **Less Time to Market**—FPGA technology offers flexibility and rapid prototyping capabilities which deals less time-to-market concerns. Developer can test an idea or concept and verify it in hardware without going through the long fabrication process of custom ASIC design.
- **Low Cost**—By using FPGA, designer has no fabrication costs or long lead times for assembly. System requirements are often changed over time, the cost of making incremental changes to FPGA designs is negligible when compared to the large expense of redesigning an ASIC.
- **Reliability**—In processor-based systems, only one instruction can be executed at a time and those are continually at risk of time-critical tasks blocking one another. FPGAs, which do not use OSs, minimize reliability concerns with true parallel execution and deterministic hardware dedicated to every task.
- **Long-term Maintenance**—FPGA chips are field-upgradable and do not require the time and expense involved with ASIC redesign. ASIC-based interfaces may cause maintenance and forward-compatibility challenges due to the change in specifications. Being reconfigurable, FPGA chips can keep up with future modifications that might be necessary.

CHAPTER 3

Design and Implementation

3.1 Introduction

This chapter discusses the details of the design and implementation of the proposed system. Details of various functional blocks as well as their FPGA implementation will be described also.

3.2 Architecture of the Design

Figure 3.1 shows the architecture of the system along with their internal connection and relations.

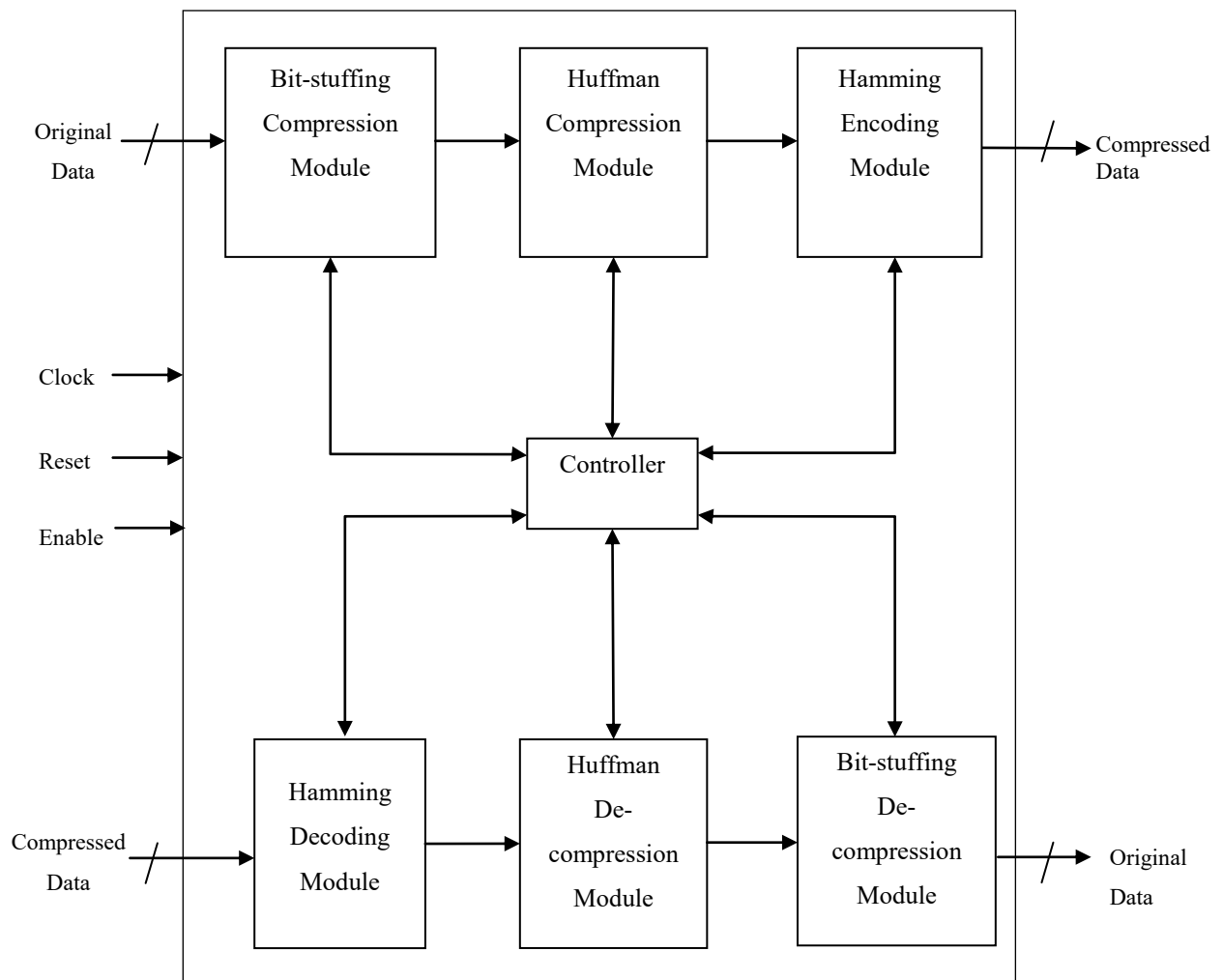


Figure 3.1 Archietecture of the Proposed System

3.2.1 Controller:

Both transmission and receiver unit have distinct controller modules. The main feature of a dynamic Huffman coder is that the histogram is calculated from the input data. The purpose of this module is to take input, processing data by enabling other modules and store output result into memory. After performing necessary operation, the data will be written into given memory location. The main purpose of this module is to co-ordinate different modules in transmission or receiver unit to process data and generates output. A simple block diagram of controller module is shown in Figure 3.2.

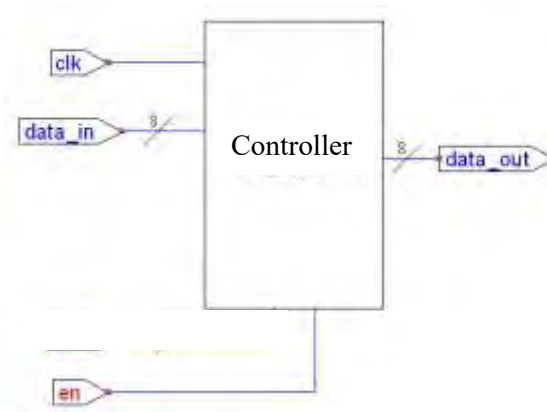


Figure 3.2 : Block Diagram of Controller

3.2.2 Bit-stuffing Compression Module:

This module performs first level compression on transmission unit. It operates on byte level. It removes MSBs from each input character and continues until the end of byte stream. On negative 'rst' signal, internal registers of this module will be reset. Then, on positive edge 'clk' cycle, this will load data on its internal register and operates on the data and write output to temporary register. Then output will be sent through 'out' port. Block diagram is shown on Figure 3.3.

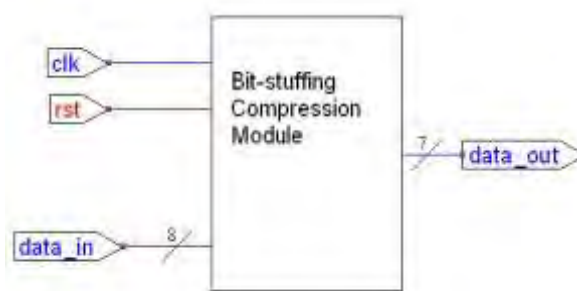


Figure 3.3 Block Diagram of Bit-stuffing Compression Module

3.2.3 Huffman Compression Module

This module is responsible for second level compression in transmission unit. At first, it computes the frequency of the input characters. Based on the frequency of input data set, it constructs Huffman tree. Later, this Huffman tree will be used to encode each character of the source data. It uses dynamic Huffman algorithm for compressing input text data.

Huffman compression module is the most complex module of this project. It is divided into several sub-modules which will be described in following sections.

3.2.3.1 7-Bit Adder

Adder module takes two 7 bit numbers— ‘NUM_1’ and ‘NUM_2’ and perform addition. Basically this addition is required to build Huffman tree where two least frequency will be summed up and added as a new node. Result of the addition is another 7 bit number which will be sent through the port ‘SUM’. Figure 3.4 shows the block diagram of this adder.

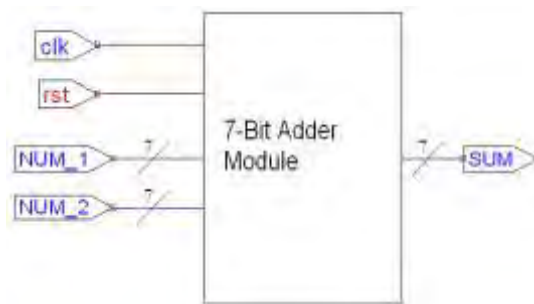


Figure 3.4 Block Diagram of 7-Bit Adder

3.2.3.2 Character Counter

The output of Bit-stuffing compression module acts as input of Character Counter Module. This module takes bit-compressed data into unpacked array ‘stringIn’ along with the ASCII value of each character into packed array ‘characters’. The size of ‘stringIn’ depends on the output size of bit-stuffing module. For example 104 bits input ASCII data generates 90 bits output from bit-stuffing module which is the input for character counter. Generated frequency of each input symbol i.e. histogram is saved into ‘freq’ which is another packed array. Each row of ‘freq’ denotes each character and corresponding frequency is saved into the columns. A block diagram of character counter is shown in Figure 3.5.

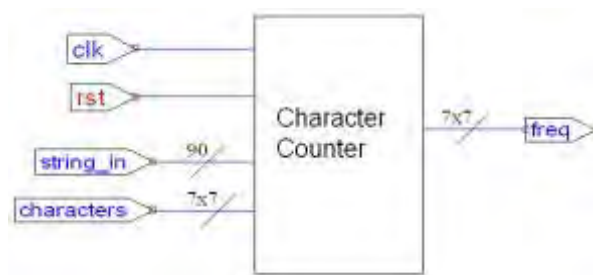


Figure 3.5 Block Diagram of Character Counter

3.2.3.3 Frequency Sorting Module

This module takes the frequencies from character counter and sorts them in ascending order. It applies bubble sorting method on frequencies 'freq' and sends sorted values in 'sorted' as shown in Figure 3.6. Characters associated with each frequency are passed in unpacked array 'charsIn' and sorted characters are sent to 'charsOut'. That means 'sorted[0]' is the frequency of the character 'charsOut[0]', 'sorted[1]' is the frequency of the character 'charsOut[1]' and so on.

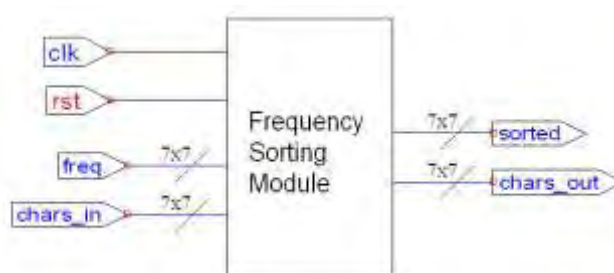


Figure 3.6 Block Diagram of Frequency Sorting Module

3.2.3.4 Node Sorting Module

Sorting module is same as frequency sorting module with the difference that it takes an extra input 'id_in' as the associate values for each frequency. This module is used for intermediate sorting of nodes when building Huffman binary tree. 'id_in' is a packed array which signifies identity of each node in the Huffman tree. Block diagram is shown in Figure 3.7.

