Efficient ASIC Design of Viterbi Decoders

By

John O'Shea

A thesis submitted by John O'Shea, (ID 9718273)
to the University Of Limerick in candidacy for the degree of

Master of Engineering
Computer Systems

undertaken in the department of Electronic and Computer Engineering,
The College of Informatics and Electronics, under the supervision of

Dr. Máirtín Ó Droma, BE, PhD, C.Eng., SMIEEE, FIEE

University Of Limerick, Ireland
Ollscoil Luimnigh, Eire
October 2001

Table of Contents

Abstract

The Viterbi decoding algorithm, is a powerful method for forward error correction. The decoding algorithm introduced in 1967 is a maximum likelihood decoding procedure of convolutional codes. Consequently the focus of research for Viterbi decoders has been on efficient of VLSI implementations for use in communication and data storage systems, and is an important field of study for many years.

A novel approach to achieving path memory savings in a Viterbi decoder based on tracking trellis survivor paths in the decoding decision process was proposed in a previous paper. This novel approach exploits the shift-register property of the trellis when using the backward label traceback technique to produce memory savings without loss of decoding performance. The savings in path memory also contribute to an improvement in decoding performance, which is to be welcomed. The purpose of this project was to investigate the exact comparison between forward and backward label Viterbi decoders, detailing the data structures required for each decoder. Following on from this analysis, a number traceback Viterbi decoders using this novel path memory technique were successfully developed. In order to implement this novel technique, a number of small changes are required to the conventional backward label traceback decoder design, including a modified traceback algorithm to manage the decoding operations. Care was taken in the designs to ensure that the techniques used in the novel decoders do not result in additional logic elsewhere in the designs.

It is shown that Viterbi decoders using this novel technique require a smaller chip size and achieve a faster decoding time without loss of decoding performance. The VLSI designs were implemented in a Xilinx FPGA. Decoders using this novel path memory can achieve savings of 20% in storage for (n,1,m) code, and <=20% for the (n,k,m) codes.

Acknowledgments

I would like to thank Dr. Máirtín O'Droma, my supervisor, for his guidance, assistance, and patience throughout the course of this project. I must express my gratitude to all secretaries, technicians, and other members of the ECE staff who have helped me throughout the Masters Degree. Finally, I would like to thank my family and friends for their encouragement.

Declaration

The work carried out in this report is my own work. The purpose of this report is to extend the initial research done on potential path memory savings in backward label Viterbi decoders. The sections that are not my original work are referenced as so within the text to the title of the work and the author in the Bibliography.

1. Introduction

1.1 Project Goal

The primary goal of the project was to investigate existing Viterbi decoder implementations, forward and backward label, and detail the data structures for each decoder. Secondly, implement decoders using a novel path memory savings technique, proposed in [4]. The approach and details of implementing the novel path memory technique is outlined in sections 5.1, followed by details of a modified algorithm to manage the novel path memory, outlined in section 5.2.

1.2 Overview of Project

1.2.1 Convolutional Codes and Decoding Techniques

Section 2.1 starts with an overview of the properties of convolutional codes. This section describes in detail what convolutional codes are, and how a convolutional code is defined.

A number of techniques exist to decode convolutional codes, the Viterbi algorithm being just one technique. Section 2.2 discusses the merits of each convolutional decoding technique. Among the techniques, the Viterbi algorithm is of major importance because it is a maximum likelihood (ML) technique. The details of the Viterbi algorithm are fully detailed in section 2.3. The operation of the algorithm with non-implementation details and a simple example are described in sections 2.3.1, and 2.3.2 respectively, followed by a detailed analysis on how the Viterbi algorithm performs ML decoding in section 2.3.3.

1.2.2 Architecture of Viterbi Decoders

In practice there are a number of distinct Viterbi decoder architecture implementations. However each architecture can be logically broken down into a number of distinct processing units. This is outlined in section 2.4. The implementation differences between each block are then explained in detail.

1.2.3 Survivor Path Memory Design In Viterbi Decoders

The survivor path memory unit is a key function in Viterbi decoders. Essentially, the survivor path memory unit in a Viterbi decoder keeps track of the information bits associated with different survivor paths along its trellis. Its design and implementation is a key focus in practical decoder implementations, and is one of the primary factors in determining the size of VLSI based decoder implementations. Therefore survivor path memory design has been a major field of study since the inception of the Viterbi algorithm. Section 3 covers this topic in detail.

A practical survivor path memory unit has a finite capacity and typically cannot hold all surviving paths at any one time, therefore a truncated path memory is used. This truncated path memory is basically a “window” that moves along the decoding process, and is typically just long enough to ensure ML decoding, [2]. Once the survivor path memory unit is full a ML decoding decision is made for one symbol. After one symbol is decoded, room is made in the survivor path memory unit for the next set of survivors. The requirement for a minimum survivor path memory size for ML decoding is thus a limiting factor in determining the size of a practical VLSI decoder. Thus any improvements to the algorithm design and implementation, which will reduce the survivor path memory requirements, are to be welcomed, [4], [5], [6]. The novel technique described in this project is a more efficient ML decoding scheme for convolutional codes. Essentially this technique takes advantage of the inherent history in a trellis structure for all surviving paths in the survivor path memory unit to make ML decisions earlier in the traceback process. Full implementation details of this technique are outlined in section 5.

1.2.4 Implementing Decoders with the Novel Path Memory Unit

Section 6.2 outlines the successful implementations of two MATLAB based Viterbi decoders which utilize the novel traceback path memory unit. The goal of this section is to prove that the novel Viterbi decoders can make ML decisions from a noisy channel without loss of decoding performance. This simulation consisted of a number of steps. First, the input sequences are randomly generated, and encoded using a model of the convolutional encoder. The encoded sequence was then subjected to a binary symmetric channel with added white Gaussian noise to produce a noisy received sequence. Finally hard decision decoding was then performed on the noisy sequence using the novel Viterbi decoder, and a measure of the BER and the number of bit errors calculated.

Once comfortable in the knowledge that the novel Viterbi decoder can operate without loss of decoding performance, a number FPGA based VLSI designs were implemented, this is detailed in sections 6.3, 6.4, and 6.5. The goal here is to implement both the conventional and novel decoders in order to generate a gates to gates comparison, and to generate data on the decoding performance improvements for the novel designs. An analysis of the decoding performance improvements for the novel technique are detailed in section 6.73.

2. Convolutional Decoding

2.1 Properties of Convolutional Codes

2.1.1 Convolutional Encoders

The general (n, k, m) non-systematic convolutional encoder, is an application of k parallel serial-in parallel-out variable length shift registers, [4] as shown in figure 2-1.

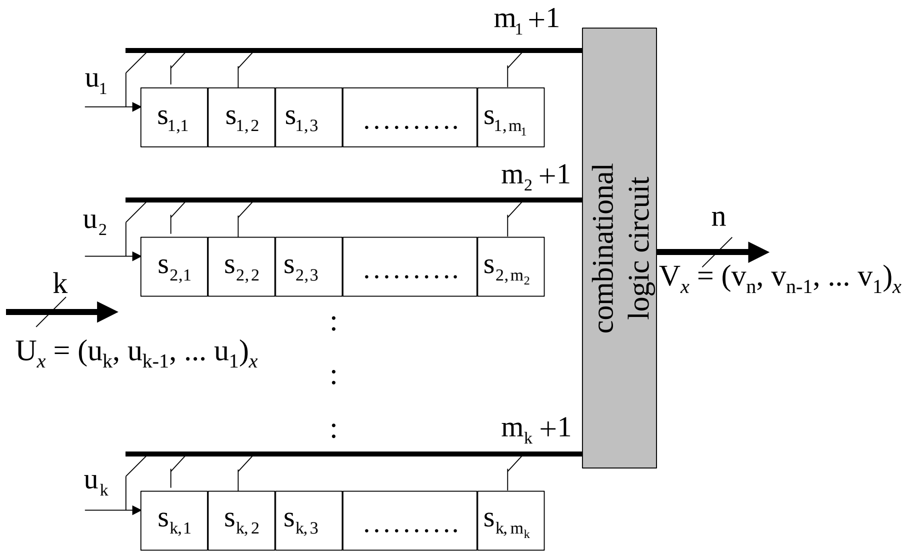

m_1+1
u_1
$s_{1,1}$ $s_{1,2}$ $s_{1,3}$ ……… s_{1,m_1}
m_2+1
u_2
$s_{2,1}$ $s_{2,2}$ $s_{2,3}$ ……… s_{2,m_2}
k
$U_x = (u_k, u_{k-1}, \ldots u_1)_x$
m_k+1
u_k
$s_{k,1}$ $s_{k,2}$ $s_{k,3}$ ……… s_{k,m_k}
combinational logic circuit
n
$V_x = (v_n, v_{n-1}, \ldots v_1)_x$

Figure 2-1 (n, k, m) non-systematic convolutional encoder

A convolutional encoder accepts k-bit input symbols U_x from the input sequence U and produces n-bit output symbols V_x which form the encoded sequence V. In convolutional coding U and V are continuous sequences, but typically are divided into pseudo-blocks of fixed length. Each output symbol V_x encoded depends not only on the present k-bit symbol U_x but also on m previous stages. Hence the encoder has a memory of order m, and will have a set of known states for the encoder memory. A trellis diagram is an extension of the convolutional encoder's state diagram, and simply represents the convolutional encoder state changes over the passage of time.

2.1.2 Polynomial Representation of Convolutional Encoders

To describe a convolutional code, one needs to characterize the encoding function G(m), so that given an input sequence U, one can readily compute the output sequence V. Several methods are used for representing a convolutional encoder, the most popular are generator polynomials. A convolutional encoder can be represented by a set of *n* generator polynomials, one for each of the *n* modulo-2 adders, [14]. Each polynomial is of degree m or less and describes the connection of the encoding shift register to that modulo-2 adder, where m is the maximal length shift register. The coefficient of each term in the polynomial represents either a 0 or 1, depending on whether the connection exists or does not exist between the shift register and the modulo-2 adder in question.

Example, consider the (2,1,3) convolutional code in figure 2-2.

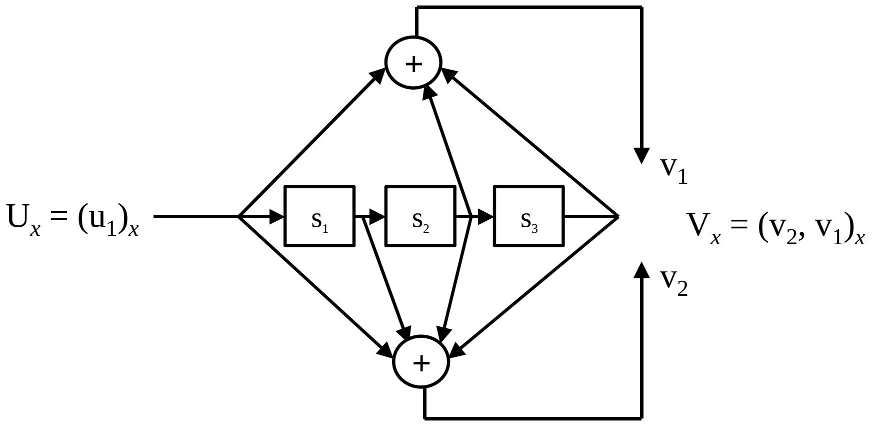


Figure 2-2 (2,1,3) convolutional encoder

This encoder has the generator polynomial: $G(x)=\begin{bmatrix}1+x^2+x^3 \\ 1+x\ +x^2+x^3\end{bmatrix}$.

Where the lowest order term in the polynomial corresponds to the input of the stage of the encoder register. Any output sequence can be determined given the input sequence and the convolutional code generator polynomial.

Example, given the input sequence, U = [0,1,1,0,1] of length N=5, the (2,1,3) convolutional encoder as shown in figure 2-2 encodes the transmitted sequence V to [00,11,10,10,11,10,11,11] as follows:

$$\mathrm{V}=\mathrm{U}.\mathrm{G}(x)$$

$$\mathrm{U}=[0,1,1,0,1]$$

$$\mathrm{U}=[x+x^2+x^4](In-Polynomial-Form)$$

$$\mathrm{V}=[x+x^2+x^4]\begin{bmatrix}1+x^2+x^3 \\ 1+x\ +x^2+x^3\end{bmatrix}$$

$$\begin{aligned}\mathrm{v}_2&=[x+x^2+x^4].[1+x^2+x^3]\\ \mathrm{v}_2&=[x+x^2+x^3+x^4+x^5+x^6+x^7]\\ \mathrm{v}_2&=[0,1,1,1,1,1,1,1]\end{aligned}$$

$$\begin{aligned}\mathrm{v}_1&=[x+x^2+x^4].[1+x+x^2+x^3]\\ \mathrm{v}_1&=[x+x^4+x^6+x^7]\\ \mathrm{v}_1&=[0,1,0,0,1,0,1,1]\end{aligned}$$

Finally, the convolutional encoded sequence V is obtained by multiplexing v_2 and v_1 such that V= [00,11,10,10,11,10,11,11]. The length of V always equals n(N + m), in this case V=2(5+3) or sixteen bits. Every convolutional encoded sequence has N n-bit symbols, plus m n-bit symbols appended to the encoded sequence called flushing symbols. When the encoder reaches the end of the encoded sequence it has to shift in m zeros to bring the encoder back to its initial state (usually state 0). This process is called flushing the encoding shift register.

2.1.3 Definition of a Trellis Stage

For an (n, k, m) non-systematic convolutional encoder, a stage denotes an instance of the encoding process, i.e. the encoders memory state, its input and its output, which lasts for the duration of one input symbol period, [4]. In any stage *x*, the n bits of the output symbol, $V_x = (v_n, v_{n-1}, \ldots v_1)_x$, are given by fixed combinations of selected bits of the input symbol, $U_x = (u_k, u_{k-1}, \ldots u_1)_x$ and selected elements of the encoder memory, i.e. the elements of previous input symbols. The state of this memory at stage *x* is denoted S_x

$$S_x = (s_{j,i})_x, \quad \text{where } j = k,\ldots 1 \text{ and } i = m_j,\ldots 1.$$

There are 2^M possible encoder states in total where, $M = \sum_{j=1}^{k} m_j$, represents the total memory. The maximal memory order of the encoder, is $m = \max(m_j)$.

Any state (state diagram or table), or node (trellis diagram), is connected directly to 2^k other states both on its input and on its output. A state transition from state S_x to state S_{x+1} occurs when the k bits of input symbol U_x are clocked in, the contents of all the registers have moved one memory element to the right, and new symbols, U_{x+1} and V_{x+1} are present at the input and output respectively.

For the (2,1,3) encoder in figure 2-2, there are 2^M = eight possible states of the encoder, where in this case the total memory M and the maximal memory order m, are both three. The state transition table for the (2,1,3) encoder is shown in table 2-1.The state transitions for the (2,1,3) encoder can also be represented by a trellis diagram. A trellis diagram is a plot of the state transition history of a convolutional encoder. A small number of state transitions for the (2,1,3) encoder are shown in figure 2-3. Each branch in the trellis is labeled as U_x / V_x, or u_x/ $(v_2, v_1)_x$.

Present State (s_3,s_2,s_1)	**Encoder Input** $U_x = (u_1)_x$	**Next State** (s_3,s_2,s_1)	**Encoder Output** $V_x = (v_2,v_1)_x$
000	0	000	00
000	1	001	11
001	0	010	01
001	1	011	10
010	0	100	11
010	1	101	00
011	0	110	10
011	1	111	01
100	0	000	11
100	1	001	00
101	0	010	10
101	1	011	01
110	0	100	00
110	1	101	11
111	0	110	01

Table 2-1 (2,1,3) encoder state transition table

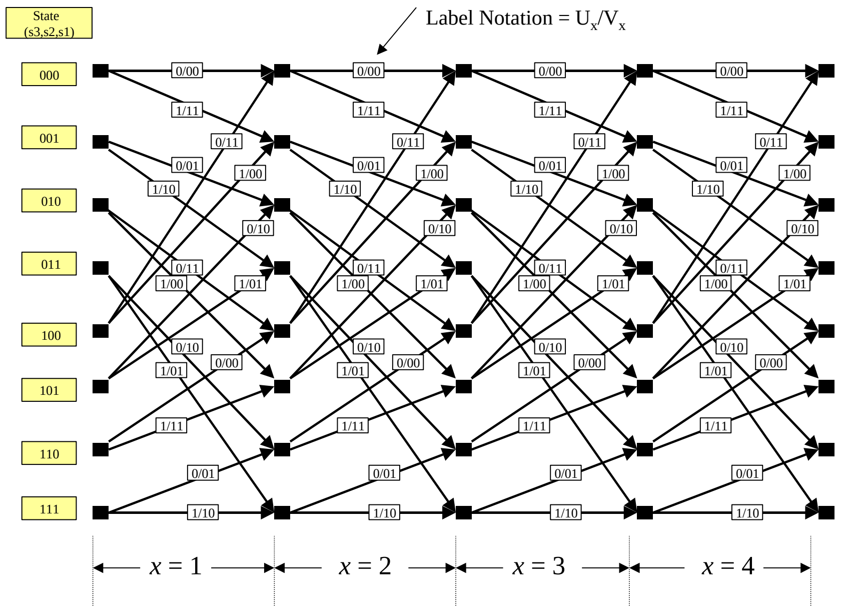


Figure 2-3 Trellis diagram for the (2,1,3) encoder.

2.2 Overview of Convolutional Decoding Algorithms

There are two widely known techniques to decode convolutional codes, sequential decoding, and Viterbi decoding, [14].

The first known convolutional decoder used the sequential decoding algorithm, originally proposed by Wozencraft, and subsequently modified by Fano, [14]. A sequential decoder works by generating hypotheses about the transmitted encoded sequence by computing a metric between these hypotheses and the received signal. It goes forward as long as the metric indicates that its choices are likely, other wise it goes backward, changing hypotheses until through systematic trial and error search, it finds a likely hypothesis. Sequential decoders can be implemented to work with hard or soft decisions, but soft decisions are usually avoided because they greatly increase the amount of storage required and the complexity of the computations. Sequential decoding can be viewed as a trial and error technique for searching out the correct path in the code tree. It performs the search in a sequential manner, always operating on just a single path at a time. If an incorrect decision is made, subsequent extensions of the path will be wrong. The decoder can eventually recognize its error by monitoring the path metric.

The second is the more widely used Viterbi algorithm. This algorithm proceeds through the decoding process by computing a hypothesis for every possible path in the trellis and retaining a fixed number of paths at any one time. At each stage, the algorithm computes a path metric for each incoming path to a trellis node and selects one path as the survivor based on a maximum likelihood criterion, all other paths are no longer considered. Each path metric is computed by summing the path metrics from nodes at the previous stage that connect to nodes with path metrics computed at the present stage. Viterbi decoders can be implemented to work with hard or soft decisions. The disadvantage of Viterbi decoding is that while error probability decreases exponentially with constraint length, the number of code states, and consequently decoder complexity grows exponentially as a function of the maximal length shift register, [14]. On the other hand, the computational complexity of the

Viterbi algorithm is independent of channel characteristics, i.e. soft decision Viterbi decoders require only a trivial increase in computation over hard decision Viterbi decoders, [14].

Sequential decoding achieves asymptotically the same error probability as maximum likelihood decoding but without searching all possible states. In fact with sequential decoding the total number of states searched is essentially independent of the constraint length, thus making it possible to use very large constraint lengths, i.e. the maximal length shift register can be $m <= 41$, [14]. This is a major factor in providing such low error probabilities.

The major drawback of sequential decoding is that the number of state metrics searched is a random variable. In other words, the expected number of poor hypotheses and backward searches is a function of the channel signal to noise ratio (SNR), [14]. Because of this variability in computational load, buffers must be provided to store the arriving sequences. Under low SNR, the received sequences must be buffered while the decoder is labouring to find a likely hypothesis. If the average symbol arrival rate exceeds the average symbol decode rate, the buffer will overflow, no matter how large it is, causing a loss of data. The sequential decoder typically puts out error-free data until the buffer overflows, at which time the decoder has to go through a recovery procedure. The buffer overflow threshold is a very sensitive function of SNR. Therefore, an important part of a sequential decoder specification is probability of buffer overflow, [14].

2.3 The Viterbi Decoding Algorithm

This section describes the operation of the Viterbi algorithm. Initially an unambiguous but simplified explanation of the Viterbi Decoding Algorithm (VDA) process for convolutional encoded sequences is outlined in section 2.3.1. This explanation will omit specific implementation details and mathematical theory and concentrate on how the VDA operates to make decoding decisions. Section 2.3.2 outlines in a simplified manner how metrics are computed for Viterbi decoding over a BSC channel, this is followed by a step by step example of Viterbi decoding using hard decisions. Finally the mathematical theory of how the Viterbi Algorithm performs maximum likelihood decisions is derived in section 2.3.3.

2.3.1 Viterbi Decoding Algorithm Procedure

The VDA is performed by maintaining memory which stores two kinds of information related to hypotheses about what sequence was encoded for transmission. There are 2^M hypotheses stored one for each encoder state, where M is the total encoder memory. Each hypothesis either corresponds to a particular sequence of encoder input symbols up to and including the current trellis stage, (forward labels), or corresponds to a particular sequence of encoder output symbols up to and including the current trellis stage, (backward labels). For each of the 2^M possible states of the encoding shift register there is one most likely decoder hypothesis for which that state could exist. One entry in the memory is the path metric which represents the likelihood of the hypothesis. The other entry represents earlier encoder symbols, which led to the presumed encoder state and is called the surviving path, [12]. As the algorithm proceeds, the path metrics of surviving hypotheses are derived from the path metrics of the hypotheses from which the survivors are descended, by adding appropriate branch metrics computed at the present stage to the corresponding path metrics. A branch metric is computed based on difference in the current received symbol and current trellis branch symbol. For example, if minimum Hamming distances are used to compute branch metrics, then the branch metrics are computed by counting the number of elements in which the received symbol and each trellis branch differ.

At the beginning of a decoding process, the algorithm has a known initial trellis state, and returns to this initial trellis state once an entire received convolutional encoded sequence has been decoded. Each received sequence is of length n(N + m), and is divided evenly into n-bit blocks, one for each decoding stage of the trellis. The VDA then proceeds by computing a path metric for each possible path in the trellis and retaining 2^M of those paths at any one time based on a maximum likelihood criterion. If the decoder uses minimum distance metrics for maximum likelihood decoding, then at the beginning of a decoding process, the path metric for the known initial state (usually state 0) is set to zero, and the remaining $2^M - 1$ path metrics set to their maximum value. At each stage, there are 2^k paths entering each trellis node. The VDA computes a new path metric for each incoming path and selects the path with the minimum path metric as the survivor, all other paths entering the trellis node are no longer considered. If the result of the computation shows that there is a tie in selecting a surviving path, i.e. two or more path metrics are equal then one is selected at random. A procedure to perform all the steps involved in the Viterbi Algorithm is shown below in table 2-2.

1. ***Initialize Data Structures***
    - 1.1. Initialize the trellis stage pointer to zero.
    - 1.2. Initialize the path metric for the known initial state to zero, with the remaining 2^M -1 path metrics to their maximum value. Go to step 2.

2. ***Compute Path Metrics and Survivors***
    - 2.1. Increment the trellis stage pointer.
    - 2.2. For every trellis node, compute 2^k path metrics by summing the path metrics from nodes at the previous stage to the corresponding branch metrics computed at the present stage[1].
    - 2.3. Compare the 2^k paths and select the path with the minimum path metric as the surviving path, all other incoming paths to the trellis node are no longer considered. If there is a tie between path metrics, the algorithm selects one path.
    - 2.4. Store the path metric and the surviving path.
    - 2.5. If the trellis stage pointer is < N + m then go to step 2.1, else if the trellis stage pointer is = N + m go to step 3

[1] If all the incoming path metrics to a trellis node have a maximum value, the path metric for that node is held at that maximum value.

3. ***Output Decision***
    - 3.1. Determine the minimum path metric state.
    - 3.2. The minimum path metric state is used to select the corresponding surviving path as the ML surviving path
    - 3.3. Release the ML path and finish.

Table 2-2. Viterbi Algorithm Procedure

2.3.2 Decoding Example Using Hard Decisions

Given the short sequence U = [0,1,1,0,1] of length N = 5, the (2,1,3) convolutional encoder as shown in figure 2-2 encodes the transmitted sequence V to [00,11,10,10,11,10,11,11]. The sequence V is transmitted over a noisy BSC channel, and the message R is received with a single bit error. The single bit error is underlined in bit position eight as follows: R = [00,11,10,1$\underline{1}$,11,10,11,11]. The final ML path chosen by the VDA is highlighted by the thick line in figure 2-4. Each trellis branch is labeled with the corresponding encoded symbol. Each trellis node is labeled with a path metric that represents the likelihood for the surviving path up that stage.

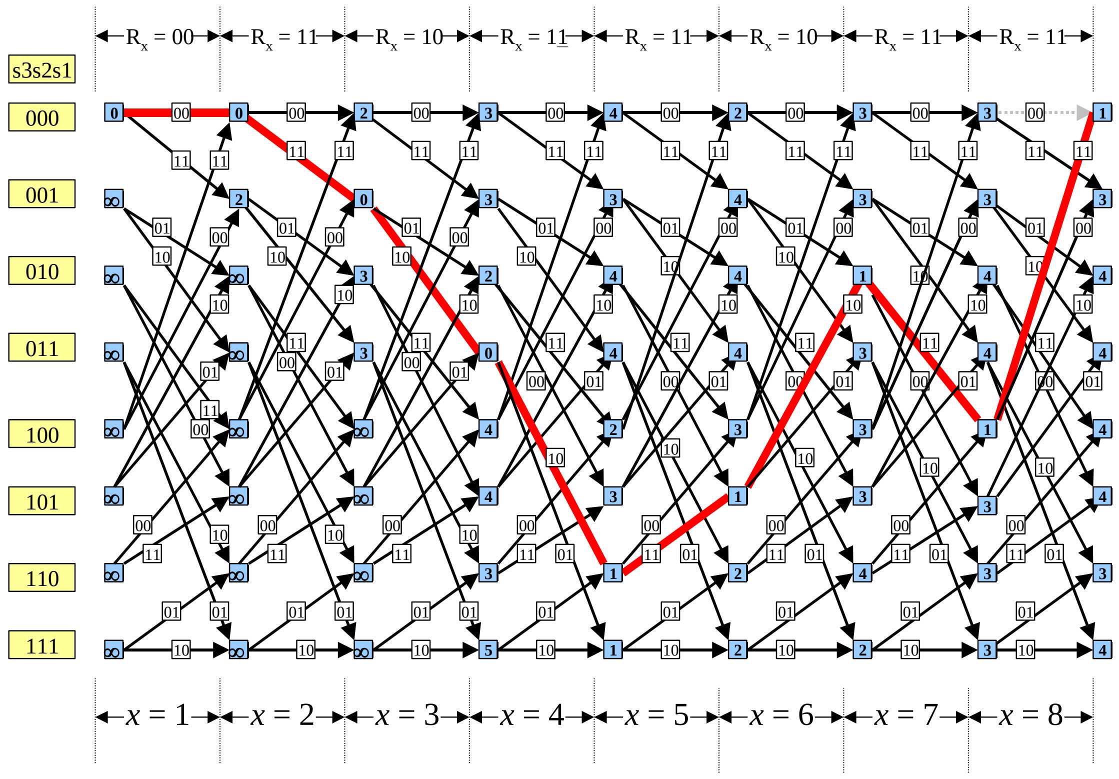


Figure 2-4 (2,1,3) Trellis showing final ML path.

The received sequence R, has (N + m) or eight decoding stages, as shown in figure 2-4. Each n-bit branch of the received sequence R is shown above the trellis, and the stage number is shown below the trellis.

In order to walk through this decoding example, the terms branch metric and path metric are introduced for making decoding decisions. Section 2.3.5 will cover the derivations of both metrics in detail. For this example, a simplified version of calculating branch metrics is introduced. The branch metric for one trellis branch is computed by finding the number of elements that differ between an n-bit trellis branch V_x and the current received n-bit symbol R_x, known as the Hamming distance between the two sequences R and V denoted $\delta(R, V)$, [14].

Example: Given the n-bit received symbol $R_x = [01]$ and an n-bit trellis branch $V_x = [10]$ the Hamming distance $\delta(R_x, V_x)$ is computed as

$$\delta([r_n, r_{n-1}, \ldots r_1]_x, [v_n, v_{n-1}, \ldots v_1]_x). = \delta([0,1], [1,0]) = 2$$

In this case R_x and V_x has two bit elements which differ, hence the Hamming distance is two.

Consider the trellis nodes Q,R, and S in figure 2-5, the path metric for trellis node S is computed as follows.

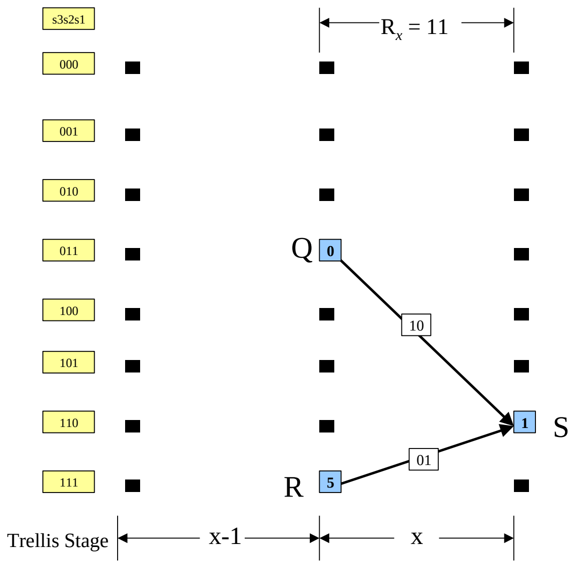


Figure 2-5 Computing an ML path metric in the VDA.

