DESIGN AND IMPLEMENTATION OF A PROGRAMMABLE CONVOLUTIONAL ENCODER AND VITERBI DECODER

by

Mohammad Bozlul Karim

MASTER OF ENGINEERING IN INFORMATION AND COMMUNICATION TECHNOLOGY

Institute of Information and Communication Technology

BANGLADESH UNIVERSITY OF ENGINEERING AND TECHNOLOGY

October, 2010

The project titled "Design and Implementation of a Programmable Convolutional Encoder and Viterbi Decoder" submitted by Mohammad Bozlul Karim, Roll No. M10063128P, Session October-2006 has been accepted as satisfactory in partial fulfillment of the requirement for the degree of Master of Engineering (ICT) held on 3rd October, 2010.

Board of Examiners

____

1. Dr. Md. Liakot Ali — Chairman
   Associate Professor
   Institute of Information and Communication Technology
   BUET, Dhaka-1000.

____

2. Dr. Md Saiful Islam — Member
   Associate Professor
   Institute of Information and Communication Technology
   BUET, Dhaka-1000.

____

3. Dr. Md. Abul Kashem Mia — Member
   Professor and Associate Director
   Institute of Information and Communication Technology
   BUET, Dhaka-1000.

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 Bozlul Karim

Dedicated to

My Daughter

Table of Contents

Title	Page No.
Board of Examiners	**ii**
Candidate Declaration	**iii**
Table of Contents	**v**
List of Figures	**vii**
List of Tables	**x**
Acknowledgement	**xi**
Abstract	**xii**
CHAPTER 1: Introduction	
1.1 Overview	1
1.2 Literature Review	1
1.3 Objectives with Specific Aims and Possible Outcome	3
1.4 Organization of the Project	3
CHAPTER 2: Convolutional Coding and Viterbi Decoding	**4**
2.1 Coding and Decoding using Convolutional Codes	4
2.2 Code Parameters and the Structure of the Convolutional Code	5
2.3 Encoding of Bit Sequence	7
2.4 Encoding technique	9
2.4.1 Look up Table	9
2.4.2 State Diagram	10
2.4.3 Tree Diagram	12
2.4.4 Trellis Diagram	13
2.5 Decoding Technique	15
2.5.1 Sequential Decoding	17
2.5.1.1 Decoding using Sequential Decoding Algorithm	17
2.5.1.2 Maximum Likelihood and Viterbi Decoding	21

CHAPTER 3: Design and Implementation **28**

3.1 Introduction 28

3.2 Verilog HDL (Hardware Definition Language) 28

3.3 Implementation using FPGA 27

3.4 Development Tool Quartus II 30

3.5 Field Programmable Gate Array (FPGA) 32

3.5.1 Cyclone II FPGA DE2 Board 32

3.5.2 Design Flow for FPGA Implementation using Quartus II 7.0 34

3.5.3 Input Output Device 36

3.6 Analysis and Design Methodology 37

3.6.1 Block Diagram 37

3.6.2 Encoder Group Design. 38

3.6.3 Decoder Group Design. 38

3.6.4 Main Module of CEVD. 39

3.6.5 Block Diagram of different Module. 40

3.6.6 Important Task and Function. 45

3.6.6.1 Next State Generator 45

3.6.6.2 Output One 45

3.6.6.3 Output Zero 45

3.6.6.4 Humming Distance 45

3.6.6.5 Maximum Path Weight 45

CHAPTER 4: Results and Discussions **46**

4.1 Introduction 46

4.2 Simulation Result 46

4.2.1 Simulation of Encoder Module 46

4.2.2 Simulation of Decoder and Encoder Integrated Module. 48

4.3 Implementation on FPGA 50

4.4 Usage of Chip Area 52

4.5 Comparison 52

CHAPTER 5: Conclusion

5.1 Conclusion 54

5.2 Future Work 54

References: 55

List of Figures

Title **Page No.**

Figure 2.1: This (3,1,3) Convolutional encoder has 3 memory registers, 1 input bit and 3 output bits. 5

Figure 2.2: The states of a code indicate what is in the memory registers 7

Figure 2.3: A sequence consisting of a solo 1 bit as it goes through the encoder. 8

Figure 2.4: A convolutional encoder with 1/2 code rate and 4 memory register. 11

Figure 2.5: State diagram of encoder (2,1,4) [8] 11

Figure 2.6: Tree diagram of encoder (2,1,4) [8] 13

Figure 2.7: Trellis diagram of encoder (2,1,4) [8] 14

Figure 2.8: Encoded sequence, input bits 1011000, output bits 11011111010111 [8] 15

Figure 2.9: Sequential decoding 1 [8] 18

Figure 2.10: Sequential decoding 2 [8] 19

Figure 2.11: Sequential decoding 3 [8] 20

Figure 2.12: Sequential decoding 4 [8] 20

Figure 2.13: Viterbi decoding step 1 [8] 22

Figure 2.14: Viterbi decoding step 2 [8] 22

Figure 2.15: Viterbi decoding step 3 [8] 23

Figure 2.16: Viterbi decoding step 4 [8] 24

Figure 2.17: Viterbi decoding step 5 [8] 24

Figure 2.18: Viterbi decoding step 6 [8] 25

Figure 2.19: Viterbi decoding step 7 [8] 25

Figure 2.20: Viterbi decoding step 8 [8] 26

Figure 2.21: Maximum weighted path 26

Figure 3.1: Cyclone II FPGA DE2 Board 34

Figure 3.2: Digital system design and implementation using FPGA 35

Figure 3.3: Block diagram of the proposed system 37

Figure 3.4: Convolutional encoder group 38

Figure 3.5: Viterbi decoder group 39

Figure 3.6: Convolutional13 40

Figure 3.7: Convolutional12 40
Figure 3.8: Encoder1315 41
Figure 3.9: Encoder1304 41
Figure 3.10: Encoder1215 41
Figure 3.11: Encoder1204 42
Figure 3.12: Decoder1315 42
Figure 3.13: Decoder1304 42
Figure 3.14: Decoder1215 43
Figure 3.15: Decoder1204 43
Figure 3.16: Encoded and decoder integrated module 44
Figure 3.17: Complete design block 44
Figure 4.1: Simulation of encoder module (3,1,3) with trellis length 15 47
Figure 4.2: Simulation of encoder module (3,1,3) with trellis length 4 47
Figure 4.3: Simulation of encoder module (2,1,3) with trellis length 15 48
Figure 4.4: Simulation of encoder module (2,1,3) with trellis length 4 48
Figure 4.5: Coding rate 1/3, trellis length 15 49
Figure 4.6: Coding rate 1/3, trellis length 4 49
Figure 4.7: Coding rate 1/2, trellis length 15 50
Figure 4.8: Coding rate 1/2, trellis length 4 50
Figure 4.9: SWITCH input LED output of coding rate 1/2 , trellis length 4 51
Figure 4.10: SWITCH input LED output of coding rate 1/2 , trellis length 15 51
Figure 4.11: SWITCH input LED output of coding rate 1/3 , trellis length 4 51
Figure 4.12: SWITCH input LED output of coding rate 1/3 , trellis length 15 51

List of Tables

Title **Page No**

Table 2.1: Generator Polynomials found by Busgang for good rate 1/2 codes 6

Table 2.2: This look up table uniquely describes the code (2,1,3) 10

Table 2.3: This look up table uniquely describes the code (3,1,3) 10

Table 2.4: Bit Agreement used as metric to decide between the received sequences and the 8 possible valid code sequences 16

Table 2.5: Hamming distance metrics 21

Table 4.1: Device utilization of Cyclone II FPGA 52

Table 4.2: Comparison of different implementations. 53

Appendix A

Project Coding

Acknowledgement

At first I would like to express my heartiest thanks to my supervisor, Dr. Md. Liakot Ali, for giving me the opportunity to do my master's project under his supervision. I am also grateful to him for all his support, advice and encouragement throughout this project.

I gratefully acknowledge the valued advice and support from Professor and Director Dr. S.M. Lutful Kabir, Professor and Associate Director, Dr. Md. Abul Kashem Mia and Associate Professor Dr. Md Saiful Islam of IICT, BUET.

Finally, I want to thank all my parents and family who helped me, making this work a success.

Abstract

This project presents the design of a programmable convolutional encoder and Viterbi decoder (CEVD) using Verilog HDL. It is implemented on FPGA platform using coding rate, trellis length as parameter for configuring the chip. High coding rate transmission is reliable but takes more time to decode comparing with low coding rate. Long trellis length causes the Viterbi algorithm to take more time to decode but reliable compare with short trellis length. These combined effects are taken as consideration for design and implementation of the proposed system. Four different CEVD are designed using 1/2, 1/3 coding rate and 4, 15 trellis length. The design is simulated using Quartus II EDA tool and then implemented on the Cyclone II FPGA device. Simulation and implementation results ensure the desired functionality of the proposed design. The proposed CEVD can be used as an intellectual property for designing application specific integrated circuit (ASIC) related to wireless communication.

Chapter 1
Introduction

1.1 Overview

At Present wireless communications methods and services have been enthusiastically adopted by people throughout the world. Particularly during the past ten years, the mobile radio communications industry has grown by orders of magnitude, fueled by digital and RF circuit fabrication improvements, new large-scale circuit integration, and other miniaturization technologies which make portable radio equipment smaller, cheaper, and more reliable. Digital switching techniques have facilitated the large scale deployment of affordable, easy-to-use radio communication networks. These trends will continue at an even greater pace during the next decade.

Field programmable gate array (FPGA) is a semiconductor device containing programmable logic components and programmable interconnects. It contains up to thousands of logic element. The programmable logic components can be programmed to duplicate the functionality of basic logic gates such as AND, OR, XOR, NOT or more complex combinational functions such as decoders or simple math functions

Increasing the speed of the wireless communication requires a reliable solution for data transfer. Now a day's error correction is one of the most crucial issue for reliable wireless data communication. The approach to error correction coding taken by modern digital communications system starts with the ground breaking work of Shannon, Hamming and Golay [1-3]. Additive white gaussian noise (AWGN) properties of most of the communication media introduce noise in original data during transmission. Channel coding is a technique to introduce redundant code in original code to remove interference and error during transmission. Coded data in transmission side thus increases by volume but error

effects become less compare with uncoded data. Receiver end receives this data and decodes the data using some technique. Viterbi decoding is one of the popular techniques to decode data effectively. Viterbi algorithm (VA) is an optimum decoding algorithm for the convolutional code. Convolutional encoder and Viterbi decoder (CEVD) is widely used for reliable data communication. Most of the wireless communications devices are portable and smaller in size. Due to this fact scientists and researchers are trying to implement the encoder and decoder in chip level. According to environment status, data speed and error tolerance a comprehensive and parameterized CEVD decoder is highly needed. This demand issues are motivating new approaches to implement configurable CEVD in FPGA.

1.2 Literature Review

A highly complex Viterbi decoder someway loses its advantages, when it is adopted to decode sequences transmitted on a low-noise channel. In this case, low minimum distance codes are more suitable for achieving a good performance, and a higher bit rate can be transmitted by lowering the coding rate. Complexity of Viterbi algorithm increase in terms of convolutionally encoded trellis length. Increasing the trellis length cause the algorithm to take more time to decode. This will cause transmission speed lower but make the transmission more reliable. Lowering the trellis length will increase the transmission speed but reliability may decrease. Coding rate of convolutional encoder and trellis length of Viterbi decoder has significant effect on reliability and speed of wireless data transmission. Implementation of CEVD is done [4-7] separately considering the different coding rate and trellis length. Implementation of CEVD using DSP [4] or µC platform is Slow. Other Implementations are on FPGA platform but fixed constraint length and code rate or with partial configuration facility [5-7]. Implementation using both configurable coding rate and

trellis length is not done. In this project a programmable convolutional with 1/2, 1/3 coding rate and 4, 15 trellis length is done using Verilog HDL on Altera platform.

1.3 Objectives with Specific Aims and Possible Outcome:

This project has following objectives:

- To design the convoutional encoder with different coding rate using Verilog HDL
- To simulate the encoder using Altera's Quartus II Software.
- To design the Viterbi decoder with different Trellis length using Verilog HDL.
- To simulate the decoder using Altera's Quartus II Software.
- To integrate the encoder and decoder and Implement it using Altera's high capacity FPGA Kit

1.4 Organization of the Project

Chapter 1 of this report starts with demand issue of a configurable CEVD followed by a brief background of latest research work.

Chapter 2 of this report describes the details of convolutional encoder encoding technique and Viterbi decoder decoding step by step with proper example.

Chapter 3 of this report describes the detail design and implementation of programmable CEVD.

Chapter 4 of this report describes the simulation result of the system and chip area used by different module of the system

Chapter 5 of this report describes the conclusion and future work of the system.

Chapter 2
Convolutional Coding and Viterbi Decoding

2.1 Coding and Decoding using Convolutional Code

Coding is process to change the input data such a way that it is unrecognizable to certain environment. This is done due to reduce the adverse effect of the environment. A convolutional code introduces redundant bits into the data stream through the use of linear shift registers. convolutional codes are commonly specified by three parameters; (n, k, m). The information bits are input into shift registers and the output encoded bits are obtained by modulo-2 addition of the input information bits and the contents of the shift registers. The connections to the modulo-2 adders were developed heuristically with no algebraic or combinatorial foundation. Convolutional codes are commonly specified by three parameters: n, k, m where

n = number of output bits

k = number of input bits

m = number of memory registers

The quantity k/n called the code rate, is a measure of the efficiency of the code. Commonly and k, n parameters range from 1 to 8, m from 2 to 10 and the code rate from 1/8 to 7/8 except for deep space applications where code rates as low as 1/100 or even longer have been employed. Often the manufacturers of convolutional code chips specify the code by parameters (n, k, L). The quantity L is called the constraint length of the code and is defined by Constraint Length, L = k (m-1)

The constraint length L represents the number of bits in the Encoder memory that affect the generation of the output bits. The constraint length L is also referred to by the capital letter K, which can be confusing with the lower case k, which represents the number of input bits. In some books K is defined as equal to product the of k and m. Often in commercial

specification, the codes are specified by (r, K), where r = the code rate k/n and K is the constraint length. The constraint length K however is equal to L - 1.

2.2 Code Parameters and the Structure of the Convolutional Code

The convolutional code structure is easy to draw from its parameters. First m boxes representing the memory registers are drawn. Then modulo-2 adders to represent the n output bits. Now the memory registers are connected to the adders based on the generator polynomial as shown in Figure 2.1

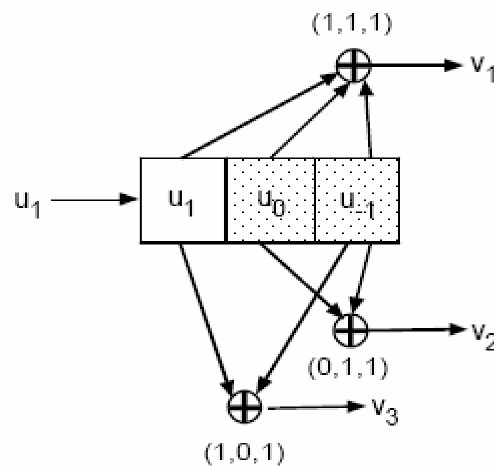

Figure 2.1: This (3, 1, 3) convolutional, 3 memory registers, 1 input bit and 3 output bits

This is a rate 1/3 code. Each input bit is coded into 3 output bits. The constraint length of the code is 2. The 3 output bits are produced by the 3 modulo-2 adders by adding up certain bits in the memory registers. The selection of which bits are to be added to produce the output bit is called the generator polynomial (g) for that output bit. For example, the first output bit has a generator polynomial of (1, 1, 1). The output bit 2 has a generator polynomial of (0, 1, 1) and the third output bit has a polynomial of (1, 0, 1). The output bits just the sum of these bits.

$v = \text{mod2}\ (u1 + u0 + u\text{-}1)$

v = mod2 (u0 + u-1)

v = mod2 (u1 + u-1)

The polynomials give the code its unique error protection quality. One (3, 1, 4) code can have completely different properties from an another one depending on the polynomials chosen. There are many choices for polynomials for any order code. They do not all result in output sequences that have good error protection properties. Petersen and Weldons book contains a complete list of these polynomials. Efficient polynomials are found from this list usually by computer simulation. A list of good polynomials for rate 1/2 codes is given below.

Table 2.1: Generator Polynomials found by Busgang for good rate 1/2 codes

Constraint Length	G1	G2
3	110	111
4	1101	1110
5	11010	11101
6	110101	111011
7	110101	110101
8	110111	1110011
9	110111	111001101
10	110111001	111001101

The (2, 1, 3) code in Figure 2.2 has a constraint length of 3. The shaded registers below hold these bits. The unshaded register holds the incoming bit. This means that 3 bits or 8 different combination of these bits can be present in these memory registers. These 8 different combinations determine what output we will get for v1 and v2, the coded sequence. The number of combinations of bits in the shaded registers are called the states of the code and are defined by

Number of states = 2^L

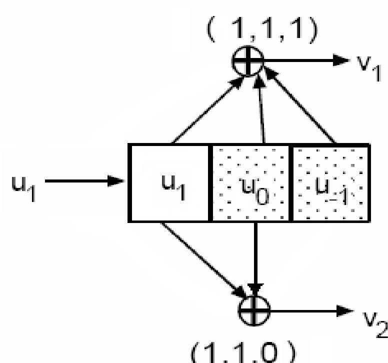

Figure 2.2: The states of a code indicate what is in the memory registers

The output bit depends on this initial condition which changes at each time tick. Lets examine the states of the code (2, 1, 3) shown Figure 2.2. This code outputs 2 bits for every 1 input bit. It is a rate 1/2 code. Its constraint length is 2. The total number of states is equal to 4. The four states of this code (2, 1, 3) are: 00, 01, 10, 11.

2.3 Encoding of Bit Sequence

First a single bit 1 is passed through this Encoder as shown in Figure 2.3.

a) At time t = 0, we see that the initial state of the encoder is all zeros (the bits in the right most L register positions). The input bit 1 causes two bits 11 to be output. Output bits are computed by a mod 2 sum of all bits in the registers for the first bit and a mod2 sum of two bits for second output bit per the polynomial coefficients.

b) At t = 1, the input bit 1 moves forward one register. The input register is now empty and is filled with a flush bit of 0. The encoder is now in state 10. The output bits are now again 11 by the same math.

c) The input bit 1 moves forward again. Now the encoder state is 01 and an another flush bit is moved into the input register. The output bits are now 10.

d) At time 3, the input bit moves from the last register and the input state is 00. The output bits are now 00. At time 3, the input bit 1 has passed completely thorough the encoder and the encoder has been flushed to an all zero state, ready for the next sequence. A single bit has produced an 8-bit output although nominally the code rate is 1/2. This shows that for small sequences the overhead is much higher than the nominal rate, which only applies to long sequences. If the same thing is done with a 0 bit, we would get an 8 bit all zero sequence.

