Implementation of Binary and Ternary Convolutional Codes on an FPGA

by

Bharath Rao Madela
B.Tech., Sardar Vallabhbhai National Institute of Technology, 2017

A Project Report Submitted in Partial Fulfillment of the
Requirements for the Degree of

Master of Engineering

in the Department of Electrical and Computer Engineering

Implementation of Binary and Ternary Convolutional Codes on an FPGA

by

Bharath Rao Madela

B.Tech., Sardar Vallabhbhai National Institute of Technology, 2017

Supervisory Committee

_____

Dr. T. Aaron Gulliver, Supervisor

(Department of Electrical and Computer Engineering)

_____

Dr. Mihai Sima, Departmental Member

(Department of Electrical and Computer Engineering)

Abstract

In modern wireless communication systems, the channels are corrupted by noise and interference. To address this issue, error control coding is employed to reliably transfer data. Convolutional codes are widely employed as they are easy to encode and decode. The focus of this work is the implementation of binary and ternary convolutional codes on an FPGA. As most data is binary, binary to ternary conversion is employed to implement ternary convolutional codes on an FPGA which is a binary device. The design architecture for both types of codes is discussed and comparisons are made based on structure, data rate, speed and resource utilization.

Contents

List of Tables

List of Figures

ACKNOWLEDGEMENTS

I am grateful to God and my loving parents who were very supportive and instrumental in my completing this project. I am also thankful to the people that provided technical advice during this project. First, I wish to express my sincere thanks to Dr. T. Aaron Gulliver whose expertise, understanding, and patience added considerably to my graduate experience. Second, I am indebted to the other member of my supervisory committee, Dr. Mihai Sima, for his time serving as a committee member. I would also like to thank Microchip Technology and Infinera Corporation for providing me an opportunity as a co-op student where I got a practical exposure to this field. I would also like to thank all my friends at the University of Victoria for being an integral part of this journey.

DEDICATION

To my parents

for their continuous guidance and support

Chapter 1 Introduction

The demand for data services is on the rise and the need for reliable and efficient data transmission is increasing, but communication channels are corrupted by interference and noise. Data transfer from source to destination involves many stages and each stage is prone to data corruption. In particular losses due to noise and interference on communication channels make the reliable transfer of data challenging.

A solution to data corruption is adding redundancy to reduce the probability of channel errors [1]. Shannon considered the reliable transmission of data over noisy communication channels. He showed that with a signalling rate less than the channel capacity, reliable communications can be achieved if proper encoding and decoding techniques are used. The design of reliable and efficient codes was initiated by Hamming [2] and Golay [3], and since has become an important area of research.

A model of a communication system is illustrated in Figure 1.1. A typical communication system involves the following steps.

1. Encoding the message at the source.
2. Transmission of encoded message through the communication channel.
3. Decoding the received message at the destination.

Figure 1.1 shows the process of communication between the source and destination. At the source, x is input to the channel encoder and redundancy is added to this message by the encoder which results in a codeword y. The codeword is transmitted over the channel and errors e are introduced by the channel which corrupts the codeword. At the destination, the corrupted codeword $r = y + e$ is received and the decoder uses the redundancy to obtain an estimate of the message $\hat{x}$ [4].

1.1 Galois Fields

A Galois field is a finite set of elements on which the mathematical operations addition and multiplication are defined. The field size must be a prime power. The smallest Galois field is GF(2) which consists of the elements 0 and 1. The addition operation in GF(2) is modulo 2 addition and multiplication is modulo 2 multiplication. Most digital systems work in GF(2) and this plays a vital role in digital communications [5]. GF(3) is the Galois field of three elements 0, 1 and 2. GF(3) addition is modulo 3 addition and multiplication is modulo 3 multiplication. In this report, codes over GF(2) and GF(3) are considered.

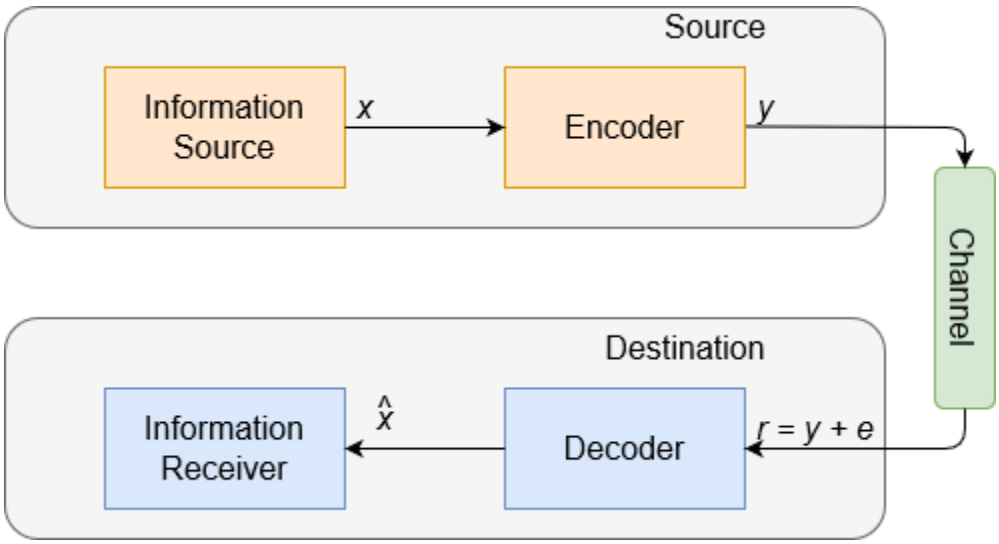

Figure 1.1 Model of a Communication System [4]

1.2 Ternary Systems

The Ericsson Networking Index study [6], indicates that there will be a huge increase in mobile data traffic by 2022 in the Middle East, Africa and Europe and the monthly data traffic per smartphone in North America will reach 25 GB. The growing data rates have created significant demands on the wireless spectrum. Thus, improving the spectral and energy efficiency of communication systems is a crucial challenge. To address this issue, communication systems should be efficient and utilize the available channel bandwidth by adjusting the transmission rate dynamically based on the channel conditions. Thus, for better use of the available bandwidth, schemes beyond binary modulation and coding should be considered.

Most computers are based on binary logic and employ binary computations. Ternary logic employs ternary arithmetic and ternary memory and is an alternative to binary logic. This can reduce the circuit overhead compared to binary logic and requires fewer chip interconnections. Due to this reduced overhead, ternary circuits can be more energy efficient compared to binary circuits. Ternary logic circuits require less energy and are faster than binary circuits [7].

The number of interconnections determines the complexity of an integrated circuit [8]. In [9], it was shown that the number of interconnections in a ternary parallel multiplier is less than two-thirds that of the binary counterpart, and the number of gates in ternary logic is about 20% lower. Further, the storage density of ternary memory cells is higher than that of binary memory cells [10].

Ternary systems can outperform binary systems in terms of energy consumption, speed and wiring complexity, but ternary systems are commercially unavailable. However, heterogeneous

computing systems having both binary and ternary logic can be employed. A system consisting of ternary computing blocks dedicated to modulation/demodulation and encoding/decoding along with binary computing blocks may improve the performance of communication systems [12]. Ternary systems can be embedded as subsystems in binary systems by using binary to ternary data conversion.

1.3 Binary to Ternary Conversion

Binary symbols can be converted to ternary data using a Binary to Ternary (BT) conversion lookup table which maps m bits to n trits (mBnT) [12]. Various BT conversions have been suggested in the literature such as three bits to two trits (3B2T) which maps 1.5 bits to a ternary symbol, six bits to four trits (6B4T) which also maps 1.5 bits to a ternary symbol, and eleven bits to seven trits (11B7T) which maps 1.571 bits to a ternary symbol. Table 1.1 shows the 3B2T conversion given in [12].

The performance of a ternary communication system which employs BT conversion depends on two factors [12]. First, the error propagation during ternary symbol to binary symbol conversion. For example, with a system employing 3B2T conversion, one trit error can result in up to 3 bit errors. Similarly, with a system employing 11B7T conversion, one trit error can result in up to 11 bit errors. Therefore, the average number of bit errors due to a single trit error, eav, has a substantial effect on the performance of the system [12].

With mBnT conversion, a ternary string of n trits has $2n$ ternary strings which differ by one trit. As an example, the length 2 ternary string 11 differs by one trit from the following four strings 10, 12, 01, 21. The number of bit errors due to a single trit error for the ith ternary string is

$$T_d(i) = \Sigma_{j=1}^{2n} T_d(i,j) \tag{1.1}$$

where $T_d(i,j)$ is the total number of bit errors due to a trit error between the ith and jth ternary strings. Then the average number of bit errors due to mBnT conversion is

$$e_{av} = \frac{\sum_{i=1}^{3^n} T_d(i)}{3^n \times 2} \tag{1.2}$$

The second factor in the performance is the conversion efficiency which is defined as the ratio of the number of input bits l_b to the number of output trits l_t [12][13] which is given by

$$\eta = \frac{l_b}{l_t \times \log_2 3} \tag{1.3}$$

The conversion efficiency is always less than 1 because n and m must be selected such that $2^m < 3^n$. However, there are several 2^m to 3^n conversions which have a high efficiency. For example, 2^{11} to 3^7 [11]. A binary to ternary conversion comparison is given in Table 1.2. This shows that 11B7T conversion has an efficiency of 99.15% which is better than the 94.65% for 3B2T

conversion. There is a tradeoff between the average number of bit errors and the conversion efficiency. For 11B7T conversion, the efficiency is 99.15% and the average number of bit errors is 4.376, while for 3B2T conversion, the efficiency is 94.65% but the average number of bit errors is 1.555 [12]. Although the conversion efficiency of 11B7T is higher, 3B2T conversion has a significantly lower average number of bit errors, so 3B2T conversion is employed here.

Binary Block Input	Ternary Block	Binary Block Output
000	12	000
001	11	001
011	01	011
111	00	111
101	02	101
100	22	100
110	21	110
010	20	010
xxx	10	010

Table 1.1 Three Bits to Two Trits (3B2T) Conversion [12]

Parameter	11B7T	3B2T
$\eta(\%)$	99.15	94.65
e_{av}	4.376	1.555

Table 1.2. Binary to Ternary Conversion Comparison [12]

1.4 FPGA Technology

Field Programmable Gate Arrays (FPGAs) are integrated circuits which can be programmed and reconfigured after they have been fabricated. They are the building blocks of reconfigurable computing, a computing paradigm that combines the power of hardware with the flexibility of software.

The basic structure of an FPGA is shown in Figure 1.2. It consists of Configurable Logic Blocks (CLBs), Input/Output Blocks (IOBs) and routing/interconnect channels. Each CLB consists of Look-up Tables (LUTs) and flip-flops that can be configured to perform combinational and/or sequential logic. CLBs are surrounded by IOBs to interface with external devices. The routing or interconnect structure connects multiple blocks such as CLBs, IOBs and memory elements based on the application [14]. Digital logic designs can be implemented on an FPGA using a Hardware Description Language (HDL), such as VHDL, Verilog or System Verilog. In this report, the designs are implemented using System Verilog on a Xilinx Artix-7 FPGA. This FPGA was chosen due to its high commercial availability.

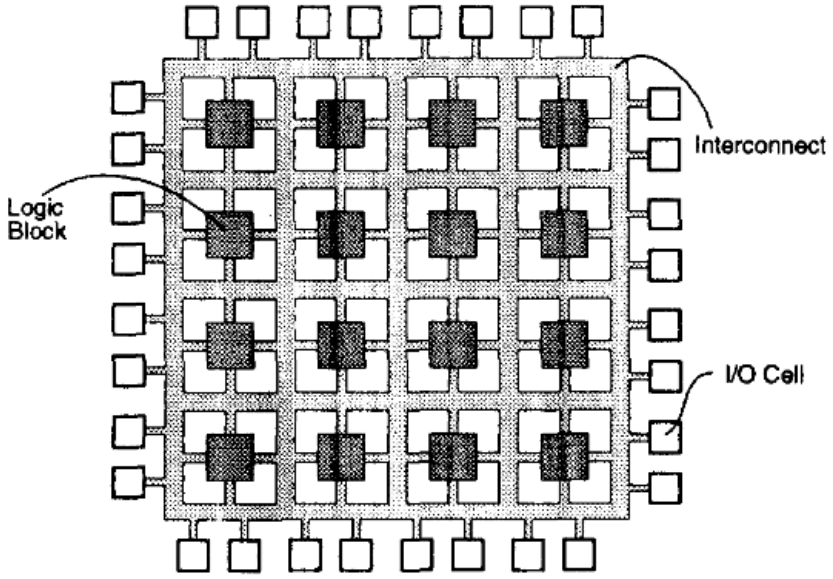

Figure 1.2 Structure of an FPGA [14]

Chapter 2 Error Control Coding

Several strategies can be employed to reduce the probability of error in a communication system. One is Automatic Repeat Request (ARQ). In this strategy, systems try to detect errors in the received data. If errors are found, the receiver informs the source and the data is retransmitted. These negative acknowledgements are sent until the data is correctly received. In many practical applications, data retransmission is not feasible. In this case, Error Correction Coding (ECC) can be used to detect and correct errors in the received data. This is achieved by adding redundancy for decoding at the destination [15].

Redundancy is added to the message at the source in a process called encoding. This redundancy is an overhead which costs resources such as transmission power and channel bandwidth. Therefore, it is important to keep it as small as possible. Consider an information source that generates messages of length k which are passed through an encoder to generate codewords of length n. The code rate R is defined as the ratio of message length to codeword length $R = k/n$. When no redundancy is added, the code rate is 1. Code rate and performance are inversely related as the error correction capability improves with the addition of redundancy but this reduces the code rate. A code should provide good performance with a code rate as high as possible [16].