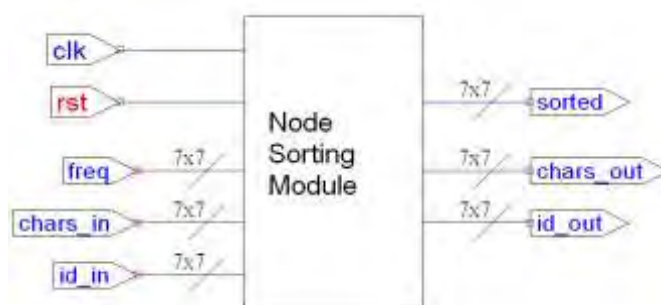


Figure 3.7 Block Diagram of Node Sorting Module

3.2.3.5 Huffman Tree Generator

This module builds Huffman binary tree to generate Huffman codes based on frequency distribution of input character set. It uses 7-bit adder and node sorting module to build the tree. It takes sorted frequency list 'freq', characters 'chars' and id set 'id' as input. 'id' contains identity of each primary nodes. Each node in constructed Huffman tree is stored in a 24-bit register. The information stored in each node mentioned in Table 3.1 and block diagram is shown in Figure 3.8.

Table 3.1 Node Structure

Bit Position	Description	Value Range
0 to 6	Character/Symbol	0-127
7 to 14	Node ID	0-255
15	Denotes if this is left or right child.	0: Left 1: Right
16 to 23	Parent ID of the node	0-255

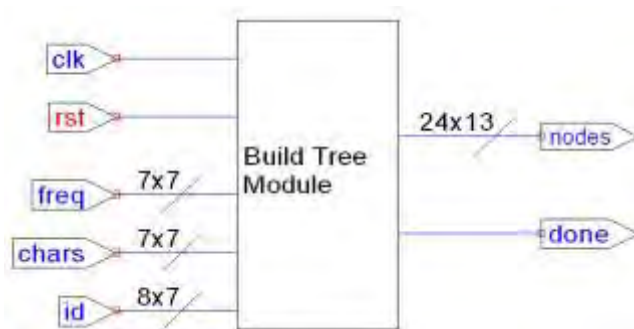


Figure 3.8 Block Diagram of Huffman Tree Generator

3.2.3.6 Code Generator

This module generates Huffman code for each input character by traversing Huffman tree. It takes character list 'characters' and Huffman tree 'node' as input. Generated codes are passed into output as 'code' with length of each code in 'codeLen'. Huffman codes are variable length code. So length of each code needs to be stored in separate registers which has been done by 'codeLen'. Codes are generated by traversing Huffman tree starting from leaf node to the root. This is done by implanting recursive function. Figure 3.9 shows the block diagram of this module.

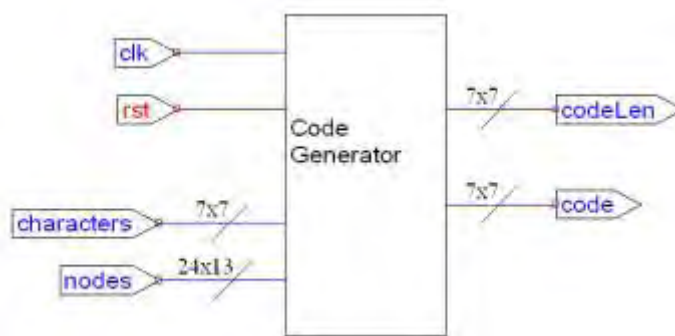


Figure 3.9 Block Diagram of Code Generator

3.2.3.7 Coding Module

This module finally compresses input data by replacing each character with corresponding Huffman code created in code generator module. It takes input data 'in', character list 'chars', generated code list 'code' and length of the each code 'codeLen' as input. Compressed data is sent through port 'out' with length of compressed data in 'length'. Figure 3.10 shows the block diagram of this module.

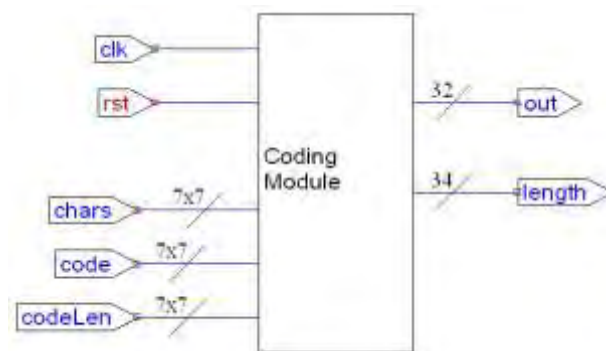


Figure 3.10 Block Diagram of Coding Module

3.2.4 Hamming Encoding Module

Hamming encoding module is responsible for secured transmission of text data. Huffman compressed data is sent to this module as input 'in'. Input data is then divided into groups of 4-bit data. Each 4-bit data generates a Hamming (7, 4) code-word with extra 3 parity bits. If input contains extra bits i.e. input length are not multiple of 4 then remaining bits are remain unchanged. Each code-word is written into memory module as final output 'out' and is sent to the receiver. The block diagram of this module is shown in Figure 3.11.

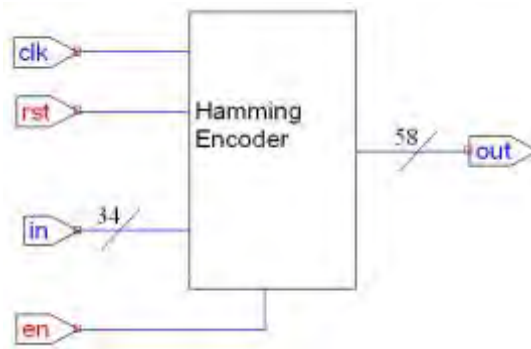


Figure 3.11 Block Diagram of Hamming Encoder

3.2.5 Hamming Decoding Module

Processing of received data in receiver unit begins with Hamming Decoding Module. Input data is passed through 'in' port. This module detects and corrects single bit error in each received code-word. It calculates check bits from each 7-bit code-word and corrects the error if any. It sends corrected data to 'out' port. If any error is found, 'error' signal will be set high and error bit number is sent to 'error_index'. Figure 3.12 shows the block diagram.

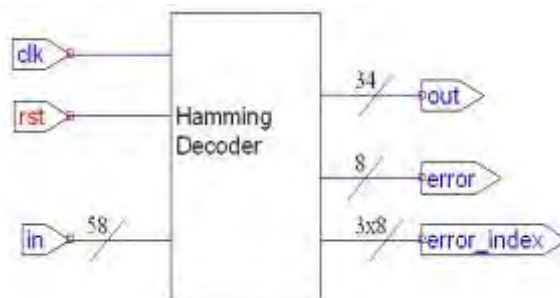


Figure 3.12 Block Diagram of Hamming Decoder

3.2.6 Huffman Decompression Module

Hamming decoded compressed data is taken as input 'in' for this module. Character list from Huffman compression module — 'chars', Huffman code list 'code' and length of each code 'codeLen' are sent to this module as other inputs. Then it identifies Huffman codes in input data and replaces those codes with original characters found in 'chars'. Decompressed data is sent to 'out' with decompressed data length 'outLen'. Figure 3.13 shows the block diagram of this module.

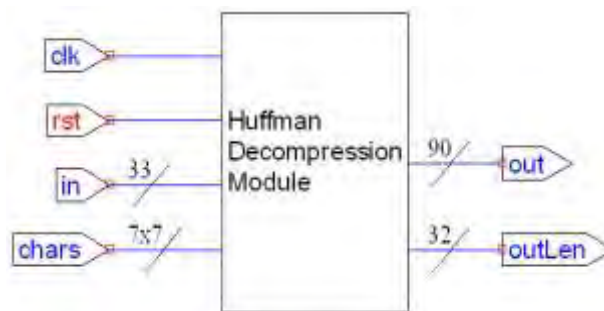


Figure 3.13 Block Diagram of Huffman Decompression Module

3.2.7 Bit-stuffing Decompression Module

This module performs the final decompression on received data. It performs the reverse process of Bit-stuffing Compression Module. It takes input 'in' from Huffman decompression module and adds '0' bit as an MSB to every 7-bit data blocks. Final output is sent to 'out'. Output of this module should be identical to original data in transmission unit which will prove the correctness of this system. Block diagram is shown in Figure 3.14.

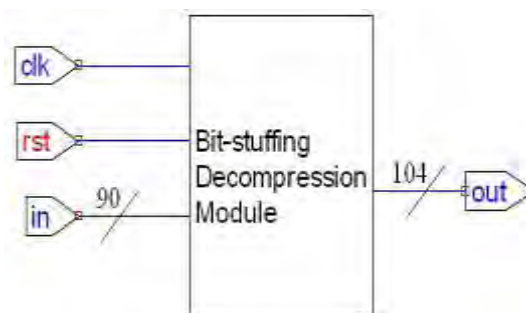


Figure 3.14 Block Diagram of Bit-stuffing Decompression Module

3.3 Flow Chart of the Design

Fig 3.15 shows the flowchart of the proposed FPGA based system.

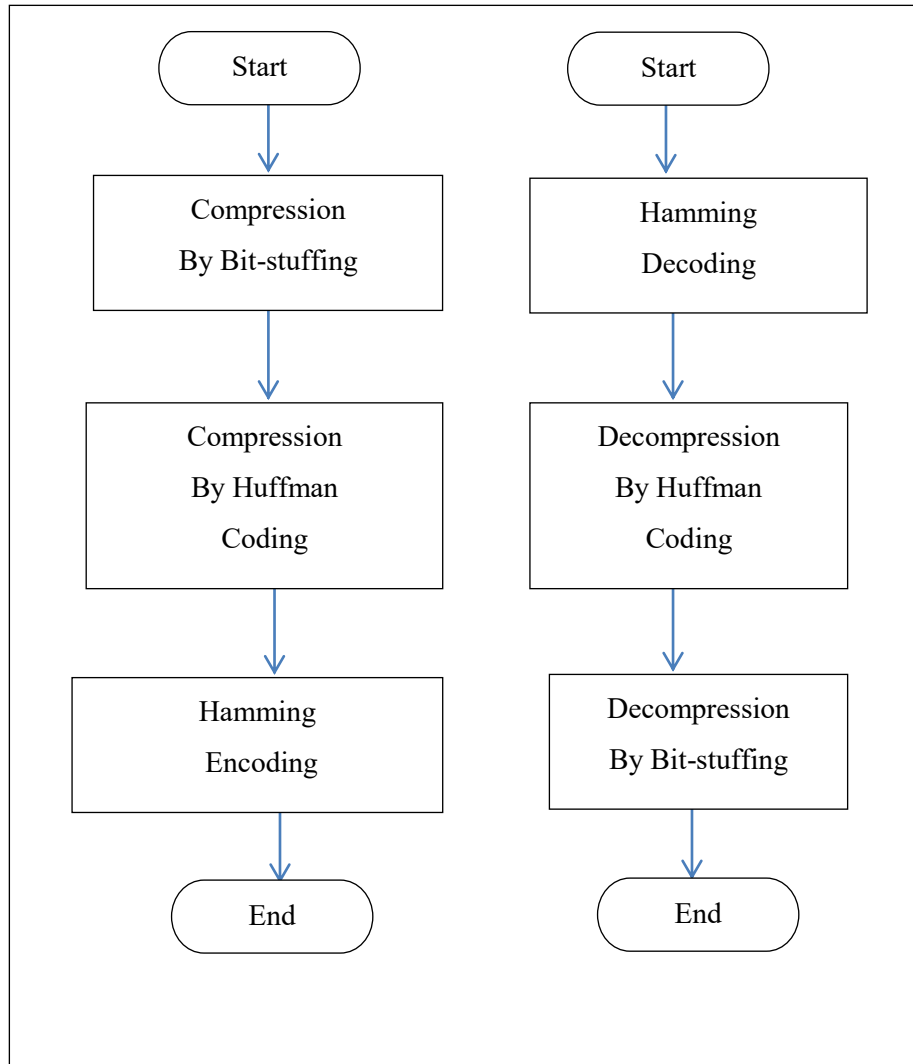


Figure 3.15: Flowchart of the Proposed FPGA based System

3.4 Algorithm of the Proposed Method

The proposed method uses bit stuffing and Huffman algorithm to do a two level compression to achieve higher compression ratio and then uses Hamming (7, 4) coding for reliable and secured transmission of text data. Using of Hamming code enables it to detect and correct single bit error when data is sent via noisy channel.