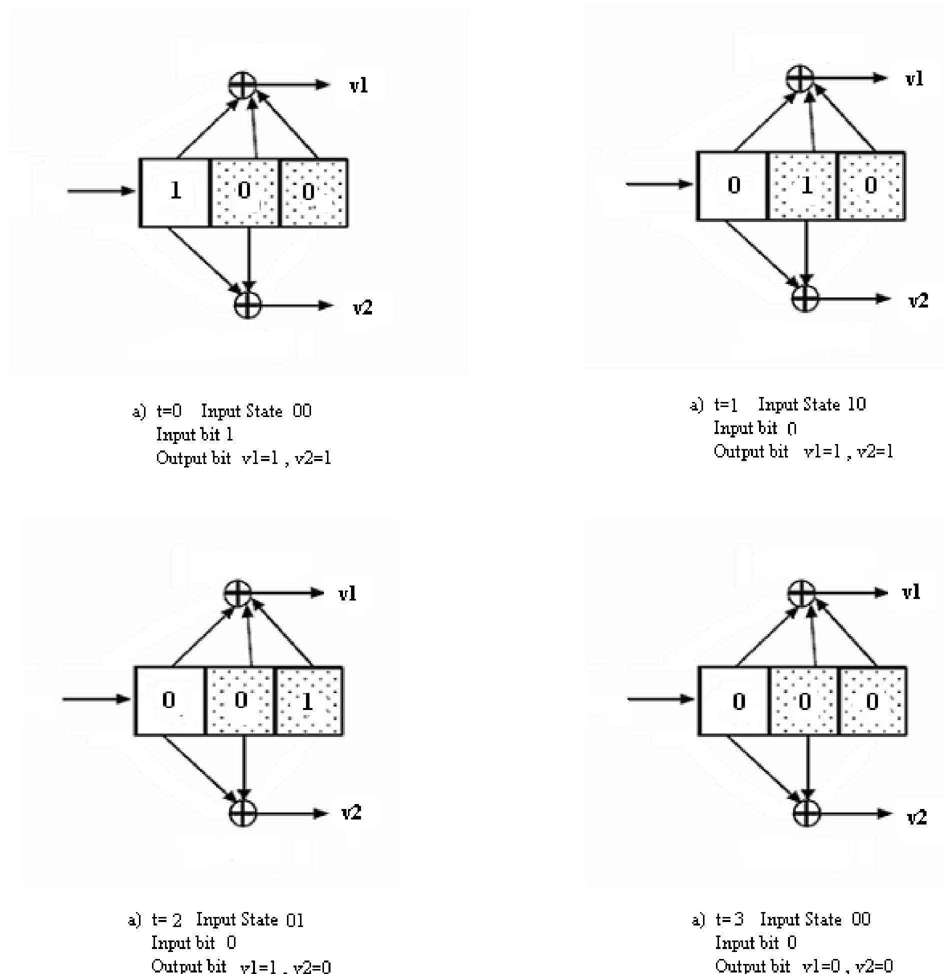

Figure 2.3: A sequence consisting of a solo 1 bit as it goes through the encoder [8]

The 1 bit has a response of 11 11 10 00 which is called the impulse response of this Encoder. Thus the 0 bit similarly has an impulse response 00 00 00 00. Convolving the input sequence with the code polynomials produced these two output sequences, which is why these codes are called convolutional codes.

2.4 Encoding Technique

There are several method of designing the Encoder. These are

1. Lookup Table
2. State Diagram.
3. Tree Diagram.
4. Trillis Diagram.

2.4.1 Lookup Table

The look up table consists of four items.

1. Input bit
2. The input state of the encoder, For code (2, 1, 3) the memory register bits are 3-1 =2 bits So four possible states will be found for code (2, 1, 3)
3. The output bits. For code (2, 1, 3) 2 bits are output and the choices are 00, 01, 10, 11. For the code (3, 1, 3) 3 bits are output and the choices are 000, 001, 010, 011, 100, 101, 110, 111.
4. The output state which will be the input state for the next bit.

Table 2.2: This look up table uniquely describes the code (2, 1, 3)

Input Bit	Input State	Output Bits	Output State
0	00	00	00
1	00	11	10
0	01	10	00
1	01	01	10
0	10	11	01
1	10	00	11
0	11	01	01
1	11	10	11

Table 2.3: This look up table uniquely describes the code (3, 1, 3)

Input Bit	Input State	Output Bits	Output State
0	00	000	00
1	00	101	10
0	01	111	00
1	01	010	10
0	10	110	01
1	10	011	11
0	11	001	01
1	11	100	11

2.4.2 State Diagram

A state diagram for code (2, 1, 4) is shown in Figure 2.5. Each circle represents a state. At any one time, the encoder resides in one of these states. The lines to and from it show state transitions that are possible as bits arrive. Only two events can happen at each time, arrival of a 1 bit or arrival of a 0 bit. Each of these two events allows the encoder to jump into a different state. The state diagram does not have time as a dimension and hence it tends to be not intuitive.

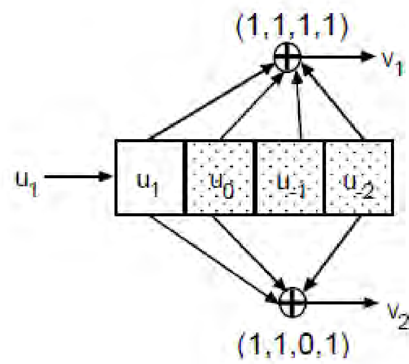

Figure 2.4: A convolutional encoder with 1/2 code rate and 4 memory register

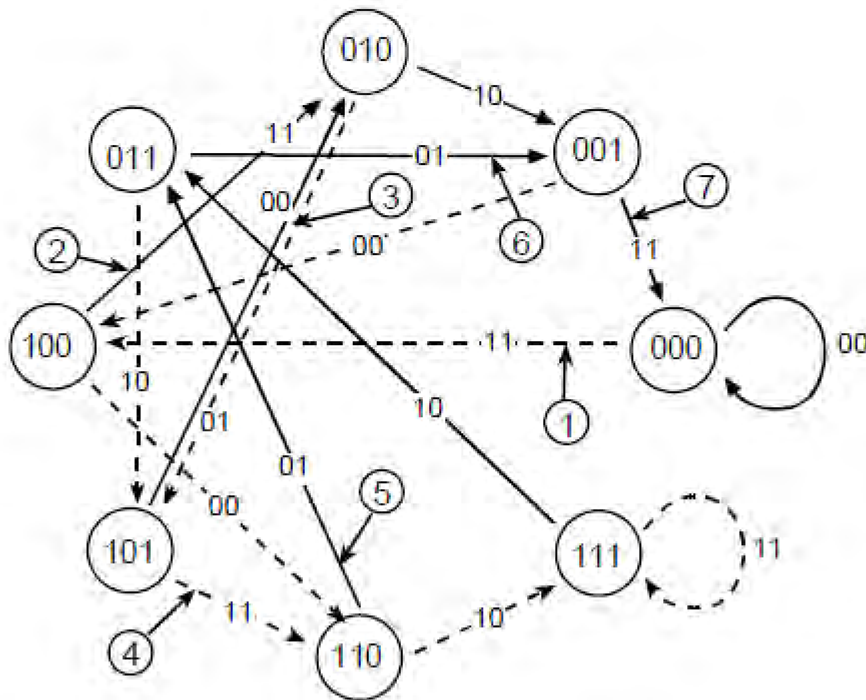

Figure 2.5: State diagram of encoder (2, 1, 4) [8]

Compare the above State diagram to the encoder lookup table. The state diagram contains the same information that is in the table lookup but it is a graphic representation. The solid lines indicate the arrival of a 0 and the dashed lines indicate the arrival of a 1. The output bits for each case are shown on the line and the arrow indicates the state transition. state

determines how many ways a travel can be done . Some encoder states allow outputs of 11 and 00 and some allow 01 and 10. No state allows all four options.

Encoding the sequence 1011 using the state diagram

(1) Always start at state 000. The arrival of a 1 bit outputs 11 and puts in state 100.

(2) The arrival of the next 0 bit outputs 11 and put in state 010.

(3) The arrival of the next 1 bit outputs 01 and puts in state 101.

(4) The last bit 1 takes to state 110 and outputs 11. So now the sequence 11 11 01 11. But this is not the end. We have to take the Encoder back to all zero state.

(5) From state 110, go to state 011 outputting 01.

(6) From state 011 next state 001 outputting 01 and then

(7) To state 00 with a final output of 11.

The final answer is : 11 11 01 11 01 01 11

This is the same answer as adding up the individual impulse responses for bits 1011000.

2.4.3 Tree Diagram

Figure 2.6 shows the tree diagram for the code (2, 1, 4). The tree diagram attempts to show the passage of time as travel deeper into the tree branches. It is somewhat better than a state diagram but still not the preferred approach for representing convolutional codes. Here instead of jumping from one state to another, branches of the tree is traverse depending on whether a 1 or 0 is received. The first branch in Figure 2.6 indicates the arrival of a 0 or a 1 bit. The starting state is assumed to be 000 as no input first arrived. If a 0 is received, upwards traverse is done and if a 1 is received, then downwards traverse is done. In Figure 2.6, the solid lines show the arrival of a 0 bit and the shaded lines the arrival of a 1 bit. The first 2 bits show the output bits and the number inside the parenthes is the output state.

Considering the sequence 1011 as before. At branch 1, downwards traverse is done . The output is 11 and the state is new 111. Now for 0 bit, upwards traverse is done. The output bits are 11 and the state is now 011. The next incoming bit is 1. Going downwards and get an output of 01 and now the output state is 101. The next incoming bit is 1 so going downwards again and gets output bits 11. From this point, in response to a 0 bit input, an output of 01 and an output state of 011 is found. If the sequence were longer, the tree diagram would have been repeated.

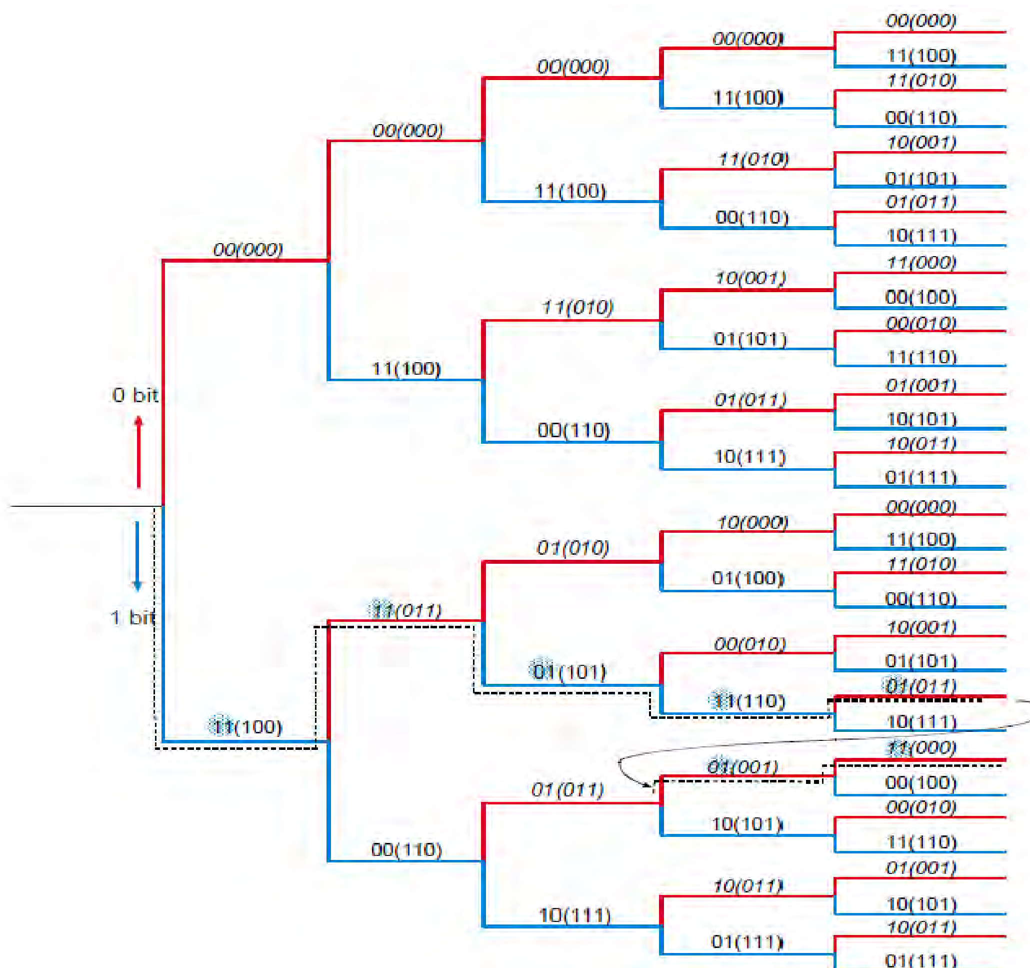

Figure 2.6: Tree diagram of encoder (2, 1, 4) [8]

2.4.4 Trellis Diagram

Trellis diagrams are messy but generally preferred over both the tree and the state diagrams because they represent linear time sequencing of events. The x-axis is discrete time and all

possible states are shown on the y-axis. We move horizontally through the trellis with the passage of time. Each transition means new bits have arrived.

Figure 2.7: Trellis diagram of encoder (2, 1, 4) [8]

The trellis diagram is drawn by lining up all the possible states (2^L) in the vertical axis. Then each state to the next state is connected by the allowable codewords for that state. There are only two choices possible at each state. These are determined by the arrival of either a 0 or a 1 bit. The arrows show the input bit and the output bits are shown in parentheses. The arrows going upwards represent a 0 bit and going downwards represent a 1 bit. The trellis diagram is unique to each code, same as both the state and tree diagrams are.

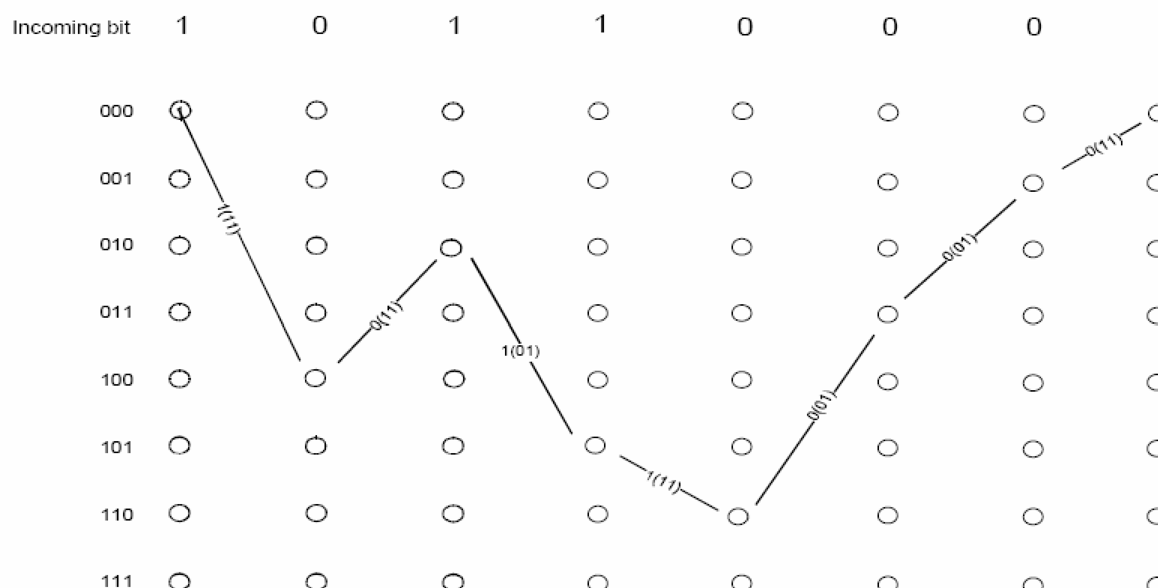

Figure 2.8: Encoded Sequence, input bits 1011000, output bits 11011111010111 [8]

In Figure 2.8, the incoming bits are shown on the top. Starting at point 1 upward traverse is done for a 0 bit and downward traverse is done for a 1 bit. The path taken by the bits of the example sequence (1011000) is shown by the lines. Trellis diagram gives exactly the same output sequence as the other three methods, namely the impulse response, state and the tree

2.5 Decoding Technique

There are several different approaches to decoding of convolutional codes. These are grouped in two basic categories.

1. Sequential decoding - Fano algorithm
2. Maximum likely-hood decoding - Viterbi decoding

Both of these methods represent 2 different approaches to the same basic idea behind decoding. If 3 bits were sent via a rate 1/2 code. 6 bits are received. (Ignoring flush bits for now.) These six bits may or may not have errors. From the encoding process these bits map uniquely. So a 3 bit sequence will have a unique 6 bit output. But due to errors, receiving

end can receive any and all possible combinations of the 6 bits. The permutation of 3 input bits results in eight possible input sequences. Each of these has a unique mapping to a six bit output sequence by the code. These form the set of permissible sequences and the decoder's task is to determine which one was sent.

Table 2.4: Bit agreement used as metric to decide between the received sequence and the 8 possible valid code sequences

Input	Valid Code Sequence	Received Sequence	Bit Agreement
000	000000	111100	2
001	000011	111100	0
010	001111	111100	2
011	001100	111100	4
100	**111110**	**111100**	**5**
101	**111101**	**111100**	**5**
110	110001	111100	3
111	110010	111100	3

If we receive 111100. It is not one of the 8 possible sequences above. The decoding can be done following two way

1. Comparing this received sequence to all permissible sequences and pick the one with the smallest hamming distance (or bit disagreement)
2. A correlation can be created and pick the sequences with the best correlation.

The first procedure is basically what is behind hard decision decoding and the second the soft-decision decoding. The bit agreements, also the dot product between the received sequence and the codeword, show that an ambiguous answer can be found and original data will not be found.

As the number of bits increase, the number of calculations required to do decoding in this brute force manner increases such that it is no longer practical to do decoding this way. a more efficient method is needed that does not examine all options and has a way of resolving ambiguity such as here with two possible answers. (shown in bold in Table 2.4).
If a message of length s bits is received, then the possible number of codewords are 2^S.

2.5.1 Sequential Decoding

Sequential decoding was one of the first methods proposed for decoding a convolutionally coded bit stream. It was first proposed by Wozencraft and later a better version was proposed by Fano.
Sequential decoding allows both forwards and backwards movement through the trellis. The decoder keeps track of its decisions, each time it makes an ambiguous decision, it tallies it. If the tally increases faster than some threshold value, decoder gives up that path and retraces the path back to the last fork where the tally was below the threshold.

Example:
If a Encoder with code (2, 1, 4) and Input Sequence is 1011 000 then the output sequence will be: 11 11 01 11 01 01 11 (If no errors occurred)
Considering one error occurred at first position and the received output is
01 11 01 11 01 01 11

2.5.1.1 Decoding using Sequential Decoding Algorithm

Decision point 1: The decoder looks at the first two bits, 01. Right away it sees that an error has occurred because the starting two bits can only be 00 or 11. Any one of this two has

error. The decoder randomly selects 00 as the starting choice. To correspond to 00, it decodes the input bit as a 0. It puts a count of 1 into its error counter. It is now at point 2.

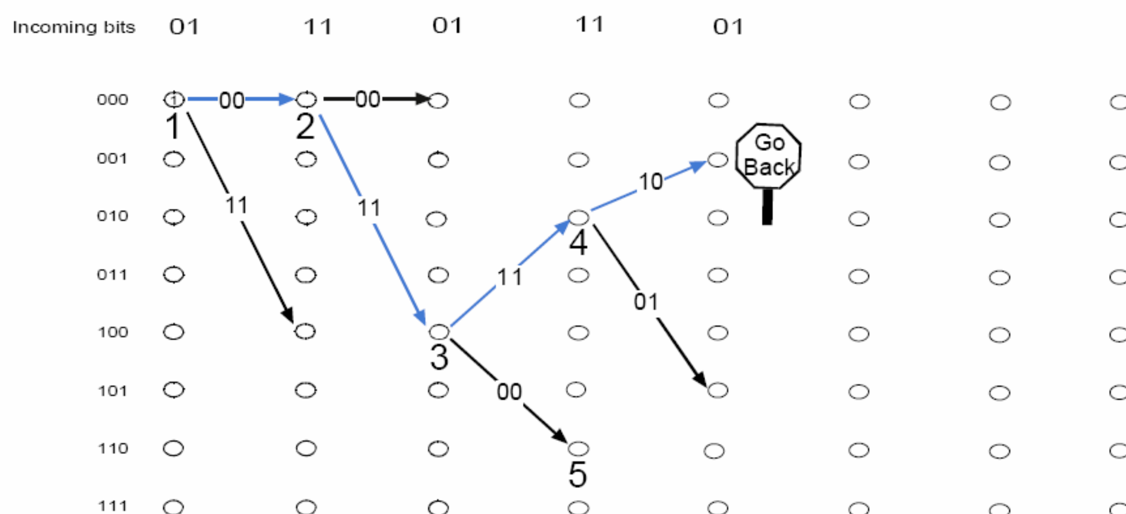

Figure 2.9: Sequential decoding 1 [8]

Decision point 2: The decoder now looks at the next set of bits, which are 11. From here, it makes the decision that a 1 was sent which corresponds exactly with one of the codewords. This puts it at point 3.