2.1 Linear Block Codes

In linear block coding, the data is segmented into messages of length k so there are 2^k possible message blocks. The encoder transforms a message block u into a codeword v of length n where $n > k$. For each block u, there should be a unique codeword [5], so there are 2^k codewords in a binary linear block code denoted by C [5]. The modulo 2 sum of any two codewords is also a codeword. A ternary linear block code with messages of length k has 3^k codewords. The ternary encoder transforms each message block u into a ternary n-tuple v where $n > k$. There are 3^k codewords in a ternary linear block code denoted by C [5].

An (n, k) linear block code C is a k-dimensional subspace of the vector space V_n of all n-tuples. It can be specified using k linearly independent codewords $(g_0, g_1, g_2, \ldots, g_{k-1})$ in C so that every codeword c is a linear combination of these codewords. They can be used as the rows of a k x n matrix

$$G = \begin{bmatrix} g_0 \\ g_1 \\ g_2 \\ \vdots \\ \vdots \\ \vdots \\ g_{k-1} \end{bmatrix} = \begin{bmatrix} g_{0,0} & g_{0,1} & \cdots & g_{0,n-1} \\ g_{1,0} & g_{1,1} & \cdots & g_{1,n-1} \\ g_{2,0} & g_{2,1} & \cdots & g_{2,n-1} \\ \vdots & \vdots & \ddots & \vdots \\ \vdots & \vdots & \ddots & \vdots \\ \vdots & \vdots & \ddots & \vdots \\ g_{k-1,0} & g_{k-1,1} & \cdots & g_{k-1,n-1} \end{bmatrix} \tag{2.1}$$

which is called a generator matrix for C. A linear block code is often defined by its generator matrix. The codeword c for a message block $u = (u_0, u_1, u_2, \dots, u_{k-1})$ is then

$$v = uG \tag{2.2}$$

$$v = (u_0, u_1, u_2, \dots\dots, u_{k-1}) \begin{bmatrix} g_0 \\ g_1 \\ g_2 \\ \vdots \\ \vdots \\ \vdots \\ g_{k-1} \end{bmatrix} \tag{2.3}$$

$$v = (u_0 g_0 + u_1 g_1 + u_2 g_2 + \cdots + u_{k-1} g_{k-1}) \tag{2.4}$$

2.1.1 Minimum Distance

The minimum distance d_{min} defines the error detection and error correction capability of a block code [5]. Let $v = (v_0, v_1, v_2, \dots, v_{n-1})$ be a codeword. The Hamming weight w(v) of v is defined as the number of nonzero elements in v. As an example, the Hamming weight of v = (10110) is 3. The Hamming distance between codewords v and a, denoted by d(v, a), is defined as the number of places in which the elements differ. If a = (10011) then d(v, a) = 2 [5].

The Hamming distance between two n-tuples v and a is equal to Hamming weight of their sum

$$\mathrm{d}(v, a) = \mathrm{w}(v + a) \tag{2.5}$$

For a block code C, the minimum distance is defined as

$$d_{min} = \min\{\, \mathrm{d}(v, a) : \; v, a \in C, v \neq a \,\} \tag{2.6}$$

For a linear block code C, the sum of two codewords is also a codeword so that

$$d_{min} = \min\{\, \mathrm{w}(x) : \; x \in C, x \neq 0 \,\} \tag{2.7}$$

Thus, the minimum distance d_{min} of a linear block code C is the minimum weight of its nonzero codewords [5]. The maximum number of errors that can be detected is d_{min}-1 [17], and the maximum number of errors that can be corrected is $\lfloor d_{min} - 1 \rfloor / 2$ [17].

2.2 Convolutional Codes

In 1955, Elias introduced convolutional codes [18]. Convolutional codes are easy to encode and decode and are employed in numerous communication systems [19]. Convolutional codes encode a continuous stream of data while block codes encode fixed length blocks of data. With convolutional codes, the encoding is typically done using shift registers and adders so the output

is a combination of the current input symbols and previous input symbols which are in the shift registers. A convolutional encoder encodes the entire stream of message into one codeword.

In a binary convolutional code, a stream of bits is passed through the shift registers. The encoded bits are a modulo 2 sum of the current input bits and the shift register contents. Convolutional codes are defined in terms of three parameters (n, k, m), where n is the number of output data streams, k is the number of input message streams and m is the memory length [12]. The code rate is $R = k/n$. The input message stream is ordered into k-tuples and the codeword is generated in continuous fashion ordered in n-tuples [5]. A ternary convolutional code (TCC) is similar to a binary convolutional code (BCC) but the arithmetic operations are ternary [13]. BCCs and TCCs of rate $R = 1/2$ with encoder memory $m = 3$ and $m = 2$, respectively, are considered in this report.

2.3 Convolutional Encoders

The encoder for a (2,1,3) convolutional code is shown in Figure 2.1. This encoder consists of a 3-stage shift register with two modulo adders and a multiplexer for serializing the outputs of the adders. The message sequence $u = (u_0, u_1, u_2, \dots, u_{k-1})$ enters the encoder one symbol at a time. The two output codeword sequences, $(v^{(0)} = (v_0^{(0)}, v_1^{(0)}, v_2^{(0)}, \dots)$ and $(v^{(1)} = (v_0^{(1)}, v_1^{(1)}, v_2^{(1)}, \dots)$, are obtained as the convolution of the input message sequence and the two encoder generator sequences. The encoder generator sequences are written as [5]

$$g^{(0)} = (g_0^{(0)}, g_1^{(0)}, g_2^{(0)}, \dots, g_m^{(0)}) \text{ and } g^{(1)} = (g_0^{(1)}, g_1^{(1)}, g_2^{(1)}, \dots, g_m^{(1)})$$

where $g^{(0)}$ (red) and $g^{(1)}$ (blue) in Figure 2.1 are the connections to the adders in the encoder for $v^{(0)}$ and $v^{(1)}$, respectively, and S_0, S_1 and S_2 are the three memory elements.

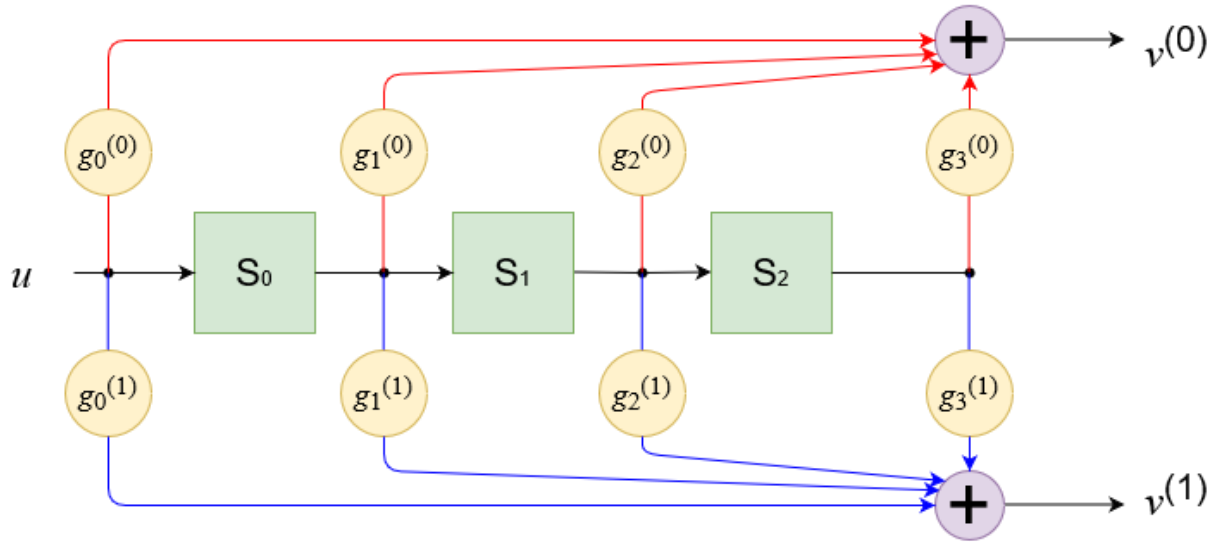

Figure 2.1 (2,1,3) Convolutional Encoder

Convolutional encoders have a semi-infinite generator matrix G with m = 1 sub matrices each of dimension $k \times n$

$$G = \begin{bmatrix} G_0 & G_1 & G_2 & \cdots & G_m & 0 & 0 & \cdots \\ 0 & G_0 & G_1 & G_2 & \cdots & G_m & 0 & \cdots \\ 0 & 0 & G_0 & G_1 & G_2 & \cdots & G_m & \cdots \\ \vdots & \vdots & \vdots & \ddots & \ddots & \ddots & \ddots & \ddots \\ \vdots & \vdots & \vdots & \ddots & \ddots & \ddots & \ddots & \ddots \end{bmatrix} \tag{2.8}$$

Each sub matrix G_j is associated with the generator sequences at time $t = j$

$$G_j = \begin{bmatrix} g_{0,j}^{(0)} & g_{0,j}^{(1)} & \cdots & g_{0,j}^{(n)} \\ g_{1,j}^{(0)} & g_{1,j}^{(1)} & \cdots & g_{1,j}^{(n)} \\ \vdots & \vdots & \ddots & \vdots \\ \vdots & \vdots & \ddots & \vdots \\ g_{k,j}^{(0)} & g_{k,j}^{(1)} & \cdots & g_{k,j}^{(n)} \end{bmatrix} \tag{2.9}$$

Substituting (2.9) in (2.8) gives [5]

$$G = \begin{bmatrix} g_0^{(0)}\, g_0^{(1)} & g_1^{(0)}\, g_1^{(1)} & \cdots & g_m^{(0)}\, g_m^{(1)} & 0 & 0 & \cdots \\ 0 & g_0^{(0)}\, g_0^{(1)} & g_1^{(0)}\, g_1^{(1)} & \cdots & g_m^{(0)}\, g_m^{(1)} & 0 & \cdots \\ 0 & 0 & g_0^{(0)}\, g_0^{(1)} & g_1^{(0)}\, g_1^{(1)} & \cdots & g_m^{(0)}\, g_m^{(1)} & \cdots \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \ddots \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \ddots \end{bmatrix} \tag{2.10}$$

For the encoder in Figure 2.1, the generator sequences are $g^{(0)}$ = (1011) and $g^{(1)}$ = (1101) and the generator matrix G for a message of four bits is

$$G = \begin{bmatrix} 11 & 01 & 10 & 11 & 00 & 00 & 00 \\ 00 & 11 & 01 & 10 & 11 & 00 & 00 \\ 00 & 00 & 11 & 01 & 10 & 11 & 00 \\ 00 & 00 & 00 & 11 & 01 & 10 & 11 \end{bmatrix} \tag{2.11}$$

The encoding can be written as

$$v^{(0)} = u \oplus g^{(0)} \tag{2.12}$$

$$v^{(1)} = u \oplus g^{(1)} \tag{2.13}$$

where $\oplus$ denotes discrete convolution. The convolution operation implies that for all time $l \geq 0$

$$v_l^{(j)} = \sum_{i=0}^{m} u_{l-i}\, g_i^{(j)} = u_l\, g_0^{(j)} + u_{l-1}\, g_1^{(j)} + \cdots + u_{l-m}\, g_m^{(j)} \qquad j = 0, 1, 2, \ldots. \tag{2.14}$$

where $u_{l-i} = 0$ for $l < i$ and $v_l^{(j)}$ is the encoder output. Hence, for the encoder in Figure 2.1

$$v_l^{(0)} = u_l + u_{l-2} + u_{l-3} \tag{2.15}$$

$$v_l^{(1)} = u_l + u_{l-1} + u_{l-3} \tag{2.16}$$

The two output sequences are then multiplexed to form the codeword [5]

$$v = (v_0^{(0)} v_0^{(1)}, v_1^{(0)} v_1^{(1)}, v_2^{(0)} v_2^{(1)}, \ldots) \tag{2.17}$$

2.3.1 Delay Operator

Convolutional codes are linear and the codewords can be represented in polynomial form. The generator matrix G can be expressed in terms of a delay operator D as [4]

$$G(D) = \begin{bmatrix} g_{00}(D) & g_{01}(D) & \cdots & g_{0n}(D) \\ g_{10}(D) & g_{11}(D) & \cdots & g_{1n}(D) \\ \vdots & \vdots & \ddots & \vdots \\ g_{k0}(D) & g_{k1}(D) & \vdots & g_{kn}(D) \end{bmatrix} \tag{2.18}$$

and the corresponding generator polynomial is

$$g_{ij}(D) = g_{ij}^{(0)} D^0 + g_{ij}^{(1)} D^1 + g_{ij}^{(2)} D^2 + \cdots + g_{ij}^{(m)} D^m \tag{2.19}$$

where $i = 0, 1, 2, \ldots, k$ and $j = 0, 1, 2, \ldots, n$. Note that the maximum degree of the generator polynomial is m. The relationship between the message polynomial and codeword polynomial is

$$v(D) = u(D)G(D) \tag{2.20}$$

where

$$u(D) = u_0 + u_1 D + u_2 D^2 + \cdots \tag{2.21}$$

$$v(D) = v_0 + v_1 D + v_2 D^2 + \cdots \tag{2.22}$$

The generator matrix can then be expressed as

$$G(D) = G_0 D^0 + G_1 D^1 + G_2 D^2 + \cdots + G_m D^m \tag{2.23}$$

2.3.2 State Diagrams

Since sequential logic elements are used in convolutional encoders, they can be defined using a finite-state machine [19]. Convolutional encoders consist of memory elements whose contents determine a mapping between the next input symbols and the output symbols. The state diagram for the binary encoder in Figure 2.2 with two shift registers is shown in Figure 2.3. A binary encoder has 2^m states while a ternary encoder has 3^m states. The encoder state at a given time l is given by $(u_{l-1}, u_{l-2}, \ldots, u_{l-m})$.