The trellis diagram in figure 2-5 corresponds to the (2,1,3) convolutional encoder in figure 2-2. Each trellis node has $2^k = 2$ paths entering each trellis node. Consider the two trellis branches QS and RS. The trellis branch label QS is labeled as 10 which corresponds to the output symbol V_x when the convolutional encoder changes from state 011 to state 101. Similarly trellis branch label RS is labeled as 01 which corresponds to the output symbol V_x when the convolutional encoder changes from state 111 to state 101. The trellis nodes Q and R have path metrics 0 and 5 respectively which were computed during the previous trellis stage. The computation of the path metric for trellis node S is now outlined as follows:

The received symbol R_x corresponding to this trellis stage and the convolutional coded symbols for trellis branches QS and RS are:

$$R_x = [r_n, r_{n-1}, \ldots r_1]_x = [r_2, r_1]_x = 11$$

$$V_x^{QS} = [v_n, v_{n-1}, \ldots v_1]_x = [v_2, v_1]_x = 10$$

$$V_x^{RS} = [v_n, v_{n-1}, \ldots v_1]_x = [v_2, v_1]_x = 01$$

The Hamming distance branch metrics, δ^{QS} and δ^{RS} are:

$$\delta^{QS} = \delta([r_n, r_{n-1}, \ldots r_1]_x, [v_n, v_{n-1}, \ldots v_1]_x)$$

$$\delta^{QS} = \delta([r_2, r_1]_x, [v_2, v_1]_x)$$

$$\delta^{QS} = \delta([1, 1]_x, [1, 0]_x)$$

$$\delta^{QS} = 1$$

Similarly, $\delta^{RS} = 1$

At trellis stage x, the xth partial path metric for a path passing through a trellis state is denoted as $A^x_{(s3s2s1)}$. For each trellis node, 2^k partial path metrics are computed as possible survivors. Each of the 2^k partial path metrics are compared and the surviving path is selected as the path with the minimum partial path metric. The compare and selecting function is denoted as follows.

$$A^x_{(s_3s_2s_1)} = \min\{[A^{x-1}_{(s_3s_2s_1)}]_{2^k-1}, [A^{x-1}_{(s_3s_2s_1)}]_{2^k-2}, \ldots, [A^{x-1}_{(s_3s_2s_1)}]_1\}$$

The subscripts (2^k-1, … ,0) denotes each of the 2^k incoming partial path metrics.

$$A^{x}_{(s_3s_2s_1)} = \min\{[A^{x-1}_{(s_3s_2s_1)}]_{2^k-1}, [A^{x-1}_{(s_3s_2s_1)}]_{2^k-2}, \ldots, [A^{x-1}_{(s_3s_2s_1)}]_1\}$$

$$A^{x}_{\mathrm{S}} = \min\{[A^{x-1}_{\mathrm{Q}} + \partial^{QS}]_2, [A^{x-1}_{\mathrm{R}} + \partial^{RS}]_1\}$$

$$A^{x}_{\mathrm{S}} = \min\{[0+1],[5+1]\}$$

$$A^{x}_{\mathrm{S}} = \min\{[1],[6]\}$$

$$A^{x}_{\mathrm{S}} = 1$$

The three step procedure outlined in section 2.3.1 is now used for walking through the decoding steps of the Viterbi algorithm.

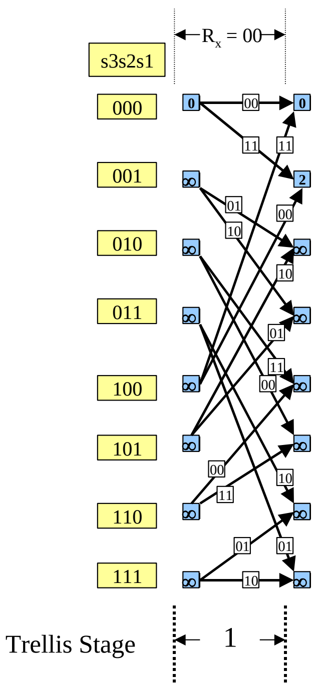
STEP 1

For this example, the initial state is state 000, therefore the path metric for state 000 is initialized to zero. The remaining seven path metrics are initialized to their maximum value denoted in this example as ∞. The stage pointer is then incremented by one.

STEP 2

This step is repeated for all eight stages of the decoding process. A total of sixteen path metrics are computed per trellis stage, two per trellis node. For each trellis node, the two path metrics computed are compared and the path with the minimum path metric is selected as the most likely path. This procedure is repeated for the remaining trellis nodes at stage 1.

At stage 1, the received symbol R_x is 00. A total of sixteen Hamming distances are computed between R_x and each of the trellis branch labels V_x. Each of the Hamming distances are added to the corresponding path metrics initialized at stage 0. The trellis at the end of stage 1 is shown in figure 2-6.



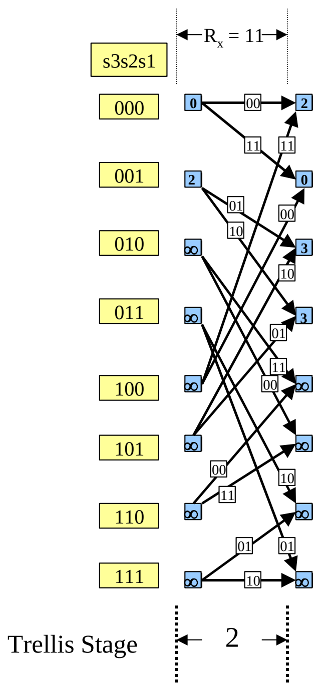
$$A^{1}_{000} = \min\{[\ 0 + 0], [\infty + 2]\} = 0$$
$$A^{1}_{001} = \min\{[\ 0 + 2], [\infty + 0]\} = 2$$
$$A^{1}_{010} = \min\{[\ \infty + 1], [\infty + 1]\} = \infty$$
$$A^{1}_{011} = \min\{[\ \infty + 1], [\infty + 1]\} = \infty$$
$$A^{1}_{100} = \min\{[\ \infty + 2], [\infty + 0]\} = \infty$$
$$A^{1}_{101} = \min\{[\ \infty + 0], [\infty + 2]\} = \infty$$
$$A^{1}_{110} = \min\{[\ \infty + 1], [\infty + 1]\} = \infty$$
$$A^{1}_{111} = \min\{[\ \infty + 1], [\infty + 1]\} = \infty$$

Figure 2-6 Path metric computation at stage 1.

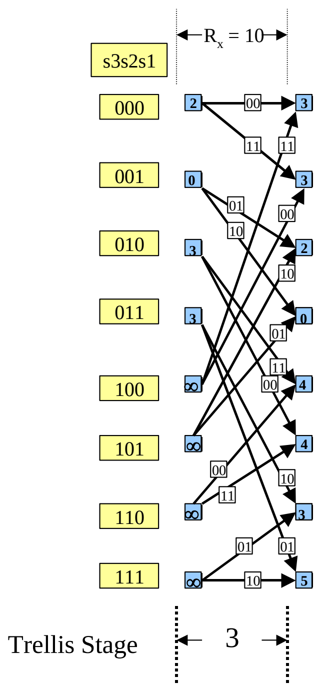
At stage 2, the received symbol R_x is 11. Again sixteen Hamming distances are computed between R_x and each of the trellis branch labels V_x. The Hamming distances are then added to the corresponding path metrics computed at stage 1. The trellis at the end of stage 2 is shown in figure 2-7.

This process of computing Hamming distances for the current received symbol R_x and each of the trellis branch symbols V_x is identical to stage 0, and stage 1. As the trellis stage pointer has not yet reached eight, the algorithm remains in step 2. The remaining stages follow the same process as illustrated by figure 2-7 through figure 2-10.

Note: In stage 4, the received symbol R_x has a single bit error, therefore the minimum path metric is no longer zero. The remaining figures show that the Viterbi algorithm can still make ML decisions in the presence of bit errors in the received sequence.



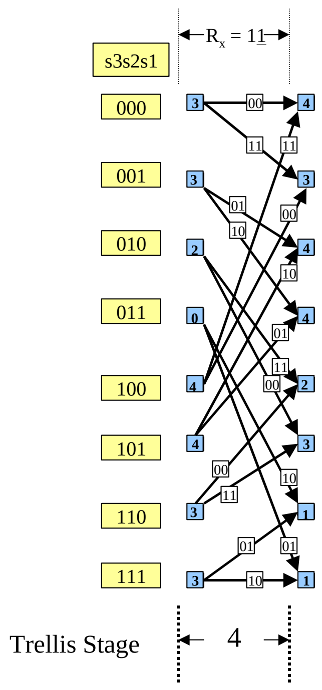
$$A^{2}_{000} = \min\{[\ 0+2],[\infty+0]\} = 2$$
$$A^{2}_{001} = \min\{[\ 0+0],[\infty+2]\} = 0$$
$$A^{2}_{010} = \min\{[\ 2+1],[\infty+1]\} = 3$$
$$A^{2}_{011} = \min\{[\ 2+1],[\infty+1]\} = 3$$
$$A^{2}_{100} = \min\{[\ \infty+0],[\infty+2]\} = \infty$$
$$A^{2}_{101} = \min\{[\ \infty+2],[\infty+0]\} = \infty$$
$$A^{2}_{110} = \min\{[\ \infty+1],[\infty+1]\} = \infty$$
$$A^{2}_{111} = \min\{[\ \infty+1],[\infty+1]\} = \infty$$



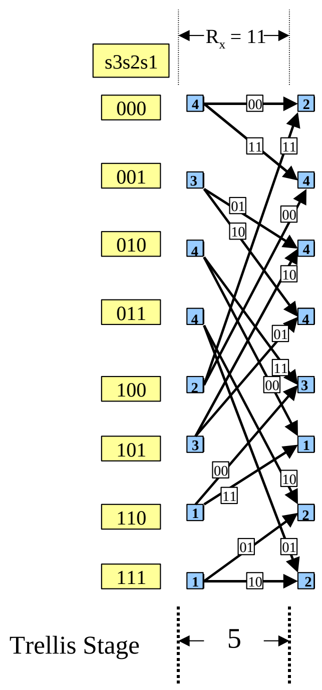
$$A^{3}_{000} = \min\{[\ 2+1],[\infty+1]\} = 3$$
$$A^{3}_{001} = \min\{[\ 2+1],[\infty+1]\} = 3$$
$$A^{3}_{010} = \min\{[\ 0+2],[\infty+0]\} = 2$$
$$A^{3}_{011} = \min\{[\ 0+0],[\infty+2]\} = 0$$
$$A^{3}_{100} = \min\{[\ 3+1],[\infty+1]\} = 4$$
$$A^{3}_{101} = \min\{[\ 3+1],[\infty+1]\} = 4$$
$$A^{3}_{110} = \min\{[\ 3+0],[\infty+2]\} = 3$$
$$A^{3}_{111} = \min\{[\ 3+2],[\infty+0]\} = 5$$

.

Figure 2-7 Path metric computation at stage 2, and 3

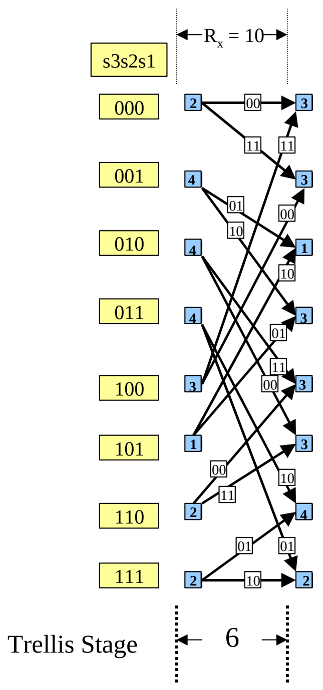


$$A^{4}_{000} = \min\{[\;3+2],[4+0]\} = 4$$
$$A^{4}_{001} = \min\{[\;3+0],[4+2]\} = 3$$
$$A^{4}_{010} = \min\{[\;3+1],[4+1]\} = 4$$
$$A^{4}_{011} = \min\{[\;3+1],[4+1]\} = 4$$
$$A^{4}_{100} = \min\{[\;2+0],[3+2]\} = 2$$
$$A^{4}_{101} = \min\{[\;2+2],[3+0]\} = 3$$
$$A^{4}_{110} = \min\{[\;0+1],[3+1]\} = 1$$
$$A^{4}_{111} = \min\{[\;0+1],[3+1]\} = 1$$

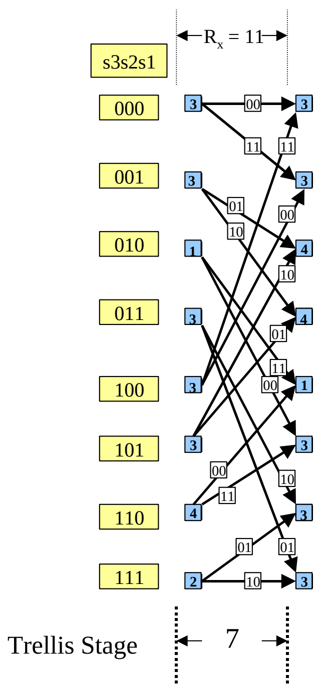


$$A^{5}_{000} = \min\{[\;4+2],[2+0]\} = 2$$
$$A^{5}_{001} = \min\{[\;4+0],[2+2]\} = 4$$
$$A^{5}_{010} = \min\{[\;3+1],[3+1]\} = 4$$
$$A^{5}_{011} = \min\{[\;3+1],[3+1]\} = 4$$
$$A^{5}_{100} = \min\{[\;4+0],[1+2]\} = 3$$
$$A^{5}_{101} = \min\{[\;4+2],[1+0]\} = 1$$
$$A^{5}_{110} = \min\{[\;4+1],[1+1]\} = 2$$
$$A^{5}_{111} = \min\{[\;4+1],[1+1]\} = 2$$

Figure 2-8 Path metric computation at stages 4, and 5.

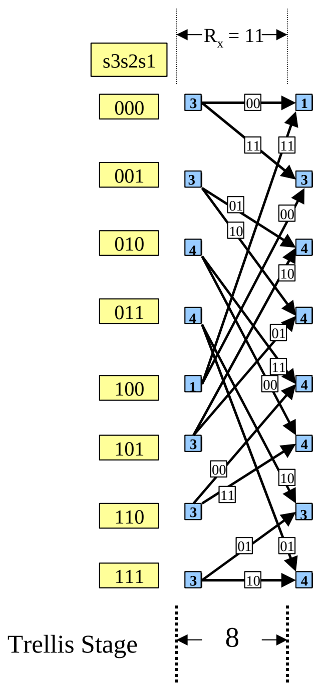


$$A^{6}_{000} = \min\{[\ 2+1],[3+1]\} = 3$$
$$A^{6}_{001} = \min\{[\ 2+1],[3+1]\} = 3$$
$$A^{6}_{010} = \min\{[\ 4+2],[1+0]\} = 1$$
$$A^{6}_{011} = \min\{[\ 4+0],[1+2]\} = 3$$
$$A^{6}_{100} = \min\{[\ 4+1],[2+1]\} = 3$$
$$A^{6}_{101} = \min\{[\ 4+1],[2+1]\} = 3$$
$$A^{6}_{110} = \min\{[\ 4+0],[2+2]\} = 4$$
$$A^{6}_{111} = \min\{[\ 4+2],[2+0]\} = 2$$



$$A^{7}_{000} = \min\{[\ 3+2],[3+0]\} = 3$$
$$A^{7}_{001} = \min\{[\ 3+0],[3+2]\} = 3$$
$$A^{7}_{010} = \min\{[\ 3+1],[3+1]\} = 4$$
$$A^{7}_{011} = \min\{[\ 3+1],[3+1]\} = 4$$
$$A^{7}_{100} = \min\{[\ 1+0],[4+2]\} = 1$$
$$A^{7}_{101} = \min\{[\ 1+2],[4+0]\} = 3$$
$$A^{7}_{110} = \min\{[\ 3+1],[2+1]\} = 3$$
$$A^{7}_{111} = \min\{[\ 3+1],[2+1]\} = 3$$

Figure 2-9 Path metric computation at stages 6, and 7.



$$A^{8}_{000} = \min\{[\ 3+2],[1+0]\} = 1$$

$$A^{8}_{001} = \min\{[\ 3+0],[1+2]\} = 3$$

$$A^{8}_{010} = \min\{[\ 3+1],[3+1]\} = 4$$

$$A^{8}_{011} = \min\{[\ 3+1],[3+1]\} = 4$$

$$A^{8}_{100} = \min\{[\ 4+0],[3+2]\} = 4$$

$$A^{8}_{101} = \min\{[\ 4+2],[3+0]\} = 3$$

$$A^{8}_{110} = \min\{[\ 4+1],[3+1]\} = 4$$

$$A^{8}_{111} = \min\{[\ 4+1],[3+1]\} = 4$$

Figure 2-10 Path metric computation at stage 8.

STEP 3

There are now eight possible surviving paths and corresponding path metrics. The final surviving path is selected by performing a comparison on each of the eight path metrics and selecting the path with the minimum path metric as the maximum likelihood surviving path. In this case state 000 has the minimum path metric of 1 and is thus selected as the ML path. All predecessor nodes on this path are selected to form this path.

2.3.3 Channel Models & ML Decoding using the VDA

The development of the ML Viterbi decoder is pursued in this section.

A typical communications link for a Viterbi decoder comprises of

- A convolutional encoder,
- A modulator,
- The channel,
- A demodulator,
- A convolutional decoder.

Typically a communications link contains many other blocks, but for the purpose of this section the communications link in figure 2-11 is sufficient.

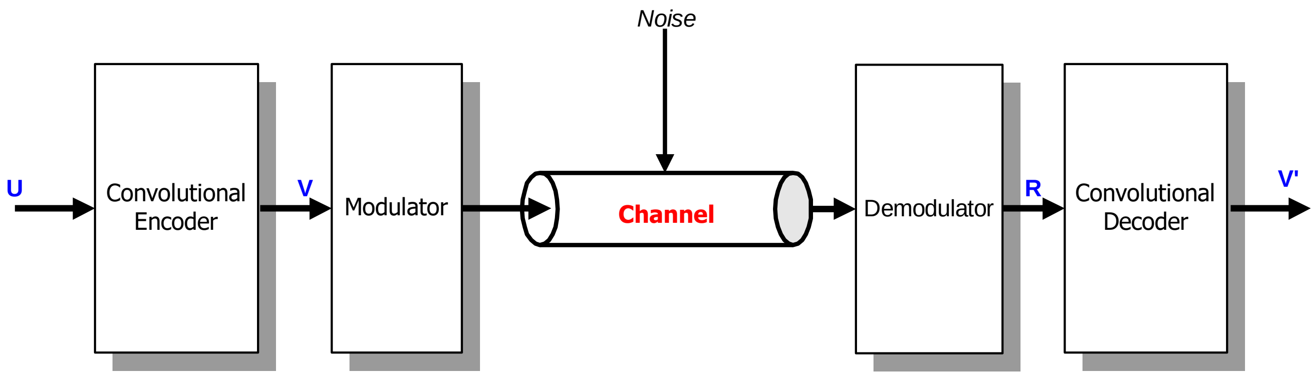


Figure 2-11 Communications Link.

The convolutional encoder generates the convolutional encoded sequence V from the input sequence U. The encoded sequence V is then modulated, where the symbols are transformed into signal waveforms. The modulation may be baseband (e.g. pulse waveforms) or bandpass (e.g. PSK) or FSK. The channel over which the waveform is transmitted is assumed to corrupt the signal with Gaussian noise, [14].

When the corrupted signal is received, it is first processed by the demodulator and then the convolutional decoder. The signal received by the demodulator can be described as $r(t) = s_i(t) + n(t)$, where $n(t)$ is a zero mean Gaussian noise process, [14]. The detection of $r(t)$ is in two steps. In the first step, the received waveform is reduced to a single number, $r(T) = a_i + n_0$, where a_i is the signal component of $r(T)$ and n_0 is the noise component. The noise component, n_0, is a zero mean Gaussian random variable, and thus is a Gaussian random variable with a mean of either a_1 or a_2 depending on whether a binary 1 or binary 0 was sent. In the second step of the detection process a decision is made as to which signal was transmitted, on the basis of comparing $r(T)$ to a threshold. The conditional probabilities of $r(T)$, $p(r|s_1)$, and $p(r|s_2)$ are shown in figure 2-12, labeled likelihood of s_1 and likelihood of s_2.

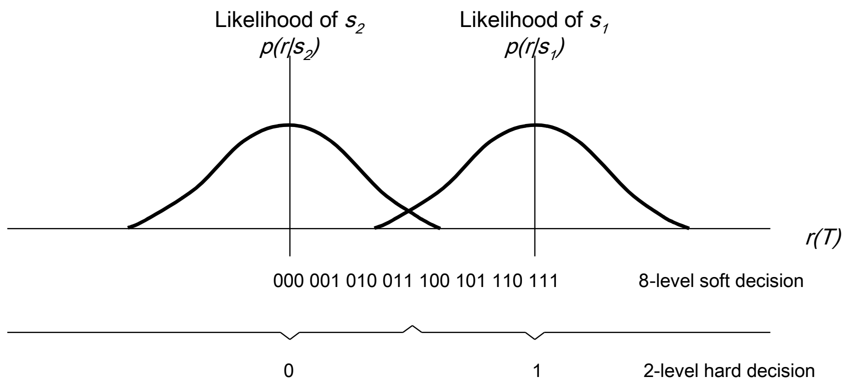


Figure 2-12 Hard and Soft Decoding Decisions.

The demodulator in figure 2-11, converts the set of time-ordered random variables, $\{r(T)\}$, into a code sequence R and passes it on to the decoder. The demodulator output can be configured in a variety of ways. It can be implemented to make a hard decision as to whether $r(T)$ represents a zero or a one. In this case the demodulator is quantized to two levels, zero and one, and fed into the decoder. The convolutional decoder then performs hard decision decoding, [14]. The demodulator can also be configured to feed the convolutional decoder with a quantized value of $r(T)$ greater than two levels, or with an unquantized or analog value $r(T)$. Such an implementation furnishes the convolutional decoder with more information than is provided in the hard decision case. When the quantization level output of the demodulator is greater than two, the decoding is called soft decision decoding, [14]. When the demodulator sends a hard binary decision to the decoder, it sends it a single binary symbol. When the demodulator sends a soft binary decision, quantized to eight levels, it sends the decoder a 3-bit word describing an interval along *r(T)*. In effect, sending such a 3-bit word in place of a single binary symbol is equivalent of sending the decoder a measure of confidence along with the encoded symbol.

For a Gaussian channel, eight-level quantization results in a performance improvement of approximately 2dB in required signal-to-noise ratio compared to two level quantization. This means that eight-level soft decision decoding can provide the same probability of bit error as that of hard decision decoding, but requires 2dB less Eb/N0 for the same performance. Analogue (or infinite level quantization) results in a 2.2dB performance improvement over two level quantization; therefore eight-level quantization results in a loss of approximately 0.2 dB compared to infinitely fine quantization. For this reason, quantization to more than eight levels can yield little performance improvement, [14]. For eight level quantized soft decision decoding 3 bits are used to describe each symbol, where as only 1 bit can describe a symbol in hard decision decoding. Therefore three times the amount of data must be handled during the decoding process. Convolutional decoding with the Viterbi algorithm can operate on both hard and soft decisions, where the soft decision case presents only a trivial increase in computation, [14].

2.3.4 Binary Symmetric Channel

A binary symmetric channel (BSC) is a discrete memoryless channel, that has a binary input and output alphabets and symmetric transition probabilities, [14]. It can be described by the transition probabilities as illustrated in figure 2-13.

$$P(0|1) = P(1|0) = p$$

$$P(1|1) = P(0|0) = 1-p$$

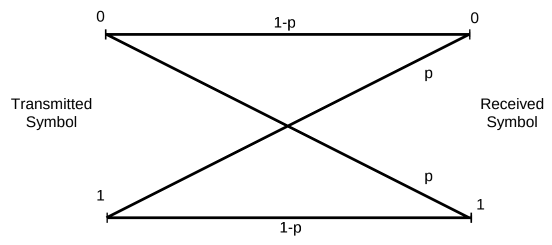


Figure 2-13 BSC model.

The probability that an output symbol will differ from the input symbol is *p*, and the probability that an output symbol will be identical is (*1-p*), [14]. The BSC is an example of a hard decision channel, which means that, even though continuous-valued signals may be received by the demodulator, a BSC allows only hard decisions such that each demodulator output symbol R_x consists of two binary values.

Let U be an input sequence to a convolutional encoder, which produces the encoded sequence V to be transmitted over a BSC with symbol error probability *p*. Let R be the corresponding received sequence from the demodulator that is input into the convolutional decoder. A maximum likelihood decoder chooses the decoded sequence U' which maximizes the likelihood *P*(R|V), [14]. For a BSC , this is equivalent to choosing the decoded sequence U' that is closest in Hamming distance to R, [14]. Thus Hamming distance is an appropriate metric to describe the distance or closeness of fit between U and R. From all the possible decoder transmitted sequences, V, the decoder chooses U' sequence for which the distance to R is minimum.

A commonly used channel modulation technique is BPSK modulation. For a BSC using BPSK modulation, the channel symbol error probability *p* is found using the following equation: $p = Q\sqrt{\frac{2E_S}{N_0}}$, where Q(x) is the complimentary error function or co-error function, [14]. Another expression for transition probability *p* using the complimentary error function, *erfc*, is as follows:

$$p = Q\sqrt{\frac{2E_b}{N_0}} = \frac{1}{2} erfc\left(\frac{\sqrt{2\frac{E_b}{N_0}}}{\sqrt{2}} \right)$$

In the case of a BSC which has added white Gaussian noise, (AWGN), the channel symbol error probability is equal to the bit error probability, [14]. The resulting bit error probability is given by: $p = Q\sqrt{\frac{2E_b}{N_0}}$

The parameter, E_b/N_0, can be expressed as the ratio of average signal power to the average noise power (SNR).

2.3.5 ML Decoding using the VDA

The ML decoder selects by definition the estimate V' that maximizes the probability *p*(R|V'). When applying the ML criterion to the convolutional decoding problem, there are typically a multitude of possible sequences that might have been transmitted. To be specific, an N-bit sequence is a member of a set of 2^N possible sequences. Therefore, in the ML context, we can say that the decoder chooses a particular sequence V' as the transmitted sequence if the likelihood *p*(R|V') is greater than the likelihood of all other possible transmitted sequences.

The maximum a posteriori (MAP) decoder selects the estimate that maximizes *p*(V'|R), [7]. If the distribution of the bits in the input sequence U is uniform, then the two decoders are identical; in general, the decoders can be related by Bayes' rule, [7].

$$p(R \mid V')\, p(V') = p(V' \mid R)\, p(R). \qquad \text{(Equation 2-1)}$$

In any stage *x*, a convolutional encoder takes the k-bit input symbol $U_x = (u_k, u_{k-1}, \ldots u_1)_x$, and produces an n-bit output symbol $V_x = (v_n, v_{n-1}, \ldots v_1)_x$. The input sequence U, consists of N k-bit symbols as follows:

$$U = ([u_k^0, u_{k-1}^0, \ldots, u_1^0],\ [u_k^1, u_{k-1}^1, \ldots, u_1^1],\ \ldots, [u_k^{N-1}, u_{k-1}^{N-1}, \ldots, u_1^{N-1})$$

The encoded sequence V will consist of (N +m) n-bit symbols. There is one n-bit symbol for each k-bit input symbol plus m additional n-bit symbols, where m is the maximal length shift register.

$$V = ([v_n^0, v_{n-1}^0, \ldots, v_1^0],\ [v_n^1, v_{n-1}^1, \ldots, v_1^1],\ \ldots, [v_n^{N+m-1}, v_{n-1}^{N+m-1}, \ldots, v_1^{N+m-1}])$$

A noise corrupted version R of the transmitted sequence arrives at the receiver, where the decoder generates a ML estimate V' of the transmitted sequence. R and V' have the following form:

$$R = ([r_n^0, r_{n-1}^0, \ldots, r_1^0],\ [r_n^1, r_{n-1}^1, \ldots, r_1^1],\ \ldots, [r_n^{N+m-1}, r_{n-1}^{N+m-1}, \ldots, r_1^{N+m-1}])$$

$$V' = ([v'^0_n, v'^0_{n-1}, \ldots, v'^0_1],\ [v'^1_n, v'^1_{n-1}, \ldots, v'^1_1],\ \ldots, [v'^{N+m-1}_n, v'^{N+m-1}_{n-1}, \ldots, v'^{N+m-1}_1])$$

To facilitate the analysis for a ML decoder, the channel is assumed to be memoryless, i.e., that the noise process affecting a given symbol in a received sequence R is independent of the noise process affecting all of the other received symbols, [7]. Since the probability of joint, independent events is simply the product of the probabilities of the individual events, [14], it follows that:

$$p(R \mid V') = \prod_{i=0}^{N+m-1} [\, p(r_n^i \mid v'^i_n)\, p(r_{n-1}^i \mid v'^i_{n-1}) \ldots p(r_1^i \mid v'^i_1)]$$

$$p(R \mid V') = \prod_{i=0}^{N+m-1} \left(\prod_{j=n}^{1} p(r_j^i \mid v'^i_j) \right) \qquad \text{(Equation 2-2)}$$

There are two sets of product indices, one i corresponding to the stage instances, and the other j corresponding to the bits within each symbol.

Eq. (2-2), is sometimes called the likelihood function for V', [7]. Generally it is computationally more convenient to use the logarithm of the likelihood function since this permits summation, instead of the multiplication of terms. We are able to use this transformation because the logarithm is a monotonically increasing function and thus will not alter the final result. By taking the logarithm of each side of Eq. (2-2), the log likelihood function is obtained, [7].

$$\log\ p(R \mid V') = \sum_{i=0}^{N+m-1} \left(\sum_{j=n}^{1} \log\ p(r_j^i \mid v'^i_j) \right) \text{(Equation 2-3)}$$

In hardware implementations of the Viterbi decoder, the summands in Eq. (2-3) are usually converted to a more easily manipulated form called the bit metrics.