Decision point 3: The received bits are 01, but the codeword choices are 11, 00. This is seen as an error and the error count is increased to 2. Since error count is less than the threshold value of 3 (which we have set based on channel statistics) the decoder proceeds ahead. It arbitrarily selects the upper path and proceeds to point 4 making a decision that a 0 was sent.

Decision point 4: It recognizes another error since the received bits are 11 but the codeword choices are 10, 01. The error tally increases to 3 and that tells the decoder to turn back.

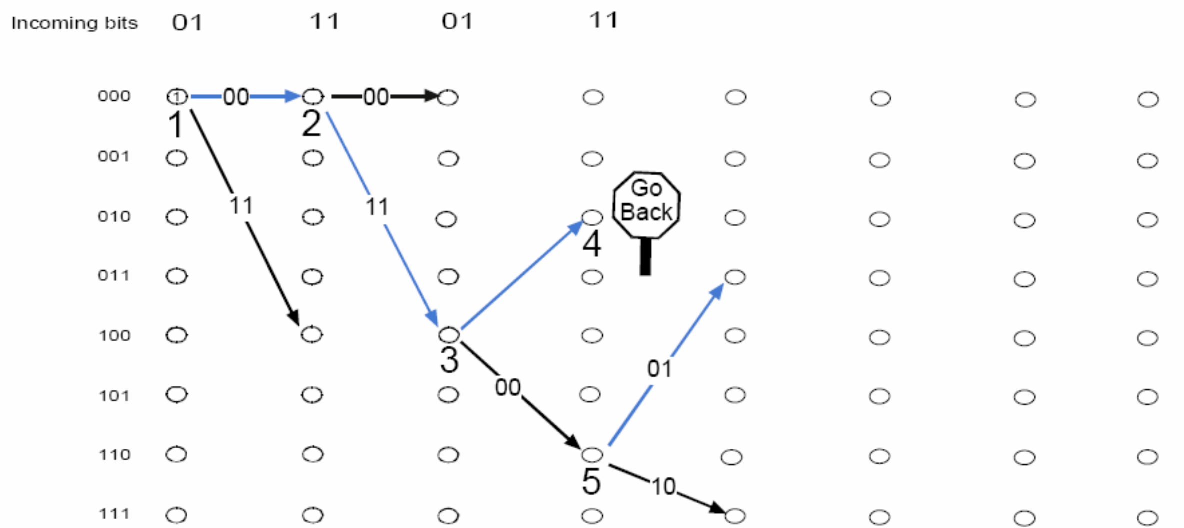

Figure 2.10: Sequential decoding 2 [8]

Decision point 5: The decoder goes back to point 3 where the error tally was less than 3 and takes the other choice to point 5. It again encounters an error condition. The received bits are 11 but the codewords possible are 01, 10. The tally has again increased to 3. It turns back again

Decision point 6: Both possible paths from point 3 have been exhausted. The decoder must go further back than point 3. It goes back to pint 2. But here if it follows to point 2 the error tally immediately goes up to 3. So it must turn back from point 2 back to point 1.

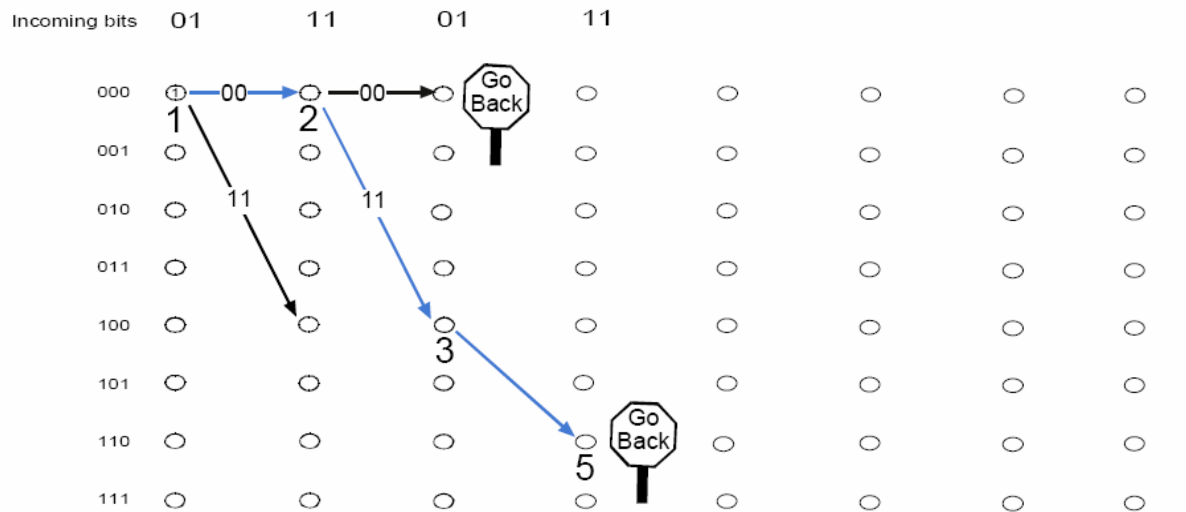

Figure 2.11: Sequential decoding 3 [8]

From point 1, all choices encountered meet perfectly with the codeword choices and the decoder successfully decodes the message as 1011000.

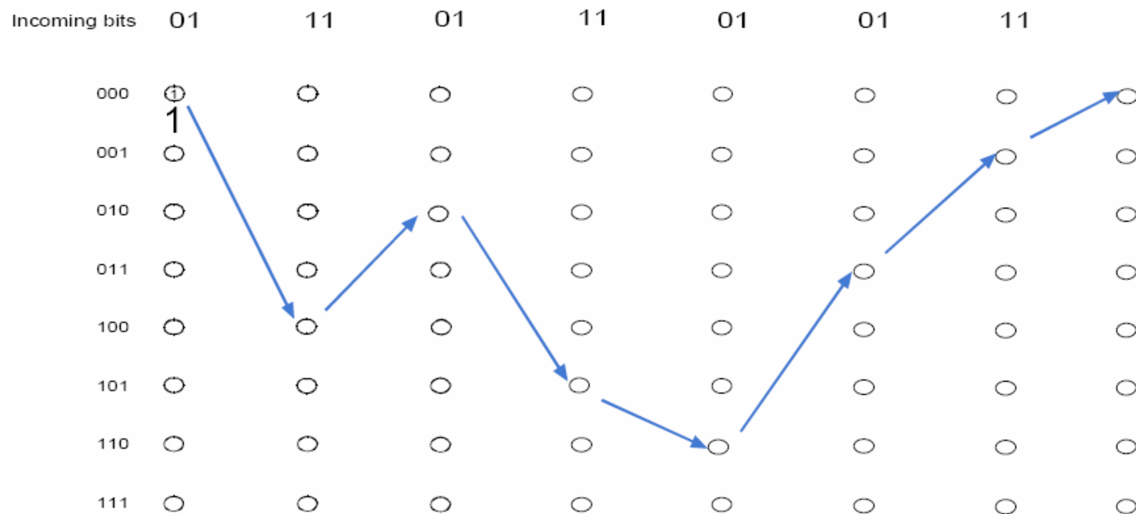

Figure 2.12: Sequential decoding 4 [8]

The memory requirement of sequential decoding is manageable and so this method is used with long constraint length codes where the S/N is also low. Some NASA planetary mission links have used sequential decoding.

2.5.1.2 Maximum Likelihood and Viterbi Decoding

Viterbi decoding is the best known implementation of the maximum likely-hood decoding. The principal used to reduce the choices is this.

1. The errors occur infrequently. The probability of error is small.
2. The probability of two errors in a row is much smaller than a single error, that is the errors are distributed randomly.

The Viterbi decoder examines an entire received sequence of a given length. The decoder computes a metric for each path and makes a decision based on this metric. All paths are followed until two paths converge on one node. Then the path with the higher metric is kept and the one with lower metric is discarded. The paths selected are called the survivors. For an N bit sequence, total numbers of possible received sequences are 2^N. Of these only 2^{KL} are valid. The Viterbi algorithm applies the maximum-likelihood principles to limit the comparison to 2 to the power of kL surviving paths instead of checking all paths. The most common metric used is the hamming distance metric. This is just the dot product between the received codeword and the allowable codeword.

Table 2.5: Humming distance metrics

Bits Received	Valid Codeword1	Valid Codeword2	Humming Metric 1	Humming Metric 2
00	00	11	2	0
01	10	01	0	2
10	00	11	1	1

These metrics are cumulative so that the path with the largest total metric is the final winner.

In the following example received sequence 01 11 01 11 01 01 11 is decoded using Viterbi decoding algorithm.

At t = 0, received bit is 01. The decoder always starts at state 000. From this point it has two paths available, but neither matches the incoming bits. The decoder computes the branch metric for both of these and will continue simultaneously along both of these branches in contrast to the sequential decoding where a choice is made at every decision point. The metric for both branches is equal to 1, which means that one of the two bits was matched with the incoming bits.

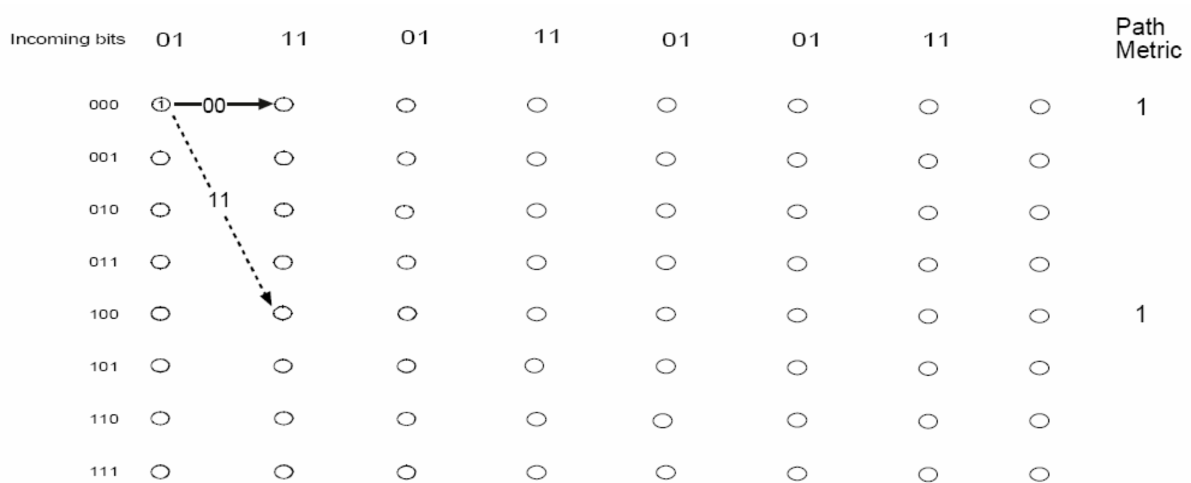

Figure 2.13: Viterbi decoding step 1 [8]

At t = 1, the decoder fans out from these two possible states to four states. The branch metrics for these branches are computed by looking at the agreement with the codeword and the incoming bits which are 11. The new metric is shown on the right of the trellis.

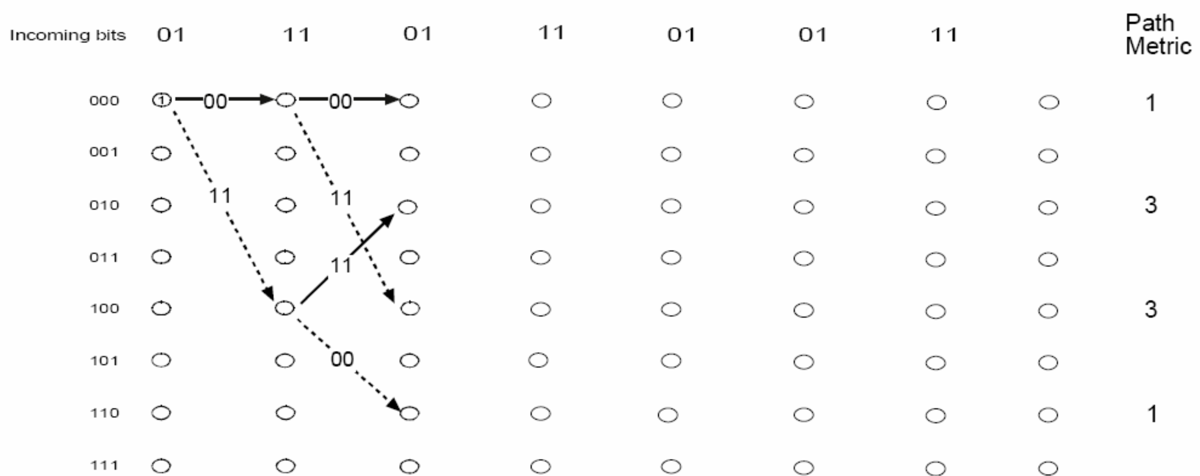

Figure 2.14: Viterbi decoding step 2 [8]

At t = 2, the four states have fanned out to eight to show all possible paths. The path metrics calculated for bits 01 and added to pervious metrics from t = 1.

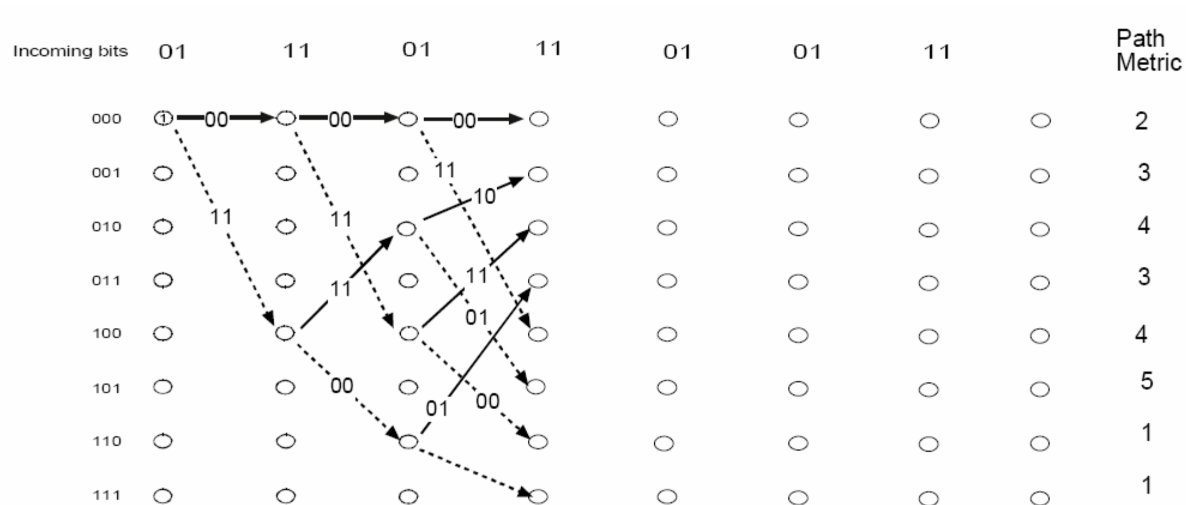

Figure 2.15: Viterbi decoding step 3 [8]

At t = 4, the trellis is fully populated. Each node has at least one path coming into it. The metrics are as shown in Figure 2.15.

At t = 5, the paths progress forward and now begin to converge on the nodes. Two metrics are given for each of the paths coming into a node. Per the maximum likelihood principle, at each node the path with the lower metric is discarded because it is least likely. This discarding of paths at each node helps to reduce the number of paths that have to be examined and gives the Viterbi method its strength.

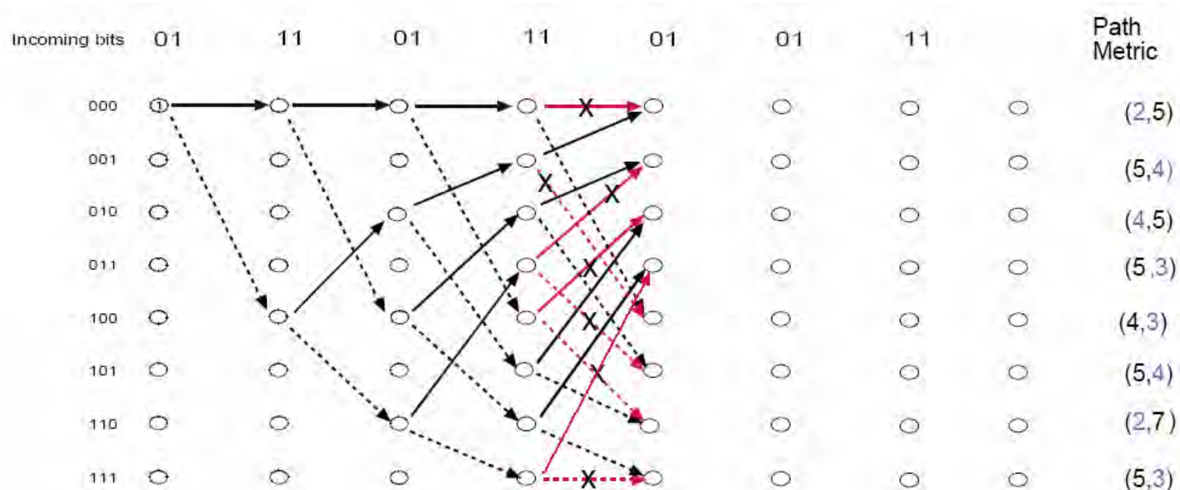

Figure 2.16: Viterbi decoding step 4 [8]

Now at each node, one or more path is available for converging. The metrics for all paths are given on the right. At each node, Only the path with the highest metric will be kept and discard all others, shown in red. After discarding the paths with the smaller metric, the following paths are left. The metric shown is that of the winner path.

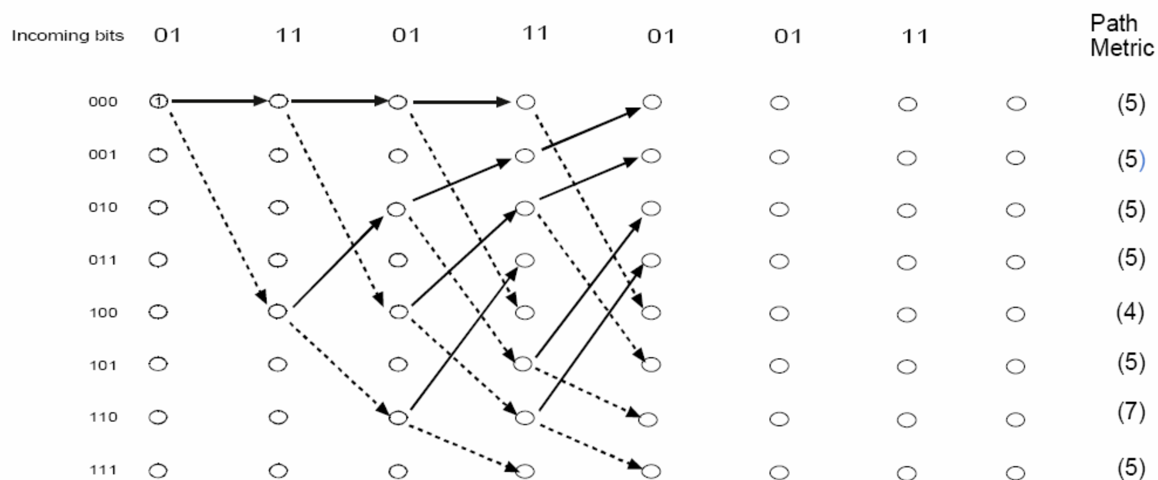

Figure 2.17: Viterbi decoding step 5 [8]

At t = 5, after discarding the paths as shown, again forward traverse and computation of new metrics is done. At the next node, paths converge occurred and paths with lower metrics is discarded.