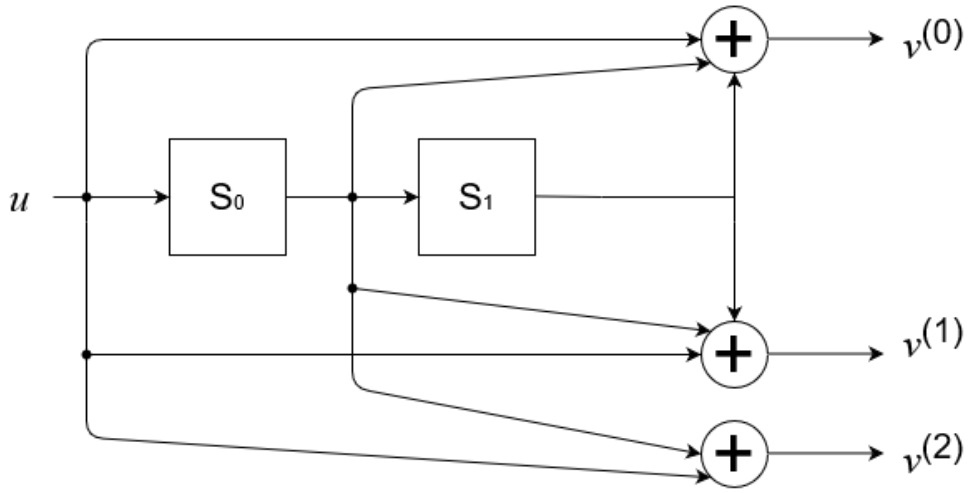

Figure 2.2 (3,1,2) Binary Convolutional Encoder

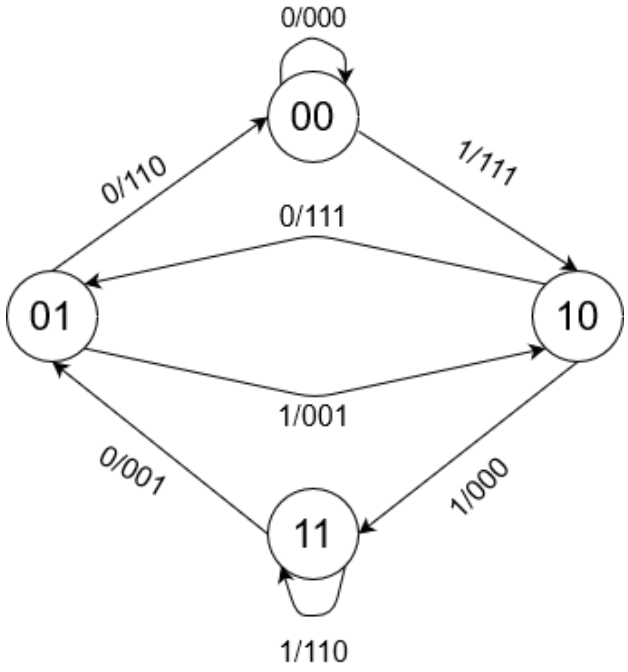

Figure 2.3 (3,1,2) Binary Convolutional Encoder State Diagram

In a convolutional encoder, each input symbol causes a transition to a new state S_1S_0 where S_0 and S_1 are the contents of the memory elements. Therefore, there are 2^k branches leaving each state in a binary encoder and 3^k branches leaving each state in a ternary encoder, each corresponding to a different input. In Figure 2.3, each branch is labelled as $u/v_0v_1v_2$ which indicates that input symbol u causes the transition and $v_0v_1v_2$ indicates the corresponding output symbols.

2.3.3 Catastrophic Generator Matrices

A convolutional code is said to be catastrophic if there is some input $u(D)$ with an infinite Hamming weight that generates an output $v(D)$ with a finite Hamming weight. This means that a finite number of errors introduced by the channel can result in an infinite number of decoding errors. This is a generator matrix property and such matrices should not be used. A $k = 1$ code with generator matrix

$$G(D) = [g_0(D) + g_1(D) + g_2(D) + \cdots + g_n(D)] \tag{2.24}$$

is non-catastrophic if and only if [20]

$$\gcd(g_0(D), g_1(D), g_2(D), \ldots, g_n(D)) = D^p \tag{2.25}$$

where gcd denotes the greatest common divisor and p is a non-negative integer. As an example, the matrix $G(D) = [1+D \quad 1+D^2]$ has gcd$(1+D\ 1+D^2) = 1 + D$ which is not in the form D^p, so it is catastrophic. In terms of the convolutional encoder state diagram, a code is catastrophic if its state diagram has a zero output weight cycle other than the self loop around the all-zero state [21].

2.4 Distance Properties of Convolutional Codes

For convolutional codes, the free distance d_{free} determines the error correction capability. This is similar to the minimum distance d_{min} of block codes [4], and is the minimum Hamming distance between two infinite length codeword sequences [5]. The free distance is defined as

$$d_{free} = \min\{d(v' - v''): u' \neq u''\} \tag{2.26}$$

where u' and u'' are message sequences and v' and v'' are the corresponding codewords. Since convolutional codes are linear

$$d_{free} = \min\{w(v' - v''): u' \neq u''\} \tag{2.27}$$

$$= \min\{w(v): u \neq 0\} \tag{2.28}$$

where u is the message sequence and v is the corresponding codeword. Hence, d_{free} is the minimum weight codeword produced by a nonzero message sequence and is the minimum weight of all paths in the state diagram that diverge from the all-zero path and then merge with this path [5].

2.5 Convolutional Decoders

Several algorithms exist to decode convolutional codes. For small values of k, the Viterbi Algorithm (VA) is typically used. This algorithm is highly parallelizable, hence it is easy to implement Viterbi decoders in VLSI hardware. For longer constraint length codes, sequential

decoding is used. Sequential decoding is not Maximum Likelihood (ML), but the complexity of the sequential decoder increases with the constraint length as opposed to the Viterbi algorithm.

2.5.1 Viterbi Decoders

The Viterbi algorithm was proposed in 1976 [22]. It is an ML decoding algorithm, i.e. the output of the decoder is the codeword which is closest to the received sequence [23]. It is commonly employed because it can easily be implemented in software and/or hardware.

A trellis diagram is the extension of the state diagram for a convolutional code that shows the time index. The trellis diagram for the binary convolutional encoder in Figure 2.2 is shown in Figure 2.4. Consider a message sequence u = (10011) encoded by the encoder in Figure 2.2 which outputs the codeword v = (111,111,110,111,000). The VA finds the most likely path through the trellis by calculating a distance metric between the possible paths and the received sequence. At each node in the trellis, the branch metric is calculated as the distance between the branch symbols as shown in Figure 2.4 and the received sequence [22]. The ML codeword is obtained by tracing back through the trellis on the minimum metric path.

Figure 2.5 shows the branch metrics (Hamming distances between the received symbols and the branch symbols) and the path metrics (red) calculated at each node. These path metrics are the minimum values for the two incoming branches at each node. Figure 2.6 shows the trace back through the trellis. The red line indicates the trace back path and this is obtained by checking the minimum path metrics at each stage. The black dotted lines signify state transitions due to the input 0 and the blue lines signify state transitions due to the input 1. The input bits corresponding to the state transitions in the trace back path are the output.

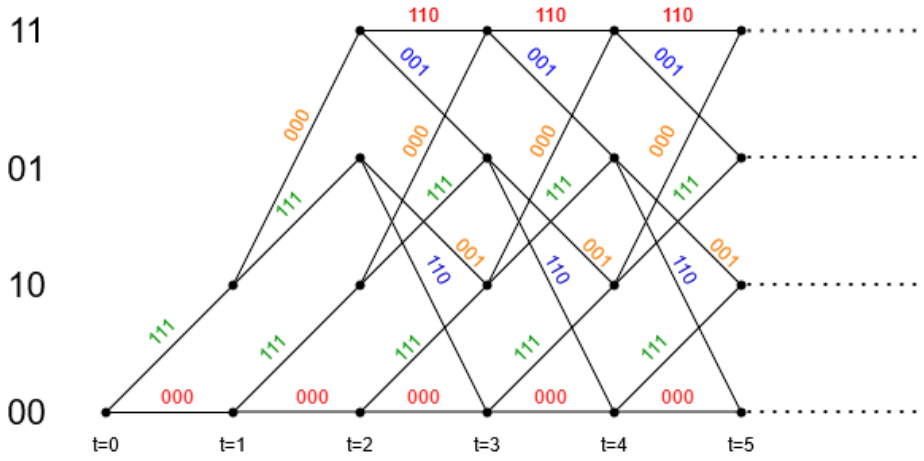

Figure 2.4 Trellis Structure for the (3,1,2) Convolutional Code

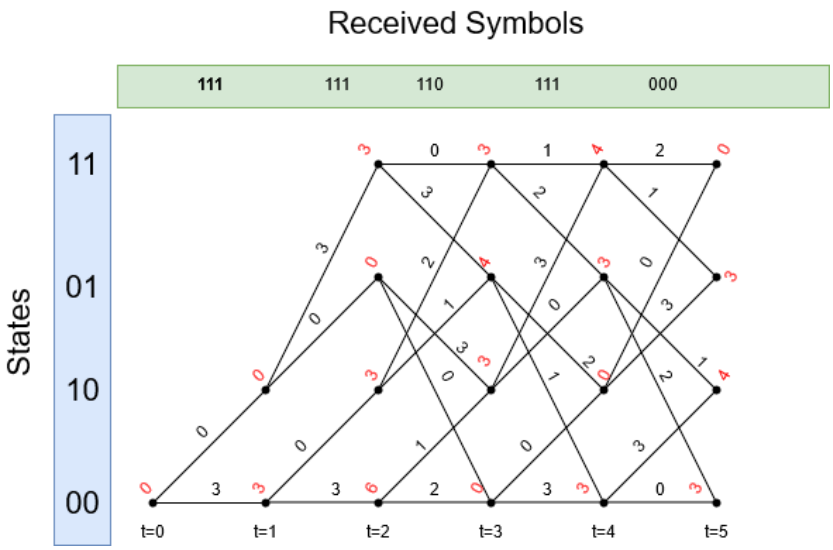

Figure 2.5 Branch and Path Metrics in the Trellis

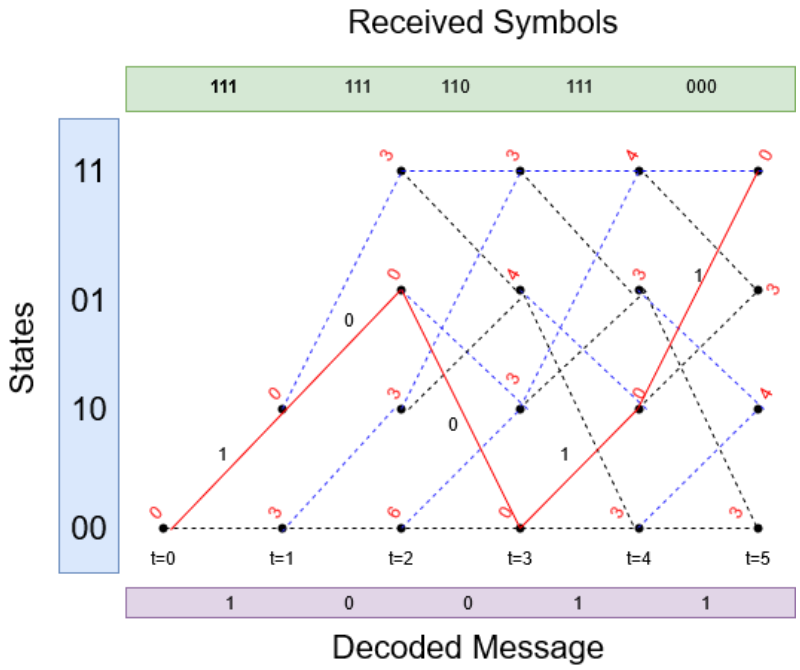

Figure 2.6 Trace Back in the Trellis

The block diagram of a Viterbi decoder is shown in Figure 2.7. It has four main components

- Branch Metric Unit (BMU)
- Add Compare and Select Unit (ACSU)
- Survivor Memory Unit (SMU)
- Trace Back Unit (TBU)

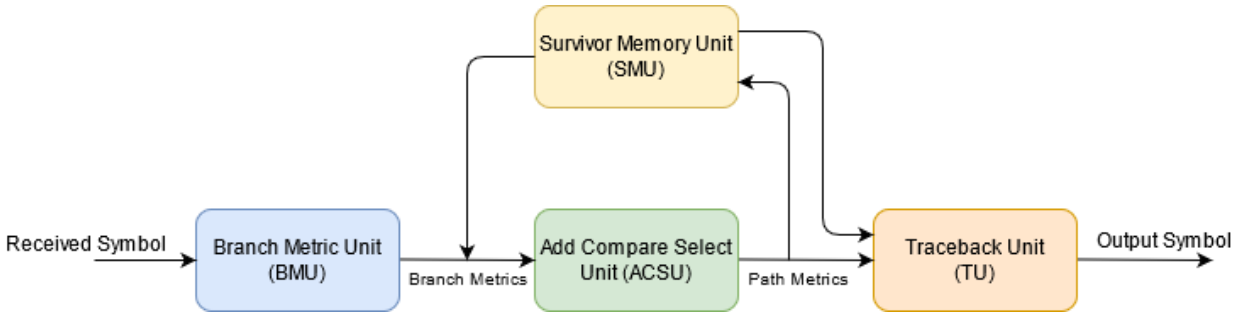

Figure 2.7 Viterbi Decoder Block Diagram

The BMU compares the received symbols with the branch symbols and calculates the distances between them. The ACSU adds the current branch metrics to the path metrics from the previous stage to obtain new path metrics at each node. These path metrics are compared at each node and the lowest metrics are selected as the new path metrics and stored in the SMU. The stored metrics are used to compute the path metrics for the next stage. The number of stages depends on the number of encoded symbols. The SMU stores the survivor path values (minimum path metrics at each node) from the ACSU module. Once the path metrics for all stages have been calculated, the minimum path metrics at all stages are found. These minimum path metrics are copied from the SMU and stored in the Trace back Memory (TM) in the TBU. The state with the minimum path metric at the last stage is input to the TBU. The TM is used to trace the survivor path from this state and output the message symbols for the corresponding transitions [24].