Let $A(\mathrm{r}_j^i \mid \mathrm{v'}_j^i)$) be the bit metric between the received symbol bit r and the a trellis branch symbol bit v

$$A(\mathrm{r}_j^i \mid \mathrm{v'}_j^i) = a[\log\ p(\mathrm{r}_j^i \mid \mathrm{v'}_j^i) + b]) \text{ (Equation 2-4)}$$

The values of *a* and *b* are chosen such that bit metrics are small positive integers that can be easily manipulated by digital logic circuits, [7]. The path metric for a sequence V', is then computed as follows:

$$A(\mathrm{R} \mid \mathrm{V'}) = \sum_{i=0}^{N+m-1} \left(\sum_{j=n}^{1} A(\mathrm{r}_j^i \mid \mathrm{v'}_j^i) \right) \text{ (Equation 2-5)}$$

If *a* in Eq. (2-4) is positive and real, while *b* is simply real, then the sequence V' that maximizes *p*(R | V') also maximizes *A*(R | V'), [7]. The value of *a* may also be chosen to be negative, in which case V' is selected so that it minimizes A(R | V').

At times it is useful for us to focus on the contribution made to the path metric by a single n-bit branch of R and V' in the trellis. The *k*th branch metric for a sequence V' is defined as the sum of the bit metrics for the *k*th block of R given V', [7].

$$A(\mathrm{r}^k \mid \mathrm{v'}^k) = \left(\sum_{j=n}^{1} A(\mathrm{r}_j^k \mid \mathrm{v'}_j^k) \right) \text{ (Equation 2-6)}$$

The *k*th partial path metric for a path is obtained by summing the branch metrics for the first *k* branches that the path traverses, [7].

$$A^k(\mathrm{r} \mid \mathrm{v'}) = \sum_{i=0}^{k-1} A(\mathrm{r}^i \mid \mathrm{v'}^i)$$

$$= \sum_{i=0}^{k-1} \left(\sum_{j=n}^{1} A(\mathrm{r}_j^i \mid \mathrm{v'}_j^i) \right) \text{(Equation 2-7)}$$

2.3.6 Bit Metric Selection for a BSC Channel

To implement a Viterbi decoder over a BSC channel, the bit metrics need to be scaled for digital hardware. We can assume that the crossover probability *p* for the BSC is less than or equal to ½ for if it were greater we could simply re-label the zeros and ones at the receiving end, converting the effective value or *p* to a value less than ½, [7] The values of a and b in Eq. (2-4) can be selected such that the bit metrics are independent of the value of the crossover probability *p*.

For a BSC channel the values of a and b can be selected according to Eq. (2-8).

$$a = \left(\frac{1}{\log_2 p - \log_2(1-p)} \right), b = \left(-\log_2(1-p) \right) \text{ (Equation 2-8)}$$

The expression for bit metrics over a BSC is thus redefined as follows

$$A(\mathrm{r}_\mathrm{j}^\mathrm{i} \mid \mathrm{v'}_\mathrm{j}^\mathrm{i}) = \left(\frac{1}{\log_2 p - \log_2(1-p)} \right) \left(\log\ p(\mathrm{r}_\mathrm{j}^\mathrm{i} \mid \mathrm{v'}_\mathrm{j}^\mathrm{i}) + \left(-\log_2(1-p) \right) \right) \text{ (Equation 2-9)}$$

Thus for the BSC case, the path metric for a sequence V given the received sequence R is simply the Hamming distance δ(R, V). The surviving paths are those paths with the minimum partial path metric at each node.

2.4 Structure of Practical Viterbi Decoders

The procedure for convolutional decoding using the Viterbi Algorithm is outlined in the previous sections. However a practical Viterbi decoder differs from the ideal image thus far considered in that it must contend with several constraints imposed by the real world, [7]. For example:

- Arbitrarily long decoding delays cannot be tolerated in most applications. The decoder therefore outputs decoded information bits before the entire encoded sequence has been received.

- If the decoder is to be implemented using digital logic, incoming analogue signals must be quantized by an analogue to digital converter.

- The decoder will frequently be brought on line in the middle of a transmission, and will thus will not know where one n-bit block ends and the next one begins. Some form of block synchronization is necessary.

Figure 2-14 shows a block diagram for a Viterbi decoder that satisfies the above constraints, [7]. For completeness, a few parts of the receiver front end have been included as well.

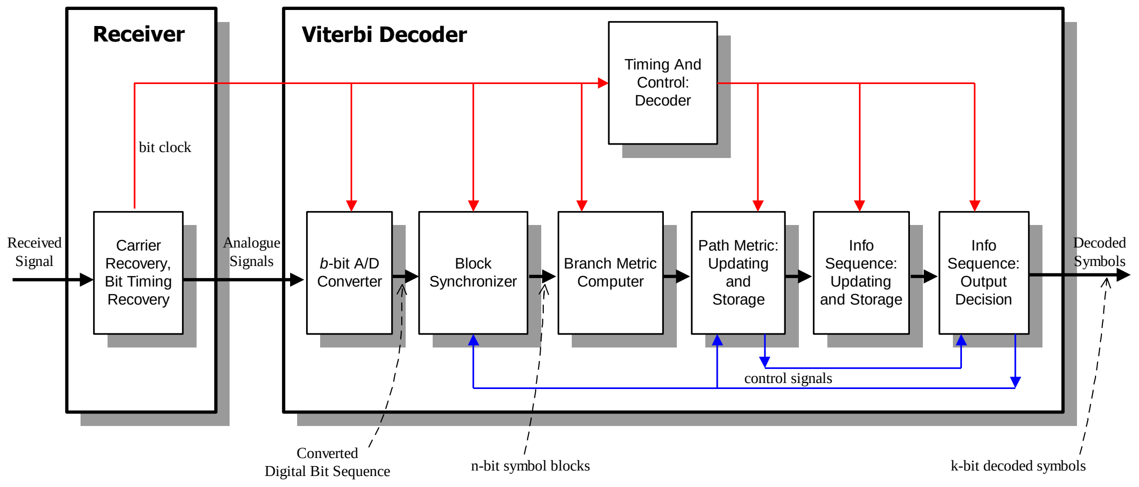

Receiver
Viterbi Decoder
Timing And Control: Decoder
bit clock
Received Signal
Carrier Recovery, Bit Timing Recovery
Analogue Signals
b-bit A/D Converter
Block Synchronizer
Branch Metric Computer
Path Metric: Updating and Storage
Info Sequence: Updating and Storage
Info Sequence: Output Decision
Decoded Symbols
control signals
Converted Digital Bit Sequence
n-bit symbol blocks
k-bit decoded symbols

Figure 2-14 Structure of a practical Viterbi decoder.

2.4.1 Quantization (*b*-bit A/D Converter)

From figure 2-18, the b-bit A/D Converter can be configured to convert the incoming analogue signal to a *b*-bit digital sequence.

If $b = 1$, then the A/D converter is quantized to two levels, zero and one, and fed into the decoder. The convolutional decoder then performs hard decision decoding, [14].

If b >1, the decoder receives a digital sequence with a quantized value greater than two levels, or with an unquantized or analog value *r(T)*. Such an implementation furnishes the convolutional decoder with more information than is provided in the hard decision case. When the quantization level output of the demodulator is greater than two, the decoding is called soft decision decoding, [14].

As outlined in section 2.3.3, quantization to more than eight levels can yield little performance improvement, [14]. For eight level quantized soft decision decoding 3 bits are used to describe each symbol, where as only 1 bit can describe a symbol in hard decision decoding. Therefore three times the amount of data must be handled during the decoding process. Convolutional decoding with the Viterbi algorithm can operate on both hard and soft decisions, where the soft decision case presents only a trivial increase in computation, [14].

2.4.2 Block Synchronization

In order to compute the branch metrics at any given point in time, the Viterbi decoder must be able to segment the received sequence into n-bit blocks, each block corresponding to a stage in the trellis. If the received bits are not properly divided up, the results are disastrous. Fortunately the nature of the disaster provides its own resolution. There are n distinct positions in the received data stream at which the decoder can draw block boundaries. Since n is usually small (n <= 4), a systematic search of the positions can be efficiently conducted. If a guess at block synchronization is incorrect , one or two partial path metrics for the survivors tend to be close together and very high, i.e. there is no dominant path. This situation is easily detected and used as an "out-of-sync" detector A simple circuit can designed that compares the minimum partial path metric to a fixed threshold after say 5 constraint lengths of the received sequence have been decoded.

The block synchronizer can also help resolve polarity ambiguity in the receiver. In many digital receivers, phase-locked loops are used to recover the carrier phase and have several stable equilibrium points. Each point assigns ones and zeros to the received signals in a different manner. If the linear convolutional encoded sequence in use does not have all-ones as part of that sequence, then given the encoded sequence V, we know that that its complement (V+1) cannot be a valid sequence, [7]. It follows that if the receiver switches its zeros and ones, a valid encoded sequence becomes invalid, and the Viterbi decoder acts as if it is out of synchronization, [7].

2.4.3 The Branch Metric Computer

The branch metric computer is typically based on a look up table containing the various bit metrics. The computer looks up n bit metrics associated with each branch and sums them to obtain the branch metric, [7]. The result is passed along to the path metric updating and storage unit. The design of the branch metric computer is made much simpler if the channel is symmetric, [7]. The same look up function is performed n times per branch for each of the $2^{(M+k)}$ branches per stage in the trellis.

2.4.4 Path Metric Updating and Storage

The path metric updating and storage unit takes the computed branch metrics and computes the partial path metrics at each node in the trellis. The surviving path at each node is identified, and the information-sequence updating and storage unit notified accordingly.

These functions are performed most efficiently by breaking the trellis up into a number of identical elements. For example the trellis diagram for (n,1,m) convolutional codes can be broken up into elements containing a pair of origin and destination states and four interconnecting branches, this is typically called a butterfly structure, as shown in figure 2-15.

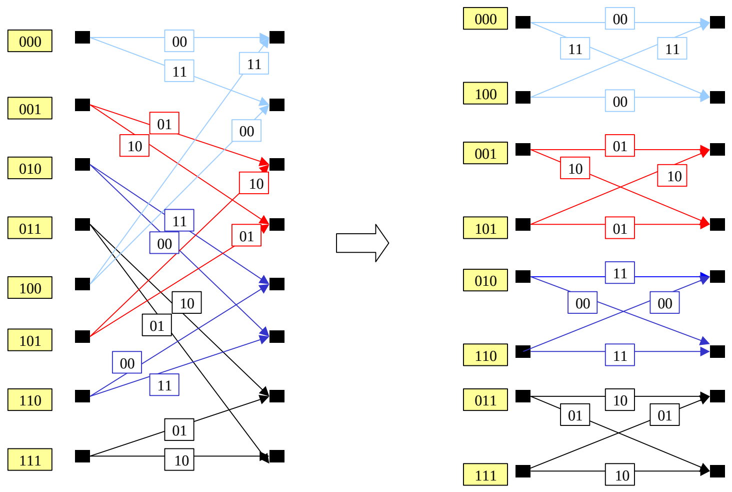


Figure 2-15 Butterfly structure of (2,1,3) trellis.

Since the entire trellis is multiple images of the same simple element, a computation circuit can be designed and used repeatedly in a decoder.

Figure 2-16 shows such a circuit for (n,1,m) codes. Since the basic functions performed by this circuit are adding comparing and selecting the circuit is commonly called an ACS block.

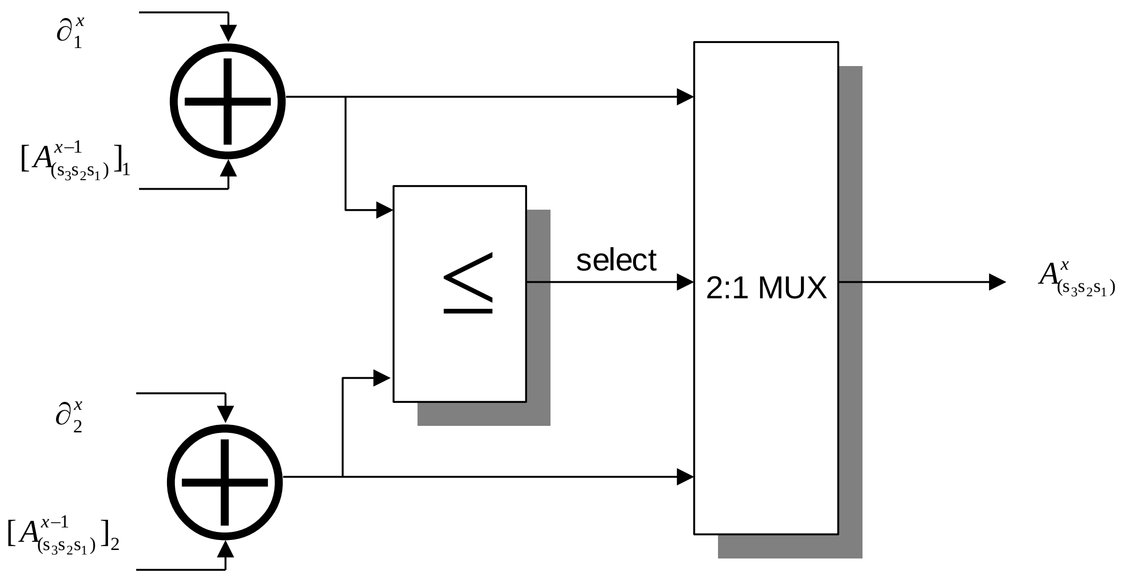


Figure 2-16 ACS circuit for (n,1,m) codes.

The ACS circuit in figure 2-16 is used twice to perform the functions associated with one butterfly trellis element in figure 2-15.

There are a number of design strategies for ACS computation within a Viterbi decoder.

An ACS operation is defined as the process carried out by an ACS circuit to select the most likely hypothesis for a surviving path on one trellis node. Given an (n,k,m) convolutional code with total memory M, N2^M ACS operations are required to decode the received sequence R of length N, [7]. One such design strategy could use one such ACS circuit, which is time shared between all trellis nodes to compute the path metric. This strategy results a low cost silicon efficient circuit, however the downside is that the decoding operations are considerably slower due to time sharing the ACS operations. At the other extreme, a separate ACS block can be dedicated to every node in the trellis, resulting in a fast, massively parallel implementation, [7,8]. A reasonable compromise between the two schemes is to assign 2^M ACS circuits to a decoder, one for each trellis node in a decoding stage. This scheme provides a 2^M improvement in ACS decoding operations over the single ACS scheme, with the same increase in silicon real estate, however the increase in decoding speed is usually desirable. The ACS block is thus a primary functional unit of the Viterbi decoder; its design and utilization are one of the principal factors in determining its complexity and speed.

For (n, k, m) decoders with k>1, a single trellis element contains two sets of 2^k states interconnected by 2^k branches. For example, for the (3,2,2) convolutional code, a single trellis node has 2^k or four incoming branches, and a four way comparison is needed to determine the surviving path. Figure 2-21 shows an ACS circuit for the (3,2,2) convolutional.

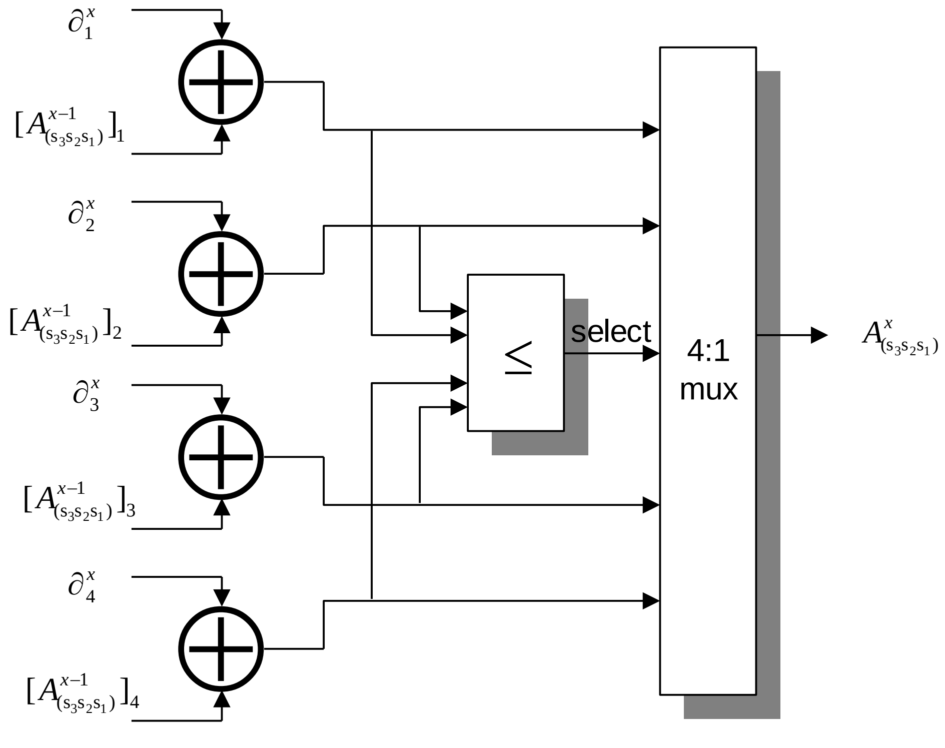

∂_1^x
$[A_{(s_3s_2s_1)}^{x-1}]_1$
∂_2^x
$[A_{(s_3s_2s_1)}^{x-1}]_2$
∂_3^x
$[A_{(s_3s_2s_1)}^{x-1}]_3$
∂_4^x
$[A_{(s_3s_2s_1)}^{x-1}]_4$
$\leq$
select
4:1
mux
$A_{(s_3s_2s_1)}^x$

Figure 2-21 ACS circuit for (3,2,2) convolutional code.

The complexity of an ACS circuit increases exponentially with k, [7]. For example, given an (n, k, m) code with say k = 3, the ACS circuit has 2^3 = eight branches entering each node and an 8-way comparison to select the surviving path. Increasing k to four results in an ACS circuit has 2^4 = sixteen branches entering each node and a 16-way comparison to select the surviving path, and so on.

2.4.5 Information Sequence Updating and Storage

The information sequence and storage unit is responsible for keeping track of the information bits associated with the surviving paths designated by the path metric updating and storage unit. There are two basic design approaches : register exchange and traceback, [7]. In both techniques a shift register is associated with every trellis node throughout the decoding operation. The number of bits that a register must be capable of storing is a function of the decoding depth, typically called the truncation length, T. Section 3 provides a detailed overview on this topic.

2.4.6 Information Sequence Output Decisions

This block is used to release decoded symbols. If a truncation length T is assumed, then at a given time *t* the decoder needs to release an n-bit symbol decoded at time (*t*-T). This raises a problem, for if at time *t* the entire received sequence has not yet been processed, there are still 2^M surviving paths from which to choose. There are three basic design solutions, [7].

- ❑ Output the decoded symbol on a randomly selected surviving path.
- ❑ Output the decoded symbol on the surviving path with the minimum partial path metric.
- ❑ Output the decoded symbol that occurs most frequently at time (*t*-T) on the 2^M surviving paths.

The first choice is simplest to implement but offers the worst performance. The second choice requires that 2^M partial path metrics be sorted to determine the minimum metric at each stage during the decoding operation, [7].

The difference in performance offered between the three choices is actually quite small if the truncation length T is sufficiently large, [7]. As the 2^M surviving paths are traced back through the trellis they tend to merge. Eventually a point is reached beyond which the 2^M survivors coincide. Clearly the symbols from this point back to the beginning of the trellis will be associated with the maximum likelihood sequence.

It also clear that the choice of output decision algorithm at time *t* is moot if all of the paths have merged at time (*t*-T), [7].

A decoded sequence error that occurs as a result of the release of the decoded symbols before the entire received sequence has been decode is called a truncation error, [7]. However, research has shown that the probability of truncation error decreases exponentially with truncation length T. Typically the probability of truncation errors is negligible if the truncation length is greater than or equal to five times the constraint length of the convolutional code, i.e. (T>= 5m), [2].

2.4.7 Initializing the Decoder

Digital receivers generally need a small amount of time to acquire an incoming signal. This is attributed to carrier-phase and bit timing synchronization. The Viterbi decoder introduces an additional requirement for block synchronization. By the time all of the preliminary operations have been concluded, the decoder has almost certainly missed the first few symbols of the transmitted code word. The decoder must thus begin its operation in midstream. Surprisingly this has virtually no impact on the decoders performance beyond the first few constraint lengths of the received sequence. Assume that the all-zero sequence has been transmitted. Since the decoder is beginning operation in midstream, branch metrics are computed for branches leaving all 2^M states at decoder time $t = 0$. An initialization error is defined as any error resulting from the all-zero path failing to survive a comparison to a non-zero path that starts at a non-zero state at time $t = 0$ and has not previously merged with the all zero path. The problem paths at time *t* are thus those non-zero paths of length greater than or equal to *t*.

The initialization problem is the dual of the truncation problem. It follows that at time (t = T) and beyond, the probability of initialization error is negligible. As a result no special initialization procedure is necessary for the Viterbi decoder in most applications, [14].

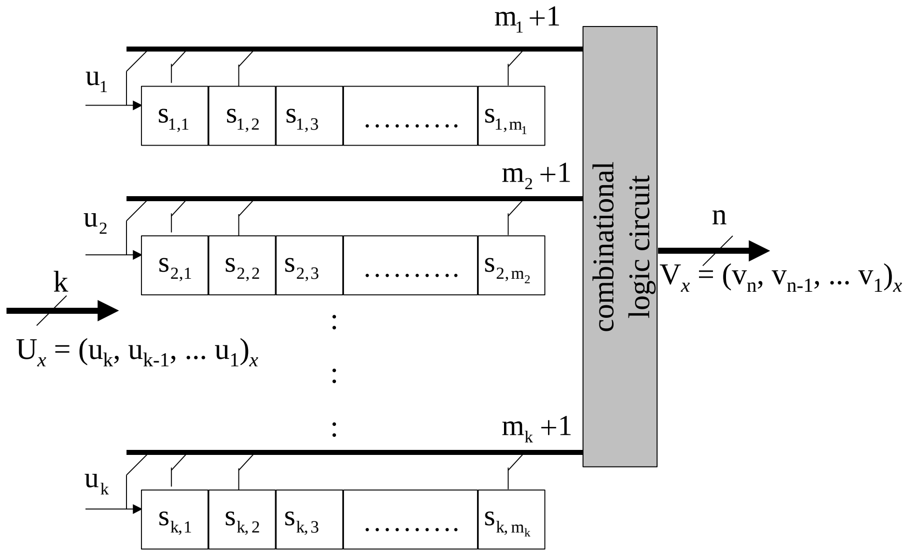
3. Survivor Path Memory Design

All the necessary blocks common to all practical Viterbi decoders were discussed in section 2.4. In practice there are a number architectural variations available. Typically this consists of performance enhancements to the path metric updating and storage unit and the information sequence updating and storage unit. Of key importance is the design of the information sequence updating and storage, or more commonly known as the survivor path memory unit. This section provides an analysis of all path memory architectures.

3.1 Trellis Branch Labeling

Section 2.1.3 introduced the definition of a trellis stage as an instance of the encoding process, i.e. the encoders memory state, its input and its output, which lasts for the duration of one input symbol period, [4]. In a trellis, a path has two ends, left and right, [3]. A path is defined by its direction. A forward path is a path starting from its left end. A backward path is a path starting from its right end. There are 2^M possible encoder states in total. Any node (state) on a trellis path is connected directly to 2^k other nodes (states) both on its input and on its output.

A trellis state transition from state S_x to state S_{x+1} occurs when the k bits of input symbol U_x are clocked into the encoder, the contents of all the registers have moved one memory element to the right, and new symbols, U_{x+1} and V_{x+1} are present at the input and output respectively, [4], as shown in figure 3-1.



m_1+1
u_1
$s_{1,1}$ $s_{1,2}$ $s_{1,3}$ ………. s_{1,m_1}
m_2+1
u_2
$s_{2,1}$ $s_{2,2}$ $s_{2,3}$ ………. s_{2,m_2}
k
$U_x = (u_k, u_{k-1}, \ldots u_1)_x$
m_k+1
u_k
$s_{k,1}$ $s_{k,2}$ $s_{k,3}$ ………. s_{k,m_k}
combinational logic circuit
n
$V_x = (v_n, v_{n-1}, \ldots v_1)_x$

Figure 3-1 (n, k, m) convolutional encoder.

In any stage *x*, the n bits of the output symbol, $V_x = (v_n, v_{n-1}, \ldots v_1)_x$, are given by fixed combinations of selected bits of the input symbol, $U_x = (u_k, u_{k-1}, \ldots u_1)_x$ and selected elements of the encoder memory, i.e. the elements of previous input symbols. The state of this memory at stage *x* is denoted S_x

$$S_x = (s_{j,i})_x, \quad \text{where } j = k,\ldots 1 \text{ and } i = m_j,\ldots 1.$$

Or in expanded form, the state, S_x of an (n, k, m) convolutional encoder can be expressed as follows:

$$S_x = \begin{bmatrix} (s_{1,1}, s_{1,2}, s_{1,3}, \ldots s_{1,m_1})_x \\ (s_{2,1}, s_{2,2}, s_{2,3}, \ldots s_{2,m_2})_x \\ : \\ : \\ (s_{k,1}, s_{k,2}, s_{k,3}, \ldots s_{k,m_k})_x \end{bmatrix}$$

The state transition information can be presented in state tables, state diagrams, or in trellis diagrams. In the latter, branch labels are used to store information about encoder state transitions and thus decode the surviving paths through the code's trellis in implementations of the Viterbi algorithm. Labels can be forward or backward, the former being identical to the k-tuple input in the same encoder stage, and the latter to the k-tuple shifted out of the encoder on transition to the next stage. Thus at stage *x* forward, (F_x), and backward, (B_x) branch label information may be identified as:

$$F_x = f_{k,x}, f_{k-1,x}, \ldots, f_{1,x} = (f_k, f_{k-1}, \ldots, f_1)_x = (u_k, u_{k-1}, \ldots, u_1)_x$$

$$B_x = b_{k,x}, b_{k-1,x}, \ldots, b_{1,x} = (b_k, b_{k-1}, \ldots, b_1)_x = (s_{k,m_1}, s_{k-1,m_{k-1}}, \ldots, s_{1,m_1})_x$$

The information sequence and storage unit introduced in section 2.4.5, more commonly known as the path memory, is responsible for keeping track of the information bits associated with the surviving paths designated by the path metric updating and storage unit. The information bits can be forward labels or backward labels. A forward label Viterbi decoder utilizes a path memory that stores k-bit forward labels at every trellis stage. Similarly, a backward label Viterbi decoder utilizes a path memory that stores k-bit backward labels at every trellis stage.

3.2 Path Memory Architectures

There are two basic design approaches : register exchange and traceback, [7]. In both techniques a shift register is associated with every trellis node throughout the decoding operation. The number of bits that a register must be capable of storing is a function of the truncation length T, [2]. Given a fixed truncation length T, the registers need only contain kT bits; once the register are full, the oldest bits in the register are output as new bits are entered, [7]. The registers are thus FIFOs of fixed length. The information sequence output device is responsible for deciding which of the 2^M register outputs is selected as the decoder symbol output at each branch, [7].

3.2.1 Register Exchange Path Memory

The Viterbi decoding algorithm procedure outlined in section 2.3.1 covered the non-implementation case. The procedure for performing the VDA when using the register exchange path memory is now outlined. Like the general VDA technique, the register exchange technique maintains a memory with two entries to perform decoding operations. One is the path metric memory, and operates as described in section 2.3.4. The other entry is the path memory and is implemented using the register exchange (RE) technique. The entries in the RE path memory contain 2^M surviving paths, and represent the input symbols shifted into the encoder, i.e. forward labels. For every write operation to path memory, every state path memory register must be read modified and rewritten.

There are 2^M state path memory registers, one for each surviving path. Each state path memory register has kT bits. This path memory register can be organized in a number of ways, however for this project each path memory register is organized into k rows and T columns, where T is the truncation length. Each register is organized as follows:

$$\begin{bmatrix} (re_{1,1}), (re_{1,2}), \ldots\ldots\ldots, (re_{1,T}) \\ (re_{2,1}), (re_{2,2}), \ldots\ldots\ldots, (re_{2,T}) \end{bmatrix}$$

Each register can indexed using $(re_{j,i})$, where j = 1,...k and i = 1,...T. In the RE technique, forward labels (F_x) are shifted into the left most elements of the path memory registers, with the remaining register contents moving one stage to the right as illustrated below:

$$\begin{bmatrix} (f_1), (re_{1,1}), (re_{1,2}), \ldots\ldots\ldots, (re_{1,T-1}) \\ (f_2), (re_{2,1}), (re_{2,2}), \ldots\ldots\ldots, (re_{2,T-1}) \end{bmatrix}$$

The RE-VDA proceeds by computing a path metric for each possible path in the trellis and retaining 2^M of those paths at any one time based on a ML criterion. The VDA computes a new path metric for each incoming path and selects the path with the minimum path metric as the surviving path. A branch label is then written to the path memory using the following procedure. First the contents of the predecessor state register are read, and represents the surviving path up to and including the predecessor stage. The branch label is then shifted into the surviving path and written to the designated state register at the present decoding stage. This procedure is repeated for each of the 2^M surviving paths. Once the path memory is full, a decoding decision is made by shifting out the oldest bits from each register into the output decision unit, along with the corresponding path metrics. The output decision unit then compares all path metrics and selects the ML survivor by selecting the path with the minimum path metric state. The branch label corresponding to the minimum path metric state is thus selected as the decoded symbol, as illustrated by figure 3-2. This decoded symbol is equivalent to the input symbol at the encoder during the encoding phase. As the register exchange technique stores forward labels in its path memory, it is known as a forward label decoder.

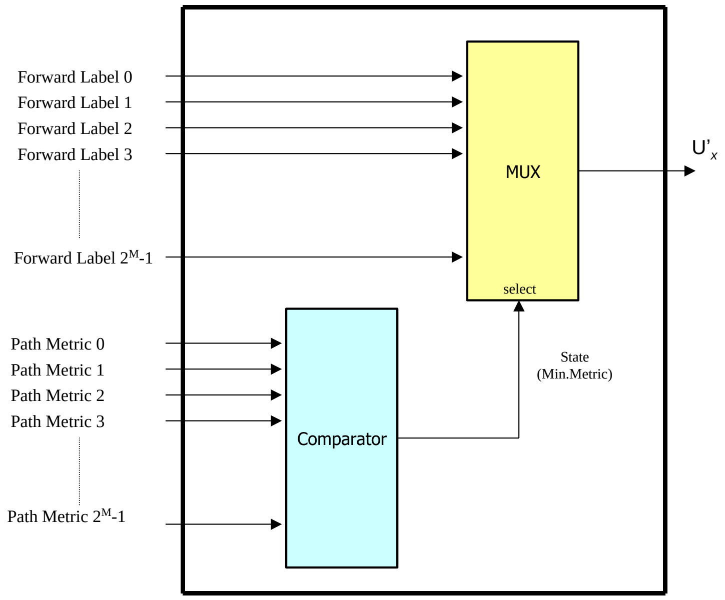


Figure 3-2 Register Exchange Output Decision Unit.