3.4.1 Transmission Unit

The process begins with two level compressions of input text data. First compression is done by bit stuffing and second level compression is done by Huffman coding. After completing two level compressions, generated output is encoded by Hamming (7, 4) code to achieve further security.

Figure 3.16 shows process flow of transmission unit

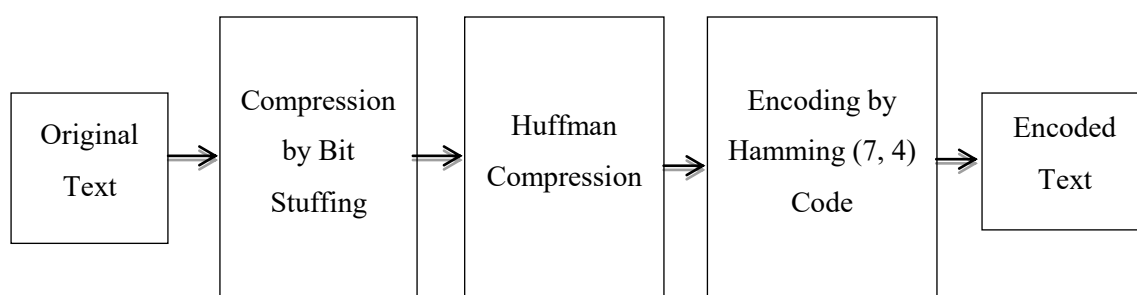


Figure 3.16: Process Flow at Transmission Unit

3.4.1.1 Level 1 Compression by Bit Stuffing

Input: Original text data.

Output: First level compressed data.

- Step—1. Load the text data into memory and initialize counter, $c = 0$.
- Step—2. Read each bit starting from LSB position into the bit-stuffing compression module and increment the counter.
- Step—3. If $c = 7$, write the bits back into register and reset the counter. Thus MSB is discarded from each character from the input data.

3.4.1.2 Level 2 Compression by Huffman Coding

Input: First level compressed data.

Output: Second level Huffman compressed data.

- Step—1. Count the frequencies of the symbols from the input and save the frequency information along with the symbols.
- Step—2. Build a Huffman tree based on the frequency information. Used Huffman algorithm is described next.

Huffman (C)

- I. $n \leftarrow |C|$
- II. $Q \leftarrow C$
- III. for $i \leftarrow 1$ to $n-1$
- IV. do allocate a new node z
- V. $\text{left}[z] \leftarrow x \leftarrow \text{EXTRACT-MIN}(Q)$
- VI. $\text{right}[z] \leftarrow y \leftarrow \text{EXTRACTMIN}(Q)$
- VII. $f[z] \leftarrow f[x] + f[y]$
- VIII. $\text{INSERT}(Q, x)$
- IX. return $\text{EXTRACT-MIN}(Q)$

Where,

- C is a set of n symbols.
- $f[c]$ denotes the frequency of c .
- Q is a min-priority queue which is used to identify the two least frequent symbols to merge together. A new symbol is created after merging whose frequency is the sum of the frequencies of the two symbols that were merged.
- $\text{EXTRACT-MIN}(x)$ is the function which returns minimum value from the queue.
- $\text{INSERT}(Q, x)$ adds value 'x' in the min-priority queue 'Q'.

3.4.1.3 Hamming (7, 4) Encoding

Input: Compressed text data.

Output: Hamming (7, 4) encoded output data.

- Step—1. Read 4 bits from compressed data and calculate 3 parity bits namely P_1 , P_2 and P_4 .
- Step—2. Insert P_1 , P_2 and P_4 into bit position 1, 2 and 4. Remaining positions (3, 5, 6 and 7) are filled up by data bits read in step 1.
- Step—3. Write resulting Hamming (7, 4) code into register.
- Step—4. Repeat steps 1-3 until the end of input compressed data.

3.4.2 Receiver Unit

Processing of receiver unit begins with Hamming (7, 4) decoding of received data. It calculates check bits to detect if any error occurs during the transmission. Hamming (7, 4) code can correct single bit error. After decoding, output data is decompressed by Huffman decompression module using Huffman codes. A second level decompression of Huffman decompressed output is required using bit stuffing decompression process.

Figure 3.17 shows different stages of the process at receiver unit.

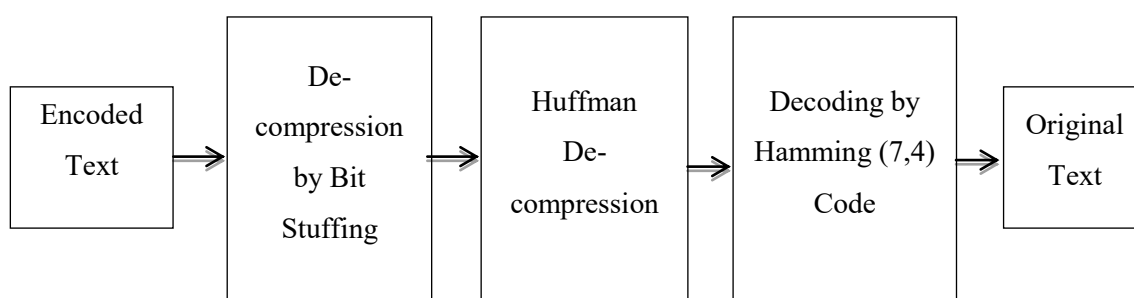


Figure 3.17: Process Flow at Receiver Unit

3.4.2.1 Hamming (7, 4) Decoding

Input: Received encoded compressed data.

Output: Hamming (7, 4) decoded compressed data.

- Step—1. Read 7 bits from the encoded received data. This is a Hamming (7, 4) code-word.
- Step—2. Calculate check bits and create a 3 bit binary number with check bits to detect single bit errors during transmission. Correct the error bit if found by flipping it.
- Step—3. Extract data bits from the position 3, 5, 6 and 7. Save the data bits into memory.
- Step—4. Repeat steps 1-3 until end of input encoded data.

3.4.2.2 Level 1 Decompression by Huffman Algorithm

Input: Hamming (7, 4) decoded compressed data.

Output: First level decompressed text data.

- Step—1. Read input character list and generated Huffman codes from transmission unit.
- Step—2. Read a bit from input data and check Huffman code list to find a match.
 - a. Go back to step-2 if there is no match found and concatenate new bit to the previous one.
 - b. If match found in step-2, find corresponding character from character list. Write the character into output register.
- Step—3. Continue step 2-3 until whole input bit stream is processed.

3.4.2.3 Level 2 Decompression by Bit Stuffing

Input: First level decompressed data.

Output: Original text data.

- Step—1. Initialize counter $c = 0$ and load data into bit-stuffing decompression module.
- Step—2. Read a bit starting from LSB position and increment the counter.
- Step—3. If $c = 7$, add a 0 as MSB and write back into memory.
- Step—4. Repeat steps 1-3 until end of the input.

3.5 MATLAB Simulation

Software implementation and intensive testing of the proposed method have been done using MATLAB. Proposed method has been implemented in popular programming language C and DLL (Dynamic Link Library) has been generated. Then this DLL is imported to MATLAB project and simulated with varying text data ranging from 100B to 8MB.

There are 6 core functions in software implementation. Function signature and description is listed on Table 3.2.

Table 3.2 Functions of Matlab Simulation

Function Signature	Description
BitCompression(string inputFile, string outputFile)	It takes input as 'inputFile' and returns bit compressed output in 'outputFile'.
HuffmanCompression(string inputFile, string outputFile)	It takes input from the output of BitCompression() in 'inputFile'. Then it compresses the data using Huffman algorithm and returns compressed output in 'outputFile'.
EncodeFileWithHamming(string inputFile, string outputFile)	This function takes input from HuffmanCompression(). After that it encodes data using Hamming (7, 4) coding and return result in 'outFile'.
DecodeFileWithHamming(string inputFile, string outputFile)	This function starts processing at receiver unit by taking input into 'inputFile' and decodes Hamming(7,4) code-words. If there are any single bit errors, it corrects those error saves output in 'outFile'.
HuffmanDecompression(string inputFile, string outputFile)	After Hamming decoding done, this function does Huffman decompression on 'inputFile' and sends it to 'outFile'.
BitDecompression(string inputFile, string outputFile)	This is the final function on receiver unit which does reverse process of BitCompression(). After decompressing this reconstruct the original data saves in 'outFile'.

3.6 Design of the System

FPGA design of the proposed system has been done using Verilog HDL. Different functional blocks of the implementation have already been described in section 3.2. A simplified description of each Verilog module is listed in Table 3.3.

Table 3.3 Modules of FPGA Implementation

Verilog File Name	Description
bit_compress.v	Bit-stuffing compression module
bit_decompress.v	Bit-stuffing decompression module
cotroller.v	Controller module for transmission unit. It interconnects different modules and generates final output from transmission unit.
humming_encoder.v	Hamming(7,4) encoder implementation
hamming_decoder.v	Hamming(7,4) decoder implementation
add_num.v	Add two 7 bit numbers and return result in SUM(7 bit)
count_characters.v	Character frequency counter.
sort_frequencies.v	Sort initial character frequency array in ascending order.
build_tree.v	Huffman binary tree builder.
sort2~6.v	Sort intermediate character frequency during Huffman tree build process.
generate_code.v	Huffman Code generator
coding.v	Compress input using generated Huffman Codes.
huffman_decoder.v	Decompress input data according to Huffman algorithm.

3.7 Software Simulation Tool—MATLAB

MATLAB (matrix laboratory) is a multi-paradigm numerical computing environment. A proprietary programming language developed by the company named MathWorks, MATLAB allows matrix manipulations, plotting of functions and data, implementation of algorithms, creation of user interfaces, and interfacing with programs written in other

languages, including C, C++, C#, Java, Fortran and Python. Although MATLAB is planned mainly for numerical computing, an optional toolbox uses the MuPAD symbolic engine, allowing access to symbolic computing abilities. An additional package, Simulink, adds graphical multi-domain simulation and model-based design for dynamic and embedded systems.

In this project, MATLAB 2016a edition is used for software implementation and simulation.

3.8 FPGA Simulation Tool— ModelSim

ModelSim is a multi-language HDL simulation environment by Mentor Graphics, for simulation of hardware description languages such as VHDL, Verilog and SystemC and includes a built-in C debugger. ModelSim can be used independently, or in conjunction with Altera Quartus or Xilinx ISE. Simulation is performed using the graphical user interface (GUI), or automatically using scripts.

ModelSim is offered in multiple editions, such as ModelSim PE, ModelSim SE, and ModelSim XE. ModelSim SE offers high-performance and advanced debugging capabilities, while ModelSim PE is the entry-level simulator for hobbyists and students. ModelSim SE is used in large multi-million gate designs, and is supported on Microsoft Windows and Linux, in 32-bit and 64-bit architectures. ModelSim XE stands for Xilinx Edition, and is specially designed for integration with Xilinx ISE. ModelSim XE enables testing of HDL programs written for Xilinx Virtex/Spartan series FPGA's without needed physical hardware. ModelSim can also be used with MATLAB/Simulink, using Link for ModelSim. Link for ModelSim is a fast bidirectional co-simulation interface between Simulink and ModelSim. For such designs, MATLAB provides a numerical simulation toolset, while ModelSim provides tools to verify the hardware implementation and timing characteristics of the design.