The Viterbi algorithm supports soft decision as well as hard decision decoding. With soft decision decoding, the decoder gets information from the demodulator indicating the confidence levels of the symbol decisions. The decoder uses this information to estimate the transmitted symbols. With hard decision decoding, the demodulator only provides the symbol decisions. A soft decision decoder provides better performance than a hard decision decoder but it is more complex to implement [25].

2.6 Decoding Complexity of Binary Convolutional Codes

For a binary convolutional code, a trellis module will have 2^m initial and final states. Each state is connected to 2^k states. Thus, there are a total of 2^{m+k} edges in a binary trellis module. For example, in a BCC (2,1,3), there are 16 edges in a trellis module. Each edge in this module is labelled with n symbols for a total of $n \times 2^{m+k}$ symbols. Hence, there will be $n \times 2^{m+k}$ BMU and ACSU operations in the binary trellis module, so the decoding complexity is [12]

$$\frac{n \times 2^{m+k}}{k} \text{ operations per encoded bit} \qquad (2.29)$$

Since there are 2^m states, the decoder must have 2 × 2^m bits of storage for the survivors. For an input message sequence of length L_b bits and a BCC with memory m, the trace back memory must have size [12]

$$\text{TM} = 2 \times L_b \times 2^m \text{ binary memory cells} \tag{2.30}$$

As an example, for BCC (2,1,3) with L_b= 30, TM = 480 binary memory cells are required.

2.7 Decoding Complexity of Ternary Convolutional Codes

For a ternary convolutional code, a trellis module will have 3^m initial and final states. Each state is connected to 3^k states. Thus, there are 3^{m+k} edges in a ternary trellis module. For example, in a TCC (2,1,3), there are 81 edges in a trellis module. Each edge in this module is labelled with n symbols for a total of $n \times 3^{m+k}$ symbols. Hence, there will be $n \times 3^{m+k}$ BMU and ACSU operations in the ternary trellis module, so the decoding complexity is [12]

$$= \frac{n \times 3^{m+k}}{k} \text{ operations per encoded trit} \tag{2.31}$$

Since there are 3^m states, the decoder requires 2 × 3^m trits of storage for the survivors. For an input message sequence of length $L_t$trits and a TCC with memory m, the trace back memory must have size [12]

$$\text{TM} = 2 \times L_t \times 3^m \text{ ternary memory cells} \tag{2.32}$$

As an example, for TCC (2,1,3) with L_t= 20, TM = 1080 ternary memory cells are required. The size of a ternary memory cell is twice the size of a binary memory cell [26], so this trace back memory in terms of binary memory cells is 1080 × 2 = 2160, which is 4.5 times larger than for BCC (2,1,3).

Chapter 3 FPGA Implementation of Convolutional Codes

In this chapter, the FPGA implementations of binary and ternary convolutional codes are presented. For a fair comparison, a (2,1,3) binary convolutional code and a (2,1,2) ternary convolutional code are considered as the BCC has 8 states and the TCC has 9 states. The (2,1,3) BCC and (2,1,2) TCC encoders are shown in Figures 3.1 and 3.2, respectively. The squares are memory elements and the circled numbers are the coefficients of the generator polynomials with which the contents of the memory elements are multiplied. The memory elements can be implemented by using flip flops and registers. For the BCC encoder, flip flops are used since they need to store just one bit. For the TCC encoder, two bit registers are used to store the ternary values. The ternary symbol to binary representation is shown in Table 3.1. The plus symbols in Figures 3.1 and 3.2 are modulo 2 adders and modulo 3 adders, respectively. The best generator polynomials for the TCC are $g^{(0)}$ = (1 1 2) and $g^{(1)}$ = (2 1 1) [12] and the best generator polynomials for the BCC are $g^{(0)}$ = (1 0 1 1) and $g^{(1)}$ = (1 1 1 1) [5]. System Verilog is used as the HDL to write the Register Transfer Level (RTL) code. A Xilinx Artix-7 FPGA with speed grade -1 was chosen to implement the Viterbi decoder due to the commercial availability of this FPGA device.

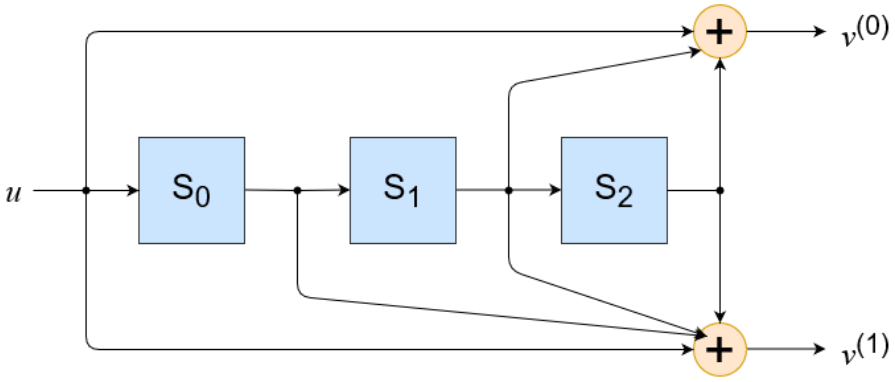

Figure 3.1. (2,1,3) Binary Convolutional Encoder

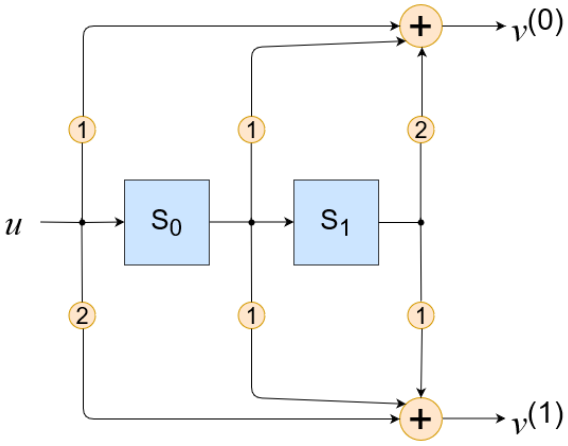

Figure 3.2. (2,1,2) Ternary Convolutional Encoder

Ternary Symbol	Binary Representation
0	00
1	01
2	10

Table 3.1 Binary Representation of Ternary Symbols

A state table describes the behaviour of a sequential circuit as a function of the current state and input variables. For each current state and input, the next state of the circuit is specified along with the output variables. The BCC and TCC state tables are given in Tables 3.2 and 3.3, respectively. (S_2, S_1, S_0) and (S_2^*, S_1^*, S_0^*) are the current and next states, respectively in the BCC, (S_1, S_0) and (S_1^*, S_0^*) are the current and next states, respectively in the TCC, u is the input and $(v^{(0)}, v^{(1)})$ are the outputs.

S_2	S_1	S_0	u	S_2^*	S_1^*	S_0^*	$v^{(0)}$	$v^{(1)}$
0	0	0	0	0	0	0	0	0
0	0	0	1	0	0	1	1	1
0	0	1	0	0	1	0	0	1
0	0	1	1	0	1	1	1	0
0	1	0	0	1	0	0	1	1
0	1	0	1	1	0	1	0	0
0	1	1	0	1	1	0	1	0
0	1	1	1	1	1	1	0	1
1	0	0	0	0	0	0	1	1
1	0	0	1	0	0	1	0	0
1	0	1	0	0	1	0	1	0
1	0	1	1	0	1	1	0	1
1	1	0	0	1	0	0	0	0
1	1	0	1	1	0	1	1	1
1	1	1	0	1	1	0	0	1
1	1	1	1	1	1	1	1	1

Table 3.2 State Table for the (2,1,3) BCC

3.1 Encoder Design

The encoder encodes a message stream into a codeword. The encoder modules for the BCC and TCC are shown Figures 3.3(a) and 3.3(b), respectively. The port descriptions of the encoders are given in Table 3.4. The clock is input to the *clk* port and the reset signal that brings the encoder to a known state is input to the *reset* port. The message stream is input to the *Tx* port. The width

of the Tx port is one for the BCC and two for the TCC. The CW port outputs the generated codeword. The width of the CW port is two for the BCC and four for the TCC. In the TCC, the CW port output is a packed array of width two bits.

S_1	S_0	u	S_1^*	S_0^*	$v^{(0)}$	$v^{(1)}$
0	0	0	0	0	0	0
0	0	1	0	1	1	2
0	0	2	0	2	2	1
0	1	0	1	0	1	1
0	1	1	1	1	2	0
0	1	2	1	2	0	2
0	2	0	2	0	2	2
0	2	1	2	1	0	1
0	2	2	2	2	1	0
1	0	0	0	0	2	1
1	0	1	0	1	0	0
1	0	2	0	2	1	2
1	1	0	1	0	0	2
1	1	1	1	1	1	1
1	1	2	1	2	2	0
1	2	0	2	0	1	0
1	2	1	2	1	2	2
1	2	2	2	2	0	1
2	0	0	0	0	1	2
2	0	1	0	1	2	1
2	0	2	0	2	0	0
2	1	0	1	0	2	0
2	1	1	1	1	0	2
2	1	2	1	2	1	1
2	2	0	2	0	0	1
2	2	1	2	1	1	0
2	2	2	2	2	2	2

Table 3.3 State Table for the (2,1,2) TCC

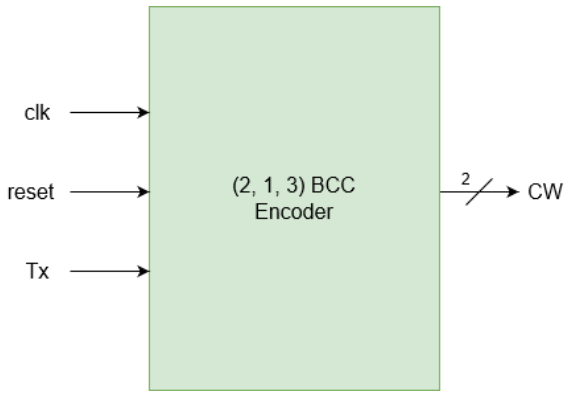

Figure 3.3(a) (2,1,3) BCC Encoder

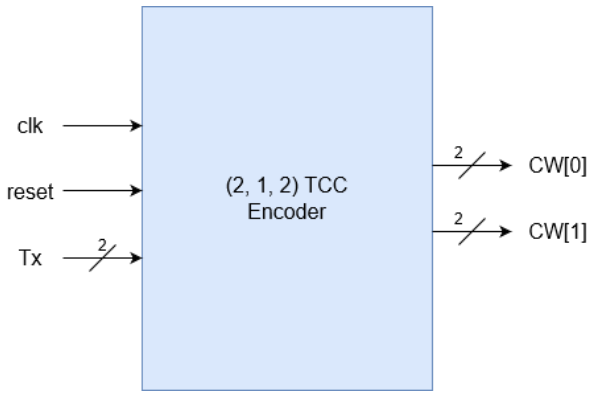

Figure 3.3(b) (2,1,2) TCC Encoder

Port	Width	Direction	Description
clk	1	input	Clock signal
reset	1	input	Reset signal
Tx	BCC=1;TCC=2	input	Message symbol input
CW	BCC=2;TCC=[2][0:1]	output	Encoded codeword

Table 3.4 Encoder Port Descriptions

3.2 Viterbi Decoder Design

The flow chart for the Viterbi decoder algorithm is shown in Figure 3.4. The VA starts when the received word is input. The BMU computes the branch metrics by calculating the Hamming distances between the received symbols and branch symbols. At each node, the ACSU adds the current branch metric to the path metric from the previous stage and computes the new path metric. These path metrics are compared and the path at each node with the lowest metric is selected and stored in the SMU. These will be used to compute the path metrics for the next stage. The minimum path metrics (survivor path metrics), are stored in the TM. Once the TM is filled, the TBU traces back through the survivor paths and outputs the decoded symbols.