The entire R.E. procedure for Viterbi decoding is shown below in table 3-1.

4. Initialize Data Structures

4.1. Initialize the trellis stage pointer to zero. Initialize the path memory write pointer to zero. Initialize the decoded symbol counter to zero.

4.2. Initialize the path metric for the known initial state to zero, with the remaining $2^M -1$ path metrics to their maximum value. Go to step 2.

5. Compute Path Metrics and Survivors

5.1. Increment the trellis stage pointer, and the path memory write pointer.

5.2. For every trellis node, compute 2^k path metrics by summing the path metrics from nodes at the previous stage to the corresponding branch metrics computed at the present stage[1].

5.3. Compare the 2^k paths and select the path with the minimum path metric as the surviving path, all other incoming paths to the trellis node are no longer considered. If there is a tie between path metrics, the algorithm selects one path.

5.4. Store the path metric. Update the surviving path by shifting in the surviving forward label to the left hand side of the path memory register where the surviving path currently terminates.

5.5. If the path memory write pointer is $< T$ then go to step 2.1, else if the path memory write pointer $= T$ go to step 3.

[1] If all the incoming path metrics to a trellis node have a maximum value, the path metric for that node is held at that maximum value.

6. Output Decision

6.1. Determine the minimum path metric state.

6.2. A decoder decision is made for one symbol by first reading in $2^M -1$ branch labels from path memory. These branch labels correspond to the oldest forward labels elements in path memory. Use the state number determined from 3.1 to select one of $2^M -1$ branch labels. This branch label corresponds to the decoded symbol.

6.3. Increment the decoded symbol counter. If the decoded symbol count $< N$, go to step 2.1, else finish.

Table 3-1 R.E. -VDA Procedure

3.2.2 Decoding Example with Register Exchange

Given the (3,2,2) convolutional encoder as shown in figure 3-3, the input sequence U = [00, 11, 01, 10, 01, 11, 00, 10, 01, 10, 01, 10, 00, 10, 01, 11, 00, 10, 01, 10] of length N = 40, is encoded to the transmitted sequence V [000, 011, 111, 011, 100, 110, 010, 000, 100, 011, 100, 011, 001, 000, 100, 110, 010, 000, 100, 011, 001, 110].

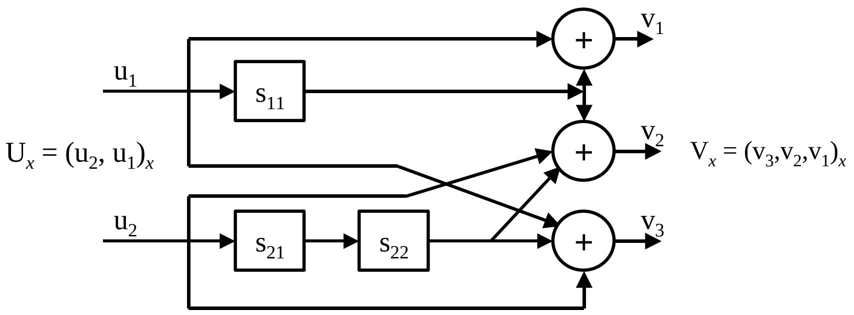


Figure 3-3 (3,2,2) Convolutional Encoder.

The sequence V is transmitted over a noisy BSC channel, and the message R is received without an error. Each trellis branch is labeled with the forward label F_x followed by the corresponding convolutional encoder output symbol, V_x for the trellis branch.

$$\begin{aligned} F_x &= (f_k, f_{k-1},\dots, f_1)_x = (u_k, u_{k-1},\dots, u_1)_x \\ &= (f_2, f_1)_x = (u_2, u_1)_x \end{aligned}$$

$$\begin{aligned} V_x &= (v_n, v_{n-1},\dots, v_1)_x \\ &= (v_3, v_2, v_1)_x \end{aligned}$$

To understand the operation of register exchange, figure 3-4 illustrates the operation for one trellis node. At stage 0, the register contents are initialized to all zeros. The initialized bits are underlined in the figure so as to distinguish the decoded bits shifted in from the initialized bits.

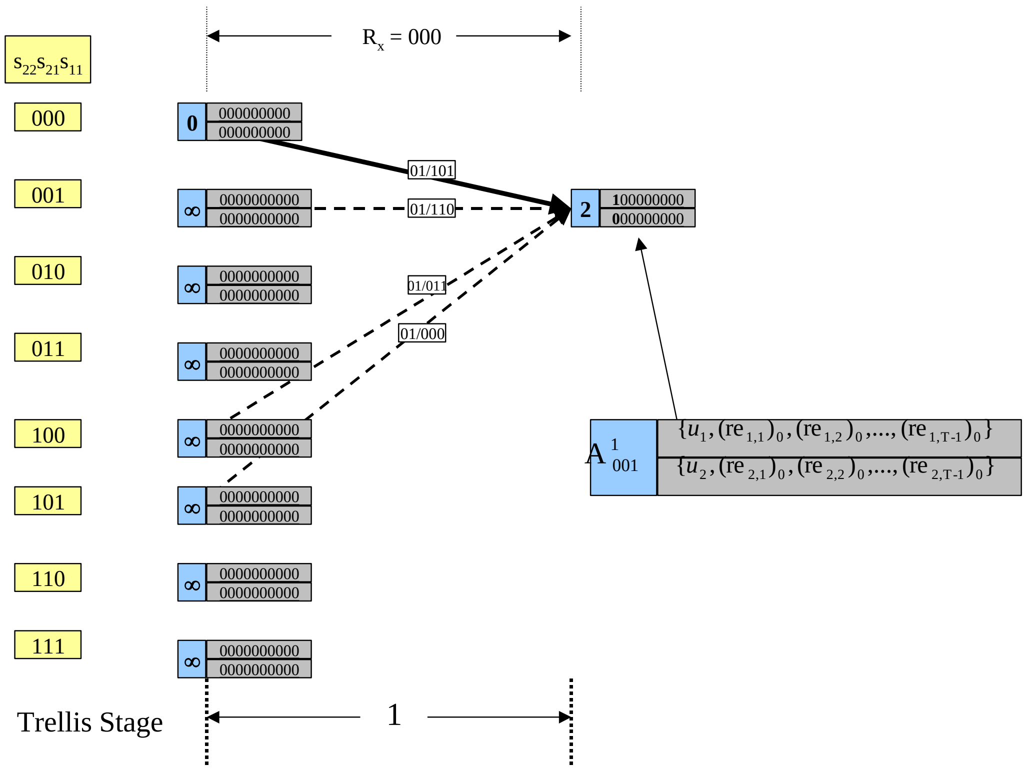


Figure 3-4 (3,2,2) RE path memory update.

For this example, it is assumed that block synchronization is accounted for, and the path metric unit computes path metrics based on Hamming distances. Also the maximum path metric is denoted as ∞. A tie between two or more path metrics is resolved by selecting the upper trellis path, i.e. the path that originates from a lower state number. The three step RE procedure outlined in section 3.2.1 is now used for walking through the decoding steps of the Viterbi algorithm.

STEP 1

For this example, the initial state is state 000, therefore the path metric for state 000 is initialized to zero. The remaining seven path metrics are initialized to their maximum value denoted as ∞.

STEP 2

This step is repeated for every trellis stage of the decoding process. At stage 1, the received symbol R_x is 000. A total of (2^{M+k}) or 32 Hamming distances are computed between R_x and each of the trellis branch labels V_x. Each of the Hamming distances are added to the corresponding path metrics initialized at stage 0. For each trellis node, the four path metrics computed are compared and the path with the minimum path metric is selected as the most likely path. If there is a tie a number of paths the uppermost "tie" path is selected. This procedure is repeated for the remaining trellis nodes at stage 1.

The stage pointer is now incremented to 2. The algorithm will stay in step 2 as the trellis stage pointer is less than ten. At stage 2 the path metrics are 0,2,2,2,∞,∞,∞,∞, and the received symbol R_x is 011. Again a total of 32 Hamming distances are computed between R_x and each of the trellis branch labels V_x. Each of the Hamming distances are added to the corresponding path metrics computed at stage 1. The trellis at the end of stage 1 and stage 2 is shown in figure 3-5.

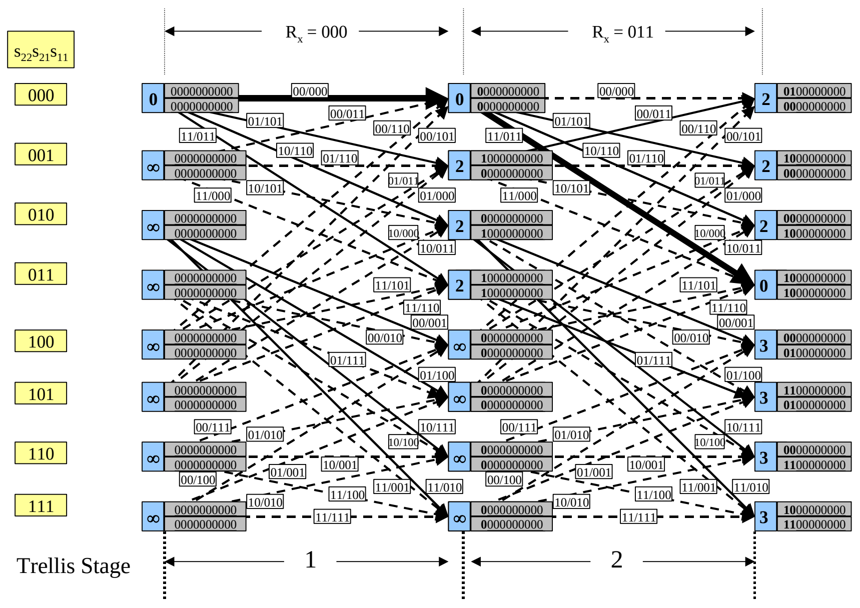


Figure 3-5 Contents of (3,2,2) RE-path memory at stages 1, and 2.

This procedure is exactly the same for each decoding stage up stage T = 10. Figures 3-6 through 3-9 show the remaining stages up until a decoding decision is made at stage T.

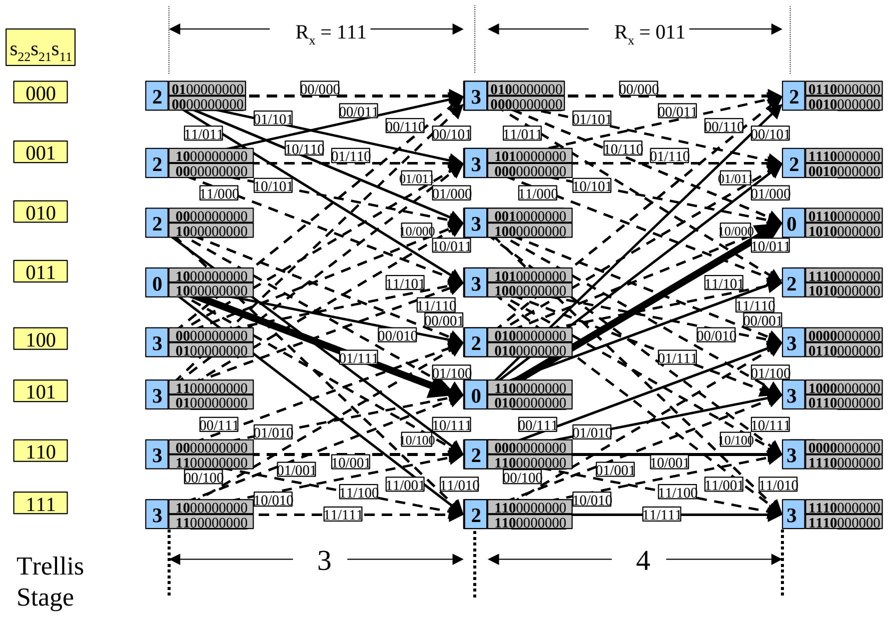


Figure 3-6 Contents of (3,2,2) RE-path memory at stages 3, and 4.

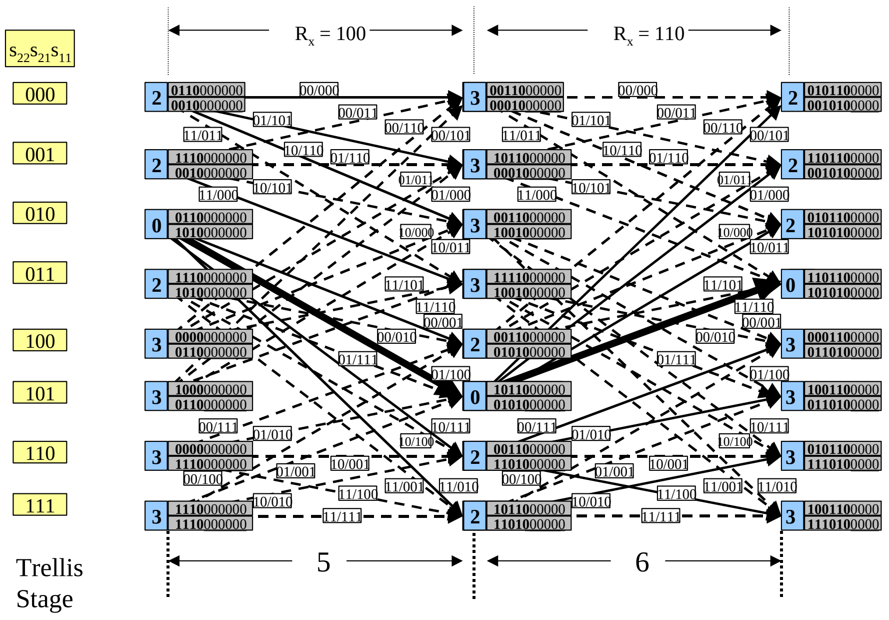


Figure 3-7 Contents of (3,2,2) RE-path memory at stage 5, and 6.

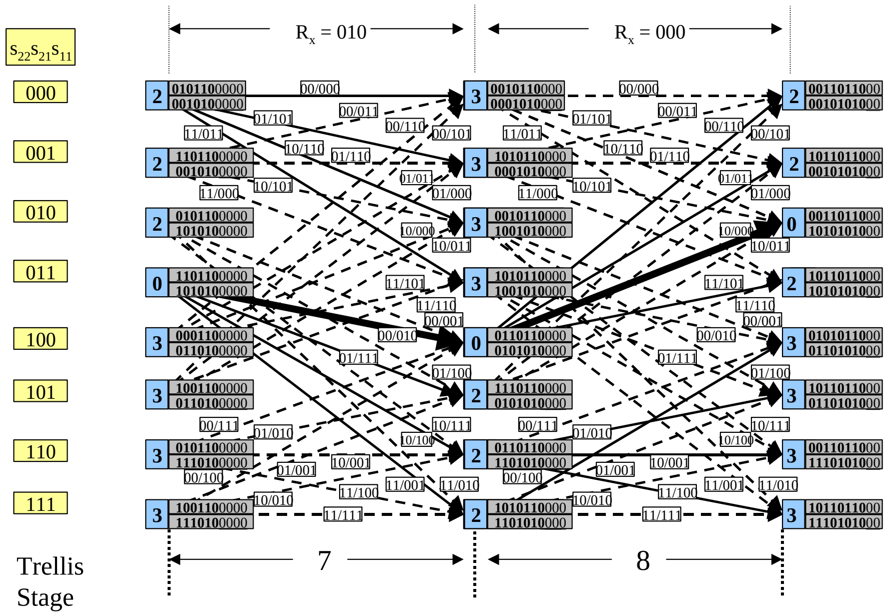


Figure 3-8 Contents of (3,2,2) RE-path memory at stages 7, and 8.

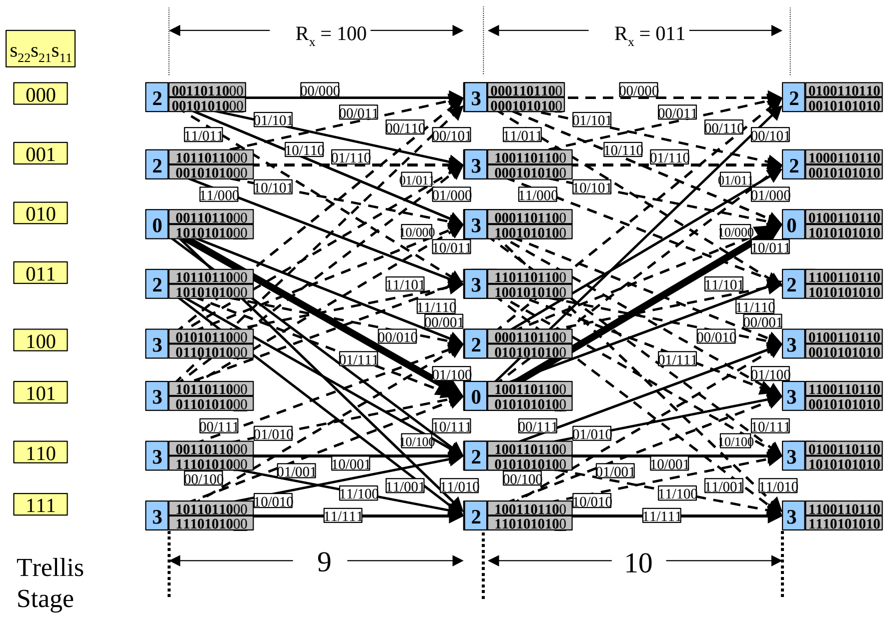


Figure 3-9 Contents of (3,2,2) RE-path memory at stage 9, and 10.

The stage pointer is now equal to the truncation length of the registers, (T=10) and the registers are full, the algorithm proceeds to step 3.

STEP 3

A decoding decision has to be made for the oldest bits in path memory before the next decoding stage. A decoding decision is made for the oldest bits in path memory by finding the most likely path metric hypothesis and then releasing the corresponding branch label as the decoded symbol. Figure 3-10, illustrates this process. In this case, the minimum path metric points to state 010. Therefore the first decoded symbol is equivalent to the forward label corresponding to the minimum path metric at state 2 (010) as shown below in figure 3-10.

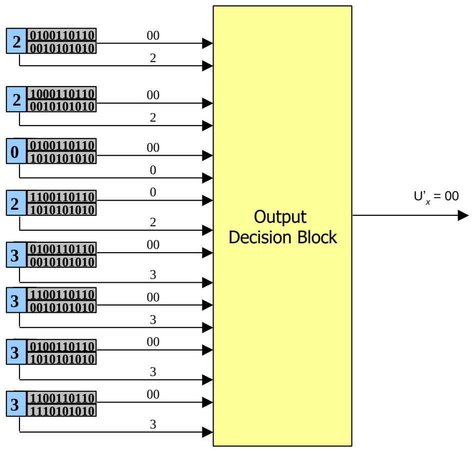


Figure 3-10 Output Decision for 1st decoded symbol.

Once a decoded symbol is selected, the next set of forward labels are shifted into the path memory and the oldest forward labels are shifted out. The path memory is full again and another decoding decision is made for the second decoded symbol. This process is repeated for each new set of decoding decisions. Therefore after an initial latency of T, a RE path memory produces a decoded symbol for every decoding instance.

3.2.3 Traceback Path Memory

In the traceback (TB) method, each state is once again assigned a register, but the contents of the registers do not move back and forth like the RE case. Each register contains the past history of the surviving branches entering that state. Once the registers are full, information bits are obtained by "tracing" back through the trellis as dictated by this connection history stored in the registers. As with the RE technique, each trace back register contains kT bits, where T is the truncation length.

The procedure for performing the VDA when using the traceback path memory is now outlined. Like the general VDA technique, the traceback technique maintains a memory with two entries to perform decoding operations. One is the path metric memory, and operates as described in section 2.4.4. The other entry is the path memory and is organized to implement the traceback (TB) technique. The entries in the TB path memory contain 2^M surviving paths, the contents of which represent symbols already shifted out of the encoder, i.e. backward labels. Once the path memory is full, the trellis connections are recalled in reverse order, i.e. right to left. Recall from section 3.1, a path that moves from right to left is called a backward path, hence to trace back along a backward path, the traceback path memory stores backward labels in order to recall the trellis connections on reverse. The traceback process is detailed in section 3.2.5.

Similar to the RE technique, there are 2^M state path memory registers, one for each surviving path, and each state path memory register has kT bits. This path memory register can be organized in a number of ways, however for this project each path memory register is organized into k rows and T columns, where T is the truncation length. Each register is organized as follows:

$$\begin{bmatrix} (tb_{1,1}),(tb_{1,2}),\ldots\ldots\ldots,(tb_{1,T}) \\ (tb_{2,1}),(tb_{2,2}),\ldots\ldots\ldots,(tb_{2,T}) \end{bmatrix}$$

During traceback backward labels are read from path memory using ($tb_{j,i}$), where j = 1,...k and i = 1,...T. In the case of writes to path memory, backward labels (B_x) are shifted into the left most elements of the path memory registers, with the remaining register contents moving one stage to the right as illustrated below:

$$\begin{bmatrix} (b_1),(tb_{1,1}),(tb_{1,2}),\ldots\ldots\ldots,(tb_{1,T-1}) \\ (b_2),(tb_{2,1}),(tb_{2,2}),\ldots\ldots\ldots,(tb_{2,T-1}) \end{bmatrix}$$

The TB-VDA proceeds by computing a path metric for each possible path in the trellis and retaining 2^M of those paths at any one time based on a maximum likelihood criterion. The VDA computes a new path metric for each incoming path and selects the path with the minimum path metric as the surviving path. A procedure to perform all the steps involved in the TB Viterbi Algorithm is shown below in table 3-2.

1. *Initialize Data Structures*

- 1.1. Initialize the trellis stage pointer to zero. Initialize the path memory write pointer to zero. Initialize the traceback pointer to zero. Initialize the decoded symbol counter to zero.
- 1.2. Initialize the path metric for the known initial state to zero, with the remaining 2^M -1 path metrics to their maximum value. Go to step 2.

2. *Compute Path Metrics and Survivors*

- 2.1. Increment the trellis stage pointer, and the path memory write pointer.
- 2.2. For every trellis node, compute 2^k path metrics by summing the path metrics from nodes at the previous stage to the corresponding branch metrics computed at the present stage[1].
- 2.3. Compare the 2^k paths and select the path with the minimum path metric as the surviving path, all other incoming paths to the trellis node are no longer considered. If there is a tie between path metrics, the algorithm selects one path.
- 2.4. Store the path metric. Update the surviving path by shifting in the surviving backward label to the left hand side of the path memory register where the surviving path currently terminates[2]
- 2.5. If the path memory write pointer is $< T$ then go to step 2.1, else if the path memory write pointer $= T$ go to step 3

[1] *If all the incoming path metrics to a trellis node have a maximum value, the path metric for that node is held at that maximum value.*

[2] *Section 3.2.4 details the path memory update.*

3. *Traceback and Output Decision*

- 3.1. Set the traceback pointer equal to T. Determine the traceback start state number as the state that corresponds to the minimum path metric.
- 3.2. The state number and the traceback pointer are combined into a row-column address used to index path memory. Use this address to read a backward label from path memory. A predecessor state on the surviving path is then determined by use of the traceback mapping function[3]. Decrement the traceback pointer. If the traceback pointer >1 repeat step 3.2, else go to step 3.3.
- 3.3. A decoder decision is made for one symbol by selecting the leftmost elements $(s_{k,1}, s_{k-1,1}, \ldots s_{1,1})_x$ of the traceback mapping register as the decoded symbol.
- 3.4. Increment the decoded symbol counter. If the decoded symbol count < N, go to step 2.1, else finish.

[3] *Section 3.2.5 details the traceback mapping process.*

Table 3-2 V.D.A Traceback Procedure.

3.2.4 Traceback Path Memory Update Example

Given the (3,2,2) convolutional encoder as shown in Figure 3-3, the same input sequence U = [00, 11, 01, 10, 01, 11, 00, 10, 01, 10, 01, 10, 00, 10, 01, 11, 00, 10, 01, 10] of length N = 40, is encoded to the transmitted sequence V [000, 011, 111, 011, 100, 110, 010, 000, 100, 011, 100, 011, 001, 000, 100, 110, 010, 000, 100, 011, 001, 110]. The sequence V is transmitted over a noisy BSC channel, and the message R is received without an error.

Each trellis branch is labeled with the backward label B_x followed by the corresponding convolutional encoder output symbol, V_x for the trellis branch.

$$\begin{aligned} B_x &= (s_{k,m_k}, s_{k-1,m_{k-1}}, \ldots, s_{1,m_1})_x = (b_k, b_{k-1}, \ldots, b_1)_x \\ &= (s_{2,2}, s_{1,1})_x = (b_2, b_1)_x \end{aligned}$$

$$\begin{aligned} V_x &= (v_n, v_{n-1}, \ldots, v_1)_x \\ &= (v_3, v_2, v_1)_x \end{aligned}$$

Figure 3-11 illustrates the update operation for one trellis node. This expression updates the surviving path at stage *x* by shifting in the backward label element B_x to the path memory register on the surviving path.

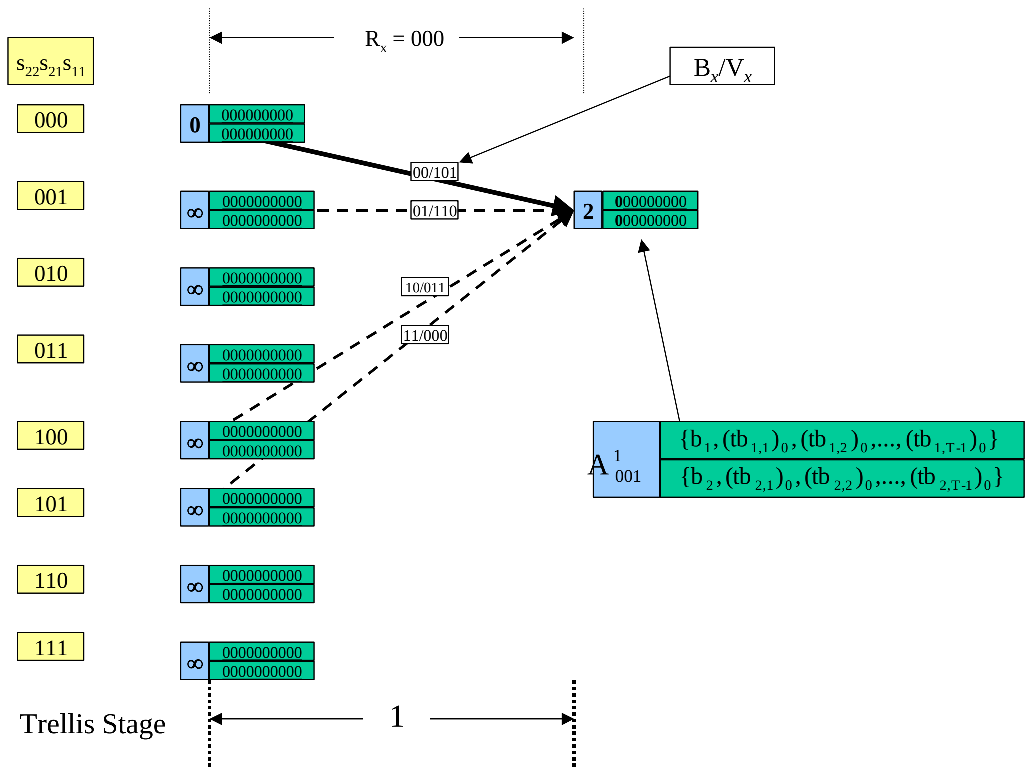


Figure 3-11 (3,2,2) TB path memory update.

From Figure 3-11, the non-survivors are denoted by the dashed lines. Again for this example, block synchronization has been accounted for and the path metric unit computes path metrics based on Hamming distances. Also the maximum path metric is denoted as ∞. A tie between two or more path metrics is resolved by selecting the upper trellis path, i.e. the path that originates from a lower state number.

STEP 1

For this example, the initial state is state 000, therefore the path metric for state 000 is initialized to zero. The remaining seven path metrics are initialized to their maximum value denoted as ∞.

STEP 2

This step is repeated for every trellis stage of the decoding process. At stage 1, the received symbol R$_x$ is 000. A total of (2^{M+k}) or 32 Hamming distances are computed between R$_x$ and each of the trellis branch labels V$_x$. Each of the Hamming distances are added to the corresponding path metrics initialized at stage 0. For each trellis node, the four path metrics computed are compared and the path with the minimum path metric is selected as the most likely path. If there is a tie between two or more paths the uppermost "tie" path is selected. This procedure is repeated for the remaining trellis nodes at stage 1. In each of the trellis stages, the non survivors are denoted by dashed lines, the survivor by a normal line, and maximum likelihood path by the thick line.

The stage pointer is now incremented to 2. The algorithm will stay in step 2 as the trellis stage pointer is less than ten, i.e. the value of T, the truncation length for this decoder. At stage 2 the path metrics are 0,2,2,2,∞,∞,∞,∞, and the received symbol R$_x$ is 011. Again a total of 32 Hamming distances are computed between R$_x$ and each of the trellis branch labels V$_x$. Each of the Hamming distances are added to the corresponding path metrics computed at stage 1. The trellis at the end of stage 1 and stage 2 is shown in figure 3-12.