ModelSim enables simulation, verification and debugging for the following languages:

- VHDL
- Verilog
- Verilog 2001
- System Verilog
- PSL
- SystemC

In this project, ModelSim SE-64 10.5 version has been used for FPGA designing, implementation and simulation.

CHAPTER 4

Results and Discussions

4.1 Introduction

The proposed method has been evaluated by considering text data of varying size. The model is effective in providing high level reliability by correcting single bit errors during transmission and higher saving percentage. It has been simulated in the MATLAB environment to ensure the accuracy of the system and then it is designed using Verilog HDL and designed for FPGA hardware. The results show high level reliability and reduction of memory which in turn produces reduction of bandwidth and transmission time.

4.2 Software Simulation

Software simulation of this project has been done in MATLAB.

The specifications of simulation environment is given below-

MATLAB 2016a, 64-bit

OS: Windows 7, 64-bit

RAM: 4GB

Processor: Core i3 2.30GHz

Figure 4.1 shows a sample of software simulation result for various input file size.

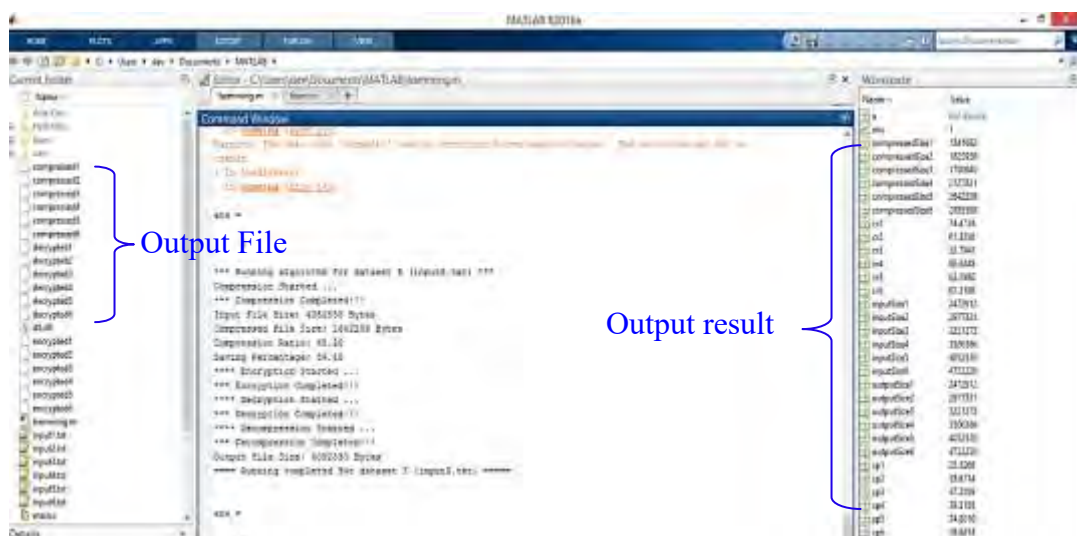
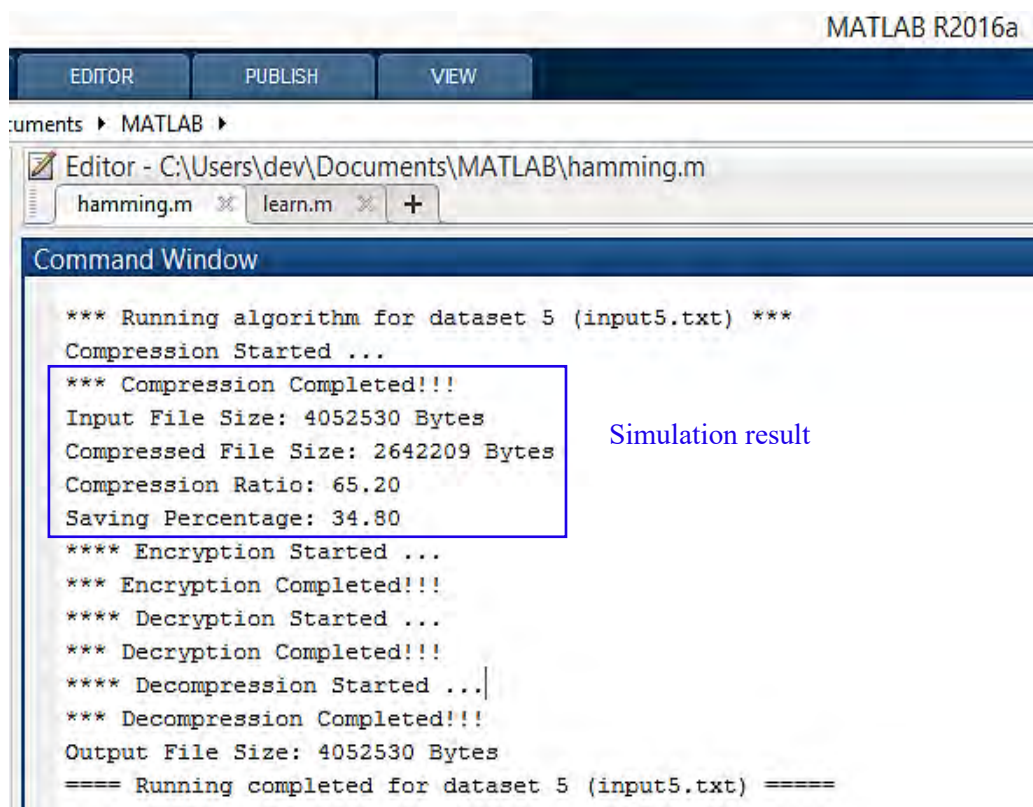


Figure 4.1 Sample of MATLAB Simulation

Figure 4.2 shows a sample result of compression in MATLAB simulation. In this particular software simulation input file size was about 4MB, generated compressed file size was about 2.6 MB and compression ratio was 65.20.



```

MATLAB R2016a
EDITOR PUBLISH VIEW
Documents > MATLAB >
Editor - C:\Users\dev\Documents\MATLAB\hamming.m
hamming.m learn.m +
Command Window
*** Running algorithm for dataset 5 (input5.txt) ***
Compression Started ...
*** Compression Completed!!!
Input File Size: 4052530 Bytes
Compressed File Size: 2642209 Bytes
Compression Ratio: 65.20
Saving Percentage: 34.80
**** Encryption Started ...
*** Encryption Completed!!!
**** Decryption Started ...
*** Decryption Completed!!!
**** Decompression Started ...|
*** Decompression Completed!!!
Output File Size: 4052530 Bytes
===== Running completed for dataset 5 (input5.txt) =====
Simulation result

```

Figure 4.2 A Sample Simulation Result on MATLAB

4.3 Generated Output File of SW Simulation

Proposed method has been tested with various file size from 100 bytes to 8M bytes. Simulation result on each simulation has been found satisfactory.

Figure 4.3 shows sample input text data which has been compressed and encoded with proposed method.

Figure 4.4 shows generated output file after encoding process. This exhibits that proposed method is also capable for adding an extra layer of security by performing triple encoding on original data.

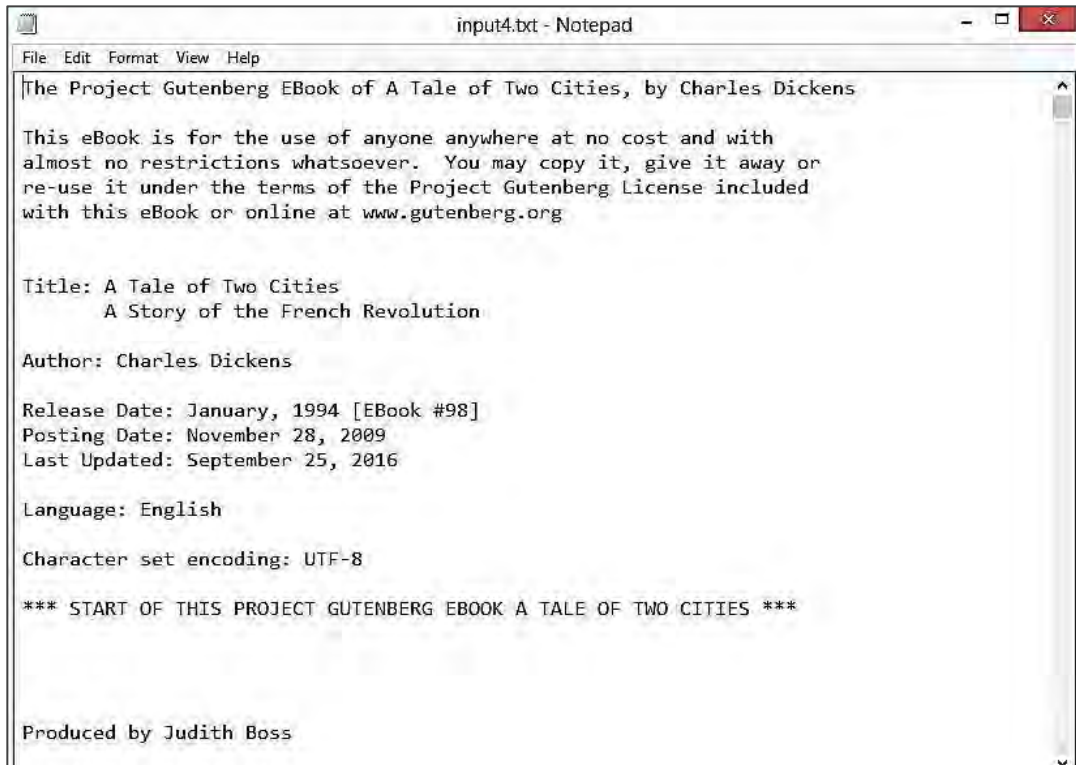


Figure 4.3: Sample Input Text File (Partial View)



Figure 4.4: Generated Output File in Transmission Unit. (Partial View)

4.4 Compilation of the FPGA Design

Mentor Graphics was the first to combine single kernel simulator (SKS) technology with a unified debug environment for Verilog, VHDL, and SystemC. The combination of industry-leading native SKS performance with the best integrated debug and analysis environment make ModelSim the simulator of choice for both ASIC and FPGA designs.

Screenshot of compilation summary Figure 4.5 shows successful compilation of the project.