The Viterbi decoder top level module is shown in Figure 3.5. This module contains all other decoder modules. The descriptions of the ports in the Viterbi decoder are given in Table 3.5. Each module of the Viterbi decoder is discussed below. The clock pulse for the decoder is input to the *clk* port and the reset signal that brings the decoder to a known state is input to the *reset* port. The *seq_ready* port validates the input. When the input is 1, the received symbols are valid. Decoding starts when the input to the *seq_ready* port is 1. The received symbols are input to the *Rx* port. The width of the *Rx* port is two for the BCC and four for the TCC. In the TCC, the *Rx* port input is a packed array of width two bits. The *symbol_out* port outputs the decoded symbols. The width of the *symbol_out* port is one and two for the BCC and TCC (extra bit to accommodate trits), respectively. The *oen* port validates the output symbols.

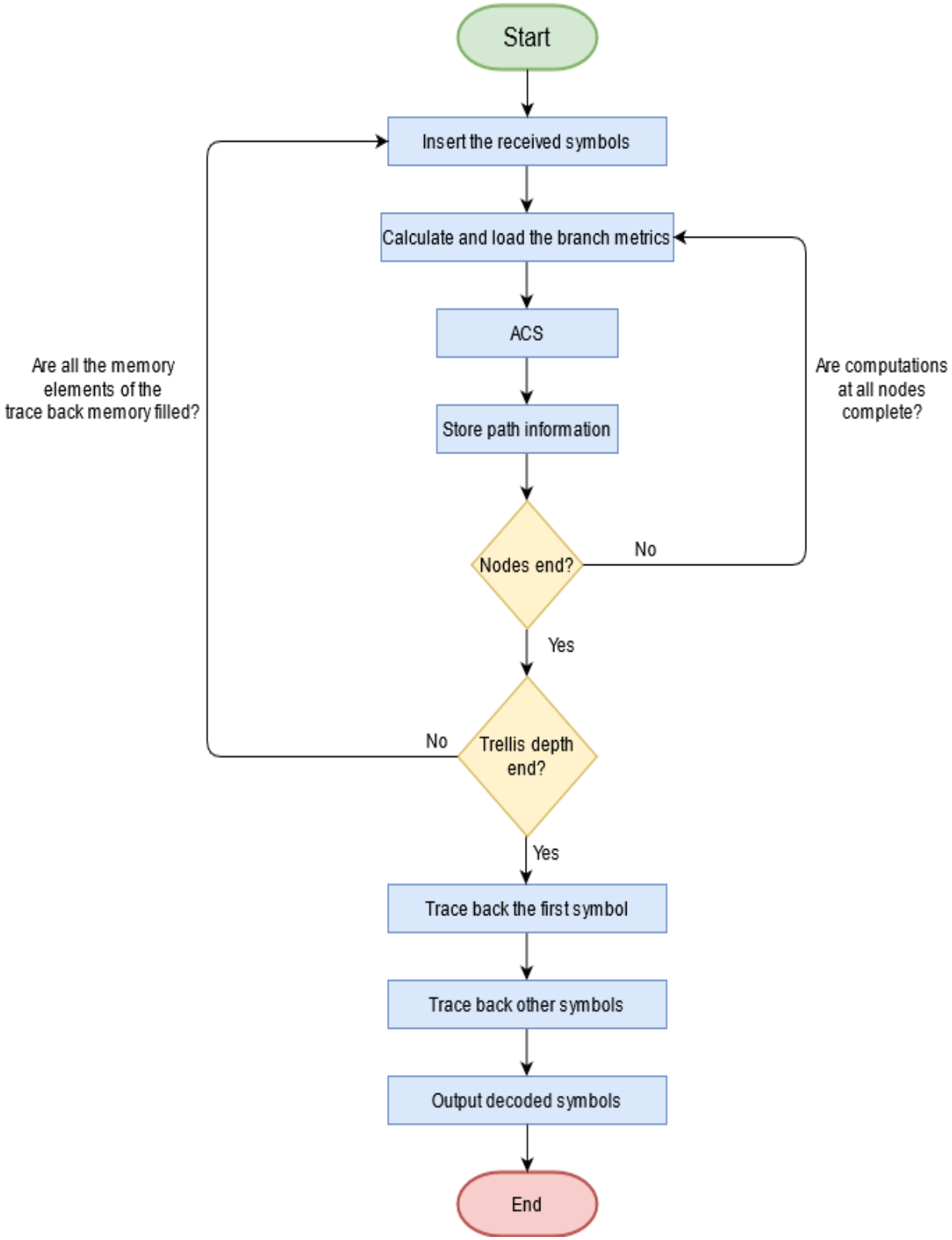

Figure 3.4 Viterbi Decoder Flow Chart

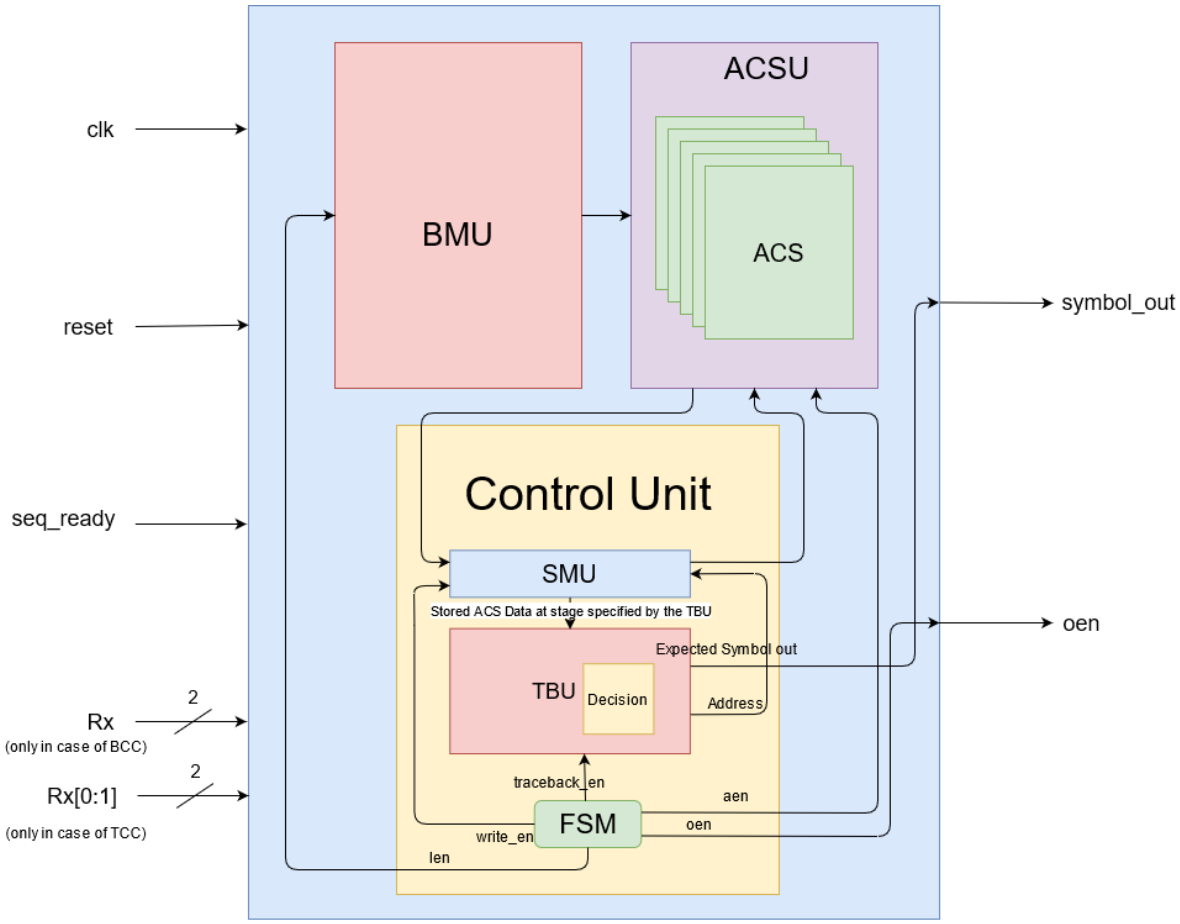

Figure 3.5 Viterbi Decoder Top Level Module

Port	Width	Direction	Description
clk	1	input	Clock signal
reset	1	input	Reset signal
seq_ready	1	input	Data ready signal (validates the input)
Rx	BCC=2; TCC=[2][0:1]	input	Received symbols
symbol_out	BCC=1; TCC=2	output	Decoded symbol
oen	1	output	Output enable signal (validates the output)

Table 3.5 Viterbi Decoder Port Descriptions

3.2.1 Branch Metric Unit (BMU)

The BMU provides the branch metrics which are the Hamming distances between the branch symbols and the received symbols for hard decision decoding. The trellis branches for the (2,1,3) BCC and (2,1,2) TCC are shown in Figures 3.6(a) and 3.6(b), respectively, where the red lines show the transitions due to an input 0, the blue lines transitions due to an input 1, and the green lines transitions due to an input 2.

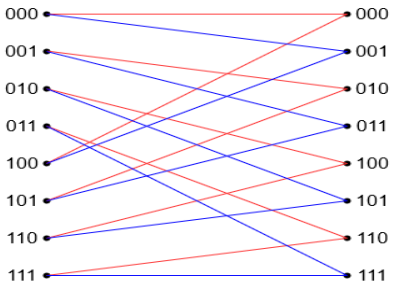

Figure 3.6(a) (2,1,3) BCC Branches

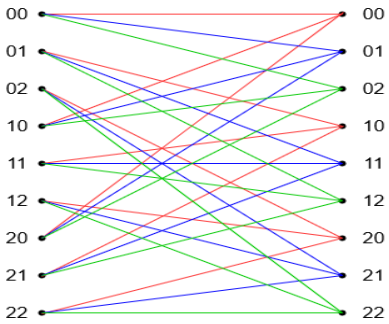

Figure 3.6(b) (2,1,2) TCC Branches

There are two ways of obtaining the branch metrics, calculating the Hamming distances between the branch symbols and the received symbols when the BMU is enabled, or storing the Hamming weights in a LUT and using it as a ROM when the BMU is enabled. The LUT method is preferred because of its simplicity and reduced computational overhead. The BMU block diagram for the BCC and TCC is shown in Figures 3.7(a) and 3.7(b), respectively. The description of the BMU ports is given in Table 3.6. The clock pulse for the BMU is input to the *clk* port. The reset signal that brings the BMU to a known state is input to the *reset* port and the received symbols are input to the *Rx* port. The width of the *Rx* port is two for the BCC and four for the TCC. The BMU is enabled only when the input to the *len* port is 1. The *HD*(*x*) ports output the Hamming distance for the *x*th branch. There are 16 *HD* ports in the BCC and 27 *HD* ports in the TCC.

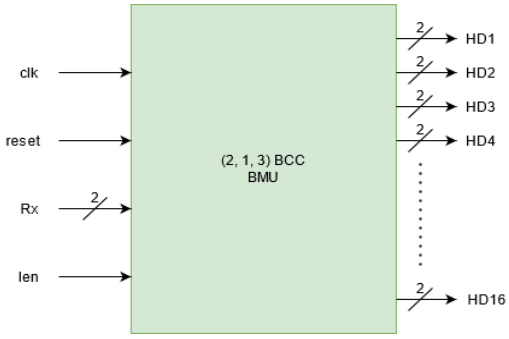

Figure 3.7(a) (2,1,3) BCC BMU

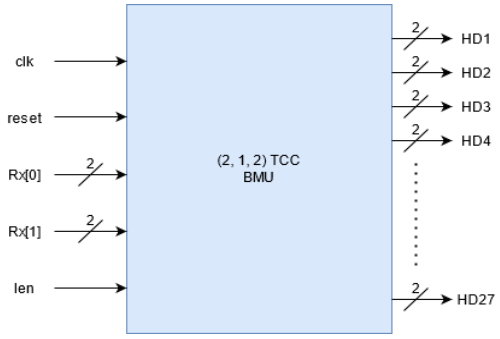

Figure 3.7(b) (2,1,2) TCC BMU

Port	Width	Description
clk	1	Input clock signal
reset	1	Input BMU reset signal
Rx	BCC=2;TCC=[2][0:1]	Input received symbols
len	1	Input BMU enable signal
HD(*x*)	2	Output Hamming distance where *x* is the branch number

Table 3.6 BMU Port Descriptions

3.2.2 Add Compare and Select Unit (ACSU)

The ACSU calculates the path metrics. Each path metric is calculated from the two BCC states or three TCC states from the previous stage. The ACSU consists of multiple ACS blocks and each block corresponds to a node in a stage. There are 2^m x N nodes in a BCC and 3^mx N nodes in a TCC, where N is the number of trellis stages. It is not silicon efficient to have a large number of ACS blocks, hence only one stage is implemented and the ACS outputs are stored in the SMU. In the next stage, the ACS blocks are reused and the stored path metrics from the previous stage are taken from the SMU for the current stage computations.

There are 8 ACS blocks for the BCC while the TCC has 9 ACS blocks. Each ACS block calculates the current path metrics by adding the current branch metrics and the path metrics from the previous stage as shown in Figure 3.8. The minimum path metric per node is chosen by the comparator and selector circuits. The survivor path which corresponds to the minimum path metric is the new path metric output and is stored in the SMU. Each BCC node gets inputs from two nodes from the previous stage so the decision value can be represented by one bit. If the bit is 0, the path metric is selected from the upper node and if it is 1, the path metric is selected from the lower node. In the TCC, each node gets inputs from three nodes from the previous stage so the decision value can be represented by two bits. If the bits are 00, the path metric is selected from the upper node, if they are 01, the path metric is selected from the central node, and if they are 10, the path metric is selected from the lower node. The decision value is the path label output and is stored in the SMU for use in back tracing. The BCC and TCC ACSU block diagrams are shown in Figures 3.9(a) and 3.9(b), respectively. The description of the ACSU ports is given in Table 3.7.