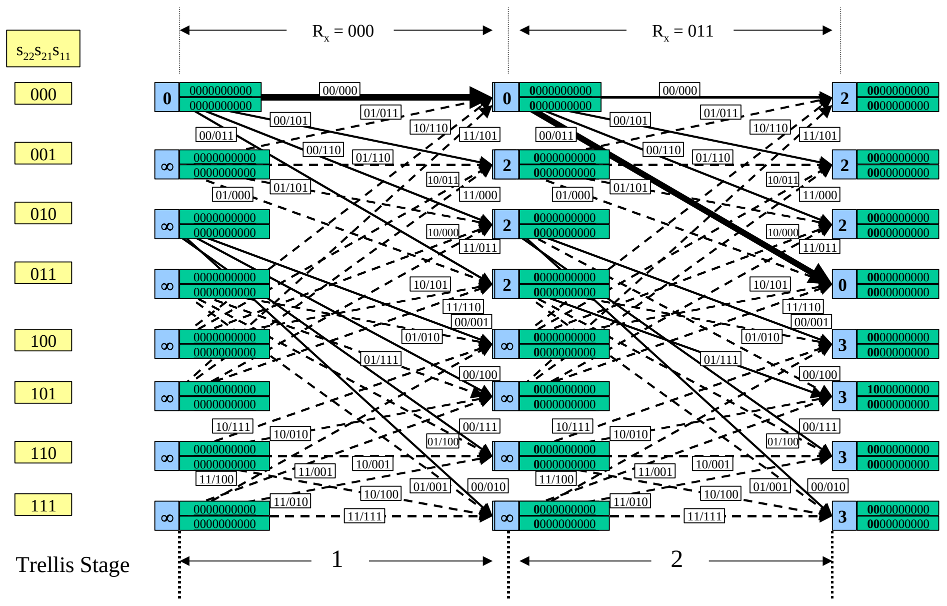


Figure 3-12 Contents of (3,2,2) TB-path memory at stages 1, and 2.

This procedure is exactly the same for each decoding stage up stage T = 10. Figures 3-13 through 3-16 show the remaining stages up until a decoding decision is made at stage T.

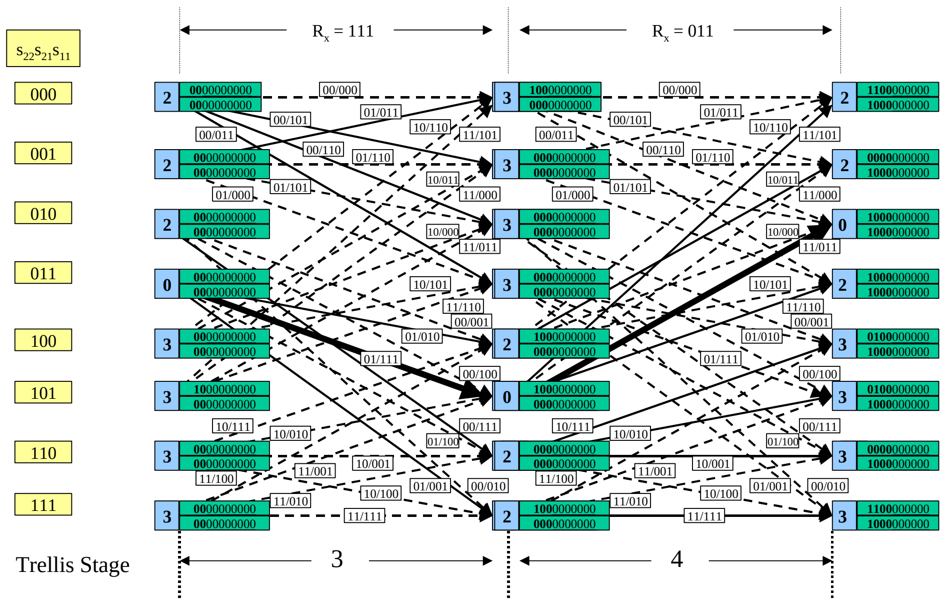


Figure 3-13 Contents of (3,2,2) TB-path memory at stages 3, and 4.

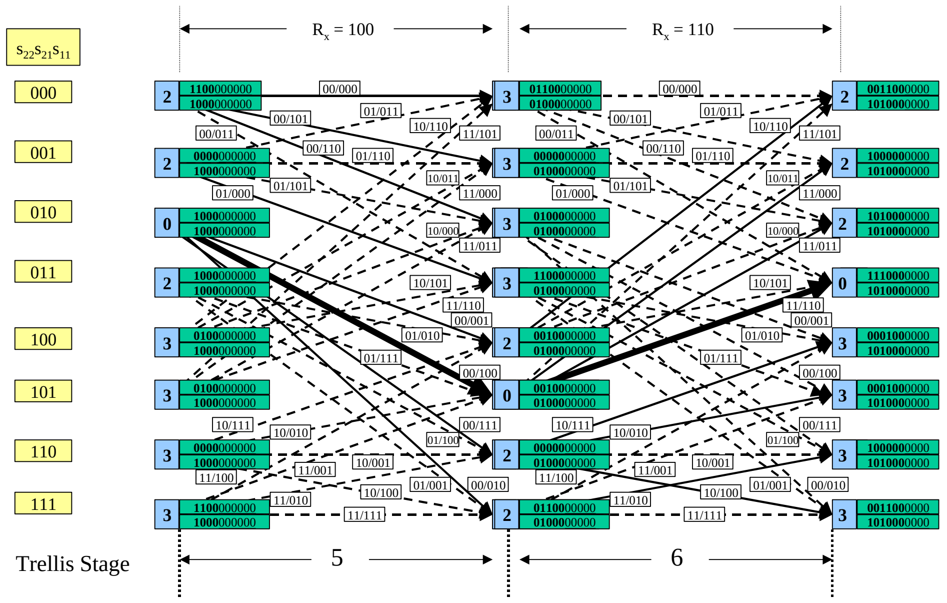


Figure 3-14 Contents of (3,2,2) TB-path memory at stages 5, and 6

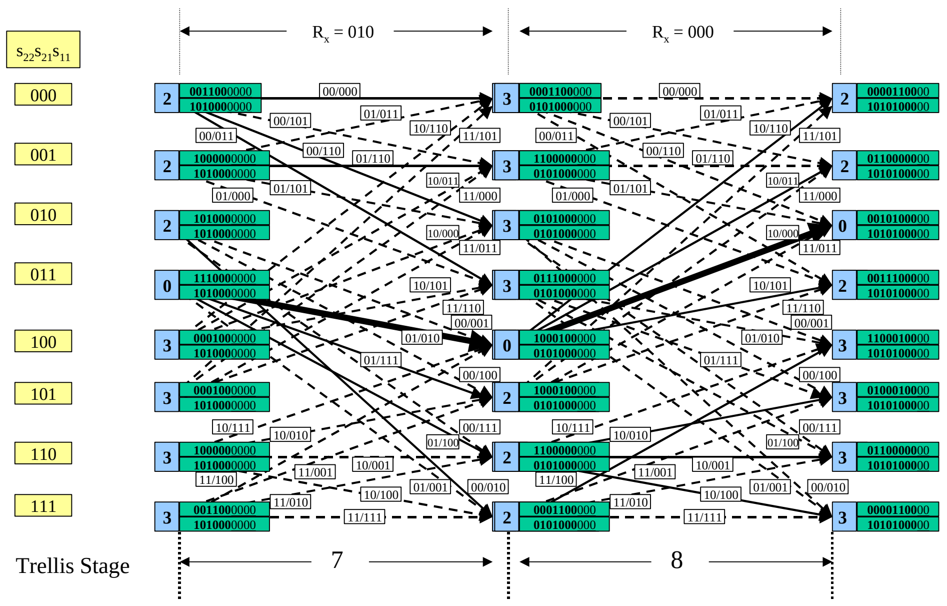


Figure 3-15 Contents of (3,2,2) TB-path memory at stages 7, and 8

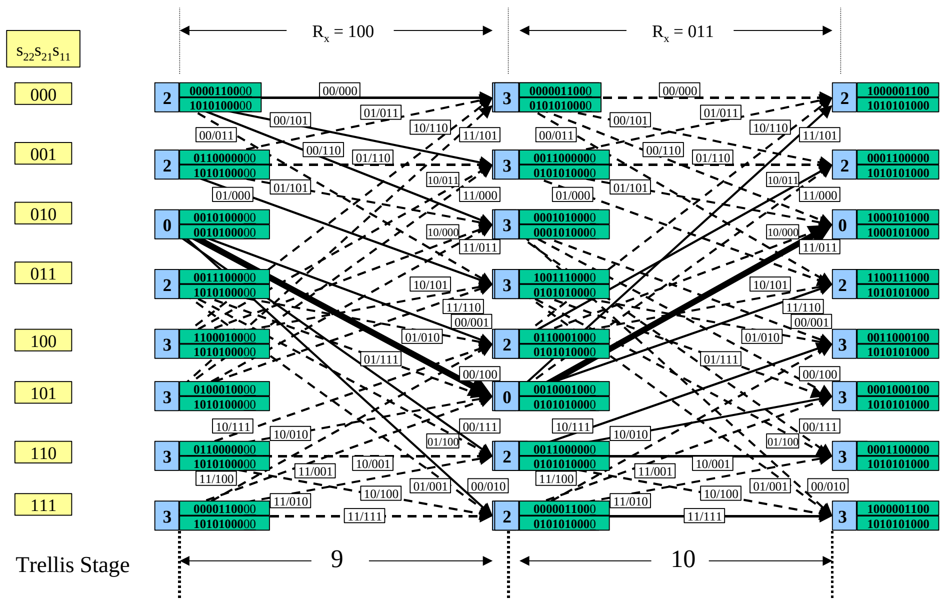


Figure 3-16 Contents of (3,2,2) TB-path memory at stages 9, and 10

Once the path memory registers are full, one of the 2^M surviving paths are selected as the ML survivor by choosing the path with the minimum path metric. This path is then traced back through the trellis to stage 1. The output decision block then selects the first decoded symbol, and section 3.2.5 will illustrate this process. The oldest bits in the path memory registers are then shifted out the right as the next set of backward labels are shifted into the leftmost elements at next trellis stage time. At the same time a new set of path metrics are also computed. In a similar manner the traceback process is repeated for the second and succeeding decoded symbols, and continues in this fashion until the entire received sequence is decoded. The process of tracing back along a surviving path and selecting a decoded symbol in outlined in the next section.

3.2.5 Traceback Process

Once the path memory is full, decoded symbols are produced by recalling the trellis connections in reverse order. This decoding method is called the traceback method. To illustrate the traceback technique, a mapping function needs to be developed to trace a state at stage *x*, back to its predecessor state at stage *x*-1 on the surviving path.

During the encoding process, the input symbol $\mathrm{U}_x = (\mathrm{u}_\mathrm{k}, \mathrm{u}_{\mathrm{k}-1}, ..., \mathrm{u}_1)_x$ is shifted into the leftmost elements of the encoder register, the right most elements, $(\mathrm{s}_{\mathrm{k},\mathrm{m}_\mathrm{k}}, \mathrm{s}_{\mathrm{k-1},\mathrm{m}_{\mathrm{k-1}}}, \ldots, \mathrm{s}_{1,\mathrm{m}_1})_x$ are shifted out of the encoder register for every state transition in the encoding process, and the encoder state changes from state S_x to state S_{x+1}. The state of the encoder before a new input symbol U_x is shifted in is as follows:

$$\mathrm{S}_x = \begin{bmatrix} (\mathrm{s}_{1,1}, \mathrm{s}_{1,2}, \mathrm{s}_{1,3}, \ldots \mathrm{s}_{1,\mathrm{m}_1})_x \\ (\mathrm{s}_{2,1}, \mathrm{s}_{2,2}, \mathrm{s}_{2,3}, \ldots \mathrm{s}_{2,\mathrm{m}_2})_x \\ : \\ : \\ (\mathrm{s}_{\mathrm{k},1}, \mathrm{s}_{\mathrm{k},2}, \mathrm{s}_{\mathrm{k},3}, \ldots \mathrm{s}_{\mathrm{k},\mathrm{m}_\mathrm{k}})_x \end{bmatrix}$$

The state of the encoder, S_{x+1}, once a new symbol has been shifted in and the right most elements, $(\mathrm{s}_{\mathrm{k},\mathrm{m}_\mathrm{k}}, \mathrm{s}_{\mathrm{k-1},\mathrm{m}_{\mathrm{k-1}}}, \ldots, \mathrm{s}_{1,\mathrm{m}_1})_x$ are shifted out is equivalent to:

$$\mathrm{S}_{x+1} = \begin{bmatrix} \{\mathrm{u}_{1,}(\mathrm{s}_{1,1}, \mathrm{s}_{1,2}, \mathrm{s}_{1,3})\}_x \\ \{\mathrm{u}_{2},(\mathrm{s}_{2,1}, \mathrm{s}_{2,2}, \mathrm{s}_{2,3})\}_x \\ : \\ : \\ \{\mathrm{u}_{\mathrm{k},}(\mathrm{s}_{\mathrm{k},1}, \mathrm{s}_{\mathrm{k},2}, \mathrm{s}_{\mathrm{k},3})\}_x \end{bmatrix}$$

This process continues for the entire encoding operation at the transmitter. At the decoder, the traceback unit uses a traceback register with the same generator polynomial as the encoding shift register, but the traceback register works in reverse order, i.e. right to left as the decoder attempts to replicate each state transition made by the encoder. Therefore at every state transition in the traceback decoding process, the left most elements $(\mathrm{s}_{\mathrm{k},1}, \mathrm{s}_{\mathrm{k-1},1}, \ldots, \mathrm{s}_{1,1})_x$ are shifted out of the traceback register as a new backward label is shifted into the right most elements.

A traceback mapping is now defined.

Let $f(.)$ denote the function that traces state S_x to S_{x-1} back along the surviving path, where S_x and $\mathrm{S}_{x-1} = (\mathrm{s}_{\mathrm{j,i}})_x$, where j = k,...1 and i = $\mathrm{m_j}$,...1.

At the encoder, the state elements $(\mathrm{s}_{\mathrm{k,m_k}}, \mathrm{s}_{\mathrm{k-1,m_{k-1}}}, \ldots, \mathrm{s}_{1,\mathrm{m}_1})_x$ are shifted out when the encoder changes to state, thus one needs to store the elements $(\mathrm{s}_{\mathrm{k,m_k}}, \mathrm{s}_{\mathrm{k-1,m_{k-1}}}, \ldots \mathrm{s}_{1,\mathrm{m}_1})_x$ for tracing back along the surviving path. These state elements are equivalent to backward label $\mathrm{B}_x = (\mathrm{b_k}, \mathrm{b_{k-1}}, \ldots, \mathrm{b}_1)_x$). In order to replicate the encoder actions in reverse during traceback, the elements are stored in path memory. Thus a traceback mapping function can be expressed in terms of the current state S_x and the backward label B_x and is defined as:

$$\mathrm{S}_{x-1} = f(\mathrm{S}_x, \mathrm{B}_x),$$

As the number of traceback stages is T, this traceback mapping function is called for T iterations until stage 1 of the path memory register is reached. For every state transition in the traceback decoding process, the left most elements

$(s_{k,1}, s_{k\text{-}1,1}, \ldots, s_{1,1})_x$ are shifted out of the traceback register as a new labels are shifted into the right most elements, as shown below :

$$S_{x-1} = \begin{bmatrix} \{(s_{1,2}, s_{1,3}, \ldots s_{1,m_1}), b_1\}_x \\ \{(s_{2,2}, s_{2,3}, \ldots s_{2,m_2}), b_2\}_x \\ : \\ : \\ \{(s_{k,2}, s_{k,3}, \ldots s_{k,m_k}), b_k\}_x \end{bmatrix}$$

These left most shifted out elements are equivalent the encoder input symbols U_x during the encoding process. Therefore at stage 1 the elements $(s_{k,1}, s_{k\text{-}1,1}, \ldots, s_{1,1})_x$ of traceback shift register are selected as the first decoded symbol.

Example. For the (3,2,2) convolutional encoder in figure 3-3, the trellis connections from state 001 at stage *x* and to states at stage *x*-1 is illustrated in figure 3-17 in order to trace state 001 back to state 000, the backward label 00 is used in the traceback mapping function.

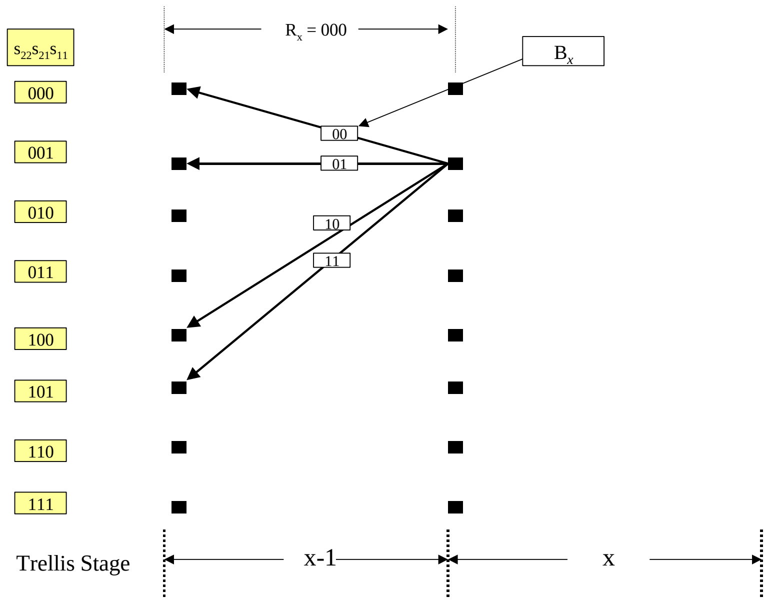


Figure 3-17 State 001 traceback paths

Table 3-3 illustrates a logic table of traceback changes for all states between stage *x* to stage *x*-1. The dash notation "-" in table 3-3 means that there is no connection between the states at stage *x* and stage *x*-1.

	S_x (s_{22},s_{21},s_{11})							
S_{x-1}(s_{22},s_{21},s_{11})	000	001	010	011	100	101	110	111
000	00	00	00	00	-	-	-	-
001	01	01	01	01	-	-	-	-
010	-	-	-	-	00	00	00	00
011	-	-	-	-	01	01	01	01
100	10	10	10	10	-	-	-	-
101	11	11	11	11	-	-	-	-
110	-	-	-	-	10	10	10	10
111	-	-	-	-	11	11	11	11

Table 3-3 (3,2,2) traceback state table

The logic connections in table 3-3, are thus equivalent to backward labels, and are necessary in order to uniquely traceback back valid paths.

3.2.6 (n, k, m) Traceback Example

The traceback process commences once the path memory is full. Given the (3,2,2) example, the state representation in matrix form is reduced to.

$$S_x = \begin{bmatrix} (s_{1,1})_x \\ (s_{2,1}, s_{2,2})_x \end{bmatrix}$$

The process of tracing back one state consists of shifting a backward label, $B_x = (b_2, b_1)_x$ to the right most elements of S_x resulting in S_{x-1} as follows

$$S_{x-1} = \begin{bmatrix} (s_{1,1})_x \\ (s_{2,1}, s_{2,2})_x \end{bmatrix} \Leftarrow \begin{bmatrix} (b_1)_x \\ (b_2)_x \end{bmatrix}$$

$$S_{x-1} = \begin{bmatrix} (b_1)_x \\ (s_{2,2}, b_2)_x \end{bmatrix}$$

At each traceback stage, a backward label is read from path memory using the ($tb_{j,i}$) row-column address. The traceback pointer provides the column address and corresponds to the column or traceback stage within the path memory register. During traceback the traceback pointer decrements from a value equivalent to the length of the path memory register to stage 1. The state S_x is used as the row address and points to the current path memory register on the surviving path. This addressing scheme is illustrated in figure 3-18.

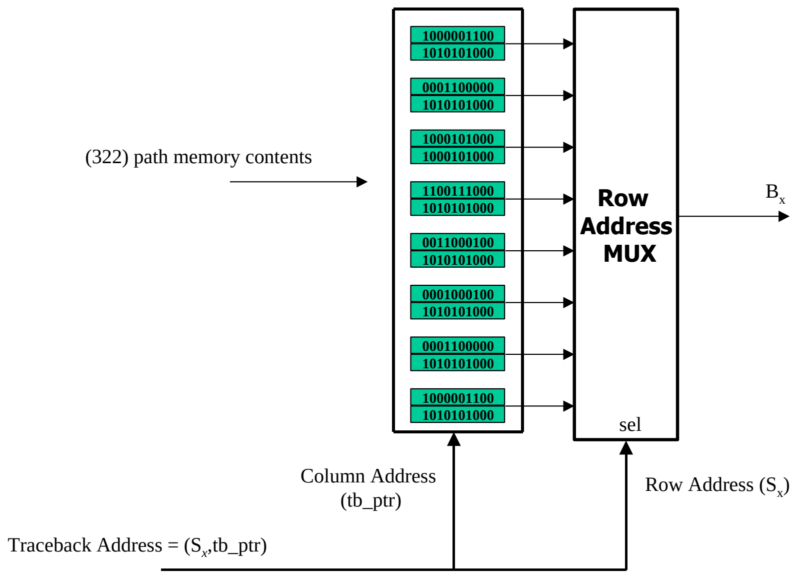


Figure 3-18 Path memory addressing scheme

From figure 3-18, the state with the smallest survivor path metric at stage 10 is 010. The backward label read from path memory using the row column address scheme above is $B_x = 11$. Then at stage 9, a new state is obtained by the traceback mapping function as follows:

$$S_{10} = \begin{bmatrix} 0 \\ 1,0 \end{bmatrix} \Leftarrow \begin{bmatrix} 1 \\ 1 \end{bmatrix}$$

$$S_9 = \begin{bmatrix} 1 \\ 0,1 \end{bmatrix}$$

Using the notation in the trellis for states s_{22},s_{21},s_{11}, this is 101. Similarly at stage 8, a new state is obtained by traceback by first reading the backward label from path memory pointed to by the row-column address as (5,9). The backward label returned is $B_x = 00$. A new state at stage 8 can then be traced back using the trace back mapping function as follows:

$$S_9 = \begin{bmatrix} 1 \\ 0,1 \end{bmatrix} \Leftarrow \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

$$S_8 = \begin{bmatrix} 0 \\ 1,0 \end{bmatrix}$$

This process is identical for all traceback stages until stage 1 is reached. Table 3-4 illustrates the traceback process for five traceback iterations. Once the traceback register reaches stage 1, a decision is made on selecting the first decoded symbol.

$$\begin{aligned} U'_x &= (s_{k,1}, s_{k\text{-}1,1}, \dots s_{1,1})_x \\ &= (s_{2,1}, s_{1,1})_x \end{aligned}$$

For example after the 1st traceback iteration, the leftmost bits (s_{21}, s_{11}) of the traceback register correspond to the decoded symbol, as shown in table 3-4.

$$\begin{aligned} U'_1 &= (s_{2,1}, s_{1,1}) \\ &= (0,0) \end{aligned}$$

1st Traceback	2nd Traceback	3rd Traceback	4th Traceback	5th Traceback
$S_{10} = \begin{bmatrix}0\\1,0\end{bmatrix} \Leftarrow \begin{bmatrix}1\\1\end{bmatrix}$	$S_{10} = \begin{bmatrix}1\\0,1\end{bmatrix} \Leftarrow \begin{bmatrix}0\\0\end{bmatrix}$	$S_{10} = \begin{bmatrix}0\\1,0\end{bmatrix} \Leftarrow \begin{bmatrix}1\\1\end{bmatrix}$	$S_{10} = \begin{bmatrix}0\\0,1\end{bmatrix} \Leftarrow \begin{bmatrix}0\\0\end{bmatrix}$	$S_{10} = \begin{bmatrix}0\\1,0\end{bmatrix} \Leftarrow \begin{bmatrix}0\\1\end{bmatrix}$
$S_9 = \begin{bmatrix}1\\0,1\end{bmatrix} \Leftarrow \begin{bmatrix}0\\0\end{bmatrix}$	$S_9 = \begin{bmatrix}0\\1,0\end{bmatrix} \Leftarrow \begin{bmatrix}1\\1\end{bmatrix}$	$S_9 = \begin{bmatrix}1\\0,1\end{bmatrix} \Leftarrow \begin{bmatrix}0\\0\end{bmatrix}$	$S_9 = \begin{bmatrix}0\\1,0\end{bmatrix} \Leftarrow \begin{bmatrix}1\\1\end{bmatrix}$	$S_9 = \begin{bmatrix}0\\0,1\end{bmatrix} \Leftarrow \begin{bmatrix}0\\0\end{bmatrix}$
$S_8 = \begin{bmatrix}0\\1,0\end{bmatrix} \Leftarrow \begin{bmatrix}0\\1\end{bmatrix}$	$S_8 = \begin{bmatrix}1\\0,1\end{bmatrix} \Leftarrow \begin{bmatrix}0\\0\end{bmatrix}$	$S_8 = \begin{bmatrix}0\\1,0\end{bmatrix} \Leftarrow \begin{bmatrix}1\\1\end{bmatrix}$	$S_8 = \begin{bmatrix}1\\0,1\end{bmatrix} \Leftarrow \begin{bmatrix}0\\0\end{bmatrix}$	$S_8 = \begin{bmatrix}0\\1,0\end{bmatrix} \Leftarrow \begin{bmatrix}1\\1\end{bmatrix}$
$S_7 = \begin{bmatrix}0\\0,1\end{bmatrix} \Leftarrow \begin{bmatrix}1\\0\end{bmatrix}$	$S_7 = \begin{bmatrix}0\\1,0\end{bmatrix} \Leftarrow \begin{bmatrix}0\\1\end{bmatrix}$	$S_7 = \begin{bmatrix}1\\0,1\end{bmatrix} \Leftarrow \begin{bmatrix}0\\0\end{bmatrix}$	$S_7 = \begin{bmatrix}0\\1,0\end{bmatrix} \Leftarrow \begin{bmatrix}1\\1\end{bmatrix}$	$S_7 = \begin{bmatrix}1\\0,1\end{bmatrix} \Leftarrow \begin{bmatrix}0\\0\end{bmatrix}$
$S_6 = \begin{bmatrix}1\\1,0\end{bmatrix} \Leftarrow \begin{bmatrix}1\\1\end{bmatrix}$	$S_6 = \begin{bmatrix}0\\0,1\end{bmatrix} \Leftarrow \begin{bmatrix}1\\0\end{bmatrix}$	$S_6 = \begin{bmatrix}0\\1,0\end{bmatrix} \Leftarrow \begin{bmatrix}0\\1\end{bmatrix}$	$S_6 = \begin{bmatrix}1\\0,1\end{bmatrix} \Leftarrow \begin{bmatrix}0\\0\end{bmatrix}$	$S_6 = \begin{bmatrix}0\\1,0\end{bmatrix} \Leftarrow \begin{bmatrix}1\\1\end{bmatrix}$
$S_5 = \begin{bmatrix}1\\0,1\end{bmatrix} \Leftarrow \begin{bmatrix}0\\0\end{bmatrix}$	$S_5 = \begin{bmatrix}1\\1,0\end{bmatrix} \Leftarrow \begin{bmatrix}1\\1\end{bmatrix}$	$S_5 = \begin{bmatrix}0\\0,1\end{bmatrix} \Leftarrow \begin{bmatrix}1\\0\end{bmatrix}$	$S_5 = \begin{bmatrix}0\\1,0\end{bmatrix} \Leftarrow \begin{bmatrix}0\\1\end{bmatrix}$	$S_5 = \begin{bmatrix}1\\0,1\end{bmatrix} \Leftarrow \begin{bmatrix}0\\0\end{bmatrix}$
$S_4 = \begin{bmatrix}0\\1,0\end{bmatrix} \Leftarrow \begin{bmatrix}1\\1\end{bmatrix}$	$S_4 = \begin{bmatrix}1\\0,1\end{bmatrix} \Leftarrow \begin{bmatrix}0\\0\end{bmatrix}$	$S_4 = \begin{bmatrix}1\\1,0\end{bmatrix} \Leftarrow \begin{bmatrix}1\\1\end{bmatrix}$	$S_4 = \begin{bmatrix}0\\0,1\end{bmatrix} \Leftarrow \begin{bmatrix}1\\0\end{bmatrix}$	$S_4 = \begin{bmatrix}0\\1,0\end{bmatrix} \Leftarrow \begin{bmatrix}0\\1\end{bmatrix}$
$S_3 = \begin{bmatrix}1\\0,1\end{bmatrix} \Leftarrow \begin{bmatrix}1\\0\end{bmatrix}$	$S_3 = \begin{bmatrix}0\\1,0\end{bmatrix} \Leftarrow \begin{bmatrix}1\\1\end{bmatrix}$	$S_3 = \begin{bmatrix}1\\0,1\end{bmatrix} \Leftarrow \begin{bmatrix}0\\0\end{bmatrix}$	$S_3 = \begin{bmatrix}1\\1,0\end{bmatrix} \Leftarrow \begin{bmatrix}1\\1\end{bmatrix}$	$S_3 = \begin{bmatrix}0\\0,1\end{bmatrix} \Leftarrow \begin{bmatrix}1\\0\end{bmatrix}$
$S_2 = \begin{bmatrix}1\\1,0\end{bmatrix} \Leftarrow \begin{bmatrix}0\\0\end{bmatrix}$	$S_2 = \begin{bmatrix}1\\0,1\end{bmatrix} \Leftarrow \begin{bmatrix}1\\0\end{bmatrix}$	$S_2 = \begin{bmatrix}0\\1,0\end{bmatrix} \Leftarrow \begin{bmatrix}1\\1\end{bmatrix}$	$S_2 = \begin{bmatrix}1\\0,1\end{bmatrix} \Leftarrow \begin{bmatrix}0\\0\end{bmatrix}$	$S_2 = \begin{bmatrix}1\\1,0\end{bmatrix} \Leftarrow \begin{bmatrix}1\\1\end{bmatrix}$
$S_1 = \begin{bmatrix}\underline{0}\\\underline{0},0\end{bmatrix}$	$S_1 = \begin{bmatrix}\underline{1}\\\underline{1},0\end{bmatrix}$	$S_1 = \begin{bmatrix}\underline{1}\\\underline{0},1\end{bmatrix}$	$S_1 = \begin{bmatrix}\underline{0}\\\underline{1},0\end{bmatrix}$	$S_1 = \begin{bmatrix}\underline{1}\\\underline{0},1\end{bmatrix}$

Table 3-4 First 5 traceback iterations for (3,2,2) decoder example.

These decoded symbols are identical to the corresponding first five n-bit symbols in the input sequence U, i.e. U = [00, 11, 01, 10, 01, 11, 00, 10, 01, 10, 01, 10, 00, 10, 01, 11, 00, 10, 01, 10].