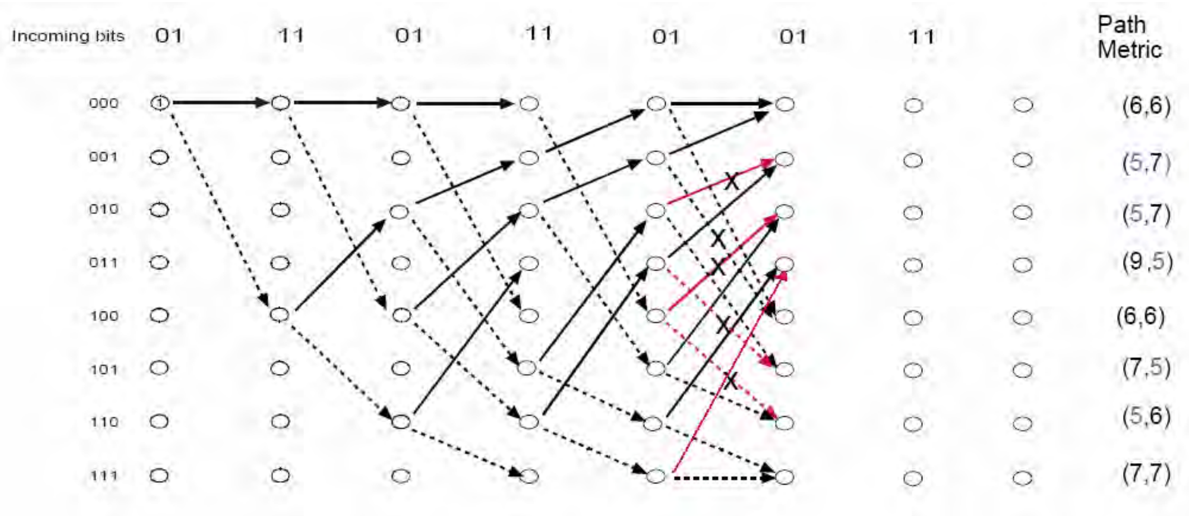

Figure 2.18: Viterbi decoding step 6 [8]

At t= 6, the received bits are 11. Again the metrics are computed for all paths. All smaller metrics paths are discarded but kept anyone if they are equal.

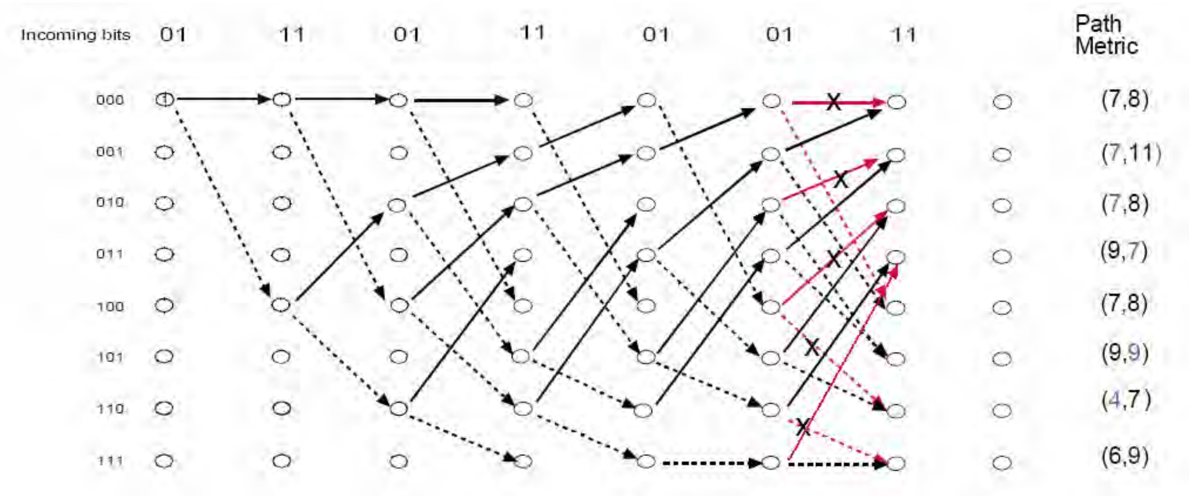

Figure 2.19: Viterbi decoding step 7 [8]

The 7-step trellis is complete. Now the path with the highest metric is winner. The path traced by states 000, 100, 010, 101, 110, 011, 001, 000 and corresponding to bits 1011000 is the decoded sequence.

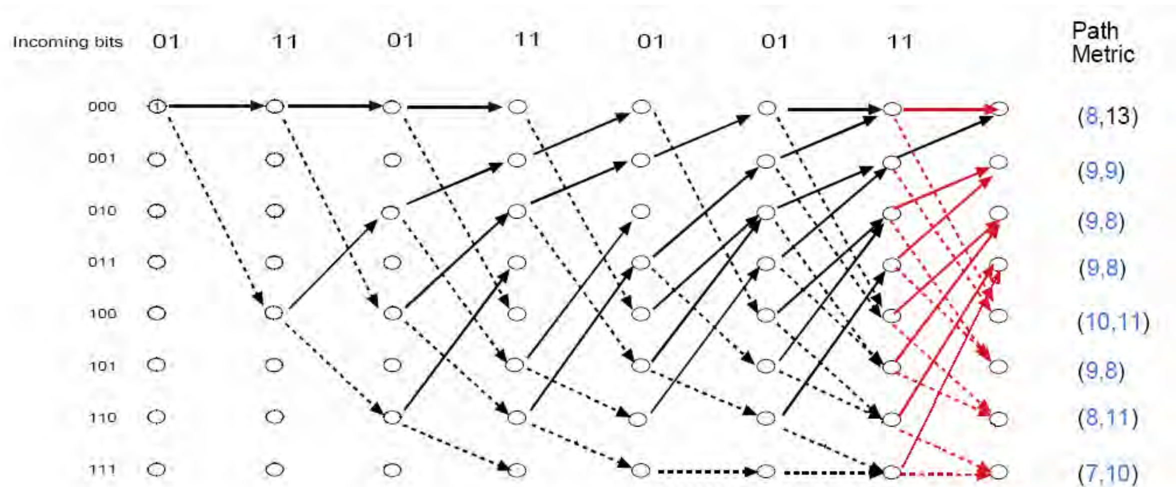

Figure 2.20: Viterbi decoding step 8 [8]

The maximum weighted path is shown in Figure 2.20

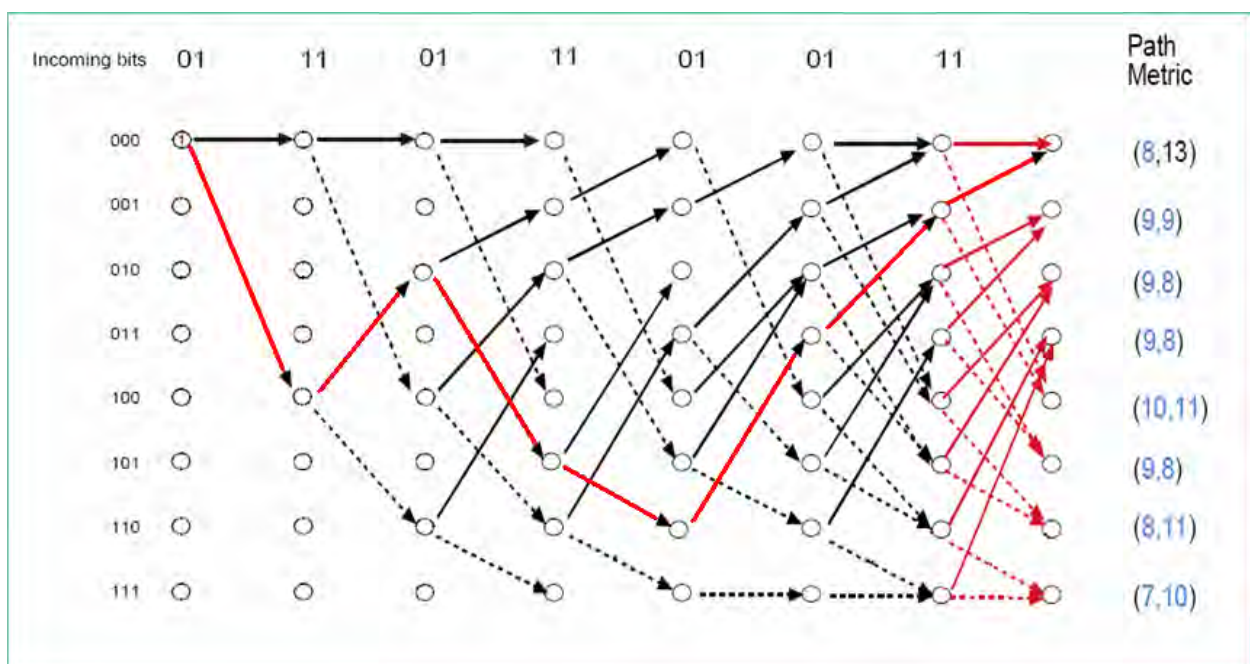

Figure 2.21: Maximum weighted path

The length of this trellis was 4bits + m bits. Ideally this should be equal to the length of the message, but by a truncation process, storage requirements can be reduced and decoding need not be delayed until the end of the transmitted sequence.

Chapter 3
Design and Implementation

3.1 Introduction

Verilog HDL is one of the most popular high definition language to design digital circuit. In this project CEVD is designed and simulated using Verilog HDL in Altera's Quartus II environment. After simulating the result the design is implemented using Cyclone II FPGA board. Convolutional encoder uses the lookup table to generate output data and next state of the Encoder. Viterbi decoder is designed using the maximum likelihood algorithm.

3.2 Verilog HDL (Hardware Definition Language)

In the earlier, the conventional approach such as hand-draw and schematic based design technique was the only choice to the designer to design a digital system. But now millions of transistors are being integrated on a single chip integrated circuit (IC) where the conventional design technique is insufficient to be used. It points towards having a new approach for designing today's complex digital system and that is hardware description language (HDL).

HDL based design technique has been emerged as the most efficient solution. It offers the following advantages over conventional based design approaches.

- It is technology independent. If a particular IC fabrication process becomes outdated, it is possible to synthesize a new level design by only changing the technology file but using the same HDL code.
- HDL shortens the design cycle of a chip by efficiently describing and simulating the behavior of the chip. A complex circuit can be designed using a few lines of HDL code.
- It lowers the cost of design of an IC.

- It improves design quality of a chip. Area and timing of the chip can be optimized and analyzed in different stages of design.

There are different types of HDL available in the market. Some of these are vendor dependent where the HDL code is only useable under the software provided by the specific vendor. For example, Altera hardware description language (AHDL) from Altera company, Lola (Logic Language) from European Silicon Structure (ES2) company etc. However Verilog and VHDL (very high speed IC hardware description language) are the two vendor independent HDL which are now widely accepted industry standard electronic design automation (EDA) tool for designing digital system. Verilog HDL is introduced by Cadence Data Systems, Inc. and later its control is transferred to a consortium of companies and universities known as open Verilog international (OVI) whereas VHDL is used primarily by defense contractors. Currently Verilog is widely used by IC designers. Verilog HDL is IEEE standard and easier than VHDL. It is less error prone. It has many pre-defined features very specific to IC design.For this reason Verilog is chosen to design and implement AES processor.

3.3 Implementation using FPGA

In most FPGAs, programmable logic components (or logic blocks, in FPGA parlance) also include memory elements, which may be simple flip-flops or more complete blocks of memories. These logic blocks and interconnects can be programmed after the manufacturing process by the customer/designer (hence the term "field programmable", i.e. programmable in the field) so that the FPGA can perform whatever logical function is needed.

There are various vendor manufacturers for different types of FPGA chip such as Altera, Xilinx, Lattice Semiconductor, Actel, Quick Logic, Cypress Semiconductor, Atmel, Achronix Semiconductor etc. Among them Altera and Xilinx are the most famous FPGA

companies since both of the companies have lot of varieties of FPGA device from small number of gate counts to higher number of gate counts. However Altera devices offer the general benefits of PLDs as innovative architectures, advanced process technologies, state-of-the-art development tools, and a wide selection of mega function. The common advantages of Altera devices include: high performance, high-density logic integration, cost-effectiveness, short development cycles with the Quartus II software, Mega core functions, Benefits of in-system programming. In this work the FPGA device used is Altera provided EP2C35F672C6 from Cyclone II family.

3.4 Development Tool Quartus II

The AES processor is designed using Quartus II EDA tool (provided by Altera Company) which provides graphical user interface (GUI) to download the digital design AES into the Cyclone II FPGA.

Quartus II software provides a simple, automated mechanism to allow designers to obtain the best performance for their designs. This software provides the way to design the solution through Verilog HDL and complile the design to ensure the workability and efficiency logically. The tool programmer allows using files generated by the compiler to program and/or configuring all devices supported by the Quartus II software. Programmer and supported programming hardware tool is used to easily program or configure a working device in minutes. After a successful compilation, download configuration data into a device through the, ByteBlaster or USB-Blaster communications cables, or through the Altera Programming Unit (APU).The program or configure devices can be in Passive Serial mode, Active Serial Programming mode, JTAG mode, or In-Socket Programming mode.

Program an Altera Device: When the design is ready to program or configure a device, it needs to open the programmer and create a chain description file (.cdf) that stores device

name, device order, programming and hardware setup information. CDFs can be used to program or configure one or more devices in a JTAG chain or a passive serial chain.

Compiling Mode: The Quartus II Compiler consists of a set of independent modules that check the design for errors, synthesize the logic, fit the design into an Altera device, and generate output files for simulation, timing analysis, software building, and device programming. The basic compiler consists of the analysis & synthesis, fitter, assembler, and timing analyzer modules. Each of the compiler modules can be run individually or together from the Quartus II user interface. Alternatively, these modules can be run independently with the appropriate command line executable.

Compile the Design: The compiler automatically locates and uses all non-design files associated with the design, such as include files (.inc) containing AHDL function prototype statements; memory initialization files (.mif) or hexadecimal intel-format files (.hex) containing the initial content of memories; as well as Quartus II project Files (.qpf) and Quartus II settings files (.qsf) containing project and setting information. During compilation, the compiler generates information, warning, and error messages that appear automatically in the messages window.

Simulation Mode: Simulation allows testing a design thoroughly to ensure that it responds correctly in every possible situation before configuring a device. Depending on the type of information need, functional or timing simulation can be performed with the simulator. Functional simulation tests only the logical operation of a design by simulating the behavior of flattened netlist extracted from the design files, while timing simulation uses a fully compiled netlist containing timing information to test both the logical operation and the worst-case timing for the design in the target device. Before running a simulation, it is necessary to specify input vectors as the stimuli for the Quartus II Simulator. The simulator

uses these input vectors to simulate the output signals that a programmed device would produce under the same conditions. The simulator supports input vector stimuli in the form of a vector waveform file (**.vwf**), vector table output file (**.tbl**), power input file (**.pwf**), or a Quartus II generated vector file (**.vec**) or simulator channel file (**.scf**).

3.5 Field Programmable Gate Array (FPGA)

A field programmable gate array (FPGA) or programmable logic device (PLD) is a semiconductor device containing programmable logic components and programmable interconnects. It contains up to thousands of gates. The programmable logic components can be programmed to duplicate 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, programmable logic components also include memory elements, which may be simple flip-flops or more complete blocks of memories. These logic blocks and interconnects can be programmed after the manufacturing process by the customer/designer (hence the term" field programmable") so that the FPGA can perform whatever logical function is needed.

3.5.1 Cyclone II FPGA DE2 board

Altera's Cyclone II FPGA family is designed on an all-layer-copper, low-k, 1.2-V SRAM process and is optimized for the smallest possible die size. Built on TSMC's highly successful 90-nm process technology using 300-mm wafers, the Cyclone II FPGA family offers higher densities, more features, exceptional performance, and the benefits of programmable logic at ASIC prices. The Cyclone II FPGA family extends the reach of FPGAs further into cost-sensitive, high-volume applications, continuing the success of the Cyclone FPGA family. Cyclone II FPGA has following facilities.

- Altera Cyclone II 2C35 FPGA with 35000 LEs
- Altera Serial Configuration deivices (EPCS16) for Cyclone II 2C35

- USB Blaster built in on board for programming and user API controlling
- JTAG Mode and AS Mode are supported
- 8Mbyte (1M x 4 x 16) SDRAM
- 512K byte(256K X16) SRAM
- 4Mbyte Flash Memory (upgradeable to 4Mbyte)
- SD Card Socket
- 4 Push-button switches
- 18 DPDT switches
- 9 Green User LEDs
- 18 Red User LEDs
- 16 x 2 LCD Module
- 50MHz Oscillator and 27MHz Oscillator for external clock sources
- 24-bit CD-Quality Audio CODEC with line-in, line-out, and microphone-in jacks
- VGA DAC (10-bit high-speed triple DACs) with VGA out connector
- TV Decoder (NTSC/PAL) and TV in connector
- 10/100 Ethernet Controller with socket.
- USB Host/Slave Controller with USB type A and type B connectors.
- RS-232 Transceiver and 9-pin connector
- PS/2 mouse/keyboard connector
- IrDA transceiver
- Two 40-pin Expansion Headers with diode protection

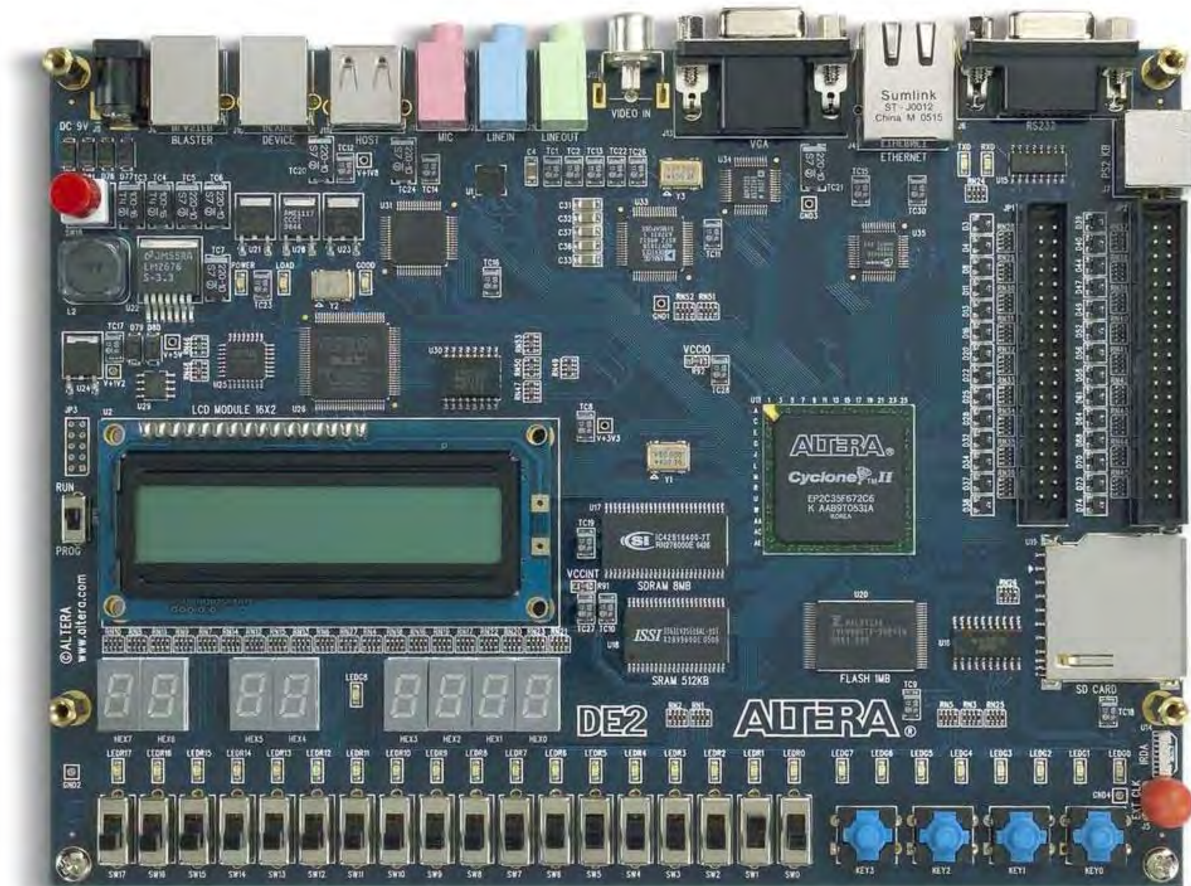

Figure 3.1: Cyclone II FPGA DE2 Board

3.5.2 Design Flow for FPGA Implementation using Quartus II 7.0

Figure 3.2 shows flow diagram of a design to be realized into FPGA hardware. Once the sub-modules of a design are identified, each of the modules is designed, compiled and synthesized using FPGA vendor provided software. Then functional simulation is performed upon each module. The correct simulation results ensure the proper functionality of a design. Once the simulation results of all the sub-modules are as desired then they are integrated and simulated again. Then for hardware realization, suitable FPGA device is selected for the design, inputs and outputs are assigned to specific pins of the FPGA. It is again compiled and synthesized.