The figure shows two screenshots from the ModelSim environment. The top screenshot is a 'Compile Report Summary' window for 'ss_tb.v'. It shows the compilation of 'bit_decompress_tb' and 'controller' modules. The bottom screenshot is a 'Transcript' window showing a list of 32 successful compilation messages for various testbenches and modules, followed by the summary: '# 32 compiles, 0 failed with no errors.' The ModelSim prompt is visible at the bottom of the transcript window.

```

M Compile Report Summary
ss_tb.v
Model Technology ModelSim SE-64 vlog 10.5 Compiler 2016.02 Feb 13 2016
-- Compiling module bit_decompress_tb

Top level modules:
    bit_decompress_tb

vlog -work work -vopt -sv -stats=none F:/MSc/Project/Verilog_Project/controller.v
Model Technology ModelSim SE-64 vlog 10.5 Compiler 2016.02 Feb 13 2016
-- Compiling module bit_compress
-- Compiling module count_characters
-- Compiling module sort_frequencies
-- Compiling module build_tree
-- Compiling module generate_code
-- Compiling module coding
-- Compiling module hamming_encoder
-- Compiling module controller

Top level modules:
    controller

vlog -work work -vopt -sv -stats=none F:/MSc/Project/Verilog_Project/controller_tb.v
Model Technology ModelSim SE-64 vlog 10.5 Compiler 2016.02 Feb 13 2016
-- Compiling module controller_tb

Transcript
# Compile of count_characters_tb.v was successful.
# Compile of generate_code_tb.v was successful.
# Compile of sort_frequencies_tb.v was successful.
# Compile of hamming_decoder_tb.v was successful.
# Compile of hamming_encoder_tb.v was successful.
# Compile of bit_compress_tb.v was successful.
# Compile of bit_decompress_tb.v was successful.
# Compile of controller_tb.v was successful.
# Compile of add_num.v was successful.
# Compile of build_tree.v was successful.
# Compile of count_characters.v was successful.
# Compile of generate_code.v was successful.
# Compile of sort_frequencies.v was successful.
# Compile of coding.v was successful.
# Compile of hamming_decoder.v was successful.
# Compile of hamming_encoder.v was successful.
# Compile of bit_compress.v was successful.
# Compile of bit_decompress.v was successful.
# Compile of controller.v was successful.
# 32 compiles, 0 failed with no errors.

ModelSim>
Project : hhb_verilog

```

Figure 4.5 Result of Compilation on ModelSim

4.5 Analysis of FPGA Simulation Results

Each module of the FPGA design is simulated separately by implementing testbench. The project has also been simulated as a whole. In either simulator, results have been found as expected and satisfactory which proves the correctness and efficiency of the design. Following sections describe simulation result for each module of the system.

To analyze FPGA simulation let us consider the string “**hip hop happy**” as input. This string has total 13 characters and total 104 bits. Frequency of the characters with corresponding ASCII[22] values are shown in Table 4.1.

Table 4.1 Frequency Distribution and ASCII values of a Test String

Char	Frequency	ASCII
h	3	01101000
a	1	01100001
p	4	01110000
y	1	01101001
i	1	01010011
o	1	01100111
space	2	00100000

If ASCII values of the test string are considered for data transmission, then 104 bits have to be sent. But two level compressions of the proposed method reduce total bits of the test string to 34 bits.

4.5.1 Simulation of Bit-Stuffing Compression Module

Simulation of the input of 104 bits results in reduction of 13 bits as shown in Figure 4.6. When ‘clk’ is on positive edge, data is loaded into this module. From timing diagram it can be observed that output time is 1 ns.¹

Here original data is 104'h68697020686f70206861707079 and compressed output data is 91'h68d3c1068dfc1068c3c3879.

¹ 1 ns=10⁻⁹s

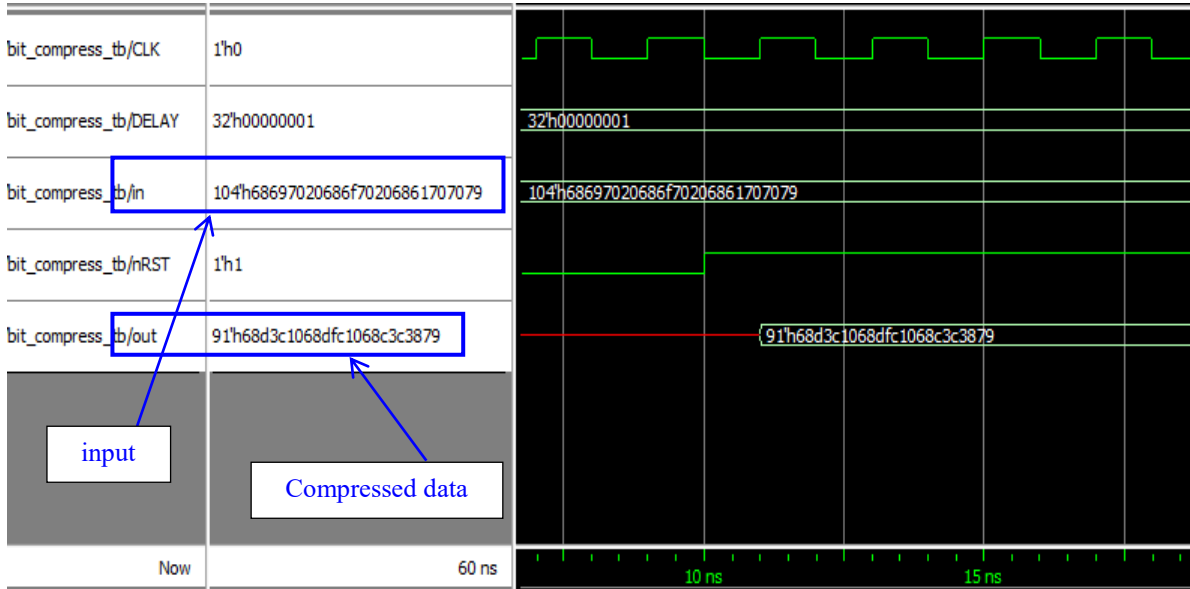


Figure 4.6 Simulation of Bit-stuffing Compression Module with 104 bits Input Data.

4.5.2 Simulation of Character Counter

Character counter is a part of Huffman compression module. For the input on port ‘CHARACTER_IN’, counted frequency is shown on FREQUENT_OUT. The result of the simulation matches with frequency distribution mentioned in Table 4.1. This proves correctness of this module. Figure 4.7 shows the simulation results.

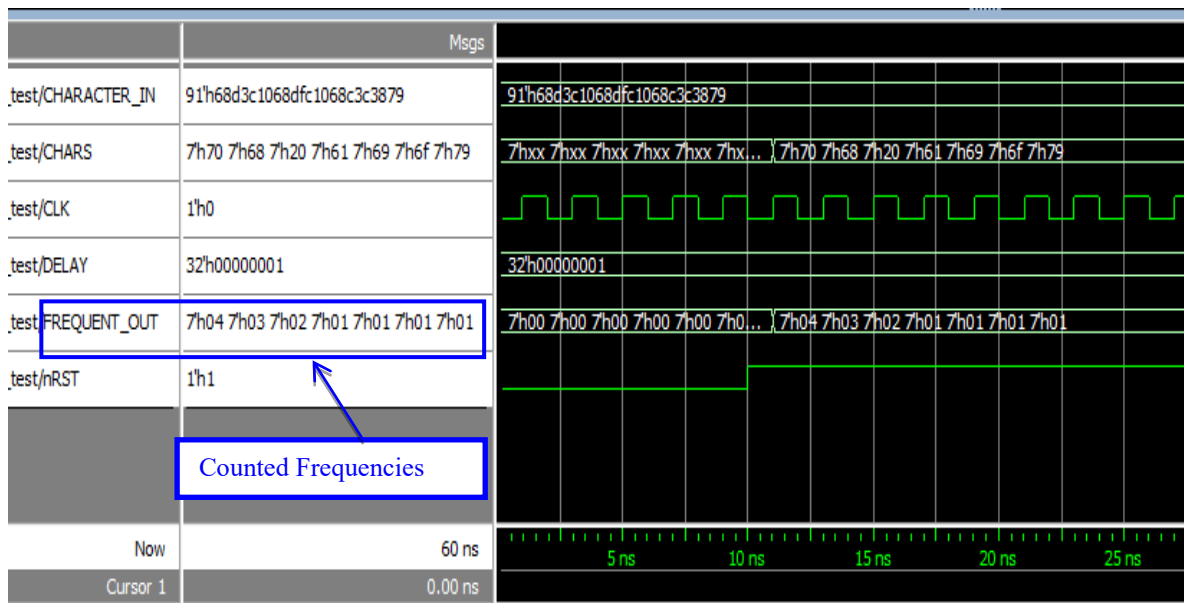


Figure 4.7 Simulation Result of Character Counter. Counted frequencies match with frequency distribution mentioned in Table 4.1.

4.5.3 Simulation of Frequency Sorting Module

This module sorts the character frequencies in ascending order. Input frequencies are given at port 'freq' and sorted output is sent to 'sorted' port. Figure 4.8 shows the sorting result.

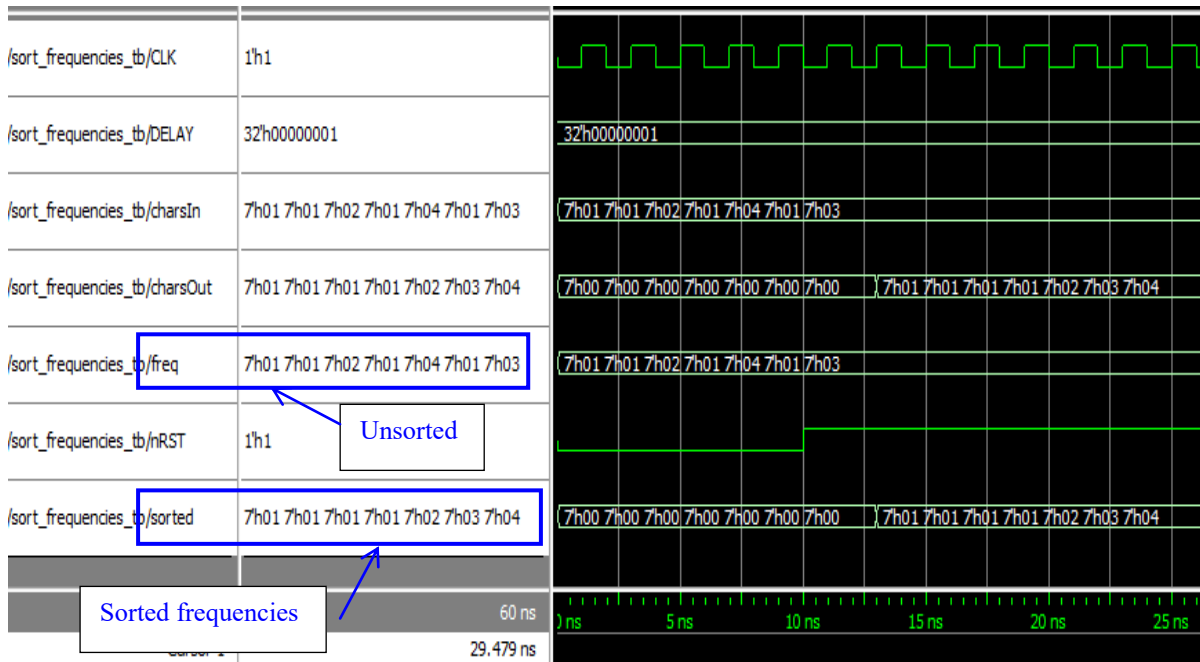


Figure 4.8 Simulation Result of Frequency Sorting Module.

4.5.4 Simulation of Huffman Tree Generator

This module builds Huffman tree based on sorted frequency list. Basically a priority queue has been implemented in this module with adder and sort modules. Generated tree is a collection of nodes where each node has a size of 24 bits. Output is passed to 'nodes'. Figure 4.9 shows the simulation result of this module.