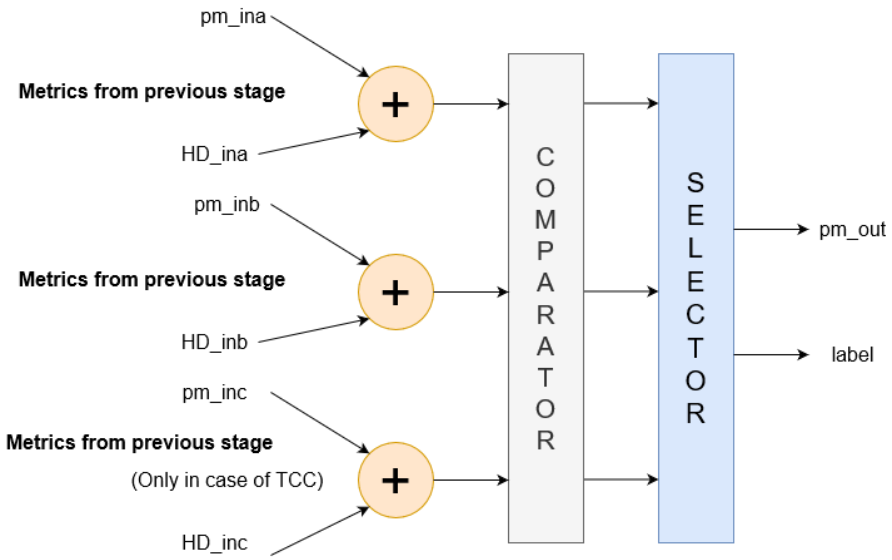

Figure 3.8 ACS Block

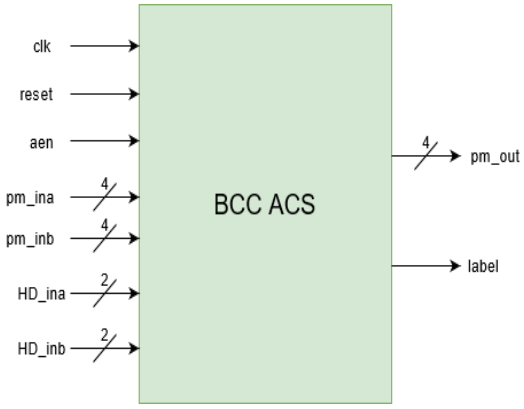

Figure 3.9(a) BCC ACS

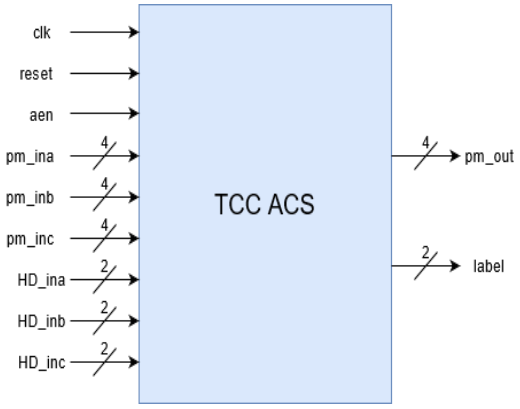

Figure 3.9(b) TCC ACS

The clock pulse for the ACSU is input to the *clk* port and the reset signal that brings the ACSU to a known state is input to the *reset* port. The ACSU is enabled only when 1 is input to the *aen* port. Each path metric in the current stage is calculated from the two BCC states or three TCC states from the previous stage. Hence, two path metrics from the previous stage in the BCC are input to the *pm_ina* and *pm_inb* ports and three path metrics from the previous state in the TCC are input to the *pm_ina*, *pm_inb* and *pm_inc* ports. The two branch metrics for the present stage computation in the BCC are input to the *HD_ina* and *HD_inb* ports and the three branch metrics for the present stage computation in the TCC are input to the *HD_ina*, *HD_inb* and *HD_inc* ports. The path metric calculated in the current stage is output from the *pm_out* port. The path label output port *label* in the BCC is one bit wide since one bit is needed to represent the surviving path label. The path label output port *label* in the TCC is two bits wide since two bits are needed to represent the surviving path label.

Port	Width	Direction	Description
clk	1	input	Clock signal
reset	1	input	Reset signal
aen	1	input	ACS enable signal
pm_ina	4	input	First path metric from the previous stage
pm_inb	4	input	Second path metric from the previous stage
pm_inc	4	input	Third path metric from the previous stage (only in TCC)
HD_ina	2	input	First branch metric from the previous stage
HD_inb	2	input	Second branch metric from the previous stage
HD_inc	2	input	Third branch metric from the previous stage (only in TCC)
pm_out	4	output	New minimum path metric
label	BCC=1; TCC=2	output	Symbol corresponding to the minimum path

Table 3.7 ACSU Port Descriptions

3.2.3 Survivor Memory Unit (SMU)

The SMU is a two-dimensional memory array which stores the data from the ACSU, i.e. the path metrics and the corresponding labels. The SMU is an array of shift registers where each shift register corresponds to a node. There are 8 nodes in the BCC and 9 nodes in the TCC in each stage. Hence, the BCC has 8 shift registers and the TCC has 9 shift registers. The depth of each shift register in the SMU depends on the constraint length K of the convolutional code. The constraint length is the number of message symbols that affect an output symbol, so $K = m + 1$. The depth of each shift register in the SMU should be equal to $5 \times K$ [27]. Therefore, for the (2,1,3) BCC the depth of each SMU register is 20 bits while the depth of each SMU register in the (2,1,2) TCC is 15 trits.

3.2.4 Trace Back Unit (TBU)

The TBU is responsible for tracing back through the trellis and outputting the decoded message symbols. The TBU is enabled after the SMU is filled. The TBU starts tracing through the SMU in a backward direction until the first symbol. The TBU has a decision unit which calculates the state at a particular stage with the lowest path metric and the corresponding survivor path symbol in the SMU. After tracing through the SMU, the TBU outputs the decoded symbol. Figure 3.10 shows the TBU and SMU.

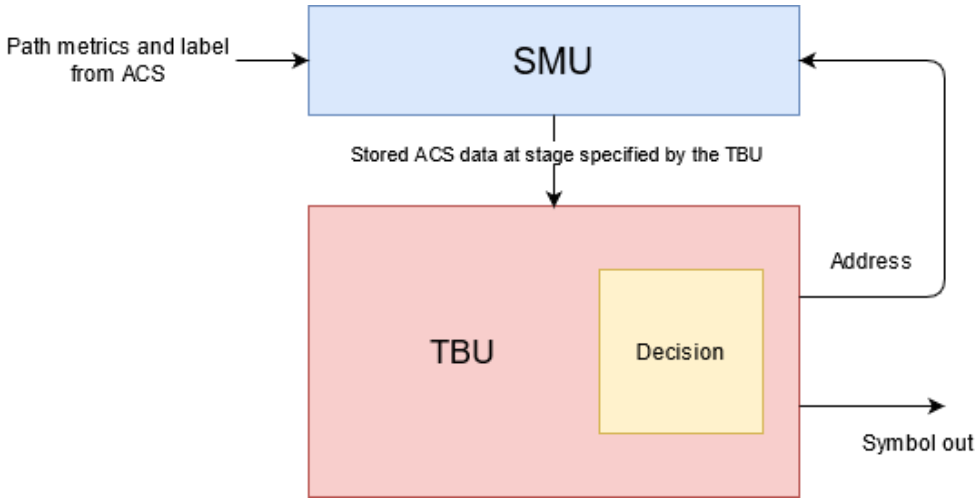

Figure 3.10 TBU and SMU

3.2.5 Control Unit

The control unit directs the data flow in the Viterbi decoder. The TBU and the SMU are inside the control unit since the trace back is controlled by this unit. The control unit has a Finite State Machine (FSM) which controls the flow of data and has the following states.

- **Reset:** The FSM is in the reset state whenever the global reset is applied. This resets all the enable signals of the other modules and the contents of the registers in the SMU. When the reset is deasserted and the input sequence ready signal is high, the FSM moves to the Calculate_BM state in the next clock cycle.
- **Calculate_BM:** In this state, the control unit enables the BMU and the branch metrics are calculated.
- **ACS_en:** In this state, the control unit enables the ACSU and the path metrics are calculated.
- **Write_state:** In this state, the control unit enables the SMU and the outputs from the ACSU are stored in the SMU.
- **Best_state:** In this state, the control unit sets the trace pointer to the maximum value which is the depth of the SMU registers. The trace pointer is a register whose value corresponds to the current stage during trace back in the trellis. In the next clock cycle the FSM moves to the Traceback state and the trace pointer is decremented every clock cycle until it reaches zero.
- **Traceback:** In this state, the control unit enables the TBU and trace back through the trellis is done. A symbol is obtained at each stage during trace back. Once the trace back reaches the first symbol in the TBU it moves to the Output_flush state.
- **Output_flush:** In this state, the control unit enables the output valid signal and outputs the decoded symbol.

A diagram of the control unit is shown in Figure 3.11. The control unit is in reset state until a codeword is received. When the *seq_ready* port input is 1, the control unit enables the BMU for one clock cycle and then disables it. Then the control unit enables the ACSU in the next clock cycle for path metric computations and then disables it. After the path metrics are computed, the control unit enables the SMU in the next clock cycle to store the path metrics and the labels from the ACSU. This process continues until the SMU is full, i.e. 20 × 3 = 60 clock cycles for the BCC and 15 × 3 = 45 clock cycles for the TCC. Then the control unit sets the trace pointer to the maximum value i.e. 20 for the BCC and 15 for the TCC, in one clock cycle and enables the TBU in the next clock cycle. The trace pointer is decremented until it reaches zero. It takes 20 clock cycles and 15 clock cycles in the BCC and TCC, respectively, to trace back to the first symbol in the TBU. After the trace back stage is completed, the control unit enables the output valid signal and the last symbol in the TBU is output as the decoded symbol. The control unit again enables the BMU, ACS and SMU for one clock cycle each and then enables the TBU for 20 cycles and 15 cycles for the BCC and TCC, respectively, and a decoded symbol is output. This process is iterated until the codeword is decoded. Block diagrams of the control unit for the BCC and TCC are shown in Figures 3.12(a) and 3.12(b), respectively. The ports in the control unit are described in Table 3.8.

The clock pulse for the decoder is input to the *clk* port. The reset signal that brings the control unit to a known state is input to the *reset* port and the *seq_ready* port validates the input. Decoding begins when a 1 is input to the *seq_ready* port. Since the control unit has the SMU, the outputs from the ACSU are input to the *acs*(i)*_pm_in* and *acs*(i)*_label* ports of the SMU, where i = 0-7 for the BCC and i = 0-8 for the TCC. In the next stage, the path metrics needed for ACSU computation are output from the *acs*(i)*_pm_out* ports of the SMU, where i = 0-7 for the BCC and i = 0-8 for the TCC. The decoded symbols are output from the *symbol_out* port. The width of the *symbol_out* port is one and two for the BCC and TCC, respectively. The *oen* port validates the output symbols. The control unit enables the BMU if 1 is input to the *len* port and disables the BMU if 0 is input to this port. The control unit enables the ACSU if 1 is input to the *aen* port and disables the ACSU if 0 is input to this port.