After that timing simulation of the design is performed to ensure that the design functions in real time. Then the design is downloaded into the FPGA.

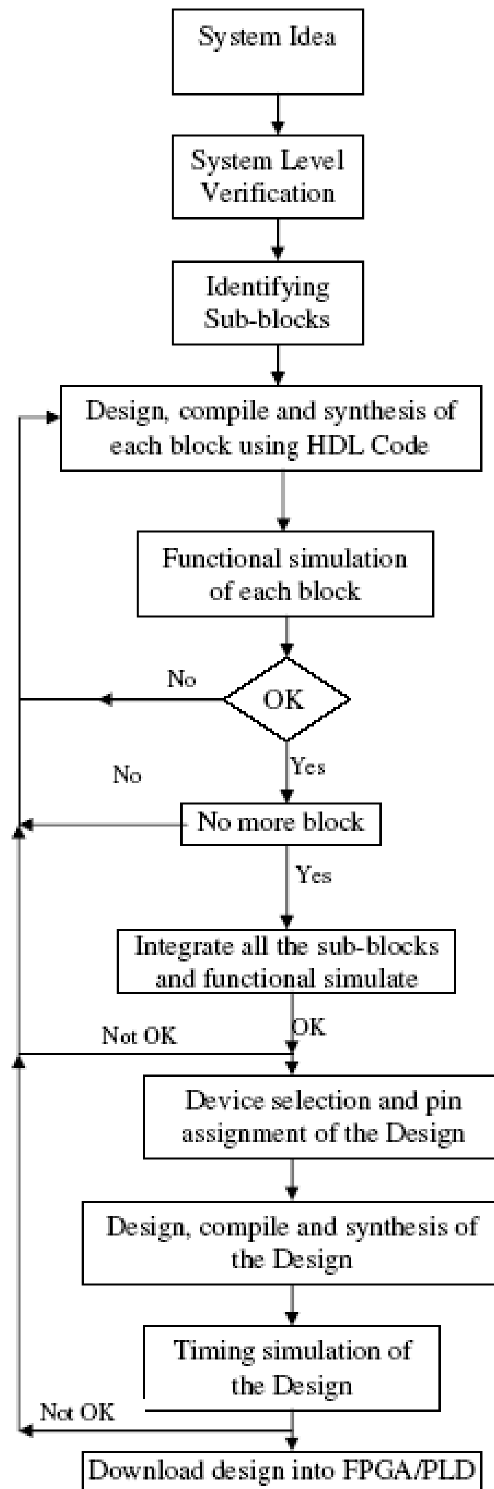

Figure 3.2: Digital system design and implementation using FPGA

So to implement the circuits that will be designed on the FPGA there are few key steps.

1. Start new project
2. Create a new Verilog HDL file
3. Write the program using Verilog HDL.
4. Compile the code.
5. Correct any syntax errors.
6. Create a vector web form file
7. Pin assignment
8. Simulate the circuit to make sure that getting the expected behavior.
9. Download the program onto the FPGA
10. Test the operation of circuit.

Quartus II helps to implement all of the above easily.

3.5.3 Input Output Device

The DE2 board provides 18 toggle switches, called *SW*17¡0, that can be used as inputs to a circuit, and 18 red lights, called *LEDR*17¡0, that can be used to display output values. Single assignment statement has been used for all 18 input output device.

3.6 Analysis and Design Methodology

3.6.1 Block Diagram

The project block diagram (Figure 3.3) shows the total design at a glance with functionality of different module

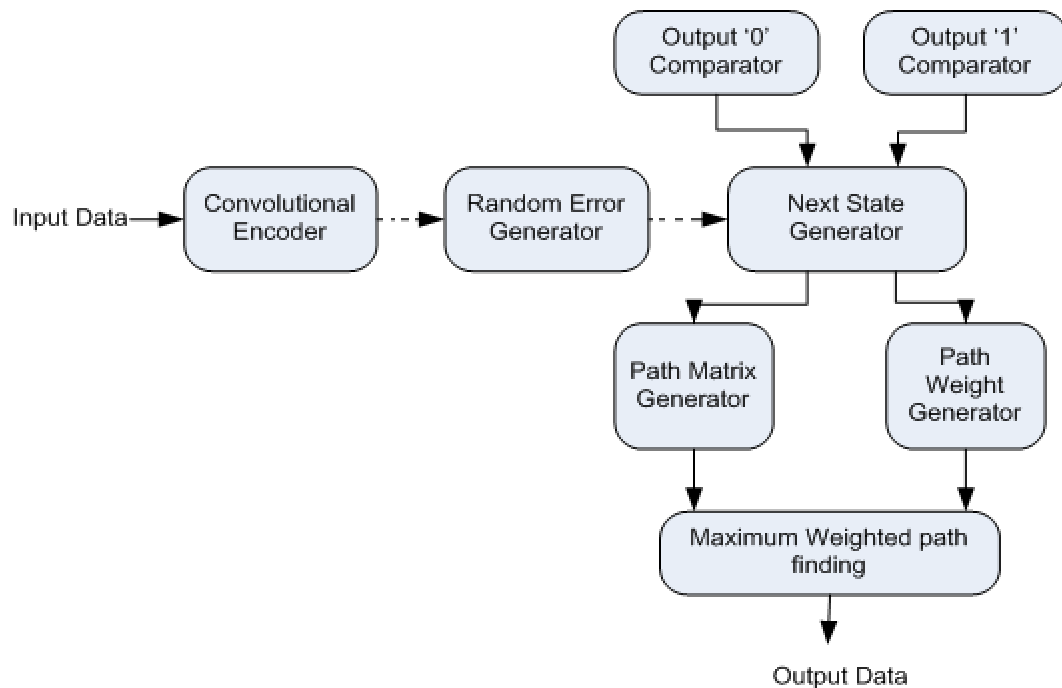

Figure 3.3: Block diagram of the proposed System

Block diagram shows different functions designed to modularized the total system. Input data first feed into convolutional encoder. Convolutional encoder make this data two or three fold according to different code rate. Using random error generator errors in different bit position are introduced in encoder output data. Next state generator function used this erroneous data in sequential manner combining 2 bits for 1/2 code rate and 3 bits for 1/3 code rate. This combined bits are compared with '0' output comparator and '1' output comparator and path matrix with path weights are generated. After completing the pattern matching maximum weighted path finding function is used to find the best survival path.

3.6.2 Encoder Group Design

Four separate encoder blocks are integrated in this encoder group design. These are

1) Encoder module 1 with 1/2 code rate, 3 constraint length, 4 trellis length
2) Encoder module 2 with 1/2 code rate, 3 constraint length, 15 trellis length
3) Encoder module 1 with 1/3 code rate, 3 constraint length, 4 trellis length
4) Encoder module 2 with 1/3 code rate, 3 constraint length, 15 trellis length

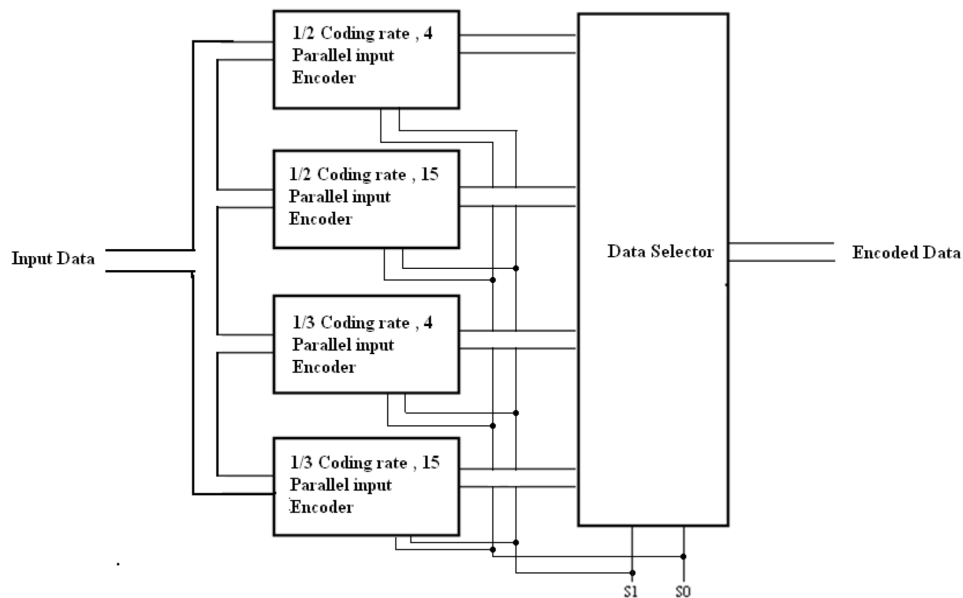

Figure 3.4: Convolutional encoder group

Selection variable S1 and S0 is used to select different module of convolutional encoder and corresponding output of this module is taken to transmit. Internal calculation of each module will be activated after receiving the predefined selection bit combination. The module without valid selection bit combination will always output '0' bit.

3.6.3 Decoder Group Design

Four separate decoder blocks are integrated in this decoder group design. These are

1) Decoder module 1 with 1/2 code rate, 3 constraint length, 4 trellis length
2) Decoder module 2 with 1/2 code rate, 3 constraint length, 15 trellis length

3) Decoder module 3 with 1/3 code rate, 3 constraint length, 4 trellis length
4) Decoder module 4 with 1/3 code rate, 3 constraint length, 15 trellis length

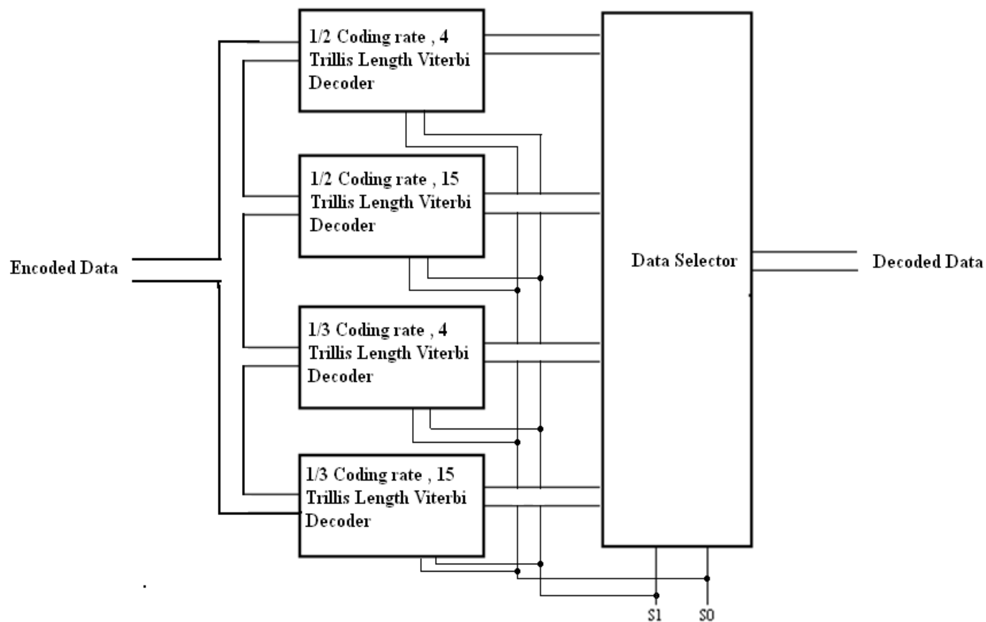

Figure 3.5: Viterbi decoder group

Verilog HDL provide task and function facility to modularize the large design. Each decoder module is sub divided into different tasks and function. Next state generator, hamming distance calculation, branch matrices calculation all this are implemented using task and function. S1 and S0 Selection variable are used to select different Viterbi decoders. This Decoder selection variable has one to one relation with encoder selection variable.

3.6.4 Main Module of CEVD

Each module is individually design with corresponding input data, output data and Enable signal. Enable signal is used to choice different encoder and decoder. According to four binary combination of two enable signal corresponding encoder and decoder output is selected.

3.6.5 Block Diagram of different Module

Total project work is divided into ten different module. Smaller modules are embedded to large module. Block diagram shows the functionality, input, output and interface facility to other module. Each module is described below.

Convolutional13 Module: 1/3 code rate convolutional encoder is Shown in figure 3.6. Convolutional encoder takes three binary data as input. In these data two LSB bits are previous memory register data and MSB bit is the new data. After processing the module 2 operation three output bits are generated. rst is use to enable the circuit.

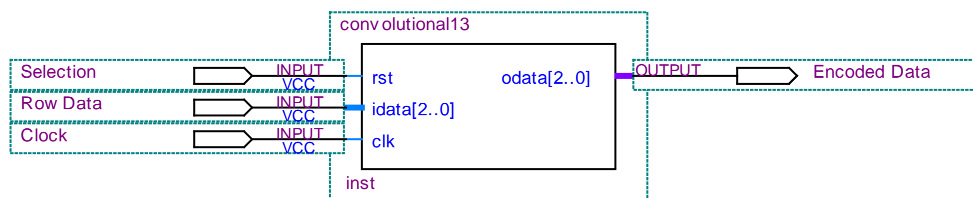

Figure 3.6: Convolutional13

Convolutional12 Module: This is same as convolutional encoder with code rate 1/3 except that this encoder produce two output bit at a time. Only two module 2 operation is done here. Figure 3.7 shows this 1/2 code rate convolutional encoder.

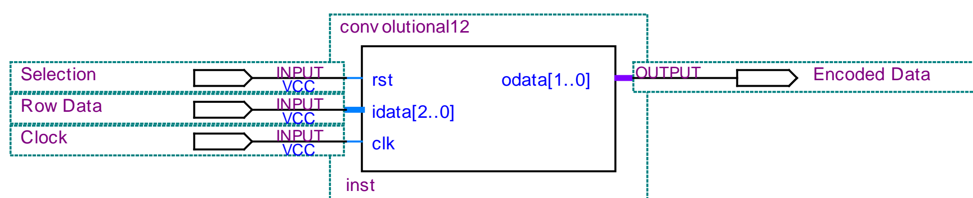

Figure 3.7: Convolutional12

Encoder1315 Module: According to the figure 3.8 this module use 15 binary data as input and produce 51 binary data as output . This module use convolutional13 module as sub module for fifteen times. Output is generated using parallel execution of fifteen sub modules. rst is used as selection variable where '100' value is fixed for this module to be activated.

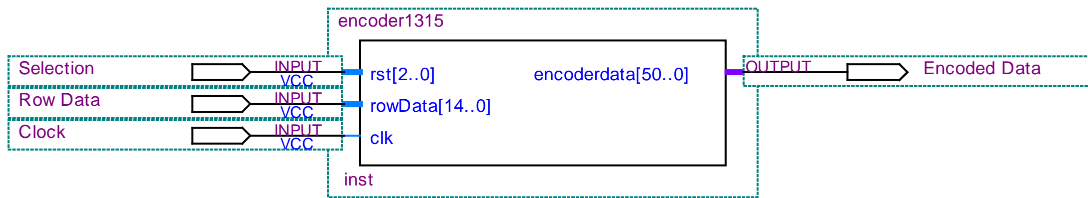

Figure 3.8: Encoder1315

Encoder1304 Module: This module use 4 binary data as input and produce 18 binary data as output shown in figure 3.9. Here extra 33 bits of output port remain '0' . This is done due to make the same length of output register for whole design. This module use convolutional13 module as sub module for four times. Output is generated using parallel execution of four sub modules . rst is used as selection variable where '101' value is fixed for this module to be activated.

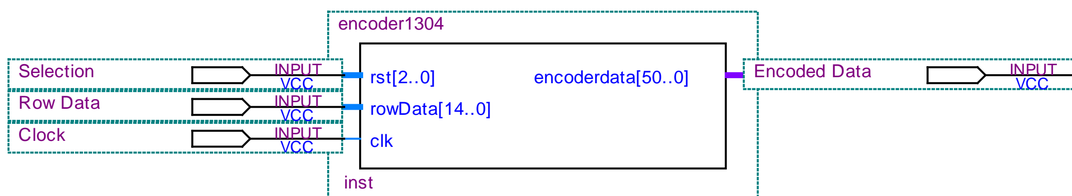

Figure 3.9: Encoder1304

Encoder1215 Module: This module use 15 binary data as input and produce 34 binary data as output shown in Figure 3.10. This module uses convolutional12 module as sub module for fifteen times. Output is generated using parallel execution of fifteen sub modules. rst is used as selection variable where '110' value is fixed for this module to be activated

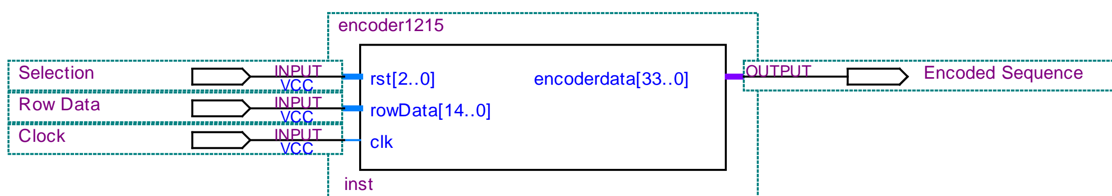

Figure 3.10: Encoder1215

Encoder1204 Module: This module use 4 binary data as input and produce 12 binary data as

output shown in Figure 3.11. Here extra 21 bits of output port remain '0'. This module use convolutional12 module as sub module for four times. Output is generated using parallel execution of four sub modules. rst is used as selection variable where '111' value is fixed for this module to be activated.

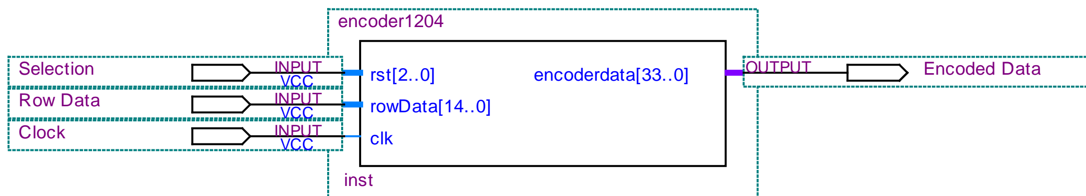

Figure 3.11: Encoder1204

Decoder1315 Module: According to Figure 3.12 this module decode 1/3 code rate and 15 trellis length encoded data at a time. rst is use to activate this decoder where '100' is fixed for this module to be activated. Input data is 51 bit length and output is 15 bit length.

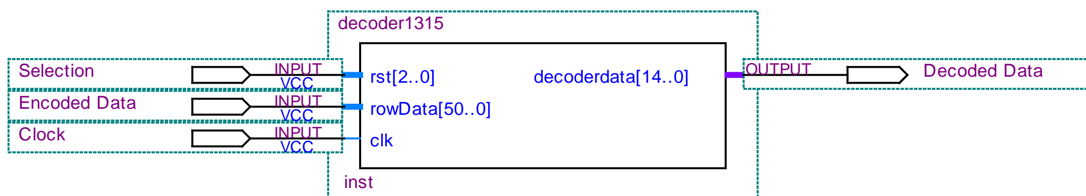

Figure 3.12: Decoder1315

Decoder1304 Module: This module decode 1/3 code rate and 4 trellis length encoded data at a time. rst is use to activate this decoder where '101' is fixed for this module to be activated. Input data is 18 bit length and output is 4 bit length. The remaining 33 input bit and 11 output bit will be ‘0’. The Figure 3.13 depict the module.