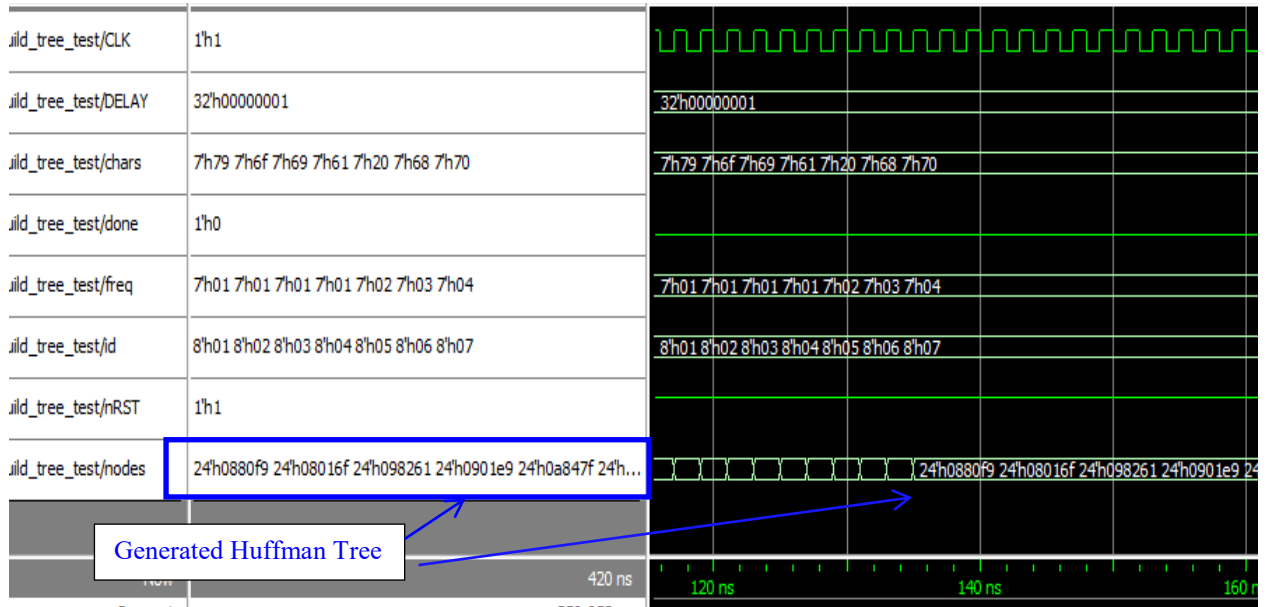


Figure 4.9 Simulation Result of Huffman Tree Generator

4.5.5 Simulation of Code Generator

It generates Huffman codes by traversing the input ‘nodes’. Generated codes are stored in ‘codes’ and length of each code is saved into ‘codeLen’. Figure 4.10 shows the simulation result. From simulation it can be observed that first code is 7’h0f and its code length is 04. According to the implementation, Huffman code is extracted from ‘code’ by copying the number of bits equal to code length starting from LSB. So the Huffman code is ‘1111’ after extracting from 7’h0f.

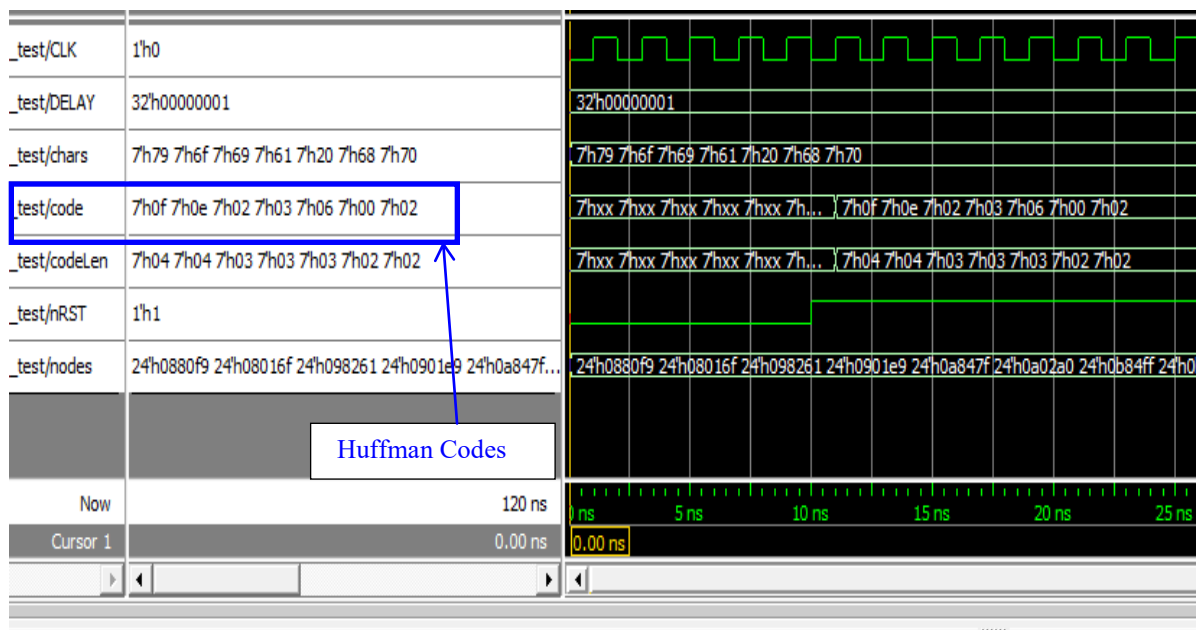


Figure 4.10 Simulation Result of Code Generator

4.5.6 Simulation of Coding Module

Coding module compresses the input string with generated Huffman codes. Compressed data is stored in 'compress'. Figure 4.11 shows the simulation result. It can be observed from the simulation that compressed size 34 for original input size of 104 bits.

Simulation results of figure and 4.6 and 4.11 show that saving or memory shrinkage percentage for current input data set is 67.30%.

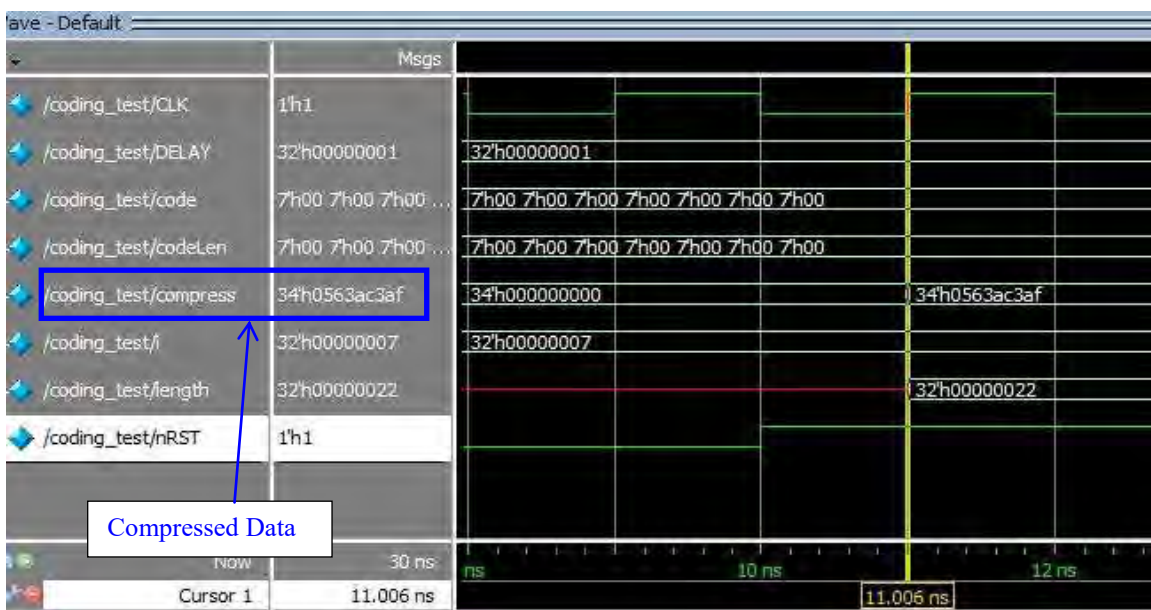


Figure 4.11 Simulation of Coding Module

4.5.7 Simulation of Hamming Encoding Module

Figure 4.12 shows the simulation results of Hamming Encoding module. It generates 58 bit data as output.

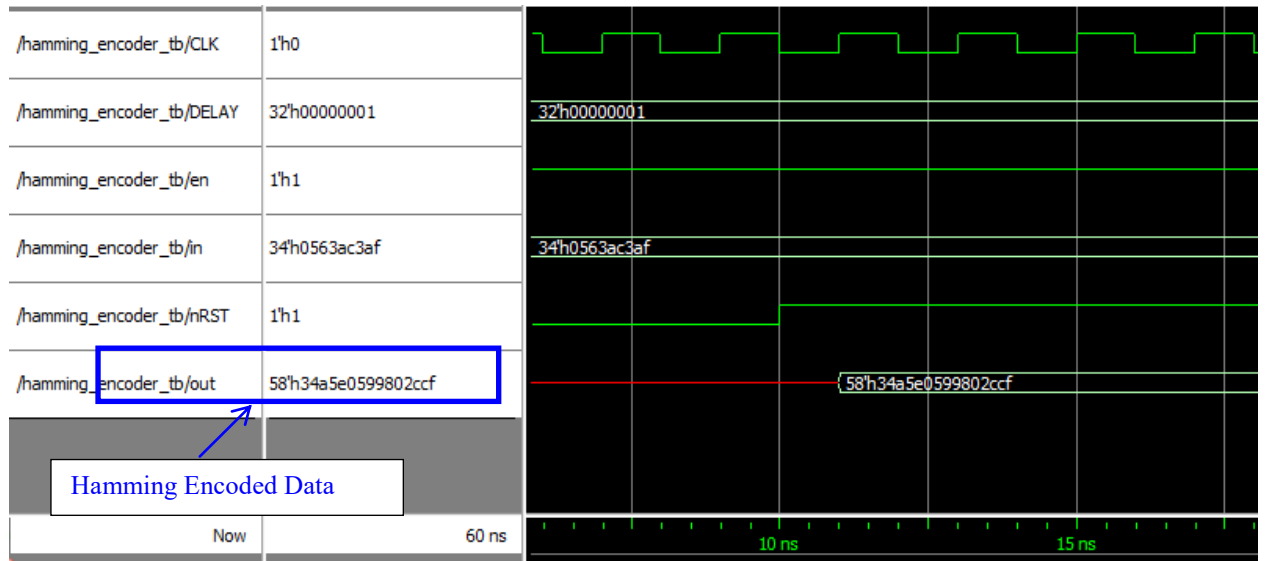


Figure 4.12 Simulation of Hamming Encoding Module

4.5.8 Simulation of Hamming Decoding Module

Processing of received data in receiver unit begins with Hamming Decoding Module. Input is received through 'in' port. This module detects and corrects single bit error in each received code-word. If any error is found, 'errorFound' signal will be high to show which received code-word has error. Error bit number is set high in 'error_index'. It sends corrected data to 'out' port.

For example, let's consider there is no error in received data. For our current simulation data set mentioned in Table 4.1, error free input to this module is

58'h34a5e0599802ccf.

So, all array indices of 'errorFound' and 'errorIndex' will be set to low. Output data is **34h0563ac3af** which is a Huffman compressed data and equal to the output of Figure 4.11. Simulation results shown in Figure 4.13 verify that.