After each traceback iteration, a decoded symbol is released from the decoder and the path metric memory and path memory are updated with the next set of decisions. In the path memory, the oldest backward labels, $(\mathrm{tb}_{2,T}, \mathrm{tb}_{1,T})_x$ are right shifted out as a new set of backward labels B_x are shifted into the left most elements. At the same time each of the corresponding path metric elements are updated with values computed for the present stage time. Once a new set of labels are shifted into the path memory it is now full again and the next decoded symbol is produced by traceback.

The process to decode the remaining symbols is identical, and thus will not be illustrated. This process of updating one stage of the path memory and performing traceback is repeated until all N symbols of the message are decoded.

3.2.7 (n, 1, m) Traceback Example

Given the input sequence U=[0,1,1,0,1,1,0,0,1,0,1,0,0,0,1,1,0,0,1,0], the (2,1,3) encoder in figure 3-19 generates the encoded sequence V =[00,11,10,10,11,01,10,00,00,01,00,10,11,11,11,10,10,00,00,01,11,11,00].

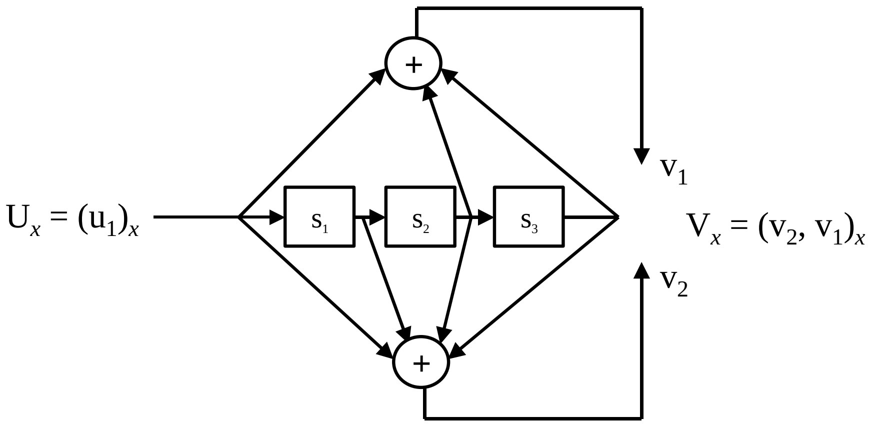


Figure 3-19 (2,1,3) encoder.

The path memory in the (2,1,3) decoder is of length T. In this case T=5m or fifteen. The decoding operations of updating the path memory are identical to the (3,2,2) method except two paths are compared instead of four paths. The path memory update operation is illustrated in figure 3-20 which shows one state.

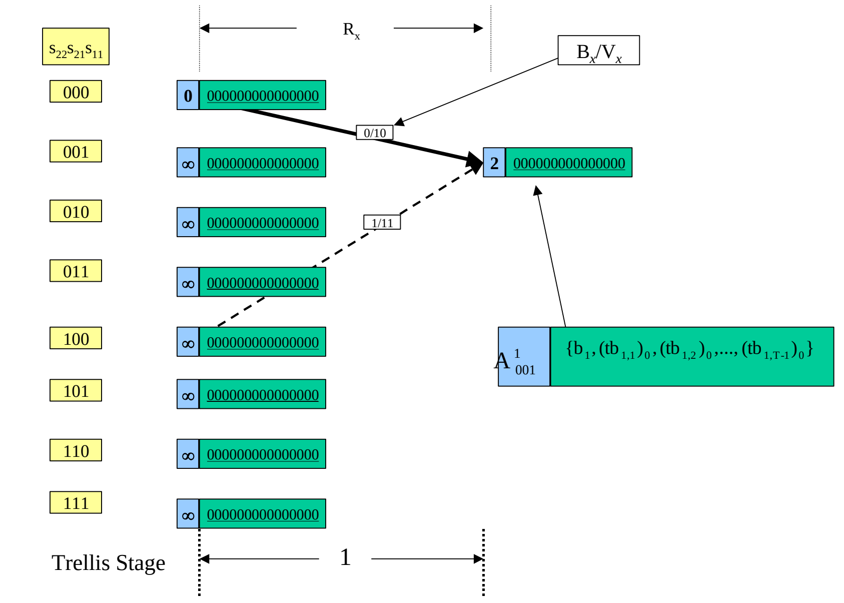


Figure 3-20 (2,1,3) path memory update.

After fifteen stages the path memory is full. The path memory register contents are shown below in figure 3-21.

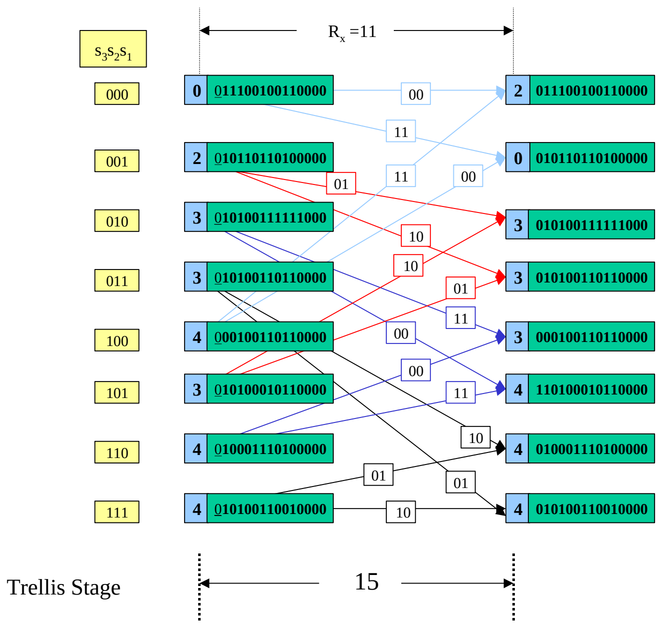


Figure 3-21 (2,1,3) path memory contents at stage 15.

Table 3-5 illustrates a logic table of traceback changes for all states between stage x to stage x-1 for the (2,1,3) traceback process. The dash notation “-“ in table 3-5 means that there is no connection between the states at stage *x* and stage *x*-1.

	S_x (s_3,s_2,s_1)							
S_{x-1}(s_3,s_2,s_1)	000	001	010	011	100	101	110	111
000	0	0	-	-	-	-	-	-
001	-	-	0	0	-	-	-	-
010	-	-	-		0	0	-	-
011	-	-	-		-	-	0	0
100	1	1	-	0	-	-	-	-
101	-	-	1	1	-	-	-	-
110	-	-	-	-	1	1	-	-
111	-	-	-	-	-	-	1	1

Table 3-5 (2,1,3) traceback state table

Once the path memory is full at stage 15, the traceback process commences to determine the first decoded symbol. Given the (2,1,3) example, the state representation is reduced to.

$$\mathrm{S}_x = \left[(\mathrm{s}_1, \mathrm{s}_2, \mathrm{s}_3)_x \right]$$

The process of tracing back one state consists of shifting a backward label, $\mathrm{B}_x = (\mathrm{b}_1)_x$ to the right most element of S_x resulting in S_{x-1} as follows

$$\begin{aligned} \mathrm{S}_{x-1} &= \left[(\mathrm{s}_1, \mathrm{s}_2, \mathrm{s}_3)_x \right] \Leftarrow \left[(\mathrm{b}_1)_x \right] \\ &= \left[(\mathrm{s}_2, \mathrm{s}_3, \mathrm{b}_1)_x \right] \end{aligned}$$

From figure 3-21, the state with the smallest survivor path metric at stage 15 is 001. The backward label read from path memory using the ($tb_{j,i}$) row column address scheme above is B_x = 0. Then at stage 14, a new state is obtained by the traceback mapping function as follows:

$$S_{15} = [1, 0,0] \Leftarrow [0]$$
$$S_{14} = [0, 0,0]$$

Similarly at stage 13, a new state is obtained by traceback by first reading the backward label from path memory pointed to by the row-column address. The backward label returned is B_x=1. A new state at stage 13 can then be traced back using the trace back mapping function as follows:

$$S_{14} = [0, 0,0] \Leftarrow [1]$$
$$S_{13} = [0, 0,1]$$

Like the (3,2,2) traceback process, this is identical for all traceback stages until stage 1 is reached. Table 3-6 illustrates the traceback process for five traceback iterations. Once the traceback register reaches stage 1, a decision is made on selecting the first decoded symbol.

$$U'_x = (s_{k,1}, s_{k\text{-}1,1}, \ldots s_{1,1})_x$$
$$U'_x = (s_1)_x$$

For example after the 1st traceback iteration, the leftmost bit (s_1) of the traceback register corresponds to the decoded symbol, as shown in table 3-6.

$$U'_1 = (s_1)$$
$$U'_1 = (0)$$

1st traceback	2nd traceback	3rd traceback	4th traceback	5th traceback
$S_{15} = [1,0,0] \Leftarrow [0]$	$S_{15} = [1,1,0] \Leftarrow [0]$	$S_{15} = [0,1,1] \Leftarrow [0]$	$S_{15} = [0,0,1] \Leftarrow [1]$	$S_{15} = [1,0,0] \Leftarrow [1]$
$S_{14} = [0,0,0] \Leftarrow [1]$	$S_{14} = [1,0,0] \Leftarrow [0]$	$S_{14} = [1,1,0] \Leftarrow [0]$	$S_{14} = [0,1,1] \Leftarrow [0]$	$S_{14} = [0,0,1] \Leftarrow [1]$
$S_{13} = [0,0,1] \Leftarrow [0]$	$S_{13} = [0,0,0] \Leftarrow [1]$	$S_{13} = [1,0,0] \Leftarrow [0]$	$S_{13} = [1,1,0] \Leftarrow [0]$	$S_{13} = [0,1,1] \Leftarrow [0]$
$S_{12} = [0,1,0] \Leftarrow [1]$	$S_{12} = [0,0,1] \Leftarrow [0]$	$S_{12} = [0,0,0] \Leftarrow [1]$	$S_{12} = [1,0,0] \Leftarrow [0]$	$S_{12} = [1,1,0] \Leftarrow [0]$
$S_{11} = [1,0,1] \Leftarrow [0]$	$S_{11} = [0,1,0] \Leftarrow [1]$	$S_{11} = [0,0,1] \Leftarrow [0]$	$S_{11} = [0,0,0] \Leftarrow [1]$	$S_{11} = [1,0,0] \Leftarrow [0]$
$S_{10} = [0,1,0] \Leftarrow [0]$	$S_{10} = [1,0,1] \Leftarrow [0]$	$S_{10} = [0,1,0] \Leftarrow [1]$	$S_{10} = [0,0,1] \Leftarrow [0]$	$S_{10} = [0,0,0] \Leftarrow [1]$
$S_9 = [1,0,0] \Leftarrow [1]$	$S_9 = [0,1,0] \Leftarrow [0]$	$S_9 = [1,0,1] \Leftarrow [0]$	$S_9 = [0,1,0] \Leftarrow [1]$	$S_9 = [0,0,1] \Leftarrow [0]$
$S_8 = [0,0,1] \Leftarrow [1]$	$S_8 = [1,0,0] \Leftarrow [1]$	$S_8 = [0,1,0] \Leftarrow [0]$	$S_8 = [1,0,1] \Leftarrow [0]$	$S_8 = [0,1,0] \Leftarrow [1]$
$S_7 = [0,1,1] \Leftarrow [0]$	$S_7 = [0,0,1] \Leftarrow [1]$	$S_7 = [1,0,0] \Leftarrow [1]$	$S_7 = [0,1,0] \Leftarrow [0]$	$S_7 = [1,0,1] \Leftarrow [0]$
$S_6 = [1,1,0] \Leftarrow [1]$	$S_6 = [0,1,1] \Leftarrow [0]$	$S_6 = [0,0,1] \Leftarrow [1]$	$S_6 = [1,0,0] \Leftarrow [1]$	$S_6 = [0,1,0] \Leftarrow [0]$
$S_5 = [1,0,1] \Leftarrow [1]$	$S_5 = [1,1,0] \Leftarrow [1]$	$S_5 = [0,1,1] \Leftarrow [0]$	$S_5 = [0,0,1] \Leftarrow [1]$	$S_5 = [1,0,0] \Leftarrow [1]$
$S_4 = [0,1,1] \Leftarrow [0]$	$S_4 = [1,0,1] \Leftarrow [1]$	$S_4 = [1,1,0] \Leftarrow [1]$	$S_4 = [0,1,1] \Leftarrow [0]$	$S_4 = [0,0,1] \Leftarrow [1]$
$S_3 = [1,1,0] \Leftarrow [0]$	$S_3 = [0,1,1] \Leftarrow [0]$	$S_3 = [1,0,1] \Leftarrow [1]$	$S_3 = [1,1,0] \Leftarrow [1]$	$S_3 = [0,1,1] \Leftarrow [0]$
$S_2 = [1,0,0] \Leftarrow [0]$	$S_2 = [1,1,0] \Leftarrow [0]$	$S_2 = [0,1,1] \Leftarrow [0]$	$S_2 = [1,0,1] \Leftarrow [1]$	$S_2 = [1,1,0] \Leftarrow [1]$
$S_1 = [\underline{0},0,0]$	$S_1 = [\underline{1},0,0]$	$S_1 = [\underline{1},1,0]$	$S_1 = [\underline{0},1,1]$	$S_1 = [\underline{1},0,1]$

Table 3-6 First 5 traceback iterations for (2,1,3) decoder example.

These decoded symbols are identical to the corresponding first five n-bit symbols in the input sequence U, i.e. U=[0,1,1,0,1,1,0,0,1,0,1,0,0,0,1,1,0,0,1,0],.

As in the (3,2,2) case, a decoded symbol is released from the decoder after each traceback iteration, and the path metric memory and path memory are updated with the next set of decisions. In the path memory, the oldest backward labels, $(\mathrm{tb}_{1,\mathrm{T}})_x$ are right shifted out as a new set of backward labels B_x are shifted into the left most elements. At the same time each of the corresponding path metric elements are updated with values computed for the present stage time. Once a new set of labels are shifted in to the path memory it is now full again and the next decoded symbol is produced by traceback.

The process to decode the remaining symbols is identical, and thus will not be illustrated. This process of updating one stage of the path memory and performing traceback is repeated until all N symbols of the message are decoded.

4. Decoder Architectures (Forward and Backward)

This section documents the known architectural implementations for forward and backward label decoders.

4.1 Structure of Forward & Backward Label Decoders

The necessary building blocks to construct a Viterbi decoder was outlined in section 2-5, with the block diagram repeated here in figure 4-1.

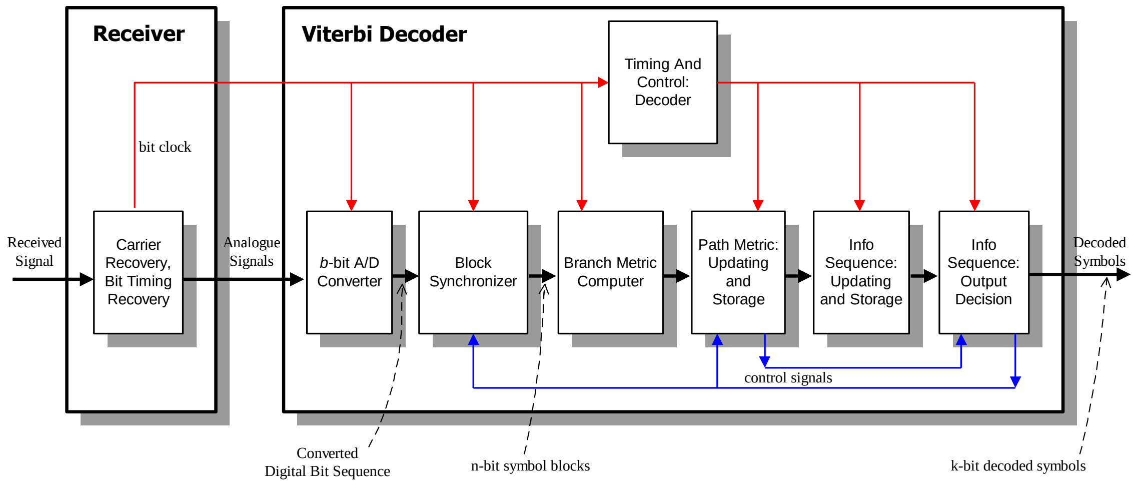

Receiver
Viterbi Decoder
Timing And Control: Decoder
bit clock
Received Signal
Carrier Recovery, Bit Timing Recovery
Analogue Signals
b-bit A/D Converter
Block Synchronizer
Branch Metric Computer
Path Metric: Updating and Storage
Info Sequence: Updating and Storage
Info Sequence: Output Decision
Decoded Symbols
control signals
Converted Digital Bit Sequence
n-bit symbol blocks
k-bit decoded symbols

Figure 4-1 Structure of a practical Viterbi decoder.

As described from the algorithms in section 3, Viterbi decoders can be classified into either forward label or backward label decoders. The register exchange technique is an example of a forward label technique, and the traceback decoder an example of the backward label technique. Viterbi decoders can thus be broadly classified by the type path memory used. Figure 4-1 shows all the blocks for a general Viterbi decoder. However, the following blocks differ between a forward and backward label decoder.

- Path Metric Unit
- Information Sequence : Updating and Storage Unit (Path Memory)
- Information Sequence : Output Decision Unit
- Timing and Control Unit

4.1.1 Path Metric Unit

The path metric unit computes path metrics in an identical fashion for both forward and backward label decoders, however the branch label output selection circuit differs in that the path metric unit produces either forward branch labels or backward branch labels, depending on the path memory implementation used.

4.1.2 Info Sequence : Updating and Storage Unit

The info. Sequence: updating and storage unit, otherwise known as the path memory unit takes the branch label output from the path metric unit and stores the branch labels in the path memory. If the Viterbi decoder uses a traceback path memory, the path memory takes the backward branch labels from the path metric unit and stores them in path memory. Similarly if a register exchange decoder is used, the path memory takes the forward labels from the path metric unit and stores them in path memory. Once the path memory registers are full, the oldest branch labels are used by the output decision unit.

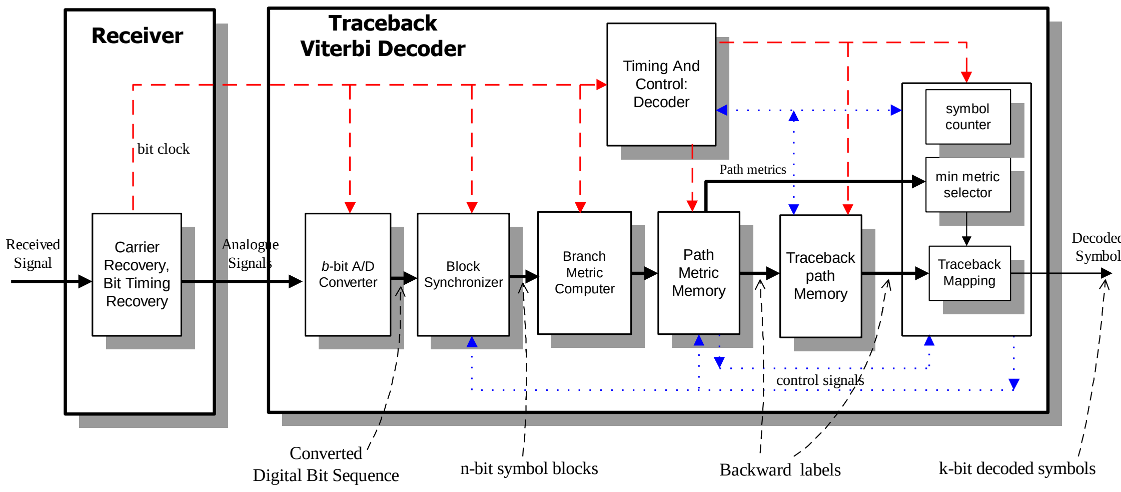
4.1.3 Info Sequence : Output Decision Unit

The output decision unit differs for both forward and backward label decoders. In forward label decoders, the output decision block is a simple design, consisting of a comparator, a multiplexor, and a symbol counter. The operation of a forward label output decision block is as follows. First, the comparator is used to determine the state with the minimum path metric, this state is then used to select one of 2^M forward labels as the decoded symbol. The symbol counter is used to track the number of symbols decoded in the pseudo block. Once all N symbols have been decoded, the output decision unit outputs a control signal to terminate the current decoding block. In traceback decoders, the output decision unit is more complex. Similar to the forward label case, a comparator is used to determine the minimum path metric state which is then used to select a backward label. The traceback mapping circuit then performs the traceback process using the backward label and the selected state. Once the traceback process reaches stage one, the circuit releases a decoded symbol. Again a symbol counter is used to track the number of decoded symbols, and terminates when all N symbols have been decoded.

4.1.4 Timing and Control Unit

The Timing and Control unit differs for both forward and backward label decoders as it needs to provide different synchronization and control signals at different times for each the blocks in the decoder.

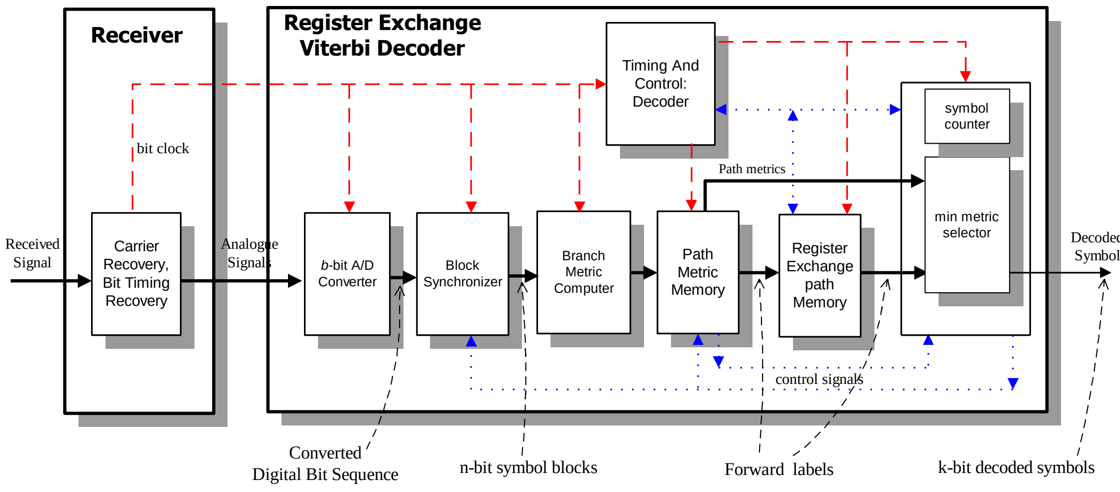
The block diagram of figure 4-1 is now redrawn in figures 4-2, and 4-3, illustrating the modified blocks for both backward and forward label decoders respectively.



Note 1 : Blue "Dotted" Lines denote Control Signals

Note 2 : Red "Dashed" Lines denote Timing or Clock Signals

Figure 4-1 Backward label Viterbi Decoder Block Diagram.



Note 1 : Blue "Dotted" Lines denote Control Signals

Note 2 : Red "Dashed" Lines denote Timing or Clock Signals

Figure 4-2 Forward label Viterbi Decoder Block Diagram.

4.2 Register Exchange Vs Traceback

In general, the register exchange approach to survivor path memory management is quite simple from a functional point of view, but suffers from the disadvantage that every register must be read modified and rewritten for every branch label written to path memory. If a fast VLSI decoder with a RE path memory is used, all of the exchanging must take place simultaneously, leading to a hardware intensive implementation. To perform the register exchange, some double buffering may be required for the exchange operations, [13], and thus the RE method requires a lot area in a VLSI implementation. In addition, the RE method can suffer from a large power consumption, due to the read modify write operations between registers, [7], [13].

As outlined in section 3, there is an initial latency equivalent to a multiple of the path memory length T, for the register exchange technique to produce the first decoded symbol. That is, after the path memory is full, the first set of branch labels are shifted out of path memory and into the output decision unit. One branch label is then selected as the decoded symbol. The remaining decoded symbols are then produced at every trellis stage. Traceback decoder implementations do not require read modify write operations of the register exchange but use simple shift register operations, hence the flip-flop toggle rate is lower and the power consumption is considerably less. This is one of the primary reasons that traceback Viterbi decoders are preferred to RE Viterbi decoders, as typically the application of these decoders are in low power applications. However the decoding operations for traceback technique are more complicated than register exchange. In addition, the initial decoding latency is worse than the register exchange technique, i.e. a latency of to fill the path memory, plus an additional traceback latency before a decoded symbol is released. Once the path memory is full, each successive symbol produced suffers a decoding latency of a multiple of T stages, refer to section 6.7.3 for more details.

4.3 Traceback Architectures

Over the last thirty years, several papers have been published on implementation improvements for traceback decoding. This research is on two fronts; one is on researching reductions to the path memory storage requirements, and the other field of study concentrates on reducing the decoding latency inherent with traceback. The literature review that follows, [8],[9],[10],[11],[13] broadly classifies the traceback architectures for use in a Viterbi decoder.

Note: The notation in the following sub-sections is that taken from the literature and not to be confused with notation used in this text.

4.3.1 One Traceback Pointer Architecture

The literature divides the survivor path memory into three distinct regions, with each region performing a specific operation, as shown in figure 4-3.

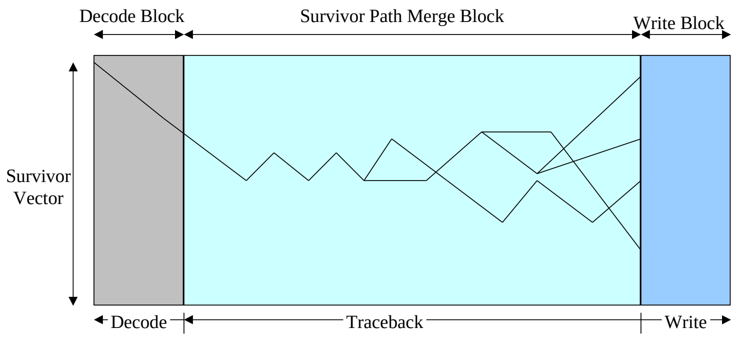


Figure 4-3 Operations on a Traceback Path Memory Unit

The survivor path memory is composed of write, merge, and decode blocks. The write block continuously writes 2^M backward labels and merge block merges the surviving paths with high probability. The length of the merge block is denoted as *D*, the decoding depth. The last part of the survivor memory is the decode block in which a decoding decision is made. These operations are as follows:

Writing New Data: At every trellis stage, the path metric unit generates 2^M k-bit backward labels, which needs to be stored into the path memory. For each new backward label written, the write-pointer moves forward (or left to right) through the path memory as the path metric memory moves on to each new stage in the code trellis.

Traceback Read: Once the path memory is full, the path memory unit, needs to start traceback in order to produce a decoded symbol. This is one of two read operations performed on the path memory unit, but in this case no decoding is done on the bits read from path memory, [9],[10].

Decode Read: Functionally the decode read operation is identical to the traceback read operation, except that pointers read from memory are decoded to form the output symbol, [9],[10]. A decode read front is started from a traceback read front once the traceback read front has reached the first trellis stage. The decode-read operation frees up memory space for use by the write operation. Since the memory in hardware is limited, the write operation must either fill the space freed up by the decode-read operation or the path memory contents are shift down one stage so that the write operation can fill the last path memory address.

On average there must be one decode-read operation for every write operation to path memory. However practical traceback architectures should achieve a throughput which is matched to the input data rate. To satisfy this condition, the traceback operation is faster then the write operation, or multiple read operations are used. The traceback architecture discussed in this project is based on the one traceback pointer architecture. The traceback pointer operates at the same speed as the write pointer and once the path memory is full there is one decode-read operation for every write operation to path memory.

A useful term discussed in the literature is the traceback recursion rate (*TRR*). The *TRR* is the number of traceback recursions per write clock cycle, [10], [11]. In this architecture, to ensure continuous operation, the traceback operations must be completed within the time to fill the write region. Therefore, the read clock is several times faster than the write clock. The required decision memory length *SL* as a function of the *TRR* is

$$SL = \left(\frac{TRR+1}{TRR-1}\right)D \quad for\, TRR > 1$$

in [11]. In this case, the required number of memory banks is (*TRR*+1), where the length of each bank is $\frac{D}{TRR-1}$. The latency of this architecture, including the LIFO buffer operation (bit order reversal), is $\left(\frac{TRR+1}{TRR-1}\right)D$.

The one pointer architecture is reasonably simple. However it requires a read operation clock much faster than the input data clock in order to achieve a throughput rate that matches the input symbol rate. A solution to this problem is to run multiple traceback process concurrently as described in the section 4.3.2

4.3.2 The "k- pointer" Traceback Architecture

The *k*-pointer traceback architecture is named from the fact that it supports traceback of *k* survivor paths concurrently. When the write operation has been completed at a bank, a traceback operation is started and this operation continues independently of other read operations. Assuming that each traceback stream is iterated at the minimum *TRR* (=1), a *k*-pointer traceback architecture requires 2*k* independent memories each of length $\frac{D}{k-1}$, equivalent to a total memory length of *SL*

$$SL = \left(\frac{2k}{k-1}\right)D \quad for\, k > 1$$

as addressed in [17]. In k-pointer traceback architectures, the latency including the LIFO buffer (bit order reversal) is $\left(\frac{2k}{k-1}\right)D$. As the memory length is reduced, the number of memory partitions is increased and hence the peripheral overhead logic is increased.

4.3.3 The Trace Delete Method (TDM)

This is a survivor memory management method called the trace delete method (*TDM*) proposed in [11] to provide a solution for reducing latency and the control logic requirements for Viterbi decoding.

When elements are written to survivor memory, there exists some states that are not connected to current states. The main idea of the *TDM* is to delete recursively the paths coming into these states. To realize this algorithm, the information about the existence of a path and the path deleting method are taken into account.

The existence of a path is a function of the next state and the decision vector (backward label). Therefore the path existence information can be extracted for the TDM from the decision vector and a given state in the traceback.