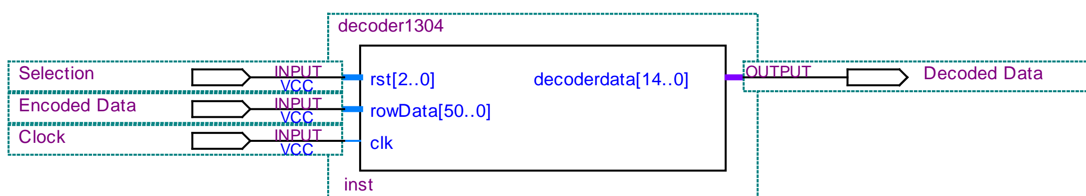

Figure 3.13: Decoder1304

Decoder1215 Module: According to Figure 3.14 this module decode 1/2 code rate and 15 trellis length encoded data at a time. rst is use to activate this decoder where '110' is fixed for this module to be activated. Input data is 34 bit length and output is 15 bit length.

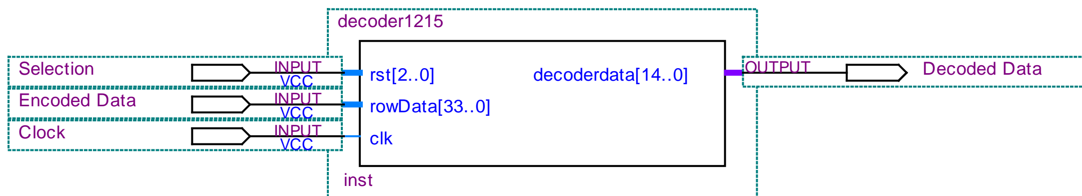

Figure 3.14: Decoder1215

Decoder1204 Module: According to Figure 3.15 this module decode 1/2 code rate and 4 trellis length encoded data at a time. rst is use to activate this decoder where '111' is fixed for this module to be activated. Input data is 12 bit length and output is 4 bit length. The remaining 21 input bit and 11 output bit will be '0' .

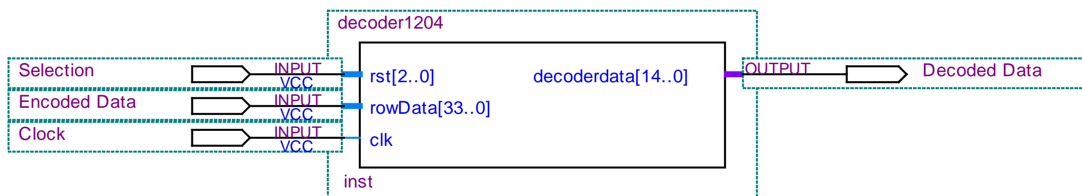

Figure 3.15: Decoder1204

Encoded and Decoder Integrated Module: All encoder and decoder modules are integrated with this module shown in Figure 3.16. Selection variable is used to choice the encoder and decoder module group at a time. There is no separate module for convolutional13 and convolutional12 due to the fact that these are integrated within each encoder module.

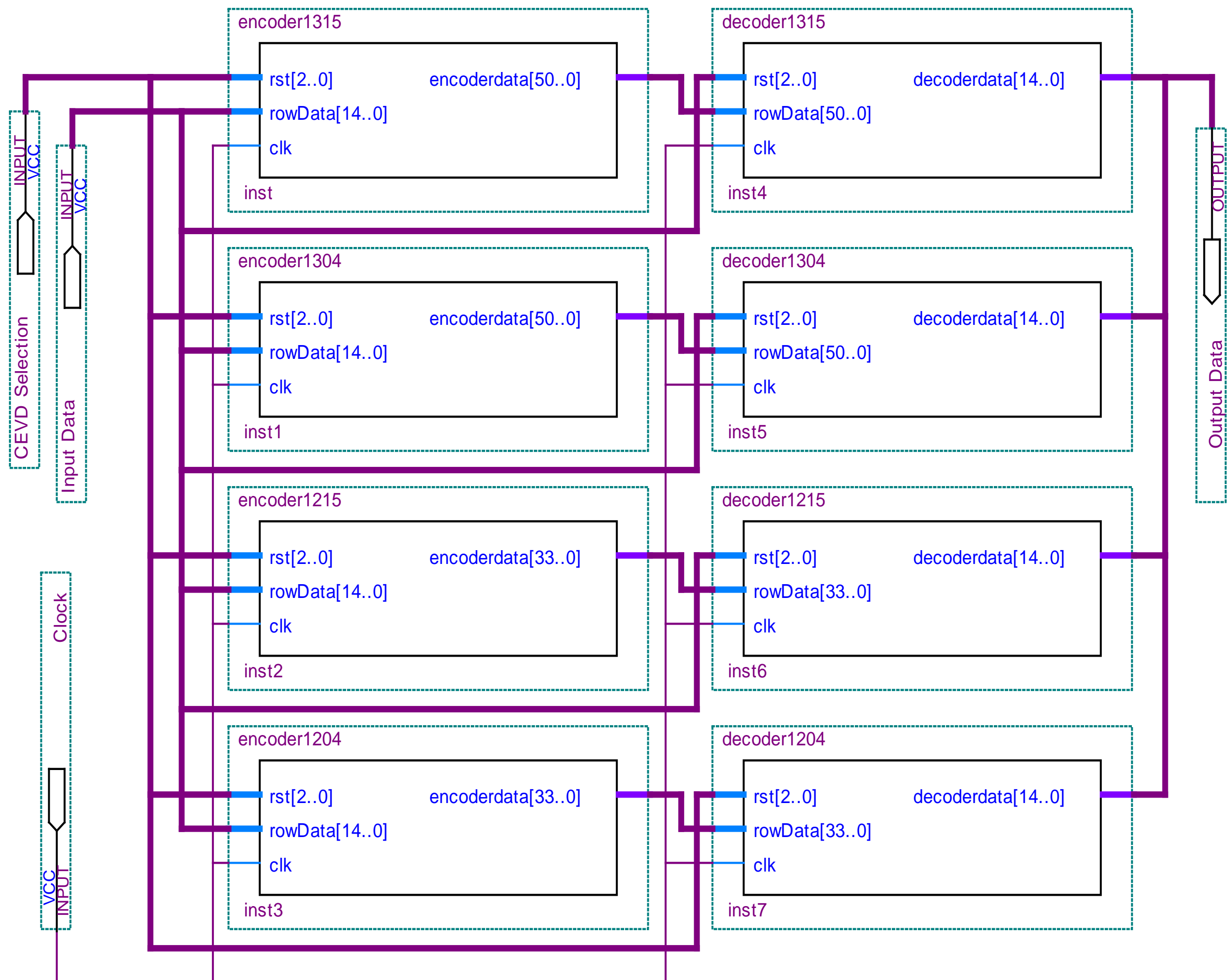

Figure 3.16: Encoded and decoder integrated module

Complete Design Block: Figure 3.17 depict the complete design block of CEVD. Three input pin and one output is shown in this block. CEVD group selector select the encoder decoder group. Row data is binary data and output data is the same to row data after performing the encode and decode operation.

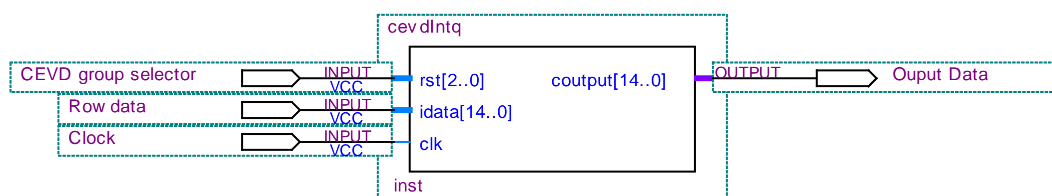

Figure 3.17: Complete design block

3.6.6 Important task and function

3.6.6.1 Next State Generator

This function take memory register data, one binary input data and generate the next state of the memory register.

3.6.6.2 Output One

This function take memory register, '1' as incoming binary data and generate the 2 or 3 output according to the Encoder coding rate.

3.6.6.3 Output Zero

This function take memory register, '0' as incoming binary data and generate the 2 or 3 output according to the Encoder coding rate.

3.6.6.4 Hamming Distance

This function take two same length binary string as input and output the humming distance of these two strings.

3.6.6.5 Maximum Path Weight

This function use global array of total path and total weight of different path as input and return maximum weighted path.

Chapter 4
Results and Discussions

4.1 Introduction

This Chapter show the simulation result and usage of chip area of the proposed system. Quartus II 7.0 web edition is used for simulation of entire design. Simulation is done using timing diagram.

4.2 Simulation Result

The simulation of the total project is done step by step to check the validity of encoder and decoder module. After performing different smaller modules simulation modules are integrated part by part according to design consideration. The integrated modules are then simulated to justify the overall outcome. Thus simulation phase is solely divided into two part

1) Simulation of encoder module

2) Simulation of decoder and encoder integrated module.

4.2.1 Simulation of Encoder Module

In this simulation phase input data is considered as 'idata' in the following Figure 4.1. 'rst' is selection variable to chose different coding rate encoder and 'odatathree' is the output coded data from 1/3 encoder. According to Figure 4.1 'idata' is 15 bit long to produce the trellis length 15. 'rst' is set to 100 which is fixed for 1/3 coding rate and 15 trellis length.

Figure 4.1: Simulation of encoder module (3,1,3) with trellis length 15

Figure 4.2 below depict the simulation of 1/3 coding rate, 4 trellis length. Here the 'rst' value is set to 101 fixed for this coding rate and trellis length. From the input bit 101100100111010 only 1010 from LSB is converted by encoder leaving the rest of 11 bits to binary 0 in 'odatathree'.

Figure 4.2: Simulation of encoder module (3,1,3) with trellis length 4

The next two Figure 4.3 and Figure 4.4 are same to above simulation but only difference is the code rate 1/2 so the output sequence is double by input sequence. Here value of 'rst' is also different and unique in order to maintain selection criteria.

Figure 4.3: Simulation of encoder module (2,1,3) with trellis length 15

Figure 4.4: Simulation of encoder module (2,1,3) with trellis length 4

4.2.2 Simulation of Decoder and Encoder Integrated Module

In this simulation phase the decoder module is designed separately and integrated with Encoder module due to the fact that the output of encoder will be the input of decoder. Here 'idata' is input data in binary format . 'odatatwo', 'odatathree' is for rate 1/2 and 1/3 encoding output. 'rst' is used to select different coding rate and trellis length. 'rst' is three bit and MSB of 'rst' is always high when the system is in operation . Deep purple area of 'coutput' Indicate the total delay it taken for decoding algorithm. For long trellis length and high code rate it will take more time so the region is expanded compare with sort trellis length and low code rate. After encoding the input data bit the sequence in output is altered randomly to test error

recovery functionality of decoder. Thus output in from encoder module is erroneous output rather then original output.

In the following Figure 4.5 as 'rst' is 100 the overall integrated module activates the code rate 1/3 and trellis length 15 encoder and decoder. The Figure show that 'odatatwo' is 0 bit sequence as it is inactivated.

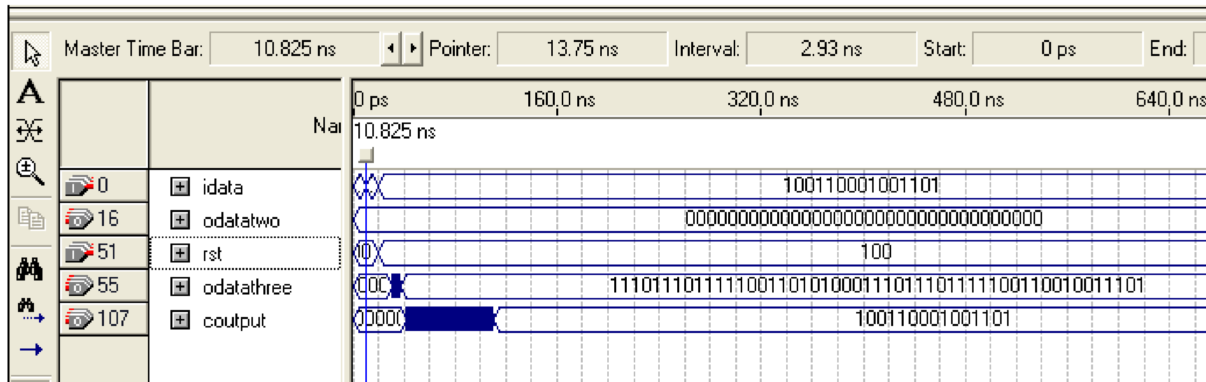

Figure 4.5: Coding rate 1/3, trellis length 15

In the following Figure 4.6 when 'rst' is 101 the overall integrated module activates the code rate 1/3 and trillis length 4 encoder and decoder. The Figure show that 'odatatwo' is 0 bit sequence as it is inactivated and only four LSB bit of input data is encoded and decoded remaining the rest of the bit intact.

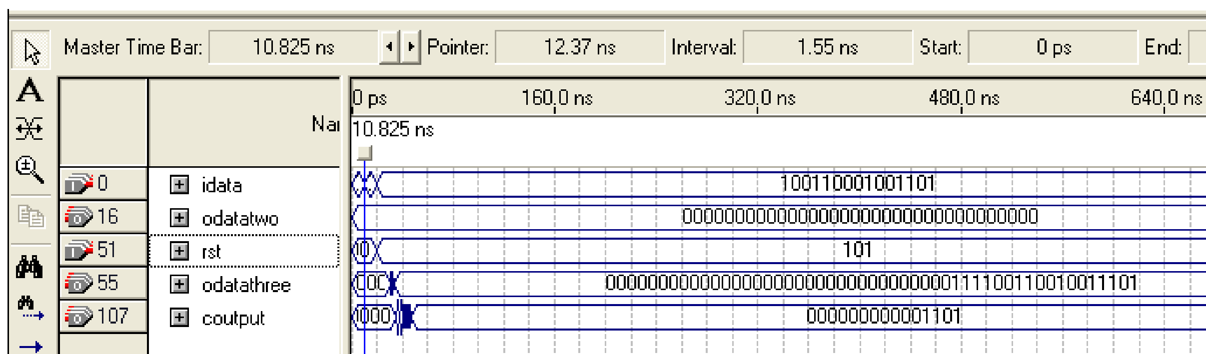

Figure 4.6: Coding rate 1/3, trellis length 4

Figure 4.7 shows the 'rst' variable 110 which imply that code rate 1/2 and trillis length 15 is activated.

Figure 4.7: Coding rate 1/2, trellis length 15

Figure 4.8 shows the 'rst' variable 111 which imply that code rate 1/2 and trillis length 15 is activated.

Figure 4.8: Coding rate 1/2, trellis length 4

4.3 Implementation on FPGA

IICT,BUET owns a Altera’s DE2 FPGA board to implement any complex digital design. The proposed design has been implemented on the FPGA mounted on the DE2 board. Leftmost three SWITCH on this board used as selection variable and remaining fifteen SWITCH is for binary input. Fifteen red LED is used as output binary data.

Figure 4.9: SWITCH input LED output of coding rate 1/2, trellis length 4

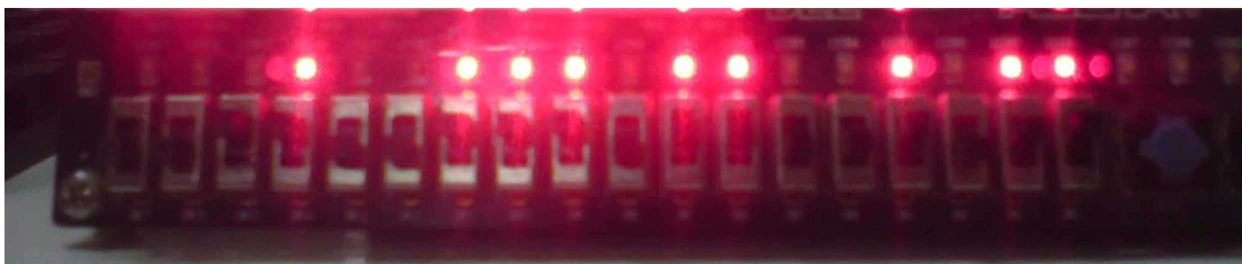

Figure 4.10: SWITCH input LED output of coding rate 1/2, trellis length 15

Figure 4.11: SWITCH input LED output of coding rate 1/3, trellis length 4

Figure 4.12: SWITCH input LED output of coding rate 1/3, trellis length 15

For input data 110100110111001 Figure 4.9 shows coding rate 1/2, trellis length 4 output. Figure 4.10 shows coding rate 1/2, trellis length 15 output. Figure 4.11 shows coding rate 1/3, trellis length 4 outputs. Figure 4.12 shows coding rate 1/3, trellis length 15 output. So it shows that the CEVD is working correctly.

4.4 Usage of Chip Area

Cyclone II FPGA has total 33216 logic cell. Among these only 22.2904% is used for implementing this encoder and decoder. Area usage for each module is given bellow. The proposed system can expand five times with same complexity.

Table 4.1: Device utilization of Cyclone II FPGA

Module	Code Rate	Trellis Length	Logic Cell	Percent of total Logic Cell
Encoder	1/3	15	23	0.0692 %
Encoder	1/3	4	2	0.0060 %
Encoder	1/2	15	16	0.0481 %
Encoder	1/2	4	6	0.0180 %
Decoder	1/3	15	3310	9.9650 %
Decoder	1/3	4	115	0.3462 %
Decoder	1/2	15	3598	10.8321 %
Decoder	1/2	4	287	0.8640 %

4.5 Comparison

A tabular comparison of different system of research work is given below

The performance of the CEVD achieved in this research has been compared with that of other researchers [4-6]. It has been shown in the Table 4.2.

Table 4.2**:** Comparison of different Implementations

Research work	Variable Trillis Length	Variable Code Rate
Azim, C.F. al[4] DSP	No	No
Bissi, L et. al.[6] Xilinx XCV300PQ240-4 FPGA	No	Yes
Wong, Y.S. al [5] Xilinx Virtex-II Pro,XC2vp30	Yes	No
This research	Yes	Yes

From Table 4.2 it is observed that variable trellis and variable code rate are separately implemented in different research. Simultaneous variation on this two parameter is only done in this research project. Research paper [4] shows the simple implementation of the encoder and decoder with DSP which is slow in operation. Research paper [6] use variable code rate without changing the trellis length. Research paper [5] user variable trellis length in same code rate. So dynamicity is done in this research work.

Chapter 5
Conclusion

5.1 Conclusion

The objective of the proposed research was to design a programmable CEVD using code rate and trellis length as parameter. The proposed CEVD has been designed using Verilog HDL. The simulation result of each module and that of whole integrated module of the CEVD verify its desired functionality. Hardware implementation results also validate the truth. Comparison with different researcher's approaches to implement the CEVD is also done. Comparison shows limited configuration facility of different researcher's approaches. According to the demand of environment and error recoverability flexible configuration facility is introduced in proposed system. Thus whenever the proposed system operates on low noisy environment high code rate and long trellis length Viterbi decoder can be switched to low code rate and short trellis length Viterbi decoder. Also for high noisy environment low code rate and short trellis length Viterbi decoder can be switched to high code rate and long trellis length Viterbi decoder The proposed system is fully code based using verilog HDL modularized facility of task and function. Thus the system can be easily expandable in compare with other.

5.2 Future Work

Wireless environment can be classified according to different noise level where each environment will use fixed code rate, trellis length and constraint length. According to the chip area used by the project the design can be expanded seven times with same complexity. Thus the total system may be parameterized on different environmental aspect. Also power analysis can be done on the proposed system.

References:

[1] Shannon, C.E. “A mathematical theory of Communication”, Bell Sys.Tech .J. vol. 27, pp. 379-423 and 23- 656, 1948.

[2] Hamming, R.W. “Error detecting and correcting codes”, Bell Sys.Tech .J. vol. 29, pp. 147-160, 1960.

[3] Golay, M. J. E. “Notes on digital coding”, Proc. IEEE, Vol. 37, p. 657, 1949.

[4] Azim, C.F., Monir, M.G. “Implementation of Viterbi Decoder for WCDMA System”, Proceedings of IEEE International Conference (NMIC 2005), pp. 1-3, 2005.