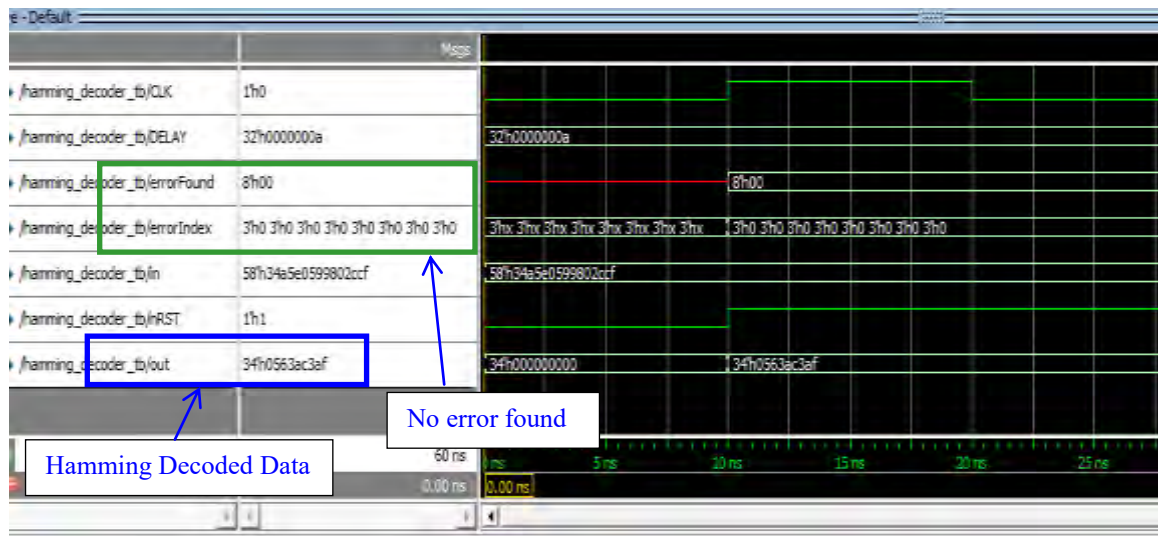


Figure 4.13 Simulation of Hamming Decoder Module for Error Free Input Data

Again, let's consider that there are 3 single bit errors in received data considering current simulation data mentioned in Table 4.1. Assume erroneous input data with 3 single bit errors— $58'h34b5e1599803ccf$, where error-free data should be $58'h34a5e0599802ccf$. For error free input data, output data is $34'h0563ac3af$ (Figure 4.13). This module can correct single bit error in each Hamming (7, 4) code-word. So, this erroneous input data will be corrected and 'errorFound' will be set high. Also 'errorIndex' will point to the bit positions where error occurred. Even though there are 3 single bit errors in received data, simulation results in Figure 4.14 show output is $34'h0563ac3af$ which is the same output for error-free data (Figure 4.13). So it corrects the errors in input data. This proves effectiveness of this module.

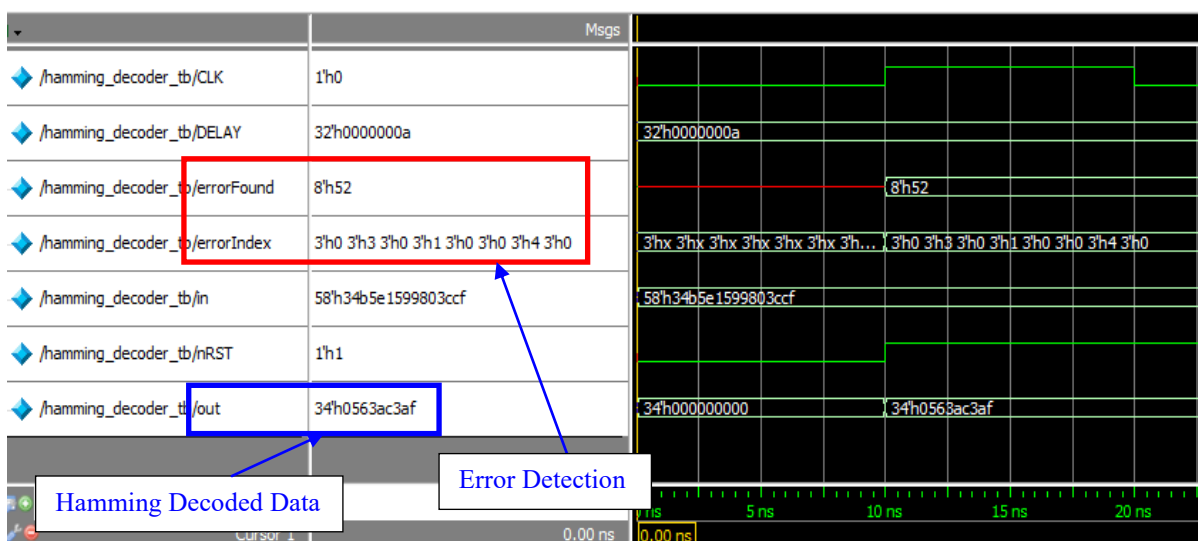


Figure 4.14 Simulation of Hamming Decoder Module for Erroneous Input Data

4.5.9 Simulation of Huffman Decompression Module

Hamming decoded compressed data will be taken as input 'in' for this module. Symbol list from Huffman compression module — 'chars', Huffman code list 'code' and length of each code 'codeLen' will be sent to this module as other inputs. For current data set for simulation output is generated as 91'h68d3c1068dfc1068c3c3879. This output value is same as the output value of Bit-stuffing compression value which proves the correctness of this module. Figure 4.15 shows the simulation results.

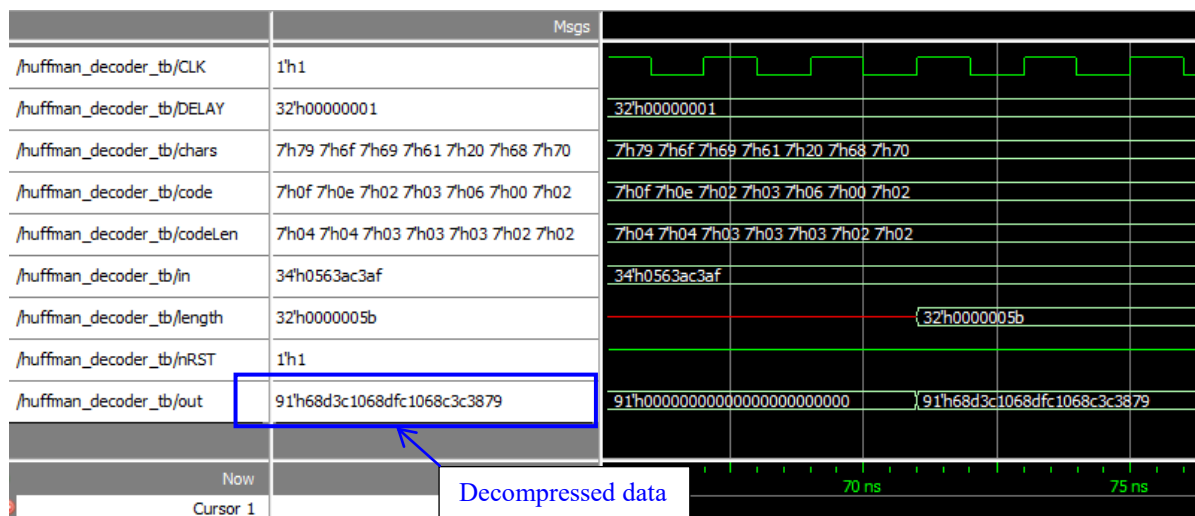


Figure 4.15: Simulation Result of Huffman Decompression Module

4.5.10 Bit-stuffing Decompression Module

This module performs final operation on the received data and reconstructs original data as output. It follows the reverse process of the bit compression module.

So, the input of 91 bits ('in') should generate original 104 bits back. Simulation shows that the output of this module is 104'h68697020686f70206861707079 which is the original data (ref. to Figure 4.6). So it verifies correctness and effectiveness of overall system. Figure 4.16 shows the simulation results.

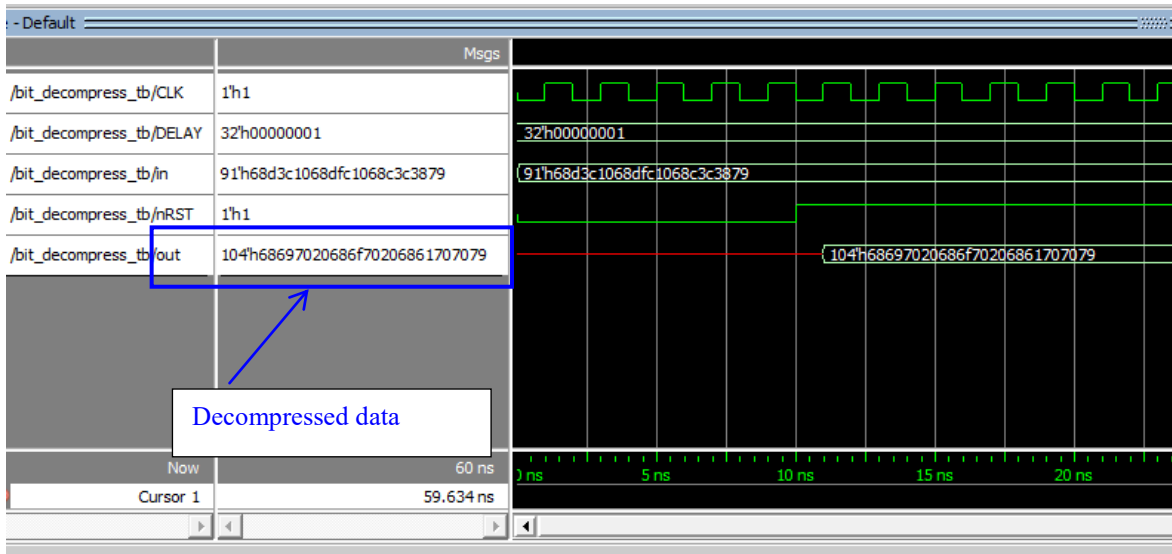


Figure 4.16: Simulation Result of Bit-stuffing Decompression Module

4.6 Error Correction Rate

To find the error correction rate of the implemented system, some random bits have been changed intentionally in received encoded file. The proposed mechanism corrects error almost 100% efficiently. Technically each input character is encoded into two (7, 4) Hamming code-word. Each code-word can correct a single bit error. So if 25% of receiving data get corrupted during transmission, implemented system can correct 100% of them.

Table 4.2 shows the result of error corrections of proposed system for an 8KB text data.

Table 4.2: Error Correction Rate of Received Data (800 bits)

Number of Corrupted Bits	Number of Corrected Bits	Error Rate	Error Correction Rate
49	49	6.125 %	100 %
53	53	6.625 %	100 %
126	125	15.75 %	99.20 %
165	161	20.63 %	97.58 %
193	191	24.13%	98.86%

4.7 Compression Ratio

Depending on the nature of the application there are various criteria to measure the performance of a compression system. When measuring the performance, the main concern would be the space efficiency. Following are some measurements used to evaluate the performances of lossless algorithms.

Compression Ratio (CR): The ratio between the size of the compressed data and the size of the original data.

$$CR = (C_2/C_1) * 100\% \quad (1)$$

- C_1 = Original data size
- C_2 = Compressed data size

Compression Factor (CF): The inverse of the compression ratio. That is the ratio between the size of the original data and the size of the compressed data.

$$CF = 1/CR \quad (2)$$

Saving Percentage (SP): Calculates the shrinkage of the source data as a percentage.

$$SP = [(C_1-C_2)/C_1] * 100\% \quad (3)$$

A closer examination of (1) and (3) reveals that as the compression ratio decreases, the shrinkage of memory i.e. SP increases. All the above methods evaluate the effectiveness of compression algorithms using data sizes. There are some other methods to evaluate the performance of compression systems such as compression time, computational complexity and probability distribution.

Figure 4.17 shows result of compression for different file size. Here X axis denotes file number of test data and Y axis denotes file size in KB. Blue bar denotes input file size and yellow bar denotes compressed file size.