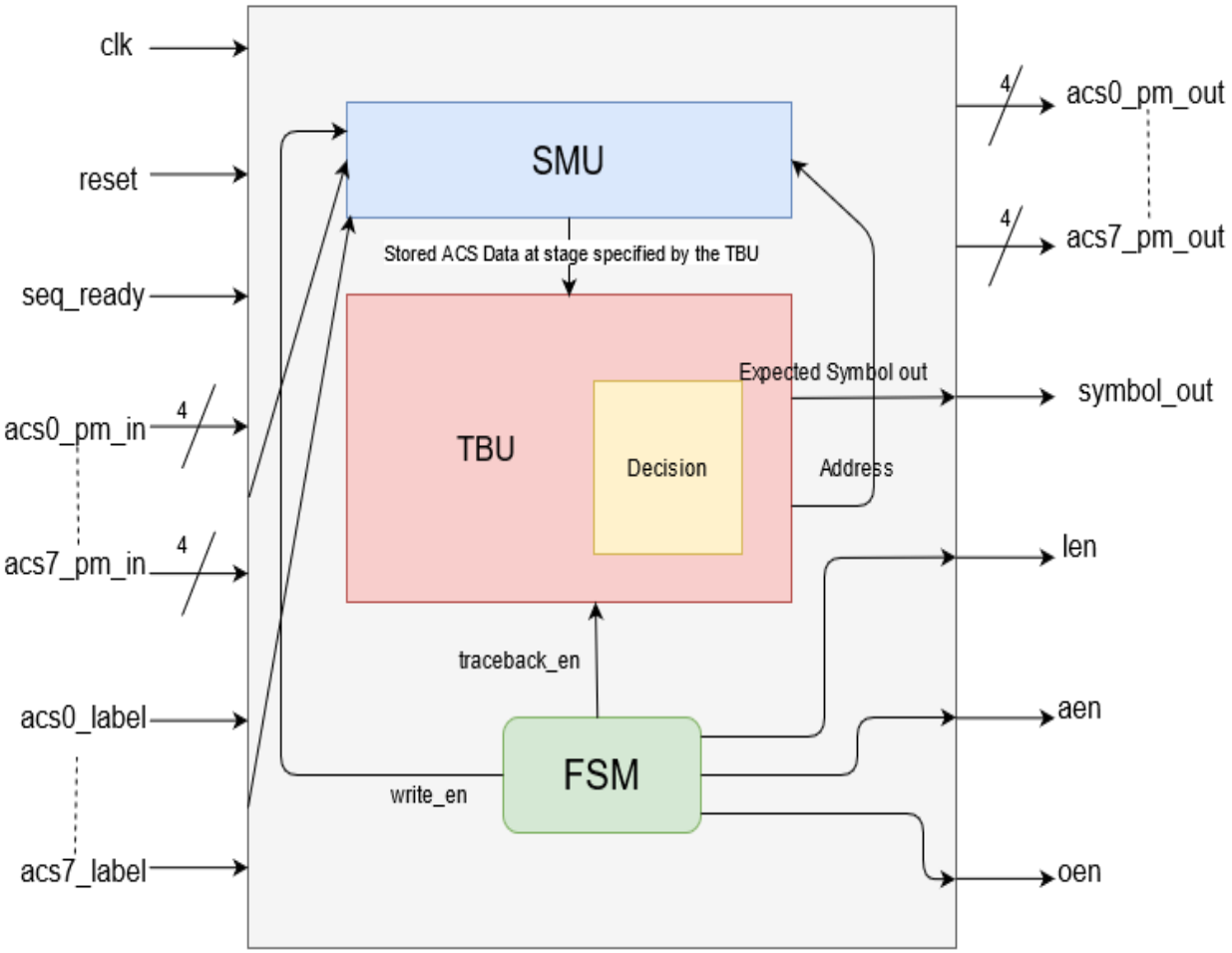

Figure 3.11 Control Unit for the Viterbi Decoder

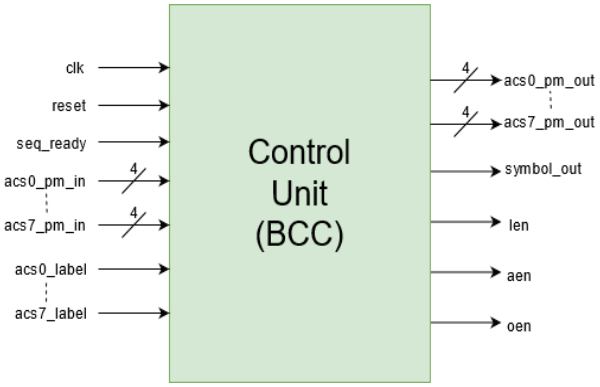

Figure 3.12(a) BCC Control Unit

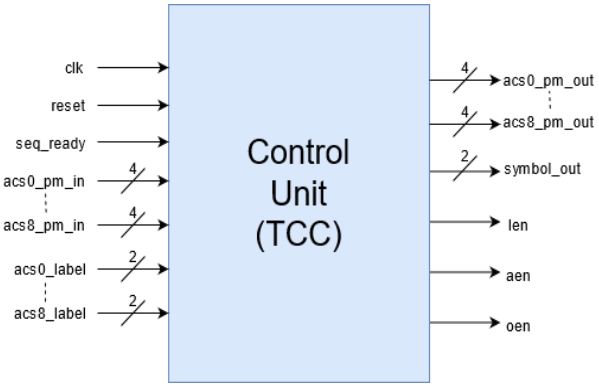

Figure 3.12(b) TCC Control Unit

Port	Width	Direction	Description
clk	1	input	Clock signal
reset	1	input	Reset signal
seq_ready	1	input	Data stream ready signal (validates the input)
acs(*i*)_pm_in	4	input	Path metric from the ACS (*i*=0-7 for BCC, *i*=0-8 for TCC)
acs(*i*)_label	4	input	Path label from the ACS (*i*=0-7 for BCC, *i*=0-8 for TCC)
acs(*i*)_pm_out	4	output	Previous path metric to the ACS
symbol_out	BCC=1; TCC=2	output	Decoded symbol
len	1	output	BMU enable signal
aen	1	output	ACSU enable signal
oen	1	output	Output enable signal (validates the output)

Table 3.8 Control Unit Port Descriptions

3.3 Design Considerations

The following principles should be considered in the design and implementation of a Viterbi decoder on an FPGA.

- **Extensibility:** The encoder and decoder parameters such as the constraint length can be changed.
- **Modularity:** The system contains separate modules and allows the modules to work in parallel independently.
- **Availability:** The FPGA chosen to implement the Viterbi decoder should be commercially available.

Chapter 4 Behavioural Simulation Results and Discussion

In this chapter, the FPGA implementation behavioural simulation results for the (2,1,3) BCC and (2,1,2) TCC are presented and discussed. Timing constraints are used to specify the timing characteristics of a design. A clock is used with the timing constraints to obtain the maximum frequency of the circuits. The clock frequency is increased while running the synthesis until the timing constraints fail, and the highest frequency at which the timing constraints pass is the maximum frequency.

Thirty test bits were randomly generated and passed through the BCC encoder. These bits are converted to 20 trits using the 3B2T conversion in Table 1.1 and these trits are passed to the TCC encoder. The data is given in Table 4.1. The BCC and TCC encoder waveforms are shown in Figures 4.1 and 4.2, respectively. The message stream is input to the Tx port and the clock is input to the *clock* port. The encoder is reset at the start of the behavioural simulation and a pulse is input to the *reset* port. The CW port outputs the generated codeword. The codewords are given in Table 4.2 and shown in red rectangles in the figures.

Test Data	Units	Message
Random data	Bits	10101000101001110001001101l000
3B2T conversion	Trits	02101120012220010112

Table 4.1 Test Data

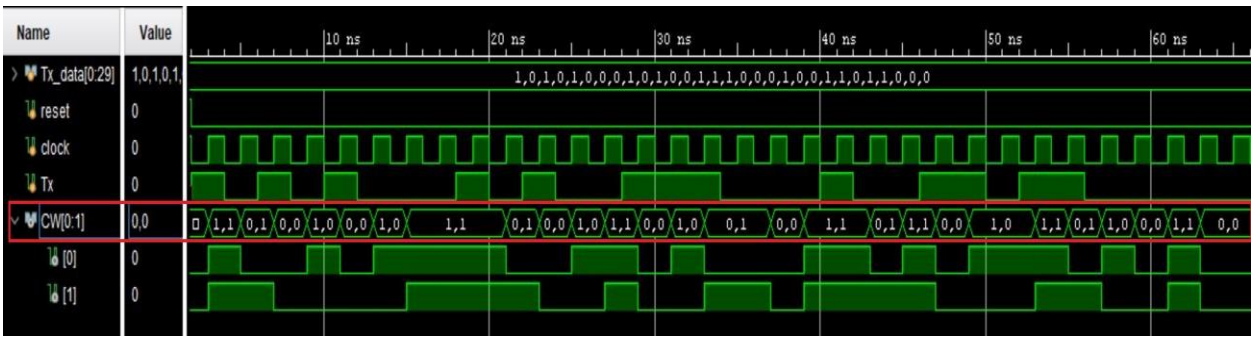

Figure 4.1 BCC Encoder Waveform

Figure 4.2 TCC Encoder Waveform

Code	Codeword
BCC	11,01,00,10,00,10,11,11,11,01,00,10,11,00,10,01,01,00,11,11,01,11,00,10,10, 11,01,10,00,11
TCC	00,21,01,20,00,20,20,10,12,12,02,01,22,01,12,12,11,00,20,20

Table 4.2 The Binary and Ternary Codewords

The codewords were decoded using the Viterbi decoder. The Viterbi decoder behavioural simulation results for the BCC are divided into two halves for better visibility and are shown in Figures 4.3(a) and 4.3(b). The corresponding waveform of the TCC Viterbi decoder is shown in Figure 4.4. The outputs are marked in red rectangles. The clock pulse is input to the *clk* port and the reset signal to bring the decoder to a known state is input to the *reset* port. Decoding begins when 1 is input to the *seq_ready* port. The received symbols are input to the *Rx* port. The width of the *Rx* port is two for the BCC and four for the TCC. The decoded symbols are output from the *symbol_out* port. The width of the *symbol_out* port is one and two for the BCC and TCC, respectively. The output symbols are validated when *oen* is 1 and the decoded symbols are output from the *symbol_out* port.

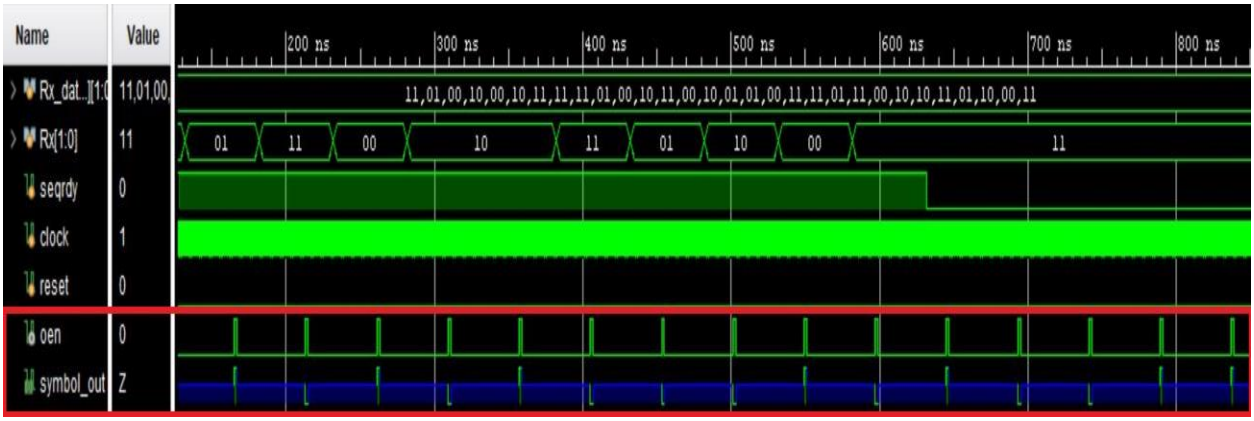

Figure 4.3(a) BCC Viterbi Decoder Waveform (left)

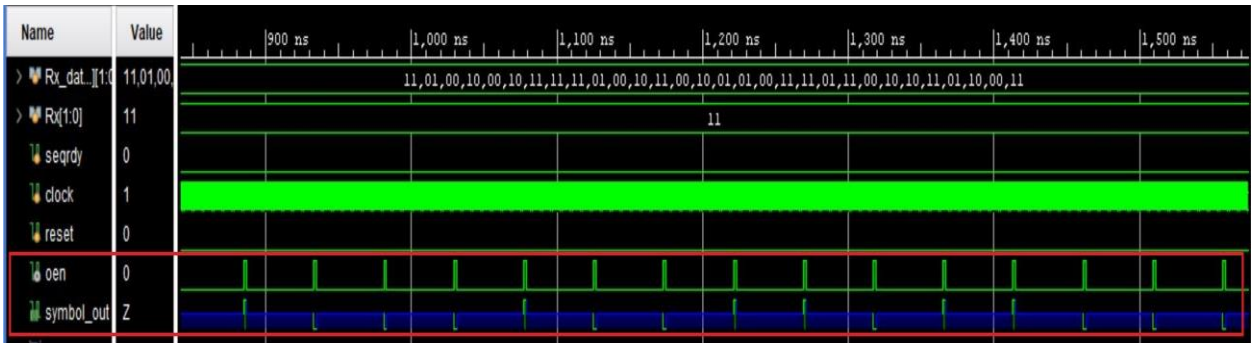

Figure 4.3(b) BCC Viterbi Decoder Waveform (right)

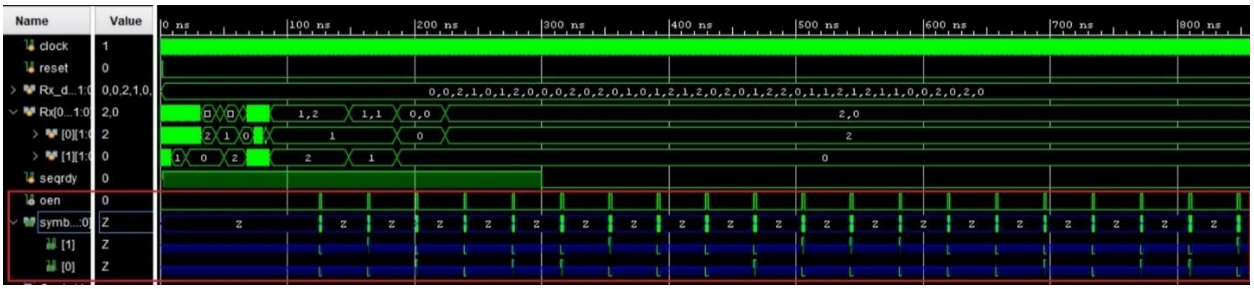

Figure 4.4 TCC Viterbi Decoder Waveform

The decoders successfully decoded the received symbols as shown in Table 4.3. The maximum number of errors that can be corrected by a Viterbi decoder is $\lfloor d_{free} - 1 \rfloor / 2$. d_{free} for the BCC is 7 so a maximum of 3 bit errors can be corrected, and d_{free} for the TCC is 6 so a maximum of 2 trit errors can be corrected. Thus, random symbol errors satisfying these conditions were added to the codewords. The erroneous codewords are shown in Table 4.4. The Viterbi decoder behavioural simulation for the BCC with the erroneous codeword is divided into two halves for better visibility and are shown in Figures 4.5(a) and 4.5(b). The Viterbi decoder behavioural simulation for the TCC with the erroneous codeword is shown in Figure 4.6. The errors are circled in yellow and the outputs are marked in red rectangles. The messages obtained are the same as in Table 4.3. These results show that the messages are correctly decoded which validates the operation of the Viterbi decoders.