[5] Wong, Y.S., Jian, W., Hui, C. O., Kyun, Ng. C., Noordi, N.K. “Implementation of Convolutional Encoder and Viterbi Decoder using VHDL”, In Proceedings of IEEE International conference on Research and Development Malaysia, November 2009.

[6] Bissi, L., Placidi, P., Baruffa, G., Scorzoni, A. “A Multi- Standard Reconfigurable Viterbi Decoder using Embedded FPGA blocks”, In Proceedings of the 9th EUROMICRO Conference on Digital System Design (DSD’06),pp. 146-153, 2006.

[7] Shaker, S.W., Elramly, S.H., Shehata, K.A. “FPGA Implementation of a reconfigurable Viterbi Decoder for WiMax Receiver”, IEEE International conference on Microelectronics, pp. 246-267, 2009.

[8] Charan, L., Editor www.complextoreal.com, Tutorial 12, Coding and decoding with Convolutional Codes.

Appendix A

--Project Coding--

-- Convolutional Encoder Module (3,1,3)

```
module convolutional13(rst,idata , odata) ;
input rst ;
input [2:0]idata;
output [2:0]odata;
reg [2:0] odata;
always @(rst or idata   )
 begin
    if (rst==1'b1)
     begin
               case(idata)
                 3'b000: odata=3'b000;
                 3'b001: odata=3'b111;
                 3'b010: odata=3'b011;
                 3'b011: odata=3'b100;
                 3'b100: odata=3'b101;
                 3'b101: odata=3'b010;
                 3'b110: odata=3'b110;
                 3'b111: odata=3'b001;
                endcase
       odata=odata;
     end
     else
     begin
     odata=0;
     end
 end
endmodule
```

--Convolutional Encoder Module (2,1,3)

```
module convolutional12(rst,idata , odata) ;
input rst ;
input [2:0]idata;
output [1:0]odata;
reg [1:0] odata;
```

```
always @(rst or idata   )
 begin
    if (rst==1)
     begin
                  case(idata)
                    3'b000: odata=2'b00;
                   3'b001: odata=2'b01;
                   3'b010: odata=2'b11;
                   3'b011: odata=2'b10;
                   3'b100: odata=2'b11;
                   3'b101: odata=2'b10;
                   3'b110: odata=2'b00;
                   3'b111: odata=2'b01;
                  endcase

        odata=odata;
     end

    else
     begin
     odata=0;
     end
 end
```

--Encoder Module (3,1,3) With Trillis length 15

```
module encoder1315(rst, rowData , encoderdata );
input [2:0] rst;
input [14:0]rowData;

output [50:0] encoderdata;
reg [50:0] encoderdata;
wire [14:0] mdata ;
wire [2:0] ndata [16:0] ;

assign  mdata = rowData;
```

```
convolutional13 c131(1'b1, {mdata[0],1'b0    ,    1'b0} ,ndata[0] );
convolutional13 c132(1'b1, {mdata[1],mdata[0],    1'b0} ,ndata[1] );
convolutional13 c133(1'b1, {mdata[2],mdata[1],mdata[0]} ,ndata[2] );
convolutional13 c134(1'b1, {mdata[3],mdata[2],mdata[1]} ,ndata[3] );
convolutional13 c135(1'b1, {mdata[4],mdata[3],mdata[2]} ,ndata[4] );
convolutional13 c136(1'b1, {mdata[5],mdata[4],mdata[3]} ,ndata[5] );
convolutional13 c137(1'b1, {mdata[6],mdata[5],mdata[4]} ,ndata[6] );
convolutional13 c138(1'b1, {mdata[7],mdata[6],mdata[5]} ,ndata[7] );
convolutional13 c139(1'b1, {mdata[8],mdata[7],mdata[6]} ,ndata[8] );
convolutional13 c13a(1'b1, {mdata[9],mdata[8],mdata[7]} ,ndata[9] );
convolutional13 c13b(1'b1, {mdata[10],mdata[9],mdata[8]} ,ndata[10] );
convolutional13 c13c(1'b1, {mdata[11],mdata[10],mdata[9]} ,ndata[11] );
convolutional13 c13d(1'b1, {mdata[12],mdata[11],mdata[10]} ,ndata[12] );
convolutional13 c13e(1'b1, {mdata[13],mdata[12],mdata[11]} ,ndata[13] );
convolutional13 c13f(1'b1, {mdata[14],mdata[13],mdata[12]} ,ndata[14] );
convolutional13 c13g(1'b1, {    1'b0 ,mdata[14],mdata[13]} ,ndata[15] );
convolutional13 c13h(1'b1, {    1'b0 ,    1'b0,mdata[14]} ,ndata[16] );

always @(rst or rowData   )
 begin
    if (rst==3'b100)
     begin

encoderdata={ndata[16],ndata[15],ndata[14],ndata[13],ndata[12],ndata[11],ndata[10],ndata[9],ndata[8],ndata[7]
,ndata[6],ndata[5],ndata[4],ndata[3],ndata[2],ndata[1],ndata[0]};
      encoderdata=encoderdata;
     end

    else
     begin
     encoderdata=0;
     end

 end
endmodule
```

--Encoder Module (3,1,3) With Trillis length 4

```
module encoder1304(rst, rowData , encoderdata );
input [2:0]rst;
input [14:0]rowData;

output [50:0] encoderdata;
reg [50:0] encoderdata;
wire [14:0] mdata  ;
wire [2:0] ndata [5:0]  ;

assign  mdata = rowData;

convolutional13 d131(1'b1, {mdata[0],1'b0    ,    1'b0} ,ndata[0] );
convolutional13 d132(1'b1, {mdata[1],mdata[0],    1'b0} ,ndata[1] );
convolutional13 d133(1'b1, {mdata[2],mdata[1],mdata[0]} ,ndata[2] );
convolutional13 d134(1'b1, {mdata[3],mdata[2],mdata[1]} ,ndata[3] );
convolutional13 d135(1'b1, { 1'b0   ,mdata[3],mdata[2]} ,ndata[4] );
convolutional13 d136(1'b1, { 1'b0,       1'b0,mdata[3]} ,ndata[5] );

always @(rst or rowData   )
 begin
    if (rst==3'b101)
     begin


encoderdata={32'b0000000000000000000000000000000000000,ndata[5],ndata[4],ndata[3],ndata[2],ndata[1],ndata[0]
};
      encoderdata=encoderdata;
     end

    else
     begin
     encoderdata=0;
     end

 end
endmodule
```

-- Encoder Module (2,1,3) With Trillis length 15

```
module encoder1215(rst, rowData , encoderdata );
input [2:0] rst;
input [14:0]rowData;

output [33:0] encoderdata;
reg [33:0] encoderdata;
wire [14:0] mdata ;
wire [1:0] ndata [16:0]  ;
assign  mdata = rowData;
convolutional12 c121 (1'b1, {mdata[0], 1'b0     ,1'b0      } ,ndata[0] );
convolutional12 c122 (1'b1, {mdata[1], mdata[0] ,1'b0      } ,ndata[1] );
convolutional12 c123 (1'b1, {mdata[2], mdata[1] ,mdata[0]  } ,ndata[2] );
convolutional12 c124 (1'b1, {mdata[3], mdata[2] ,mdata[1]  } ,ndata[3] );
convolutional12 c125 (1'b1, {mdata[4], mdata[3] ,mdata[2]  } ,ndata[4] );
convolutional12 c126 (1'b1, {mdata[5], mdata[4] ,mdata[3]  } ,ndata[5] );
convolutional12 c127 (1'b1, {mdata[6], mdata[5] ,mdata[4]  } ,ndata[6] );
convolutional12 c128 (1'b1, {mdata[7], mdata[6] ,mdata[5]  } ,ndata[7] );
convolutional12 c129 (1'b1, {mdata[8], mdata[7] ,mdata[6]  } ,ndata[8] );
convolutional12 c12a (1'b1, {mdata[9], mdata[8] ,mdata[7]  } ,ndata[9] );
convolutional12 c12b (1'b1, {mdata[10],mdata[9] ,mdata[8]  } ,ndata[10] );
convolutional12 c12c (1'b1, {mdata[11],mdata[10] ,mdata[9]  } ,ndata[11] );
convolutional12 c12d (1'b1, {mdata[12], mdata[11] ,mdata[10]  } ,ndata[12] );
convolutional12 c12e (1'b1, {mdata[13],mdata[12] ,mdata[11]  } ,ndata[13] );
convolutional12 c12f (1'b1, {mdata[14],mdata[13] ,mdata[12]  } ,ndata[14] );
convolutional12 c12g (1'b1, {1'b0 ,    mdata[14],mdata[13]  } ,ndata[15] );
convolutional12 c12h (1'b1, {1'b0 ,    1'b0     ,mdata[14] }  ,ndata[16] );

always @(rst or rowData   )
 begin
    if (rst==3'b110)
     begin

encoderdata={ndata[16],ndata[15],ndata[14],ndata[13],ndata[12],ndata[11],ndata[10],ndata[9],ndata[8],ndata[7]
,ndata[6],ndata[5],ndata[4],ndata[3],ndata[2],ndata[1],ndata[0]};
      encoderdata=encoderdata;
     end
        else
     begin
     encoderdata=0;
```

```
    end

 end
endmodule
```

-- Encoder Module (2,1,3) With Trillis length 4

```
module encoder1204(rst, rowData , encoderdata );
input [2:0] rst;
input [14:0]rowData;

output [33:0] encoderdata;
reg [33:0] encoderdata;
wire [14:0] mdata ;
wire [1:0] ndata [5:0]  ;
assign  mdata = rowData;

convolutional12 d121 (1'b1, {mdata[0], 1'b0     ,1'b0      } ,ndata[0] );
convolutional12 d122 (1'b1, {mdata[1], mdata[0] ,1'b0      } ,ndata[1] );
convolutional12 d123 (1'b1, {mdata[2], mdata[1] ,mdata[0]  } ,ndata[2] );
convolutional12 d124 (1'b1, {mdata[3], mdata[2] ,mdata[1]  } ,ndata[3] );
convolutional12 d125 (1'b1, {1'b0 ,    mdata[3], mdata[2]  } ,ndata[4] );
convolutional12 d126 (1'b1, {1'b0 ,    1'b0     ,mdata[3] }  ,ndata[5] );

always @(rst or rowData   )
 begin
    begin
    if (rst==3'b111)
     begin
      encoderdata={22'b0000000000000000000000,ndata[5],ndata[4],ndata[3],ndata[2],ndata[1],ndata[0]};
      encoderdata=encoderdata;
     end
   else
      begin
       encoderdata=0;
      end

 end

 end
```

endmodule

--Decoder Module (3,1,3) With Trillis length 4

```
module decoder1304(rst , rowData , decoderdata );
input [2:0] rst;
input [50:0]rowData;
reg [50:0]rowData1;

output [14:0] decoderdata;
reg [14:0] decoderdata;

reg [35:0]pathmatrix[7:0] ;
integer pathweight[7:0];
integer pWeight;
reg [35:0]pString;

function [1:0]nextstateOneThird(input [1:0]pstate , input inputdata  );
  begin

          nextstateOneThird = 2'b00;
        if ((pstate == 2'b00) &&  (inputdata == 1'b0 ))
           nextstateOneThird = 2'b00;
      else if ((pstate == 2'b00) && (inputdata == 1'b1 ))
           nextstateOneThird = 2'b10;
      else if ((pstate == 2'b10) && (inputdata == 1'b0 ))
           nextstateOneThird = 2'b01;
      else if ((pstate == 2'b10) && (inputdata == 1'b1 ))
           nextstateOneThird = 2'b11;
      else if ((pstate == 2'b01) && (inputdata == 1'b0 ))
           nextstateOneThird = 2'b00;
      else if ((pstate == 2'b01) && (inputdata == 1'b1 ))
           nextstateOneThird = 2'b10;
      else if ((pstate == 2'b11) && (inputdata == 1'b0 ))
           nextstateOneThird = 2'b01;
      else if ((pstate == 2'b11) && (inputdata == 1'b1 ))
           nextstateOneThird = 2'b11;
      end
endfunction

function [2:0]outputOneOneThird(input [1:0]pstate );
```

```
  begin

      outputOneOneThird = 3'b000;
                  if  (pstate == 2'b00)
                   outputOneOneThird = 3'b101;
              else if  (pstate == 2'b01 )
                   outputOneOneThird = 3'b010;
              else if  (pstate == 2'b10 )
                   outputOneOneThird = 3'b110;
              else if  (pstate == 2'b11 )
                   outputOneOneThird = 2'b001;
  end
endfunction

function [2:0]outputZeroOneThird(input [1:0]pstate );
 begin
                 outputZeroOneThird = 3'b000;
                  if (pstate == 2'b00)
                   outputZeroOneThird = 3'b000;
              else if (pstate == 2'b10)
                   outputZeroOneThird = 3'b011;
              else if (pstate == 2'b01)
                   outputZeroOneThird = 3'b111;
              else if (pstate == 2'b11)
                   outputZeroOneThird = 3'b100;

 end
endfunction

function [31:0]hummingDistanceOneThird(input [2:0]p1String, input [2:0]p2String );
  begin
   hummingDistanceOneThird = 0;

   if  (p1String[0]==p2String[0])
       begin
       hummingDistanceOneThird = hummingDistanceOneThird + 1;
       end
   if  (p1String[1]==p2String[1])
       begin
```

```
      hummingDistanceOneThird = hummingDistanceOneThird + 1;
      end
   if  (p1String[2]==p2String[2])
      begin
      hummingDistanceOneThird = hummingDistanceOneThird + 1;
      end

  end
endfunction

function  [31:0] findMaxpathWeightOneThird(input a1 );
  integer i;
  integer j;
  begin
   findMaxpathWeightOneThird=0;
   j=0;
  for(i=0; i<=3; i=i+1)
     begin
      if (pathweight[i]> j)
        begin
        j=pathweight[i];
        findMaxpathWeightOneThird=i;
        end
     end
   end
endfunction

task optimizePathMatrixOneThird(input a1);

  reg [35:0]temppathmatrix[7:0];
  integer temppathweight[7:0];
  integer i;



  begin

  for (i=0; i<=3; i=i+1 )
   begin
    if  (pathweight[2*i] > pathweight[2*i+1])
```

```
      begin
      temppathmatrix[i]=  pathmatrix[2*i];
      temppathweight[i]=  pathweight[2*i];
      end
    else
      begin
      temppathmatrix[i]=  pathmatrix[2*i+1];
      temppathweight[i]=  pathweight[2*i+1];
      end
   end

  for (i=0; i<=7; i=i+1)
   begin
     pathmatrix[i] = temppathmatrix[i];
     pathweight[i] = temppathweight[i];

   end

 end
endtask

task viterbi12(input [50:0]adata , output [14:0]bdata  );
reg errorFlag;
reg [1:0]nstate;
reg [1:0]nstatep;
reg [2:0]soutput;
integer a;
integer b;
integer i;
integer p;

begin
              b=0;
              p=0;
              errorFlag=0;
              nstate=2'b00;
              nstatep=2'b00;
    for (a=0; a<=5; a=a+1)
                  begin
```

```
soutput = {adata[3*a+2],adata[3*a+1],adata[3*a]};
if (errorFlag==0)
  begin
    if   (outputOneOneThird(nstate) == soutput  )
      begin
        nstate = nextstateOneThird(nstate, 1);
        pathmatrix[0][2*a+3] = nstate[1];
        pathmatrix[0][2*a+2] = nstate[0];
      end
    else if (outputZeroOneThird(nstate) == soutput)
     begin
        nstate = nextstateOneThird(nstate, 0);
        pathmatrix[0][2*a+3] = nstate[1];
        pathmatrix[0][2*a+2] = nstate[0];
     end
    else
      begin
        errorFlag=1;
      end
  end
if (errorFlag==1)
  begin
    p=3;
    for(b=0; b<=3; b=b+1)
      begin
              nstate = { pathmatrix[b][2*a+1],pathmatrix[b][2*a]};
              pString=pathmatrix[b];
              pWeight=pathweight[b];
              pathweight[b] = pWeight + hummingDistanceOneThird(outputOneOneThird(nstate), 
soutput);
              nstatep = nextstateOneThird(nstate, 1);

              pathmatrix[b][2*a+3] = nstatep[1];
              pathmatrix[b][2*a+2] = nstatep[0];



              pathweight[b+p+1]  = pWeight + 
hummingDistanceOneThird(outputZeroOneThird(nstate), soutput);
```

```
                                        nstatep = nextstateOneThird(nstate, 0);
                                        pathmatrix[b+p+1]=pString;
                                        pathmatrix[b+p+1][2*a+3] = nstatep[1];
                                        pathmatrix[b+p+1][2*a+2] = nstatep[0];
                                       end
                                optimizePathMatrixOneThird(1);
                            end

                    end
                   p = findMaxpathWeightOneThird(1);
    bdata=15'b000000000000000;
    for(i=0; i<=3; i=i+1)
     begin
       bdata[i]= pathmatrix[p][2*i+3];
     end

end
endtask

task clearPathWeightmatrix(input a1);
 integer i;
 begin
  for(i=0; i<=7; i=i+1)
     begin
      pathmatrix[i] = 0;
      pathweight[i] = 0;
     end
 end
endtask


always @( rst or rowData   )
 begin
    if (rst==3'b101 )
      begin
              clearPathWeightmatrix(1);
              rowData1=rowData;
              rowData1[0]=0;
              viterbi12(rowData1,decoderdata);
```

```
                decoderdata=decoderdata;
      end
    else
      begin
                decoderdata=0;
      end

 end

endmodule
```