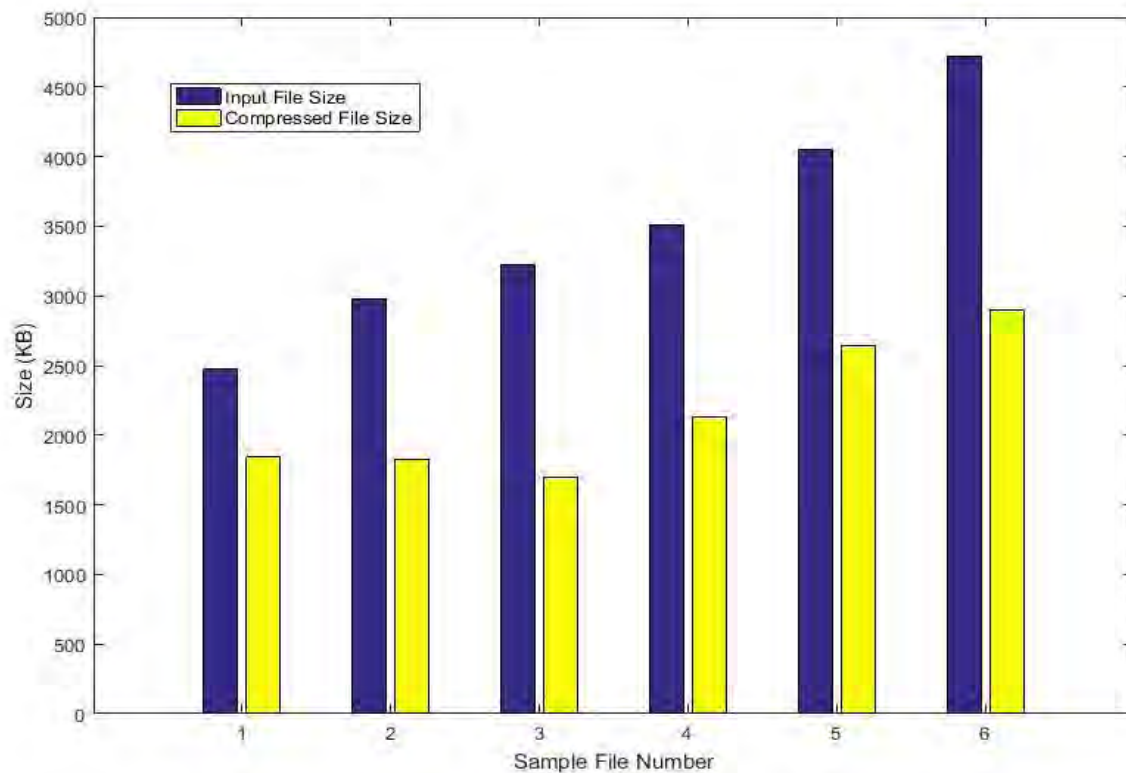


Figure 4.17: Result of Proposed 2 Level Compressions

Table 4.3 shows results of compression in terms of CR (1) and SP (3) for different file sizes.

Table 4.3: Result of Proposed Two Level Compression Method

File Number	Input File Size (in bytes)	Compressed File Size (in bytes)	CR (%)	SP (%)
1	2977331	1825956	61.33	38.67
2	3221272	1700640	52.79	47.21
3	3506366	2127821	60.68	39.32
4	4052530	2642209	65.20	34.80
5	4722220	2895598	61.32	38.68

4.8 Comparison with Other Compression Methods

Comparison of CR (1) and SP (3) between implemented methodology and related algorithms are shown in Table 4.4.

Table 4.4: Comparison of Saving Percentage (SP) and Compression Ratio (CR) between Proposed Method and Other Compression Techniques

	Huffman [23]	RLE [24]	Adaptive Huffman [24]	LZW [25]	Shannon Fano [26]	LZOP [27]	This work
SP (%)	45.41	11.79	42.11	69.03	40.83	61.58	67.30
CR (%)	54.59	88.21	57.89	30.97	59.17	38.42	32.70

Values obtained by all the algorithms listed in Table 4.4 imply that the proposed methodology furnishes encouraging saving percentage. It shows lower compression ratio and better saving percentage to the most of the compression techniques which have been compared except LZW [25] which is a fairly newer compression scheme than Huffman coding and has the potential for very high throughput in hardware implementations. The difference between proposed technique and LZW in terms of saving percentage is negligible. Moreover, this work presents a system which provides better approach for reliable transmission of data by adding error correction mechanism.

CHAPTER 5

Conclusion

5.1 Conclusion

Secured transmission and compression of data is important for most business and even home computer users. Researchers around the world are conducting research to make the transmission of data more reliable, cost effective and less time consuming. Compression plays a vital role to reduce bandwidth during transmission which results in reduction of time and cost. On the other hand, the adoption of FPGA technology continues to increase as higher-level tools evolve to deliver the benefits of reprogrammable silicon to engineers and scientists at all levels of expertise. FPGA based implementation further improves the system performance in terms of security, reliability, speed, cost etc. The proposed FPGA based system is a blend of bit stuffing, Huffman and Hamming algorithm for high speed data compression and secured transmission. The proper functionality of the design has been tested by simulating the design using MATLAB environment. Results of FPGA simulations verify correctness and effectiveness of the overall system. The simulation results show very impressive results in terms of storage savings and error correction capability. The results obtained by the proposed system are compared with the existing data compression techniques in terms of compression ratio and saving percentage. The results of comparison imply that proposed system has encouraging saving percentage of memory. Moreover, implemented system can achieve 100% error correction rate even if 25% of transmitting data get corrupted during transmission through a noisy medium. The proposed two level compression processes can achieve saving percentage of 67.30% based on input data. Since this implementation is written in Verilog HDL, it is fully portable to a variety of hardware architectures. As FPGA technology improves, the performance of the hardware implementation of the proposed system will improve without the need to change the design.

5.2 Future Works

The system has been designed using Verilog and simulated in ModelSim as a FPGA simulator. Due to time constraint, the design was not implemented on actual FPGA hardware which can be considered as future work of this project. This system can be further secured by adding symmetric or asymmetric cryptographic module which can also be considered as its future expansion.

References

- [1] Hofstee P., “The Big Deal about Big Data.” In Proceedings of the 8th *IEEE International Conference on Networking, Architecture, and Storage*, 2013.
- [2] Carlos A. J., Carlos F. A., Oscar M. R. and Javier C., “FPGA implementation of a Huffman decoder for high speed seismic data decompression”, *IEEE Data Compression Conference(DCC)*, 2014, Snowbird, UT, USA
- [3] Smith S. V., “Big Data creates big industry for storing data”. *marketplace.org*[Online]. Available at: <http://www.marketplace.org/topics/business/big-data-creates-big-industry-storing-data> [Accessed 7 Oct. 2017]
- [4] Chakraborty R. S. and Bhunia S., “HARPOON: An obfuscation based SoC design methodology for hardware protection”. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 28, no. 10, pp. 1493-1502, 2009
- [5] Maity S. P., Kundu M. K., “Distortion free image-in-image communication with implementation in FPGA”. *Int. J. Electron. Commun.* 67 (I), pp. 438–447, 2013
- [6] Wang W. and Zhang W., “Adaptive Spatial Modulation Using Huffman Coding”. *IEEE Global Communications Conference (GLOBECOM)*, 2016.
- [7] Sorokin A. and Makushenko E., “Identification of JPEG files fragments on digital media using binary patterns based on Huffman code table”. *IEEE 3rd International Conference on Digital Information Processing, Data Mining, and Wireless Communications (DIPDMWC)*, 2016.
- [8] Djusdek D., Studiawan H. and Ahmad T., “Adaptive image compression using Adaptive Huffman and LZW”. *IEEE International Conference on Information & Communication Technology and Systems (ICTS)*, 2016.
- [9] Potthuri S., Shankar T. and Rajesh A., “Cluster head selection using Huffman coding algorithm for Wireless Sensor Networks”. *IEEE International Conference on Innovations in Information, Embedded and Communication Systems (ICIIECS)*, 2015.
- [10] Navarro G. and Ordonez A., “Compressing Huffman Models on Large Alphabets”. *Data Compression Conference*, Snowbird, UT, USA, 2013.
- [11] Kim D., Song J., Kim D. and Lee S., “Fixed cycle Huffman decoding instruction for multi-format decoder”. *IEEE International Conference on Consumer Electronics (ICCE)*. Las Vegas, NV, USA, 2014.
- [12] Marti D., Jamsed K., and Agarwal K., “FPGA-Based Application Acceleration: Case Study with GZIP Compression/Decompression Streaming Engine”. *International Conference on Computer-Aided Design (ICCAD)*, Nov 2013.

- [13] Dandalis N., and Prasanna V., "Configuration compression for FPGA-based embedded systems", *ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2001
- [14] Jamro M.W.E., Wiatr K., "FPGA implementation of the dynamic Huffman encoder", *Proceedings of IFAC Workshop on Programmable Devices and Embedded Systems*, 2006
- [15] Shcherbakov I., Weis C., and Wehn N., "A High-Performance FPGA-Based Implementation of the LZSS Compression Algorithm", *Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW) 2012 IEEE 26th International*, pp. 449-453, 2012
- [16] Dias W., David Moreno E. and Palmeira I., "A new code compression algorithm and its decompressor in FPGA-based hardware". *IEEE 26th Symposium on Integrated Circuits and Systems Design (SBCCI)*, 2013
- [17] Rigler, S., Bishop, W. and Kennings, A., "FPGA-Based Lossless Data Compression using Huffman and LZ77 Algorithms". *Canadian Conference on Electrical and Computer Engineering*, 2007.
- [18] Khalid Sayood, *Introduction to Data Compression* , San Francisco, USA: Elsevier Inc., 2006
- [19] Martha R., Quispe Ayala , Krista Asalde-Alvarez , Avid Roman-Gonzalez, "Image Classification Using Data Compression Techniques", *IEEE 26-th Convention of Electrical and Electronics Engineers*, Israel, 2010
- [20] Huffman D., "A method for the construction of minimum-redundancy codes", *Proceedings of the Institute of Radio Engineers*, vol. 40, no. 9, pp. 1098-1101, September 1952
- [21] Hamming, R. W., "Error Detecting and Error Correcting Codes". *Bell SystemTech. Jour.*, 29 pp. 147–160, 1950.
- [22] ASCII, *Wikipedia*, [Online]. Available:<http://en.wikipedia.org/wiki/ASCII>, accessed on May 2017.
- [23] Kodabagi M.M., Jerabandi M.V, and Gadagin N., "Multilevel Security and Compression of Text Datausing Bit Stuffing and Huffman Coding". *International Conference on Applied and Theoretical Computing and Communication Technology (iCATccT)*, 2015
- [24] Kodituwakku S. R., Amarasinghe U. S., "Comparison of Lossless Data Compression Algorithms for Text Data", *Indian Journal of Computer Science and Engineering*, Vol 1 No 4 416-425, ISSN : 0976-5166, 2015.

- [25] Terry W., "A Technique for High-Performance Data Compression". *Computer*. 17 (6): 8–19. DOI:10.1109/MC.1984.1659158,
- [26] Fano, R. M., "The transmission of information". *Technical Report No. 65. Cambridge (Mass.), USA: Research Laboratory of Electronics at MIT, 1949.*
- [27] *Comparison of compression*, [Online]. Available: <https://binfalse.de/2011/04/04/comparison-of-compression/>, accessed on September, 2017.

Outcome of this Research Work

The following paper based on the research work of this project has been accepted in 3rd International Conference on Electrical Information and Communication Technology

Rakib M. and Ali L., “Design of an FPGA Based System for High Speed Data Compression and Secured Transmission”, *IEEE 3rd International Conference on Electrical Information and Communication Technology (EICT)*, December 2017, KUET, Khulna.