To distinguish each path, a path is denoted which starts from state S_j at time j with input *k as* $P_{S_j}^k$.If we know the state $S_{n,j}$ at time j and the state $S_{n',j+1}$ at time j+1, the existence of a path $P_{S_j}^k$ can be inferred as

$$g : P_{S_{n,j}}^{a_{j+1}} \leftarrow \left(S_{n,j}, S_{n',j+1}\right).$$

If a path exists it is tagged with a 1, i.e. $P_{S_n}^k$ *is* 1 otherwise it is tagged with a 0. For paths tagged with a 0, a path deleting signal D_S is asserted. The path deleting signal must be transferred to the possible previous states. The relationship between the present state and the previous state depends on the structure of the trellis. Therefore possible states are known by the structure of the trellis. If the state receives a path deleting signal $D_{s,}$ the path relating to this signal must be deleted. The path delete operation is represented as:

$$P_{S_{n,j-1}}^{a_j} \leftarrow \quad if \sum_{all\, a_j+1} P_{S_{n,j}}^{a_j+1} = 0$$

where $S_{n,-j-1}$ is obtained from the structure of the trellis and a_j can be obtained from $S_{n,j}$.

A path at time j-1 before deletion is denoted as $P'^{a_j}_{S_{n,j-1}}$. The path existence information $P_{S_{n,j-1}}^{a_j}$ is updated according to the following equation

$$P_{S_{n,j-1}}^{a_j} \leftarrow \quad D_s \; and \; P'^{a_j}_{S_{n,j-1}}$$

The TDM method in [11] demonstrates that it can outperform the existing traceback technique in terms of latency and the required number of memory elements.

4.3.3 Systolic Arrays

Systolic arrays are arrays of processing units, which are connected to a small number of nearest neighbors in a mesh-like topology. The processing units perform a sequence of operations on data that flows between them. Generally the operations will be the same in each unit, with each unit performing an operation (or small number of operations) on a data item and them passing it on to its neighbor.

A particularly clever solution to the traceback problem for the Viterbi algorithm was shown using a systolic array, [8]. The systolic array keeps the high throughput for the traceback operations by launching a new traceback operation into the pipeline on every cycle. The pipeline has two components, propagating through the stages:

- the traceback states
- the decision vectors (backward labels read from memory)

While the decision vectors move one step ahead on every clock cycle, the ML state stages actually jump ahead two steps at a time. That is on the next clock cycle, decision vector for stage *x* moves one stage, but the ML traceback state jumps forward to the decision vector for stage (*x*-1). Again the previous state is calculated and as the state travels into the pipeline, it moves closer to the final traceback stage. The pipeline is sized such that the traceback state will reach the final state where a symbol decision is made at the end of the pipeline, and at that point the decision vector bit can be released as a true decoded symbol.

5. Storage Efficient Path Memory Design

This chapter is focused on the design of a new type of path memory that can operate in Viterbi decoders without loss of decoding performance. This memory is a storage efficient version of the conventional backward label path memory unit. The concept is introduced by two theorems and a modified traceback algorithm to manage the novel technique.

5.1 Code Theorems

The size of the path memory required for either ML decoding or truncated decision decoding is presented in section 3. Further research in Viterbi decoding [4], [5], [6], shows that it is possible to reduce path memory requirements without loss of decoding performance. A property of the trellis diagram, consequent on the shift register structure of the encoder and linking elements of the forward and backward labels of branches of a path through the trellis, is stated and derived below for both the general (n,k,m) case and the simpler, more popular, (n,1,m) case. The encoder in the latter, having only one shift register (length m), has thus only one forward $\left(F_x = f_{1,x}\right)$, and one backward, $\left(B_x = b_{1,x}\right)$, label element per branch, [4].

5.1.1 (n, k, m) Code Theorem

In a valid path through the trellis diagram of a (n, k, m) convolutional code each of the jth elements, $f_{j,x}$, of the forward label F_x of the branch in the path at trellis stage x is identical to the corresponding jth element, $b_{j,x+m_j}$ of the backward label B_{x+m_j} of the branch at the stage (x+m_j) of the path, where m_j, j = 1,...k, are the lengths of the corresponding k shift registers of the encoder, [4].

That is $f_{j,x} = b_{j,x+m_j}$

and thus $F_x = (f_k, f_{k-1}, \dots f_1)_x = (b_{k,x+m_k}, b_{k-1,x+m_{k-1}}, \dots b_{1,x+m_1})$

5.1.2 Proof of (n, k, m) Code Theorem

Any element $f_{j,x}$ in the forward branch label F_x of a branch emanating from a state S_x at stage x is identical to the j^{th} element or bit, $u_{j,x}$ of the input symbol U_x. This same bit is shifted out of the j^{th} register m_j stages later, on the state transition from $S_{x+}m_j$ to $S_{x+}m_{j+1}$. Thus it is also identical to backward label elements $b_{j,x}+m_j$ of all backward labels $B_{x+}m_j$, of branches at stages stage $x+m_j$ which are on valid trellis paths from the originating state at stage x, [4].

Since this is true for the pair of elements $f_{j,x}$ and $b_{j,x}+m_j$, of branch labels F_x and $B_{x+}m_j$ respectively, then it is true for all pairs of elements $f_{j,x}$ and $b_{j,x}+m_j$, where j=1,2, … k, connected by valid paths through the trellis. In addition, since no one specific state S_x of the 2^M possible states at any stage x has been indicated, the property is true for valid trellis paths emanating from all states, [4].

5.1.3 (n, 1, m) Code Theorem

In a valid path through the trellis diagram of a (n,1,m) convolutional code the forward label F_x of the branch in the path at trellis stage x is identical to the backward label B_{x+m} of the branch at the stage (x+m) of the path, [4]. That is: $F_x = B_{x+m}$

5.1.4 Proof of (n, 1,m) Code Theorem

This follows the same line of argument for the (n, k, m) case, [4].

5.2 Novel Path Memory Traceback Algorithm

Significant savings in storage can be achieved in Viterbi decoders using a backward label path memory. This can be implemented by exploiting the trellis property outlined in the (n, k, m), and the (n, 1, m) code theorems. In order to take advantage of this property, the conventional backward label traceback algorithm needs a small modification to realize the novel path memory.

As outlined in sections 5.1.1 through 5.1.4, (n, k, m) convolutional codes have the property where every j^{th} element of a forward label branch and a backward label branch in a trellis path are separated by a distance equal to the memory order of the encoder's j^{th} shift register. Any j^{th} element of a backward label branch is equivalent to the corresponding j^{th} element in the forward label branch m_j stages earlier. The converse of this property can be used at any stage *x*, during the traceback process of a backward label decoder. That is, in any stage *x*, a decoder decision can be made on the j^{th} input element of the n-bit symbol, which stretches, back to stage $x+ m_j$. Thus for the j^{th} element of the encoder shift register there is a saving of m_j stages. As the backward label at stage *x* is equivalent to the forward label, m_j stages previous, the storage requirements can be reduced for this m_j length shift register without loss of decoding performance. Consider the 3-stage paths *QPCD* and *NPCD* highlighted in the (3,2,2) trellis in figure 5-1, where the maximal memory order is ($m_j = m_2 = 2$). In any stage *x*, a decoder decision can to be made on the j^{th} input element of the n-bit symbol, which stretches, back to stage $x+ m_j$. Thus for the j^{th} element of the encoder shift register there is a saving of m_j stages.

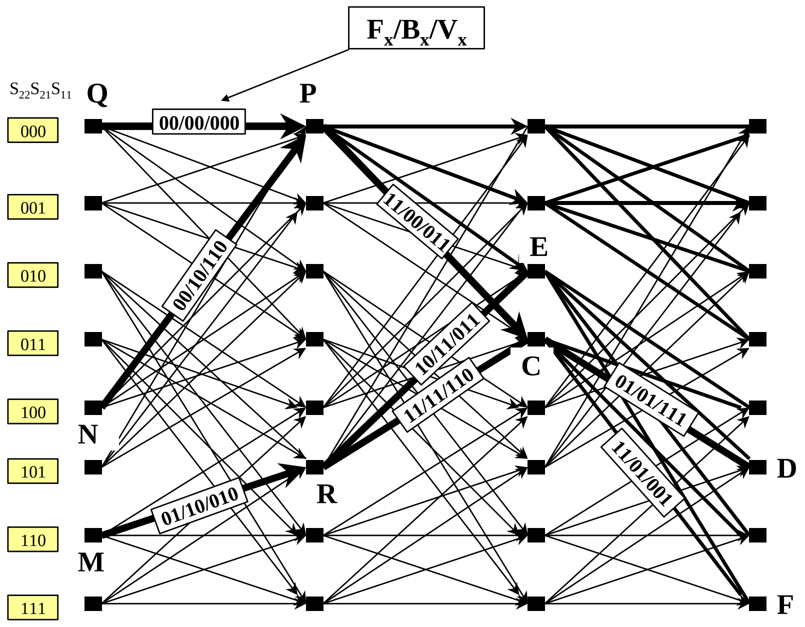


Figure 5-1. (3,2,2) Trellis Structure.

Any trace forward of 2 stages ($m_j = m_2 = 2$), from a node in stage *x*, say node *Q*, to a node in stage *x+2*, (= m_j+2), will <u>always</u> arrive at a node, e.g. *D*, such that the second, (i.e. the jth element), element of the forward branch label of the branch *QP* in stage *x* is identical to the second element of the backward branch label of the branch *CD* in stage *x+2*, (i.e. both are 0 in this case). Thus for a decoder decision to be made on the 2nd input element of the 2-bit input symbol in stage *x* of the encoder, when using a Viterbi algorithm implementation based on a backward label traceback scheme, a decoder survivor path memory stretching back only to stage *x*+2 is sufficient. Similarly, any trace forward of 1 stage ($m_j = m_1 = 1$), from a node in stage *x*, say node *M*, to a node in stage *x*+1, will <u>always</u> arrive at a node, e.g. *C*, such that the first element of the forward branch label of the branch *MR* in stage *x* is identical to the first element of the backward branch label of the branch *RC* in stage *x*+1, (i.e. both are 1 in this case).This trellis property can be exploited at any stage *x*, during the traceback process of a decoder. That is, in any stage *x*, a decoder decision can be made on the jth input element of the n-bit symbol, which stretches, back to stage x+ m_j. Thus for the j^{th} element of the encoder shift register there is a saving of m_j stages.

In the conventional traceback process, the traceback process includes a traceback register with the same generator polynomial as the encoding shift register. Each of the k traceback shift registers are of length m_j. During traceback mapping, the traceback shift register operations work right to left by shifting backward labels from path memory, and producing a predecessor state on every clock cycle back along the surviving path. In the conventional case the left most elements of S_x are shifted out to produce a decoded symbol, when stage 1 of the path memory is reached. However when using the novel technique, the same elements are available m_j stages earlier, and the decoded symbol elements are typically resident in the right most elements of the traceback register.

5.2.1 Example: (n, k, m) Efficient Traceback

For the (3,2,2) encoder each of the parallel shift registers are of different length, in this case there are two parallel shift registers, one of length one (s_{11}), and the other of length two (s_{21},s_{22}) as shown in figure 3-3. The second element (b_2) of a backward label branch is equivalent to the corresponding second element (f_2) in the forward label branch two stages earlier. Once the traceback pointer reaches stage three this second element (b_2) of a backward label branch is also equivalent to the second element (u'_2) of decoded symbol at stage one, i.e. two stages earlier. Similarly the first element (b_1) of a backward label branch is equivalent to the corresponding first element (f_1) in the forward label branch one stage earlier. In other words once the traceback pointer reaches stage two, the first element (b_1), of a backward label branch is equivalent to the first element (u'_1) of decoded symbol at stage one, i.e. one stage earlier. The traceback example in section 3 for (3,2,2) decoder is now repeated below in table 5-1 showing the first five traceback iterations to realize the storage efficient path memory. Note: the decoded symbol elements are underlined at stages 1 and 2 of each traceback iteration.

1st traceback	2nd traceback	3rd traceback	4th traceback	5th traceback
$S_9 = \begin{bmatrix}0\\1,0\end{bmatrix} \Leftarrow \begin{bmatrix}1\\1\end{bmatrix}$	$S_9 = \begin{bmatrix}1\\0,1\end{bmatrix} \Leftarrow \begin{bmatrix}0\\0\end{bmatrix}$	$S_9 = \begin{bmatrix}0\\1,0\end{bmatrix} \Leftarrow \begin{bmatrix}1\\1\end{bmatrix}$	$S_9 = \begin{bmatrix}0\\0,1\end{bmatrix} \Leftarrow \begin{bmatrix}0\\0\end{bmatrix}$	$S_9 = \begin{bmatrix}0\\1,0\end{bmatrix} \Leftarrow \begin{bmatrix}0\\1\end{bmatrix}$
$S_8 = \begin{bmatrix}1\\0,1\end{bmatrix} \Leftarrow \begin{bmatrix}0\\0\end{bmatrix}$	$S_8 = \begin{bmatrix}0\\1,0\end{bmatrix} \Leftarrow \begin{bmatrix}1\\1\end{bmatrix}$	$S_8 = \begin{bmatrix}1\\0,1\end{bmatrix} \Leftarrow \begin{bmatrix}0\\0\end{bmatrix}$	$S_8 = \begin{bmatrix}0\\1,0\end{bmatrix} \Leftarrow \begin{bmatrix}1\\1\end{bmatrix}$	$S_8 = \begin{bmatrix}0\\0,1\end{bmatrix} \Leftarrow \begin{bmatrix}0\\0\end{bmatrix}$
$S_7 = \begin{bmatrix}0\\1,0\end{bmatrix} \Leftarrow \begin{bmatrix}0\\1\end{bmatrix}$	$S_7 = \begin{bmatrix}1\\0,1\end{bmatrix} \Leftarrow \begin{bmatrix}0\\0\end{bmatrix}$	$S_7 = \begin{bmatrix}0\\1,0\end{bmatrix} \Leftarrow \begin{bmatrix}1\\1\end{bmatrix}$	$S_7 = \begin{bmatrix}1\\0,1\end{bmatrix} \Leftarrow \begin{bmatrix}0\\0\end{bmatrix}$	$S_7 = \begin{bmatrix}0\\1,0\end{bmatrix} \Leftarrow \begin{bmatrix}1\\1\end{bmatrix}$
$S_6 = \begin{bmatrix}0\\0,1\end{bmatrix} \Leftarrow \begin{bmatrix}1\\0\end{bmatrix}$	$S_6 = \begin{bmatrix}0\\1,0\end{bmatrix} \Leftarrow \begin{bmatrix}0\\1\end{bmatrix}$	$S_6 = \begin{bmatrix}1\\0,1\end{bmatrix} \Leftarrow \begin{bmatrix}0\\0\end{bmatrix}$	$S_6 = \begin{bmatrix}0\\1,0\end{bmatrix} \Leftarrow \begin{bmatrix}1\\1\end{bmatrix}$	$S_6 = \begin{bmatrix}1\\0,1\end{bmatrix} \Leftarrow \begin{bmatrix}0\\0\end{bmatrix}$
$S_5 = \begin{bmatrix}1\\1,0\end{bmatrix} \Leftarrow \begin{bmatrix}1\\1\end{bmatrix}$	$S_5 = \begin{bmatrix}0\\0,1\end{bmatrix} \Leftarrow \begin{bmatrix}1\\0\end{bmatrix}$	$S_5 = \begin{bmatrix}0\\1,0\end{bmatrix} \Leftarrow \begin{bmatrix}0\\1\end{bmatrix}$	$S_5 = \begin{bmatrix}1\\0,1\end{bmatrix} \Leftarrow \begin{bmatrix}0\\0\end{bmatrix}$	$S_5 = \begin{bmatrix}0\\1,0\end{bmatrix} \Leftarrow \begin{bmatrix}1\\1\end{bmatrix}$
$S_4 = \begin{bmatrix}1\\0,1\end{bmatrix} \Leftarrow \begin{bmatrix}0\\0\end{bmatrix}$	$S_4 = \begin{bmatrix}1\\1,0\end{bmatrix} \Leftarrow \begin{bmatrix}1\\1\end{bmatrix}$	$S_4 = \begin{bmatrix}0\\0,1\end{bmatrix} \Leftarrow \begin{bmatrix}1\\0\end{bmatrix}$	$S_4 = \begin{bmatrix}0\\1,0\end{bmatrix} \Leftarrow \begin{bmatrix}0\\1\end{bmatrix}$	$S_4 = \begin{bmatrix}1\\0,1\end{bmatrix} \Leftarrow \begin{bmatrix}0\\0\end{bmatrix}$
$S_3 = \begin{bmatrix}0\\1,0\end{bmatrix} \Leftarrow \begin{bmatrix}1\\1\end{bmatrix}$	$S_3 = \begin{bmatrix}1\\0,1\end{bmatrix} \Leftarrow \begin{bmatrix}0\\0\end{bmatrix}$	$S_3 = \begin{bmatrix}1\\1,0\end{bmatrix} \Leftarrow \begin{bmatrix}1\\1\end{bmatrix}$	$S_3 = \begin{bmatrix}0\\0,1\end{bmatrix} \Leftarrow \begin{bmatrix}1\\0\end{bmatrix}$	$S_3 = \begin{bmatrix}0\\1,0\end{bmatrix} \Leftarrow \begin{bmatrix}0\\1\end{bmatrix}$
$S_2 = \begin{bmatrix}1\\\underline{0},1\end{bmatrix} \Leftarrow \begin{bmatrix}1\\0\end{bmatrix}$	$S_2 = \begin{bmatrix}0\\\underline{1},0\end{bmatrix} \Leftarrow \begin{bmatrix}1\\1\end{bmatrix}$	$S_2 = \begin{bmatrix}1\\\underline{0},1\end{bmatrix} \Leftarrow \begin{bmatrix}0\\0\end{bmatrix}$	$S_2 = \begin{bmatrix}1\\\underline{1},0\end{bmatrix} \Leftarrow \begin{bmatrix}1\\1\end{bmatrix}$	$S_2 = \begin{bmatrix}0\\\underline{0},1\end{bmatrix} \Leftarrow \begin{bmatrix}1\\0\end{bmatrix}$
$S_1 = \begin{bmatrix}1\\1,0\end{bmatrix} \Leftarrow \begin{bmatrix}\underline{0}\\0\end{bmatrix}$	$S_1 = \begin{bmatrix}1\\0,1\end{bmatrix} \Leftarrow \begin{bmatrix}\underline{1}\\0\end{bmatrix}$	$S_1 = \begin{bmatrix}0\\1,0\end{bmatrix} \Leftarrow \begin{bmatrix}\underline{1}\\1\end{bmatrix}$	$S_1 = \begin{bmatrix}1\\0,1\end{bmatrix} \Leftarrow \begin{bmatrix}\underline{0}\\0\end{bmatrix}$	$S_1 = \begin{bmatrix}1\\1,0\end{bmatrix} \Leftarrow \begin{bmatrix}\underline{1}\\1\end{bmatrix}$

Table 5-1 First 5 efficient traceback iterations for (3,2,2) decoder.

5.2.2 Example: (n, 1, m) Efficient Traceback

Given the (2,1,3) traceback example from section 3, the traceback process for the storage efficient path memory is reduced by (m_j =3) stages in order to decode the input symbol. That is at any stage during traceback, the backward label B_x, is identical to the corresponding forward label branch F_x (m_j =3) stages earlier. Once the traceback pointer reaches stage four, the backward label branch is also equivalent to the decoded symbol at stage one, i.e. three stages earlier at stage 1. The traceback example in section 3 for (2,1,3) decoder is now repeated below in table 5-2, and shows how the backward label B_x= (b_1) at stage 4 produces the decoded symbol U'_x= earlier than in the conventional case. Again the decoded symbol elements are underlined at stage 1.

1st traceback	2nd traceback	3rd traceback	4th traceback	5th traceback
$S_{12} = [1,0,0] \Leftarrow [0]$	$S_{12} = [1,1,0] \Leftarrow [0]$	$S_{12} = [0,1,1] \Leftarrow [0]$	$S_{12} = [0,0,1] \Leftarrow [1]$	$S_{12} = [1,0,0] \Leftarrow [1]$
$S_{11} = [0,0,0] \Leftarrow [1]$	$S_{11} = [1,0,0] \Leftarrow [0]$	$S_{11} = [1,1,0] \Leftarrow [0]$	$S_{11} = [0,1,1] \Leftarrow [0]$	$S_{11} = [0,0,1] \Leftarrow [1]$
$S_{10} = [0,0,1] \Leftarrow [0]$	$S_{10} = [0,0,0] \Leftarrow [1]$	$S_{10} = [1,0,0] \Leftarrow [0]$	$S_{10} = [1,1,0] \Leftarrow [0]$	$S_{10} = [0,1,1] \Leftarrow [0]$
$S_9 = [0,1,0] \Leftarrow [1]$	$S_9 = [0,0,1] \Leftarrow [0]$	$S_9 = [0,0,0] \Leftarrow [1]$	$S_9 = [1,0,0] \Leftarrow [0]$	$S_9 = [1,1,0] \Leftarrow [0]$
$S_8 = [1,0,1] \Leftarrow [0]$	$S_8 = [0,1,0] \Leftarrow [1]$	$S_8 = [0,0,1] \Leftarrow [0]$	$S_8 = [0,0,0] \Leftarrow [1]$	$S_8 = [1,0,0] \Leftarrow [0]$
$S_7 = [0,1,0] \Leftarrow [0]$	$S_7 = [1,0,1] \Leftarrow [0]$	$S_7 = [0,1,0] \Leftarrow [1]$	$S_7 = [0,0,1] \Leftarrow [0]$	$S_7 = [0,0,0] \Leftarrow [1]$
$S_6 = [1,0,0] \Leftarrow [1]$	$S_6 = [0,1,0] \Leftarrow [0]$	$S_6 = [1,0,1] \Leftarrow [0]$	$S_6 = [0,1,0] \Leftarrow [1]$	$S_6 = [0,0,1] \Leftarrow [0]$
$S_5 = [0,0,1] \Leftarrow [1]$	$S_5 = [1,0,0] \Leftarrow [1]$	$S_5 = [0,1,0] \Leftarrow [0]$	$S_5 = [1,0,1] \Leftarrow [0]$	$S_5 = [0,1,0] \Leftarrow [1]$
$S_4 = [0,1,1] \Leftarrow [0]$	$S_4 = [0,0,1] \Leftarrow [1]$	$S_4 = [1,0,0] \Leftarrow [1]$	$S_4 = [0,1,0] \Leftarrow [0]$	$S_4 = [1,0,1] \Leftarrow [0]$
$S_3 = [1,1,0] \Leftarrow [1]$	$S_3 = [0,1,1] \Leftarrow [0]$	$S_3 = [0,0,1] \Leftarrow [1]$	$S_3 = [1,0,0] \Leftarrow [1]$	$S_3 = [0,1,0] \Leftarrow [0]$
$S_2 = [1,0,1] \Leftarrow [1]$	$S_2 = [1,1,0] \Leftarrow [1]$	$S_2 = [0,1,1] \Leftarrow [0]$	$S_2 = [0,0,1] \Leftarrow [1]$	$S_2 = [1,0,0] \Leftarrow [1]$
$S_1 = [0,1,1] \Leftarrow [0]$	$S_1 = [1,0,1] \Leftarrow [1]$	$S_1 = [1,1,0] \Leftarrow [1]$	$S_1 = [0,1,1] \Leftarrow [0]$	$S_1 = [0,0,1] \Leftarrow [1]$
$[1,1,\underline{0}]$	$[0,1,\underline{1}]$	$[1,0,\underline{1}]$	$[0,1,\underline{0}]$	$[0,1,\underline{1}]$

Table 5-2 First 5 efficient traceback iterations for (2,1,3) decoder example.

5.2.3 Managing an Efficient Traceback decoder

For (n,1,m) decoders, and (n, k, m), decoders where each of the k parallel shift registers are the same length, a single traceback pointer is used to track the traceback process. In this project, these are known as category 1 decoders.

However in cases for (n, k, m) decoders where each of the (k>1) parallel shift registers are of different length, traceback pointer management becomes more complex. As each of the registers are different lengths, a maximum of k traceback pointers are needed to track each of the traceback shift registers. Therefore some of the gains made in path memory savings can be lost to the extra traceback pointers needed for the novel path memory technique. These decoders are known as category 2 decoders.

In this case the (3,2,2) decoder has k=2 unequal traceback shift registers and therefore needs 2 traceback pointers. Figure 5-2 presents a pictorial view of the first (3,2,2) traceback iteration, refer to table 5-1. The shaded elements represent the traceback mapping register with the path memory elements for each traceback stage to its right. From figure 5-2, it can be seen that a decoded symbol is produced from the S_{22} and the b_1 elements, once the upper traceback pointer tb_ptr1 reaches stage 1.

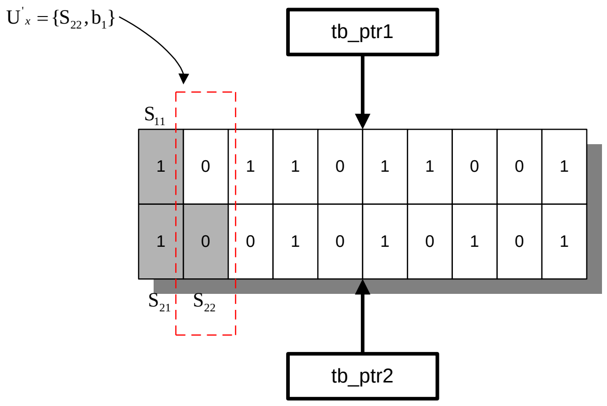


Figure 5-2 Efficient (3,2,2) Traceback Pointer Solution.

A procedure to perform the TB Viterbi Algorithm when using the novel path memory technique is now detailed in table 5-3.

```
1.   Initialize Data Structures
     1.1. Initialize the trellis stage pointer to zero. Initialize the path memory write pointer
          to zero. Initialize the traceback pointer to zero. Initialize the decoded symbol
          counter to zero.
     1.2. Initialize the path metric for the known initial state to zero, with the remaining $2^M$
          $-1$ path metrics to their maximum value. Go to step 2.
2.   Compute Path  Metrics and Survivors
     2.1. Increment the trellis stage pointer, and the path memory pointer.
     2.2. For every trellis node, compute $2^k$ path metrics by summing the path metrics
          from nodes at the previous stage to the corresponding branch metrics
          computed at the present stage.
     2.3. Compare the $2^k$ paths and select the path with the minimum path metric as the
          surviving path, all other incoming paths to the trellis node are no longer
          considered. If there is a tie between path metrics, the algorithm selects one
          path.
     3.5. Store the path metric. Update the surviving path by shifting in the surviving
          backward label to the left hand side of the path memory register where the
          surviving path currently terminates.
     2.4. If the path memory write pointer is < T then go to step 2.1, else if the path
          memory write pointer = T go to step 3
3.   Traceback and Output Decision
     3.1. Set the traceback pointer equal to T. Determine the traceback start state
          number as the state that corresponds to the minimum path metric.
     3.2. The state number and the traceback pointer are combined into a row-column
          address used to index path memory. Use this address to read a backward label
          from path memory. A predecessor state on the surviving path is then
          determined by use of the traceback mapping function. Decrement the
          traceback pointer. If the traceback pointer >1 repeat step 3.2, else go to step
          3.3.
     3.3. Produce a decoded symbol.
          3.3.1. Category 1 - A decoder decision is made for one symbol by selecting
                 the rightmost elements $(s_{k,m_k}, s_{k-1,m_{k-1}}, \dots, s_{1,m_1})_x$ of the traceback
                 mapping register.
          3.3.2. Category 2 - A decoder decision is made for one symbol by selecting a
                 combination of specific elements from the backward label read from path
                 memory and the rightmost elements $(s_{k,m_k}, s_{k-1,m_{k-1}}, \dots, s_{1,m_1})_x$ of the
                 traceback mapping register[4].
     3.4. Increment the decoded symbol counter. If the decoded symbol count < N, go to
          step 2.1, else finish.
```

Table 5-3 Efficient V.D.A Traceback Procedure.

5.2.4 Effects of "Encoder Flushing" for Novel technique

As discussed in section 2.1.2, convolutional codes are typically forced into a block structure by periodic truncation. This requires a number of zeros to be appended to the end of the input sequence, for the purpose of clearing or flushing the encoder shift register bits, [14].

One could ask the question, is “encoder flushing” lost and hence loss block synchronization between decoding blocks when using this novel technique?

This question could be asked as the novel technique has m_j less path memory registers stages than the conventional technique. As outlined in the previous examples, the savings gained are realized during the traceback process. That is, during traceback a decoded symbol is realized from the backward label elements m_j stages earlier than in the conventional case. Therefore the first m_j stages during traceback are not required to realize the novel technique.

To implement this in a decoder, this stages of novel path memory are synchronized with trellis stages m_j to T. Hence the final m_j stages of the novel path memory unit are identical to the conventional case and thus novel decoder implementations will not lose block synchronization due to “encoder-flushing”.

5.2.5 Quantifying the memory savings

The decoder survivor path memory requirement will be k bits per state by 2^M states per stage by T stages, i.e. $kT2^M$ for present forward or backward branch label systems implementing the Viterbi decoder. However, in algorithms using a backward label scheme and exploiting this memory saving property, survivor path memory savings of Δ are available:

$$\Delta = (m_1 + m_2 + ... + m_k).2^M \qquad = M.2^M \text{ bits.}$$

This is equivalent to a percentage saving of decoder traceback memory of $\frac{100M}{kT}$ %, [4].

In the (2,1,3) encoder example, with T = 5m, the savings available are 20%. For all (n,1,m) convolutional codes the savings available are $\frac{100m}{T}$ % , or 20%, when T = 5m.

For all (n,1,m) codes including the (2,1,3) decoder described in this paper, the path memory savings available are 20%.

For (n,k,m) codes including the (3,2,2) decoder described in this paper, the path memory savings available are $\leq$20%. The (3,2,2) decoder described in this paper achieves a 15% reduction in path memory size.

6. Design Of Efficient Viterbi Decoders

6.1 Overview

Section 5 shows that path memory savings are achievable without loss of decoding performance. That is by taking the conventional backward label traceback decoder, and taking advantage of the code theorems, a traceback decoder can be designed in a smaller VLSI footprint. This section provides a functional description of a number of software and hardware based designs for the conventional backward label traceback decoder, and the new efficient backward label traceback decoder.