--Decoder Module (3,1,3) With Trillis length 15

```
module decoder1315(rst , rowData , decoderdata );
input [2:0] rst;
input [50:0]rowData;
reg [50:0]rowData1;

output [14:0] decoderdata;
reg [14:0] decoderdata;

reg [35:0]pathmatrix[7:0] ;
integer pathweight[7:0];
integer pWeight;
reg [35:0]pString;

function [1:0]nextstateOneThird(input [1:0]pstate , input inputdata  );
  begin

                   nextstateOneThird = 2'b00;
                 if ((pstate == 2'b00) &&  (inputdata == 1'b0 ))
                    nextstateOneThird = 2'b00;
              else if ((pstate == 2'b00) && (inputdata == 1'b1 ))
                    nextstateOneThird = 2'b10;
              else if ((pstate == 2'b10) && (inputdata == 1'b0 ))
                    nextstateOneThird = 2'b01;
              else if ((pstate == 2'b10) && (inputdata == 1'b1 ))
                    nextstateOneThird = 2'b11;
              else if ((pstate == 2'b01) && (inputdata == 1'b0 ))
                    nextstateOneThird = 2'b00;
              else if ((pstate == 2'b01) && (inputdata == 1'b1 ))
```

```
                nextstateOneThird = 2'b10;
            else if ((pstate == 2'b11) && (inputdata == 1'b0 ))
                nextstateOneThird = 2'b01;
            else if ((pstate == 2'b11) && (inputdata == 1'b1 ))
                nextstateOneThird = 2'b11;
            end
endfunction

function [2:0]outputOneOneThird(input [1:0]pstate );
  begin

      outputOneOneThird = 3'b000;
              if  (pstate == 2'b00)
               outputOneOneThird = 3'b101;
            else if  (pstate == 2'b01 )
               outputOneOneThird = 3'b010;
            else if  (pstate == 2'b10 )
               outputOneOneThird = 3'b110;
            else if  (pstate == 2'b11 )
               outputOneOneThird = 2'b001;
  end
endfunction

function [2:0]outputZeroOneThird(input [1:0]pstate );
 begin
             outputZeroOneThird = 3'b000;
              if (pstate == 2'b00)
               outputZeroOneThird = 3'b000;
            else if (pstate == 2'b10)
               outputZeroOneThird = 3'b011;
            else if (pstate == 2'b01)
               outputZeroOneThird = 3'b111;
            else if (pstate == 2'b11)
               outputZeroOneThird = 3'b100;

 end
endfunction

function [31:0]hummingDistanceOneThird(input [2:0]p1String, input [2:0]p2String );
```

```
  begin
   hummingDistanceOneThird = 0;

   if  (p1String[0]==p2String[0])
       begin
       hummingDistanceOneThird = hummingDistanceOneThird + 1;
       end
   if  (p1String[1]==p2String[1])
       begin
       hummingDistanceOneThird = hummingDistanceOneThird + 1;
       end
   if  (p1String[2]==p2String[2])
       begin
       hummingDistanceOneThird = hummingDistanceOneThird + 1;
       end

  end
endfunction

function  [31:0] findMaxpathWeightOneThird(input a1 );
  integer i;
  integer j;
  begin
    findMaxpathWeightOneThird=0;
    j=0;
  for(i=0; i<=3; i=i+1)
     begin
      if (pathweight[i]> j)
         begin
         j=pathweight[i];
         findMaxpathWeightOneThird=i;
         end
     end
   end
endfunction

task optimizePathMatrixOneThird(input a1);

  reg [35:0]temppathmatrix[7:0];
```

```
  integer temppathweight[7:0];
  integer i;



  begin

  for (i=0; i<=3; i=i+1 )
   begin
    if  (pathweight[2*i] > pathweight[2*i+1])
      begin
      temppathmatrix[i]=  pathmatrix[2*i];
      temppathweight[i]=  pathweight[2*i];
      end
    else
      begin
      temppathmatrix[i]=  pathmatrix[2*i+1];
      temppathweight[i]=  pathweight[2*i+1];
      end
   end

  for (i=0; i<=7; i=i+1)
   begin
     pathmatrix[i] = temppathmatrix[i];
     pathweight[i] = temppathweight[i];

   end

 end
endtask

task viterbi12(input [50:0]adata , output [14:0]bdata  );
reg errorFlag;
reg [1:0]nstate;
reg [1:0]nstatep;
reg [2:0]soutput;
integer a;
integer b;
integer i;
integer p;
```

```
begin
            b=0;
            p=0;
            errorFlag=0;
            nstate=2'b00;
            nstatep=2'b00;
   for (a=0; a<=16; a=a+1)
               begin
                 soutput = {adata[3*a+2],adata[3*a+1],adata[3*a]};
                 if (errorFlag==0)
                   begin
                     if   (outputOneOneThird(nstate) == soutput  )
                       begin
                         nstate = nextstateOneThird(nstate, 1);
                         pathmatrix[0][2*a+3] = nstate[1];
                         pathmatrix[0][2*a+2] = nstate[0];
                       end
                     else if (outputZeroOneThird(nstate) == soutput)
                      begin
                         nstate = nextstateOneThird(nstate, 0);
                         pathmatrix[0][2*a+3] = nstate[1];
                         pathmatrix[0][2*a+2] = nstate[0];
                      end
                     else
                       begin
                         errorFlag=1;
                       end
                   end
                 if (errorFlag==1)
                   begin
                     p=3;
                     for(b=0; b<=3; b=b+1)
                       begin
                                    nstate = { pathmatrix[b][2*a+1],pathmatrix[b][2*a]};
                                    pString=pathmatrix[b];
                                    pWeight=pathweight[b];
                                    pathweight[b] = pWeight + hummingDistanceOneThird(outputOneOneThird(nstate),
soutput);
```

```
                              nstatep = nextstateOneThird(nstate, 1);

                              pathmatrix[b][2*a+3] = nstatep[1];
                              pathmatrix[b][2*a+2] = nstatep[0];



                              pathweight[b+p+1]  = pWeight +
hummingDistanceOneThird(outputZeroOneThird(nstate), soutput);

                              nstatep = nextstateOneThird(nstate, 0);
                              pathmatrix[b+p+1]=pString;
                              pathmatrix[b+p+1][2*a+3] = nstatep[1];
                              pathmatrix[b+p+1][2*a+2] = nstatep[0];
                             end
                        optimizePathMatrixOneThird(1);
                     end

                end
               p = findMaxpathWeightOneThird(1);
  for(i=0; i<=14; i=i+1)
   begin
     bdata[i]= pathmatrix[p][2*i+3];
   end

end
endtask

task clearPathWeightmatrix(input a1);
 integer i;
 begin
  for(i=0; i<=7; i=i+1)
   begin
    pathmatrix[i] = 0;
    pathweight[i] = 0;
   end

 end
endtask
```

```
always @( rst or rowData   )
 begin
   rowData1=0;
   if (rst==3'b100)
      begin
              clearPathWeightmatrix(1);
              rowData1=rowData;
              rowData1[0]=0;
              viterbi12(rowData1,decoderdata);
              decoderdata=decoderdata;
      end
    else
      begin
              decoderdata=0;
      end

 end

endmodule
```

--Decoder Module (2,1,3) With Trillis length 4

```
module decoder1204(rst , rowData , decoderdata );
input [2:0] rst;
input [33:0]rowData;
reg [33:0]rowData1;

output [14:0] decoderdata;
reg [14:0] decoderdata;



reg [35:0]pathmatrix[7:0] ;
integer pathweight[7:0];

integer pWeight;
reg [35:0]pString;



function [1:0]nextstateHalf(input [1:0]pstate , input inputdata  );
  begin
```

```
                    nextstateHalf = 2'b00;
                  if ((pstate == 2'b00) && (inputdata == 1'b0 ))
                    nextstateHalf = 2'b00;
              else if ((pstate == 2'b00) && (inputdata == 1'b1 ))
                    nextstateHalf = 2'b10;
              else if ((pstate == 2'b01) && (inputdata == 1'b0 ))
                    nextstateHalf = 2'b00;
              else if ((pstate == 2'b01) && (inputdata == 1'b1 ))
                    nextstateHalf = 2'b10;
              else if ((pstate == 2'b10) && (inputdata == 1'b0 ))
                    nextstateHalf = 2'b01;
              else if ((pstate == 2'b10) && (inputdata == 1'b1 ))
                    nextstateHalf = 2'b11;
              else if ((pstate == 2'b11) && (inputdata == 1'b0 ))
                    nextstateHalf = 2'b01;
     else if ((pstate == 2'b11) && (inputdata == 1'b1 ))
                    nextstateHalf = 2'b11;
  end
endfunction

function [1:0]outputOneHalf(input [1:0]pstate );
  begin
                //   110 111
               outputOneHalf = 2'b00;
                  if  (pstate == 2'b00)
                   outputOneHalf = 2'b11;
              else if  (pstate == 2'b01 )
                   outputOneHalf = 2'b01;
              else if  (pstate == 2'b10 )
                   outputOneHalf = 2'b00;
              else if  (pstate == 2'b11 )
                   outputOneHalf = 2'b10;
              end
endfunction


function [1:0]outputZeroHalf(input [1:0]pstate );
 begin
```

```
                outputZeroHalf = 2'b00;
               if (pstate == 2'b00)
                outputZeroHalf = 2'b00;
            else if (pstate == 2'b01)
                outputZeroHalf = 2'b10;
            else if (pstate == 2'b10)
                outputZeroHalf = 2'b11;
            else if (pstate == 2'b11)
                outputZeroHalf = 2'b01;

 end
endfunction



function [31:0]hummingDistanceHalf(input [1:0]p1String, input [1:0]p2String );
  begin
   hummingDistanceHalf = 0;

   if  (p1String[0]==p2String[0])
       begin
       hummingDistanceHalf = hummingDistanceHalf + 1;
       end
   if  (p1String[1]==p2String[1])
       begin
       hummingDistanceHalf = hummingDistanceHalf + 1;
       end

  end
endfunction

function  [31:0] findMaxpathWeightHalf(input a1 );
  integer i;
  integer j;
  begin
    findMaxpathWeightHalf=0;
    j=0;
  for(i=0; i<=3; i=i+1)
     begin
      if (pathweight[i]> j)
```

```
        begin
        j=pathweight[i];
        findMaxpathWeightHalf=i;
        end
      end
    end
endfunction


task optimizePathMatrixHalf(input a1);

  reg [35:0]temppathmatrix[7:0] ;
  integer temppathweight[7:0];
  integer i;



  begin
  for (i=0; i<=3; i=i+1 )
   begin
    if  (pathweight[2*i] > pathweight[2*i+1])
      begin
      temppathmatrix[i]=  pathmatrix[2*i];
      temppathweight[i]=  pathweight[2*i];
      end
    else
      begin
      temppathmatrix[i]=  pathmatrix[2*i+1];
      temppathweight[i]=  pathweight[2*i+1];
      end
   end

  for (i=0; i<=7; i=i+1)
   begin
     pathmatrix[i] = temppathmatrix[i];
     pathweight[i] = temppathweight[i];

   end

 end
```

```
endtask



task viterbi11(input [33:0]adata , output [14:0]bdata  );
        reg errorFlag;
        reg [1:0]nstate;
        reg [1:0]nstatep;
        reg [1:0]soutput;
        integer a;
        integer b;
        integer i;
        integer p;

 begin
        p=0;
        errorFlag=0;
        nstate=2'b00;

        for (a=0; a<=5; a=a+1)
          begin
            soutput = {adata[2*a],adata[2*a+1]};

            if (errorFlag==0)
              begin
                if   (outputOneHalf(nstate) == soutput  )
                  begin
                    nstate = nextstateHalf(nstate, 1);
                    pathmatrix[0][2*a+2] = nstate[1];
                    pathmatrix[0][2*a+3] = nstate[0];

                  end
                else if (outputZeroHalf(nstate) == soutput)
                 begin
                    nstate = nextstateHalf(nstate, 0);
                    pathmatrix[0][2*a+2] = nstate[1];
                    pathmatrix[0][2*a+3] = nstate[0];

                 end
```

```
        else
          begin
            errorFlag=1;
          end

      end
    if  (errorFlag==1)
      begin
        p=3;
        for(b=0; b<=3; b=b+1)
          begin
                  nstate = { pathmatrix[b][2*a],pathmatrix[b][2*a+1]};
                  pString=pathmatrix[b];
                  pWeight=pathweight[b];
                  pathweight[b] = pWeight + hummingDistanceHalf(outputOneHalf(nstate), soutput);
                  nstatep = nextstateHalf(nstate, 1);
                  pathmatrix[b][2*a+2] = nstatep[1];
                  pathmatrix[b][2*a+3] = nstatep[0];



                  pathweight[b+p+1]  = pWeight + hummingDistanceHalf(outputZeroHalf(nstate),
soutput);


                  nstatep = nextstateHalf(nstate, 0);
                  pathmatrix[b+p+1]=pString;
                  pathmatrix[b+p+1][2*a+2] = nstatep[1];
                  pathmatrix[b+p+1][2*a+3] = nstatep[0];
                end
          optimizePathMatrixHalf(1);
      end

  end
  p = findMaxpathWeightHalf(1);
  bdata=15'b000000000000000;
  for(i=0; i<=3; i=i+1)
   begin
     bdata[i]= pathmatrix[p][2*(i+1)];
   end
```

```
 end
endtask


task clearPathWeightmatrix(input a1);
 integer i;
 begin
 for(i=0; i<=7; i=i+1)
    begin
     pathmatrix[i] = 0;
     pathweight[i] = 0;
    end

  end
endtask


always @( rst or rowData   )
 begin
    if (rst==3'b111)
     begin
      clearPathWeightmatrix(1);
       rowData1=rowData;
       rowData1[0]=0;

      viterbi11(rowData1,decoderdata);

      decoderdata=decoderdata;

     end
     else
      begin
       decoderdata=0;
      end

 end

endmodule
```

--Decoder Module (2,1,3) With Trillis length 15

```
module decoder1215(rst , rowData , decoderdata );
input [2:0] rst;
input [33:0]rowData;
reg [33:0]rowData1;

output [14:0] decoderdata;
reg [14:0] decoderdata;



reg [35:0]pathmatrix[7:0] ;
integer pathweight[7:0];

integer pWeight;
reg [35:0]pString;



function [1:0]nextstateHalf(input [1:0]pstate , input inputdata  );
  begin
            nextstateHalf = 2'b00;
           if ((pstate == 2'b00) && (inputdata == 1'b0 ))
            nextstateHalf = 2'b00;
         else if ((pstate == 2'b00) && (inputdata == 1'b1 ))
            nextstateHalf = 2'b10;
         else if ((pstate == 2'b01) && (inputdata == 1'b0 ))
            nextstateHalf = 2'b00;
         else if ((pstate == 2'b01) && (inputdata == 1'b1 ))
            nextstateHalf = 2'b10;
         else if ((pstate == 2'b10) && (inputdata == 1'b0 ))
            nextstateHalf = 2'b01;
         else if ((pstate == 2'b10) && (inputdata == 1'b1 ))
            nextstateHalf = 2'b11;
         else if ((pstate == 2'b11) && (inputdata == 1'b0 ))
            nextstateHalf = 2'b01;
    else if ((pstate == 2'b11) && (inputdata == 1'b1 ))
            nextstateHalf = 2'b11;
  end
endfunction

function [1:0]outputOneHalf(input [1:0]pstate );
```

```
  begin
                //   110 111
               outputOneHalf = 2'b00;
                  if  (pstate == 2'b00)
                   outputOneHalf = 2'b11;
              else if  (pstate == 2'b01 )
                   outputOneHalf = 2'b01;
              else if  (pstate == 2'b10 )
                   outputOneHalf = 2'b00;
              else if  (pstate == 2'b11 )
                   outputOneHalf = 2'b10;
              end
endfunction


function [1:0]outputZeroHalf(input [1:0]pstate );
 begin

                   outputZeroHalf = 2'b00;
                  if (pstate == 2'b00)
                   outputZeroHalf = 2'b00;
              else if (pstate == 2'b01)
                   outputZeroHalf = 2'b10;
              else if (pstate == 2'b10)
                   outputZeroHalf = 2'b11;
              else if (pstate == 2'b11)
                   outputZeroHalf = 2'b01;

 end
endfunction


function [31:0]hummingDistanceHalf(input [1:0]p1String, input [1:0]p2String );
  begin
   hummingDistanceHalf = 0;

   if  (p1String[0]==p2String[0])
       begin
       hummingDistanceHalf = hummingDistanceHalf + 1;
```

```
        end
    if  (p1String[1]==p2String[1])
        begin
        hummingDistanceHalf = hummingDistanceHalf + 1;
        end

  end
endfunction

function  [31:0] findMaxpathWeightHalf(input a1 );
  integer i;
  integer j;
  begin
    findMaxpathWeightHalf=0;
    j=0;
  for(i=0; i<=3; i=i+1)
     begin
      if (pathweight[i]> j)
         begin
         j=pathweight[i];
         findMaxpathWeightHalf=i;
         end
     end
   end
endfunction


task optimizePathMatrixHalf(input a1);

  reg [35:0]temppathmatrix[7:0] ;
  integer temppathweight[7:0];
  integer i;



  begin
  for (i=0; i<=3; i=i+1 )
   begin
    if  (pathweight[2*i] > pathweight[2*i+1])
      begin
```

```
      temppathmatrix[i]=  pathmatrix[2*i];
      temppathweight[i]=  pathweight[2*i];
      end
     else
      begin
      temppathmatrix[i]=  pathmatrix[2*i+1];
      temppathweight[i]=  pathweight[2*i+1];
      end
    end

   for (i=0; i<=7; i=i+1)
    begin
     pathmatrix[i] = temppathmatrix[i];
     pathweight[i] = temppathweight[i];

    end

  end
 endtask



 task viterbi11(input [33:0]adata , output [14:0]bdata  );
            reg errorFlag;
            reg [1:0]nstate;
            reg [1:0]nstatep;
            reg [1:0]soutput;
            integer a;
            integer b;
            integer i;
            integer p;

  begin
            p=0;
            errorFlag=0;
            nstate=2'b00;

            for (a=0; a<=16; a=a+1)
                begin
```

```
soutput = {adata[2*a],adata[2*a+1]};

if (errorFlag==0)
  begin
    if   (outputOneHalf(nstate) == soutput  )
      begin
        nstate = nextstateHalf(nstate, 1);
        pathmatrix[0][2*a+2] = nstate[1];
        pathmatrix[0][2*a+3] = nstate[0];

      end
    else if (outputZeroHalf(nstate) == soutput)
     begin
        nstate = nextstateHalf(nstate, 0);
        pathmatrix[0][2*a+2] = nstate[1];
        pathmatrix[0][2*a+3] = nstate[0];

     end
    else
      begin
        errorFlag=1;
      end

   end
 if  (errorFlag==1)
   begin
      p=3;
      for(b=0; b<=3; b=b+1)
        begin
                    nstate = { pathmatrix[b][2*a],pathmatrix[b][2*a+1]};
                    pString=pathmatrix[b];
                    pWeight=pathweight[b];
                    pathweight[b] = pWeight + hummingDistanceHalf(outputOneHalf(nstate), soutput);
                    nstatep = nextstateHalf(nstate, 1);
                    pathmatrix[b][2*a+2] = nstatep[1];
                    pathmatrix[b][2*a+3] = nstatep[0];



                    pathweight[b+p+1]  = pWeight + hummingDistanceHalf(outputZeroHalf(nstate),
```

```
soutput);

                                nstatep = nextstateHalf(nstate, 0);
                                pathmatrix[b+p+1]=pString;
                                pathmatrix[b+p+1][2*a+2] = nstatep[1];
                                pathmatrix[b+p+1][2*a+3] = nstatep[0];
                               end
                         optimizePathMatrixHalf(1);
                       end

                  end
                  p = findMaxpathWeightHalf(1);
                  for(i=0; i<=14; i=i+1)
                   begin
                     bdata[i]= pathmatrix[p][2*(i+1)];
                   end

 end
endtask


task clearPathWeightmatrix(input a1);
 integer i;
 begin
 for(i=0; i<=7; i=i+1)
    begin
     pathmatrix[i] = 0;
     pathweight[i] = 0;
    end

  end
endtask


always @( rst or rowData   )
 begin
    if (rst==3'b110)
     begin
      clearPathWeightmatrix(1);
```

```
       rowData1=rowData;
       rowData1[0]=0;

      viterbi11(rowData1,decoderdata);

      decoderdata=decoderdata;

     end
     else
      begin
       decoderdata=0;
      end

 end

endmodule
```

--Convolutional Encoder and Viterbi Decoder Main Module

```
module cevdIntq(rst, idata, , coutput , odatathree ,odatatwo  );
input [14:0]idata;
input [2:0] rst;

 output [50:0] odatathree;
 output [33:0] odatatwo;
output [14:0] coutput;

 reg [50:0] odatathree;
 reg [33:0] odatatwo;

reg [14:0] coutput;

wire [50:0] edata1315;
wire [50:0] edata1304;
wire [33:0] edata1215;
wire [33:0] edata1204;

wire [14:0] ddata1315;
wire [14:0] ddata1304;
wire [14:0] ddata1215;
```

```
wire [14:0] ddata1204;


encoder1315 con11(rst, idata , edata1315 );
encoder1304 con12(rst, idata , edata1304 );
encoder1215 con13(rst, idata , edata1215 );
encoder1204 con14(rst, idata , edata1204 );

decoder1315 dec11(rst , edata1315 , ddata1315 );
decoder1304 dec14(rst , edata1304 , ddata1304 );
decoder1215 dec13(rst , edata1215 , ddata1215 );
decoder1204 dec12(rst , edata1204 , ddata1204 );



 always @(rst  or coutput or edata1215 or ddata1215 or edata1204 or ddata1204 or edata1315 or ddata1315 or
edata1304 or ddata1304)
      begin

         coutput=0;
         odatatwo=0;
         odatathree=0;
                        if (rst==3'b100)
                                    begin
                                     odatathree = edata1315;
                                            coutput    = ddata1315;
                                           end
                         else if (rst==3'b101)
                                    begin
                                     odatathree = edata1304;
                                     coutput    = ddata1304;
                                    end
                         else if (rst==3'b110)
                                    begin
                                     odatatwo = edata1215;
                                     coutput  = ddata1215;
                                    end
                                else if (rst==3'b111)
                                    begin
```

```
            odatatwo = edata1204;
            coutput  = ddata1204;
           end
          coutput  = coutput;
          odatatwo=odatatwo;
          odatathree = odatathree;

    end
endmodule
```