Code	Decoded Message
BCC	10101000101001110001001101l000
TCC	02101120012220010112

Table 4.3 Decoded Symbols from the Viterbi Decoder Simulations

Code	Codeword
BCC	10,01,11,10,00,10,11,11,11,01,11,00,11,00,10,01,01,00,11,11,00,11,00,10,10, 10,01,10,01,11
TCC	00,22,11,20,00,20,21,10,12,11,02,01,22,01,02,12,11,01,20,20

Table 4.4 Erroneous Codewords

Figure 4.5(a) BCC Error Correction Waveform (left)

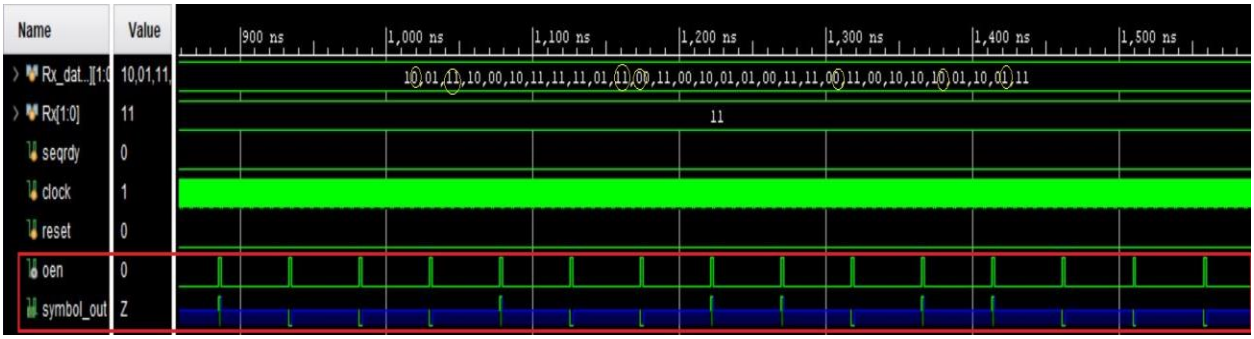

Figure 4.5(b) BCC Error Correction Waveform (right)

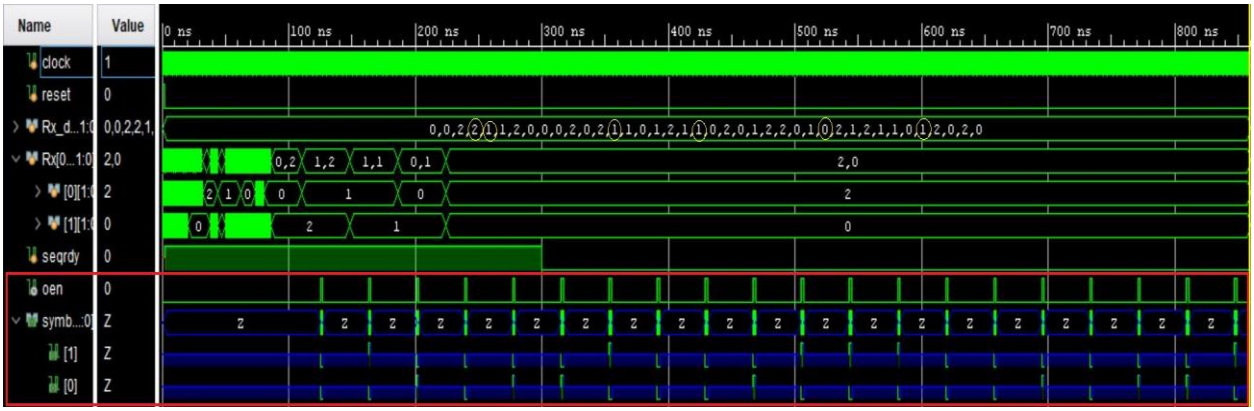

Figure 4.6 TCC Error Correction Waveform

4.1 BCC and TCC Implementation Comparison

The power, maximum frequency, maximum data rate and resource utilization of the BCC and TCC are shown in Table 4.5. The maximum frequency for the TCC implementation is lower than that of the BCC implementation due to the additional resources needed to handle ternary symbols since the FPGA only has binary logic. These additional resources also increase the chip area of the TCC. The encoders have a higher power consumption than the decoders due to the higher dynamic switching activity at the I/O ports. The encoder data rate is the speed at which message symbols are encoded. The decoder data rate is the speed at which the received symbols are decoded. The data rate ratio between the BCC and TCC encoders and decoders is approximately 3:2 since 3B2T conversion is used. Thus the TCC decoder, despite being slower, has a higher data rate due to the decoding of more data per symbol compared to the BCC. The data rates and on-chip power of the BCC and TCC decoders at 50 MHz and 90 MHz are shown in Tables 4.6 and 4.7, respectively. The on-chip power at a fixed frequency and data rate for the decoders is higher than the encoders due to the more complex circuitry of the decoders. The on-chip power for a fixed data rate of 2 Mbps is given in Table 4.8. Due to the fixed data rate, the TCC works at a lower frequency, thus reducing the dynamic switching in the circuitry and as a consequence TCC has the same on-chip power as the BCC.

	Maximum Frequency	LUT Utilization	FF Utilization	On-chip Power	Maximum Data Rate
BCC Encoder	464 MHz	4	5	78 mW	456 Mbps
TCC Encoder	452 MHz	6	8	84 mW	443 Mtps = 665 Mbps
BCC Decoder	102 MHz	301	323	76 mW	3.94 Mbps
TCC Decoder	95.8 MHz	498	497	79 mW	4.52 Mtps = 6.79 Mbps

Table 4.5 Implementation Parameters for the BCC and TCC

	Data Rate	On-chip Power
BCC Encoder	49.2 Mbps	71 mW
TCC Encoder	49.0 Mtps = 73.3 Mbps	72 mW
BCC Decoder	1.92 Mbps	73 mW
TCC Decoder	2.36 Mtps = 3.54 Mbps	75 mW

Table 4.6 Data Rates and On-chip Power for the BCC and TCC at 50 MHz

	Data Rate	On-chip Power
BCC Encoder	88.5 Mbps	72 mW
TCC Encoder	88.2 Mtps = 132 Mbps	73 mW
BCC Decoder	3.46 Mbps	75 mW
TCC Decoder	4.25 Mtps = 6.38 Mbps	78 mW

Table 4.7 Data Rates and On-chip Power for the BCC and TCC at 90 MHz

	On-chip Power	Frequency
BCC Encoder	70 mW	2.03 MHz
TCC Encoder	70 mW	1.36 MHz
BCC Decoder	73 mW	51.9 MHz
TCC Decoder	73 mW	28.2 MHz

Table 4.8 On-chip Power for the BCC and the TCC at a Data Rate of 2 Mbps

Chapter 5 Conclusions and Future Work

Encoders and Viterbi decoders for (2,1,3) binary and (2,1,2) ternary convolutional codes with code rate R = 1/2 were implemented on an FPGA. These codes were chosen for a fair comparison as the number of states only differs by one. Random message bits were generated and 3B2T conversion was employed for the ternary code which maps 3 bits to 2 trits. These symbols were encoded by the corresponding encoders and then decoded. Errors were also added to the codewords to ensure the decoders were operating correctly. The ternary system took less time due to the encoding and decoding of more data per symbol compared to the binary system. The constraint length also plays a factor in the latency since the trace back memory size and the SMU register size is five times the constraint length of the encoder. The ternary Viterbi decoder takes up more area due to the additional ACS block needed for the extra state and the additional hardware needed to support ternary logic because the FPGA only supports binary logic. The implementation of ternary logic is more complex than binary logic because of the ternary computations and the additional resources needed to store ternary symbols.

5.1 Future Work

Due to the commercial unavailability of ternary logic devices, BT conversion is employed with ternary convolutional codes. Ternary systems use more resources than binary systems due to the implementation of ternary logic on a binary system. Ternary logic devices may be commercially available in the future so the implementation of ternary encoders and decoders on ternary logic devices can be considered. Implementation of low power techniques such as clock gating to reduce the power consumption can be considered. Implementation of BCCs and TCCs with code rates such as R = 1/3 and 1/4 can be considered for greater error detection and correction. Codes with longer constraint lengths can also be investigated to increase the error correction capability, but this reduces the data rate because the depth of the SMU registers is dependent on the constraint length.

References

[1] C. E. Shannon, “A mathematical theory of communication,” Bell Systems Technical Journal, vol. 27, pp. 379–423, July 1948.

[2] R. W. Hamming, “Error detecting and error correcting codes,” Bell Systems Technical Journal, vol. 29, pp. 147–160, April 1950.

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

[4] D. Johnsson and F. Bjärkeson, “Playing with the BEAST, Efficient Error Control Coding using the Cell Broadband Engine Architecture,” Lund Institute of Technology, Lund, Sweden, January 2010.

[5] S. Lin and D. J. Costello Jr., Error Control Coding, 2nd ed., Upper Saddle River, NJ, USA, Prentice-Hall, 2004.

[6] Ericsson, “Future mobile data usage and traffic growth,” [Online]. Available: https://www.ericsson.com/en/mobility-report/future-mobile-data-usage-and-traffic-growth.

[7] S. Lin, Y.-B. Kim, and F. Lombardi, “CNTFET-based design of ternary logic gates and arithmetic circuits,” IEEE Transactions on Nanotechnology, vol. 10, no. 2, pp. 217–225, March 2011.

[8] D. Zrilic, A. Mavretic, and M. Freedman, “Arithmetic ternary operations on delta-modulated signals and their application in the realization of digital filters,” IEEE Transactions on Acoustics, Speech, and Signal Processing, vol. 33, no. 3, pp. 760–764, June 1985.

[9] Z. G. Vranesic and K. C. Smith, “Engineering aspects of multi-valued logic systems,” IEEE Computer, vol. 7, no. 9, pp. 34-41, September 1974.

[10] S. Lin, Y.-B. Kim, and F. Lombardi, “Design of a ternary memory cell using CNTFETs,” IEEE Transactions on Nanotechnology, vol. 11, no. 5, pp. 1019–1025, August 2012.

[11] T. Koike and S. Yoshida, “Space-time trellis-coded ternary PSK for mobile communications,” Electronics Letters, vol. 40, no. 16, pp. 1011–1012, August 2004.

[12] M. Abdelaziz and T. A. Gulliver, “Ternary coding and triangular modulation,” IEEE Access, vol. 7, pp. 49027–49038, April 2019.

[13] M. Tanahashi and H. Ochiai, “A multilevel coded modulation approach for hexagonal signal constellation,” IEEE Transactions on Wireless Communications, vol. 8, no. 10, pp. 4993–4997, October 2009.

[14] S. D. Brown, “An overview of technology, architecture and CAD tools for programmable logic devices,” Proceedings of the IEEE Custom Integrated Circuits Conference, pp. 69-76, May 1994.

[15] Y. Jiang, A Practical Guide to Error-Control Coding Using MATLAB, Norwood, MA, USA, Artech House, 2010.

[16] F. J. MacWilliams and N. J. A. Sloane, The Theory of Error-Correcting Codes, New York, NY, USA, North-Holland, 1977.

[17] D. Chase, “A class of algorithms for decoding block codes with channel measurement information,” IEEE Transactions on Information Theory, vol. 18, no. 1, pp. 170-182, January 1972.

[18] P. Elias, “Coding for noisy channels,” IRE International Convention Record, pp. 37–47, March 1955.

[19] A. Viterbi, “Convolutional codes and their performance in communication systems,” IEEE Transactions on Communication Technology, vol. 19, no. 5, pp. 751-772, October 1971.

[20] J. L. Massey and M. K. Sain, “Inverses of linear sequential circuits,” IEEE Transactions on Computers, vol. 17, no. 4, pp. 330-337, April 1968.

[21] D. L. Bitzer, A. Dholakia, H. Koorapaty, and M. A. Vouk, “On locally invertible rate-1/n convolutional encoders,” IEEE Transactions on Information Theory, vol. 44, no. 1, pp. 420–422, January 1998.

[22] G. D. Forney, “The Viterbi algorithm,” Proceedings of the IEEE, vol. 61, no. 3, pp. 268–278, March 1973.

[23] A. Viterbi, “Error bounds for convolutional codes and an asymptotically optimum decoding algorithm,” IEEE Transactions on Information Theory, vol. 13, no. 2, pp. 260–269, April 1967.

[24] Y. M. Sandesh and K. Rambabu, “Implementation of convolution encoder and Viterbi decoder for constraint length 7 and bit rate 1/2,” International Journal of Engineering Research and Applications, vol. 3, no. 6, pp. 42– 46, November-December 2013.

[25] R. Vijay, S. Mrinalee, and G. Mathur, “Comparison between Viterbi algorithm soft and hard decision decoding,” Journal of Information Systems and Communication, vol. 3, no. 1, pp. 193–198, March 2012.

[26] S. C. Krishnan, R. Panigrahy, and S. Parthasarathy, “Error-correcting codes for ternary content addressable memories,” IEEE Transactions on Computers, vol. 58, no. 2, pp. 275–279, February 2009.

[27] R. Manzoor, A. Rafique, and K. B. Bajwa, “VLSI implementation of an efficient pre-trace back approach for Viterbi algorithm,” Proceedings of the International Bhurban Conference on Applied Sciences & Technology, pp. 27-30, January 2007.