Both the conventional and efficient decoders have common functional blocks, such as the branch metric computation unit, and the path metric unit. However the path memory storage unit, and output decision units differ for the efficient decoders. This section will detail the designs using the (2,1,3), and (3, 2, 2) test codes.

Before discussing the hardware implementations, section 6-2 details the software simulation of the entire channel to determine a “lab” model for the novel ML decoding technique.

Section 6-3 details the hardware implementation for the (2,1,3) conventional and efficient traceback decoders. A general top level block diagram for a hardware based backward label decoder is first introduced. This top level block diagram is used to show that the conventional and efficient decoders can share common modules. Therefore for this project, the hardware design is modularized such that common modules, i.e. the BMU and ACSU can be re-used between both decoders. This also provides a mechanism to easily determine the logic utilization between both hardware implementations by only comparing the modules unique to the particular decoder.

Similarly, section 6-4 introduces a top level block diagram for the (3,2,2) traceback decoders, and breaks down the functionality of each block.

All hardware based decoders are designed using the Verilog HDL language and synthesized to fit in a Xilinx Virtex XCV50 FPGA. The FPGA design process from Verilog to gates is detailed in section 6.6. Finally, section 6-7 provides an analysis on the decoding speed improvements for the novel traceback decoder implementations.

6.2 Software Simulation Environment

Before proceeding with the hardware implementations, a software simulation is carried out on a model of the entire channel. These models provide a "lab" environment to try out these designs in a typical model of a communications link before proceeding with the ASIC design. These models can also be used for further research into the novel technique if needed. The simulation environment is used for both the conventional and efficient decoders. The simulations are comprised of the following steps:

1. Take the input sequence and produce a convolutional encoded sequence,
2. Subject the encoded sequence to a noisy channel model,
3. Perform Viterbi decoding on the noisy received sequence,
4. Calculate the bit error ratio for the decoding operation and count the number of bit errors.

6.2.1 Software Simulation Language

The software simulation language used was MATLAB, which is a widely known mathematical simulation tool. MATLAB is a very useful simulation language that can handle a range of computing tasks by using its powerful technical language. Another useful feature of MATLAB is the ability to integrate external routines written in C, C++, Fortran, and Java into MATLAB applications. A number of MATLAB based functions were written using the MATLAB language to perform each of the four steps above.

6.2.2 Block Size of Convolutional Encoded Sequence

A popular application for Viterbi decoding is the GSM cellular telephone standard. The GSM standard generates an encoded sequence of 456 bits block (partly encoded using a convolutional code). The generator polynomials used in GSM to generate convolutional coded sequences are not the same as this project, however the block size of 456 bits was chosen as a realistic block size for the simulation. This however can be easily changed in the simulation environment. Given a (2,1,3) encoder, the input sequence U needs to be 225 bits long in order to encode the 456 bit block V, i.e. 2(225+3) = 456 bits. Similarly, the input sequence U for a (3,2,2) encoder needs to be 150 bits long in order to encode the 456 bit block, i.e. 3(150+2) = 456 bits.

6.2.3 Convolutional Encoding Process

In order to test the designs with a range of test sequences, each of the N-bit encoder input sequences U is randomly generated using the MATLAB *rand* library function. The *rand* function generates a real number ranging from zero to one, however these real numbers have to be converted to ones and zeros. The procedure to carry out this process is as follows:

1. Each of N real numbers are offset by –0.5,
2. The MATLAB *ceil* library function then rounds these numbers to the next highest integer. Therefore all negative real numbers get assigned to zero, and all positive real numbers get assigned to one.

An excerpt of the (2,1,3) MATLAB code that perform the steps above is shown as follows:

```
N=225;
NUM_PSEUDO_BLOCKS=10;
U_vectors=zeros(NUM_PSEUDO_BLOCKS,N);
for v=1:10,
   U_vectors(v,:) = ceil(rand(1,N)-0.5);
end;
```

Now for the convolutional encoding function. A MATLAB function was written that takes any N-bit input binary sequence and the generator polynomial of the encoder, and generates a convolutional encoded sequence V of length n(N+m).

As described in section 2, a convolutional encoder is constructed with M encoding shift register elements, and n modulo-2 adders. The generator polynomial (or generator matrix) of a convolutional encoder describes a set of n connection vectors, one for each of the n modulo-2 adders. Each vector is of length m_j+1 and describes the connections between the encoding shift register elements to its modulo-2 adder. For example, a one in the, i^{th} position of the vector indicates that the corresponding stage in the shift register is connected to the modulo-2 adder, and a zero in a given position indicates that no connection exists between the modulo-2 adder and the shift register, [14].

The method of generating the convolutional encoded sequence V from the input sequence U is taken from, [14], and described as follows.

The encoder is viewed by its impulse response, that is the response of the encoder to a single 1 that moves through it. Using this method, the output of the encoder V is found from the input sequence U by the superposition or linear addition of the time-shifted input impulses, [14].

In MATLAB, a function exists that can be used to convolve the input sequence U. The library function *conv* is used to convolve the input sequence U with each row of the generator polynomial matrix, to produce the convolutional encoded message V. The MATLAB convolutional encoding function code, *"vit_enc.m"* can be studied further by referencing the attached MATLAB code

6.2.4 Channel Model

Next a channel model needs to be described in MATLAB. The channel model of choice for the simulation is the binary symmetric channel (BSC) induced by added white Gaussian noise, (AWGN). The theory of the BSC channel model was detailed in section 2.3.4. Hard decision decoding is then performed on the received sequence R as the implementation is less complex than the soft decision case, however soft decision decoding results in a performance improvement of approximately 2 dB in required signal to noise ratio compared to hard decision decoding. In the case of a binary symmetric channel with added white Gaussian noise, (AWGN), the channel bit error or transition probability is given by:

$$p = Q\sqrt{\frac{2E_b}{N_0}} = \frac{1}{2} erfc\left(\frac{\sqrt{2\frac{E_b}{N_0}}}{\sqrt{2}} \right)$$

Where Q(x) is the complimentary error function or co-error function, and *erfc*, is the complimentary error function, [14]. In MATLAB the library function *erfc* exists so an expression for the bit transition probability is constructed using the second equation.

A function called *"bsc_channel_errors.m"* was written in MATLAB that takes as input the convolutional encoded sequence V and a value for E_b/N_0 and generates the noise induced sequence R from V. The value for E_b/N_0 can be varied to calculate different transition probabilities, however the default value equals 10. The method of simulating a noisy channel is as follows:

1. Make a copy of the encoded sequence V and call it R.
2. Given a value for E_b/N_0, compute the transition probability, the value computed is a real number between 0 and 1.
3. The *rand* MATLAB library function is then used to generate N real numbers, where each of the N elements fall between 0 and 1.
4. Compare each of the N random elements generated in step 3, against the transition probability p computed in step 2. If an element of the random sequence is less than the transition probability p, the corresponding bit in received sequence R is flipped.

In summary, the received sequence, R is essentially a copy of the encoded sequence V, however any of the bit positions that are less than the transition probability p are flipped, similar to the noise process induced on a channel. The steps above provide an accurate model to generate the noise induced in a BSC channel. An excerpt of the MATLAB code is shown as follows:

```
R = V;
alpha =sqrt(2*EbNo);
p=0.5*(erfc(alpha/sqrt(2)));

error=rand(1,length(R));
error_pos=find(error<p);

for i=1:length(error_pos),
   %Flip the bit in error
   R(error_pos(i)) = abs(R(error_pos(i))-1);
end;
```

6.2.5 Viterbi Decoding

Finally the decoding process can be described. The Viterbi decoding function is written to perform as a backward label traceback decoder. As the Viterbi algorithm can broken into a number of distinctive steps, a number of sub-functions were written to perform these tasks. These tasks are as follows:

- Branch Metric Computation,
- Path Metric Computation,
- Traceback Mapping.

A block diagram for all MATLAB Viterbi decoders is shown in figure 6-1, which just shows the major blocks at a high level.

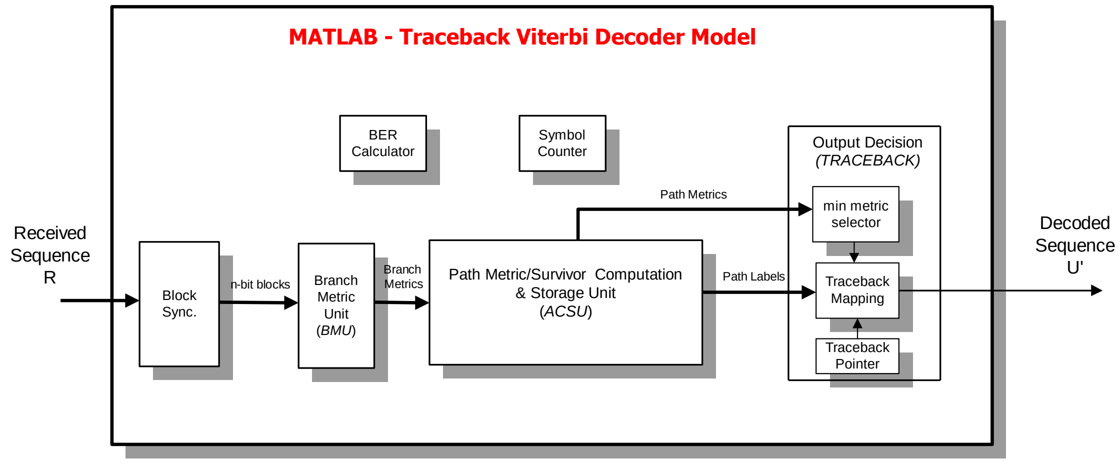


Figure 6-1 MATLAB Traceback Decoder.

From figure 6-1, the Block Synchronizer is a simple operation. In fact this function simply divides the received sequence R equally into n-bit symbols. This n-bit symbol

R_x is then used by the BMU block to produce $2^{(M+k)}$ branch metrics for one trellis stage. An excerpt of the MATLAB code for the (2,1,3) decoder is shown as follows highlighting the process of correctly computing R_x

```
    if x<=N+m,
       %Read an n-bit symbol Rx at the current trellis stage
       Rx=R(n*(x-1)+1:n*x);
       %Determine all Branch Metrics for Rx.
       [HD]=BMU_213(Rx);
    end;
```

The ACSU block in figure 6-1 is the main computational unit in the Viterbi decoder. During each trellis stage the ACSU block takes as input 2^M path metrics and $2^{(M+k)}$ branch metrics, and computes 2^M path metrics and 2^M k-bit backward labels. As previously noted a trellis is made up of identical sub-structures called a trellis butterfly. Therefore, the ACSU block contains 2^M ACS sub-functions to perform the Add-Compare-Select operations on each half of the trellis butterfly. Each ACS function then provides a new path metric and backward label.

An excerpt of the MATLAB code for state 0 of the (2,1,3) ACS block is shown below, showing the state 0 path metric and path memory updates:

```
% Generate copy of A (Path Metric Storage) for processing
A_copy = A;
%======================
%   State 0 ACS
%======================
[sel_out0, metric_out0]=ACS_213(HD(1,1),HD(1,9),
                                A_copy(1,(st0+1)),
                                A_copy(1,(st4+1)));

% Update State(000) Metrics
A(1,st0+1) = metric_out0;
%Shift in new k-bit label
P(st0+1,1:T) = [P(st0+1,2:T) D2B((sel_out0-1),k)];
```

The traceback and output decision block in figure 6-1 performs the traceback mapping function and releases decoded symbols. The traceback mapping function differs between the conventional backward label decoder and the efficient backward label decoder. Initially a function is called to determine the trellis state with the minimum path metric. The traceback function then loads the traceback pointer with a value equal to the truncation length T. The combination of the traceback pointer and the minimum metric state are then used to read a backward label from path memory. The traceback mapping function then performs traceback and determines the predecessor state on the surviving path. The traceback pointer decrements for each traceback iteration, and halts when all surviving backward labels are read from path memory. The leftmost element of the traceback register is then substituted as the decoded symbol. The MATLAB traceback function for the (2,1,3) decoder is shown below:

```
% Find Traceback start state
[min_metric,min_state]=BSU_213(A);

% Perform Traceback process
for stage_ptr = T : -1: 2,
   if stage_ptr == T,
      Bx=B2D(P((min_state+1),stage_ptr));
      Sx=D2B(tb_ptr,m);
   else
      Sx_1=B2D(Sx_1);
      Bx=B2D(P((Sx_1+1),stage_ptr));
      Sx=D2B(Sx_1,m);
   end;
   Sx_1=[Bx Sx(1,1) Sx(1,2)];
end;

% Find Decoded Symbol, Dx
Dx     =[Sx_1(1,m)];
% Add Decoded Symbol to Maximum Likelihood path, D
D      =[Dx D];
```

After the decoding operations are complete the Viterbi decoder function calculates the number of bit errors and the bit error ratio. The bit error ratio is defined as the number of erroneous bits received divided by the total number of bits transmitted over a stipulated period of time. Examples are:

- Transmission Bit Error Ratio, *i.e.*, the number of erroneous bits received divided by the total number of bits transmitted.
- Information Bit Error Ratio, *i.e.*, the number of erroneous decoded bits divided by the total number of decoded bits.

The bit error ratio is usually expressed as a coefficient and a power of 10; for example, 2 erroneous bits out of 100,000 bits transmitted would be 2 out of 10^5 or 2×10^{-5}.

The BER block in figure 6-1 calculates the Information bit error ratio by first taking the input sequence U, and the decoded sequence D and performing an XOR operation on a bit by bit basis to determine the number of erroneous bit positions. The total number of erroneous bits is then determined by summing all the ones in the XOR sequence. The Information bit error ratio is then calculated by dividing the number of erroneous bits by the total number of bits transmitted. The MATLAB code that performs this function is shown as follows.

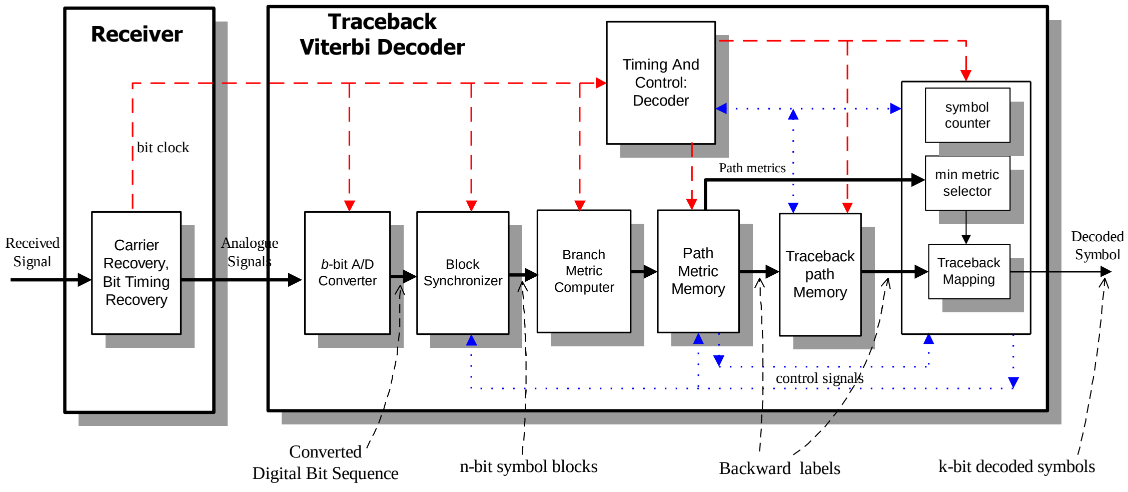
```
%Determine bit error ratio
ber=sum(xor(U(1,1:N),D(1,1:N)))/N;

str = ('==========================================');
disp(str);
str = sprintf('Pseudo Block =%d',i);
disp(str);
if (ber == 0),
   str = sprintf('Bit Error Ratio (BER) = %d',ber);
   disp(str);
else
   str = sprintf('Bit Error Ratio (BER) = %d, # Bit
Errors',ber, sum(xor(U(1,1:N),D(1,1:N))));
   disp(str);
end;
str = ('==========================================');
disp(str);
```

6.3 Hardware Design (2,1,3) Traceback Decoders

This section details the hardware design for two backward label traceback Viterbi decoders. Like the MATLAB simulation in section 6.2, the hardware decoders are designed to use bit metrics that allow hard decisions to be made on the incoming digital bit sequence. The first design details the conventional traceback decoder utilizing the “one-traceback” pointer path memory, with the traceback operations running at the same rate as the write operations. The second design then details the hardware efficient decoder.

Section 4 introduced the architectural differences between forward and backward label decoders, with the general block diagram for a backward label decoder repeated below in figure 6-2.



Note 1 : Blue "Dotted" Lines denote Control Signals

Note 2 : Red "Dashed" Lines denote Timing or Clock Signals

Figure 6-2 Backward label Viterbi Decoder Block Diagram.

For this project, the hardware design of both traceback decoders use a subset of the blocks shown in figure 6-2. The hardware traceback decoders work on the assumption that all pre-processing activity on the received sequence has been carried out correctly, and that the decoder receives a fully synchronized digital sequence, i.e. 2-bits per trellis stage on every clock cycle. Therefore the hardware designs do not include the front end receiver, A/D converter and block synchronization blocks of figure 6-2. However an out of sync detector is used to detect if no surviving path is dominant and an output signal is asserted to force resynchronization. This circuit resides in output decision block sections of each decoder. The block diagram for both (2,1,3) hardware based traceback decoders, and the associated top-level symbol is shown below in figure 6-3, followed by the top level signal descriptions in table 6-1.

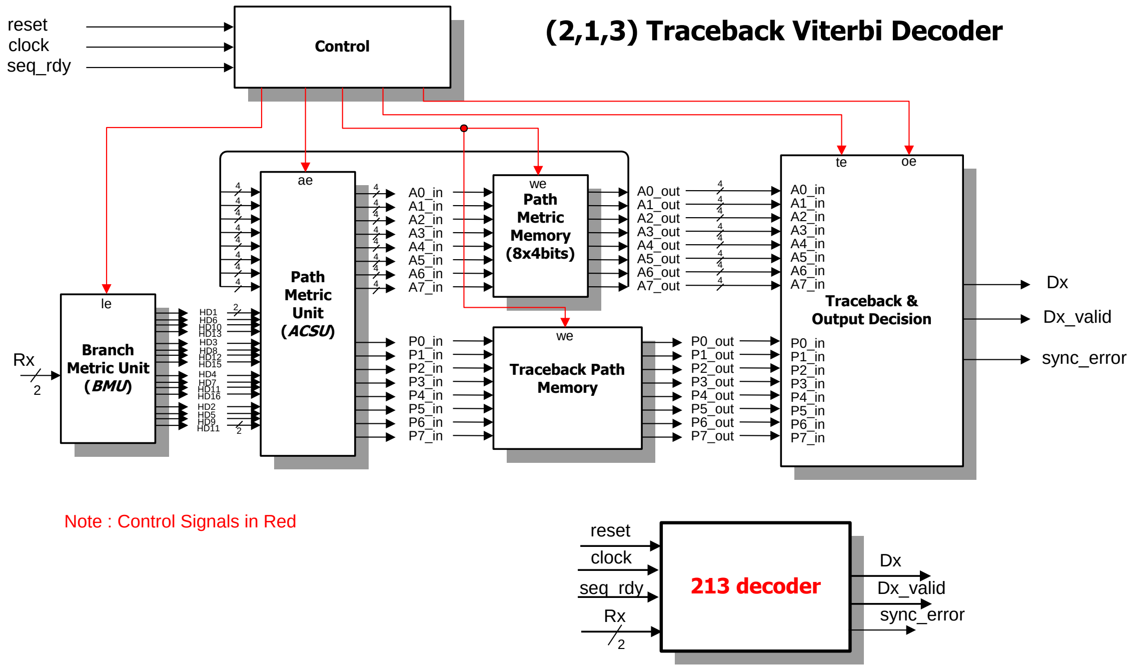


Figure 6-3 (2,1,3) H/W Viterbi Decoder.

Signal	Direction	Description
Rx[1:0]	input	A 2-bit digital sequence representing the n-bit received sequence for the current trellis stage.
seq_rdy	input	A pulse to indicate a new received sequence is ready to be decoded.
clock	input	Chip level clock input.
reset	input	Chip level reset input.
Dx	output	The decoded symbol output, which can be either a zero or a one and qualified by the assertion of *Dx_Valid*. If *Dx_Valid* is de-asserted this signal remains in tri-state.
Dx_Valid	output	An output signal used to qualify a valid decoded symbol is available.
sync_error	output	An output signal to indicate that the decoder is out of sync with the incoming received sequence.

Table 6-1 Top level Signal Description for the (2,1,3) Decoder.

6.3.1 (2,1,3) Branch Metric Unit

The hardware design of the (2,1,3) branch metric unit (BMU) is implemented to compute branch metrics based on minimum Hamming distances. In such cases, each branch metric is computed by counting the number of elements that differ between the received symbol R_x and each trellis branch.

At every trellis decoding stage, the BMU receives a 2-bit sequence R_x. and computes $2^{(M+k)}$ or sixteen Hamming distance branch metrics, one for each trellis branch in the current trellis stage. The sixteen outputs of the BMU, (HD1 to HD16) and represent the trellis branches below in table 6-2.

	S_{x+1} (s_3,s_2,s_1)							
S_x(s_3,s_2,s_1)	000	001	010	011	100	101	110	111
000	HD1	HD2	-	-	-	-	-	-
001	-	-	HD3	HD4	-	-	-	-
010	-	-	-		HD5	HD6	-	-
011	-	-	-		-	-	HD7	HD8
100	HD9	HD10	-	-	-	-	-	-
101	-	-	HD11	HD12	-	-	-	-
110	-	-	-	-	HD13	HD14	-	-
111	-	-	-	-	-	-	HD15	HD16

Table 6-2 (2,1,3) BMU outputs assignment table

For the (2,1,3) trellis, there can only be three valid Hamming distances of 0, 1, and 2. For example some (2,1,3) Hamming distances are computed as follows, $\delta(00,00) = 0$, $\delta(11,11) = 0$, $\delta(00,01) = 1$, $\delta(00,10) = 1$, and $\delta(10,01) = 2$.

For the hardware implementation, a simple circuit was designed to compute each of the outputs HD1 through HD16. This simple circuit can be described by the Verilog code, followed by the gate level representation in figure 6-5.

```
module bm213 (HD, Rx, Vx);

   output [1:0] HD;   // Hamming Distance Branch Metric
   input  [1:0] Rx;   // 2-bit Received Symbol
   input  [1:0] Vx;   // 2-bit trellis branch label

   assign HD = {((Rx[1]^Vx[1]) & (Rx[0]^Vx[0])), // HD[1]
                ((Rx[1]^Vx[1]) ^ (Rx[0]^Vx[0]))  // HD[0]
               };
endmodule
```

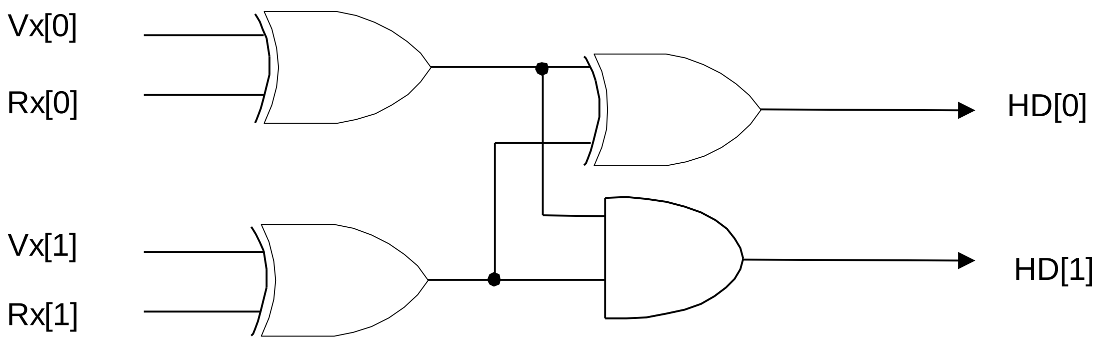


Figure 6-5 (2,1,3) HD Branch Metric Building Block.

As the sixteen trellis branches in one (2,1,3) trellis stage can be labeled with one of the four possible sequences {00,01,10,11}, this building block is instantiated four times to compute the Hamming distances. The outputs of the instantiated blocks are then registered when the load enable signal (*le*) is asserted to load and hold the branch metrics for the current trellis stage. The outputs of the flip-flops are then assigned to four of the BMU outputs as shown in figure 6-6.

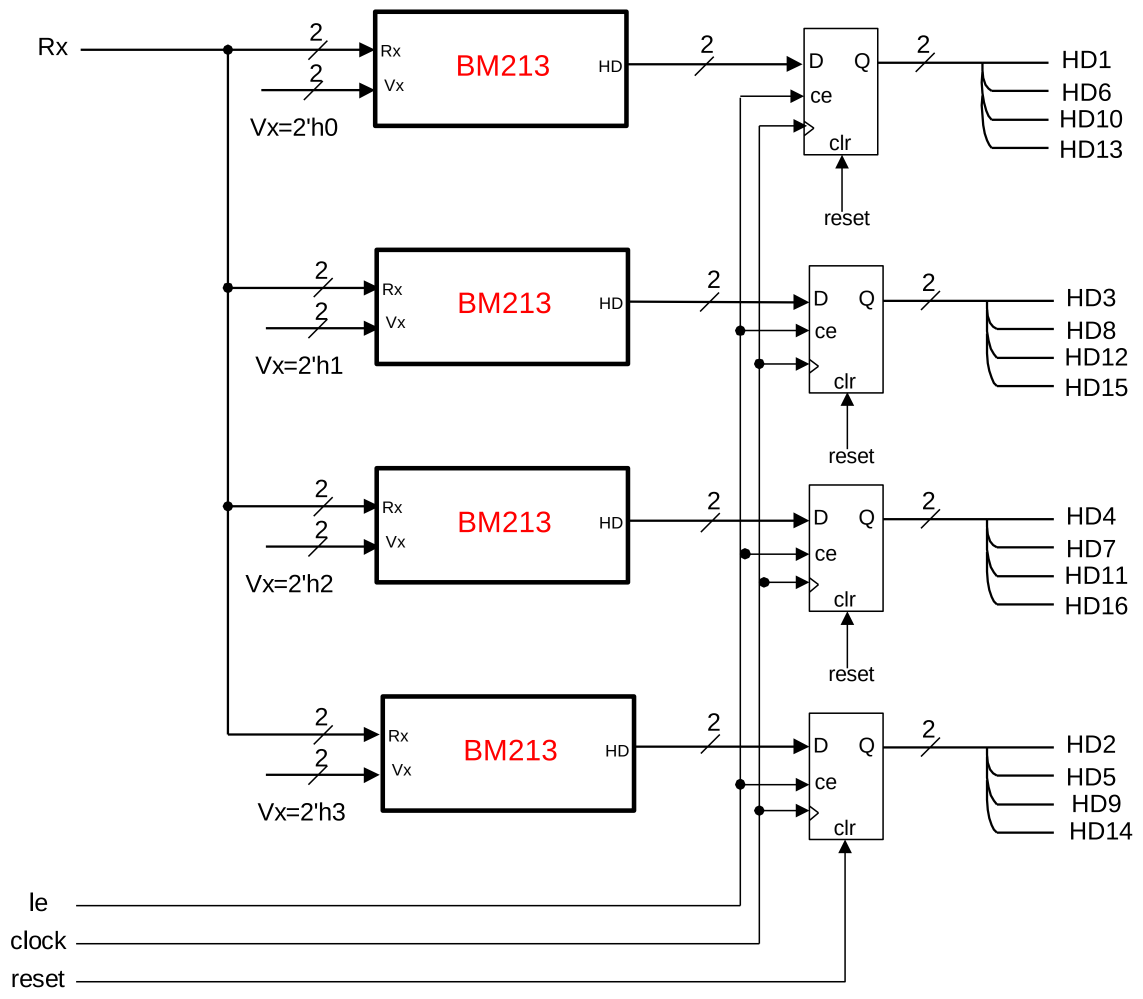


Figure 6-6 (2,1,3) BMU.

6.3.2 (2,1,3) Add Compare Select Unit

The hardware design of the Add-Compare-Select Unit (ACSU) is implemented to compute the path metrics and produce backward labels for the Viterbi decoder. For the (2,1,3) decoder, a new path metric and backward label is produced by the Add-Compare-Select (ACS) operation in one clock cycle. The ACS block works as follows. The range of metrics that can be computed by the each state path metric ACS is (0000_2-1111_2) or 0 – 15 in decimal. At the beginning of a new decoding sequence, the path metric for state (000), is initialized to (0000_2), and the remaining seven path metric memories are initialized to their maximum value (1111_2). As the algorithm proceeds, if both incoming path metrics equals (1111_2), the outgoing path metric of the ACS block is held at (1111_2) to prevent overflow problems in the ACS, otherwise if one or both incoming path metrics is less than (1111_2), the ACS block determines the minimum path metric as follows:

If the add enable signal is asserted (*ae*), each incoming path metric and the corresponding branch metric are added together. Both sums from the Add operation are then fed into a $\leq$ comparator, the output of the comparator generates the surviving backward label output. The comparator output is also used as the select signal for the 2:1 path metric MUX. This ACS circuit is described by the Verilog code below. The equivalent circuit is then shown in figure 6-7.

```
wire         hold_a = (ae & acs_ppm_ina==4'b1111); // Hold metric a to its max value
wire         hold_b = (ae & acs_ppm_inb==4'b1111); // Hold metric b to its max value

   // Compute the partial path metrics
   always @(posedge clock or posedge reset)
      begin
         if (reset)
           begin
              suma <=4'h0;
              sumb <=4'h0;
           end
         else
       begin
              suma <= (hold_a) ? acs_ppm_ina : (ae) ? (acs_ppm_ina+HD_ina) : suma;
              sumb <= (hold_b) ? acs_ppm_inb : (ae) ? (acs_ppm_inb+HD_inb) : sumb;
       end
      end

   // Compare Path Metrics and Select One Surviving Path
   always @(suma or sumb)
      begin
         acs_ppm_out =(suma <=sumb & ae) ? suma : sumb;
         acs_Bx_out  =(suma <=sumb & ae) ? 1'b0 : 1'b1;
      end
```

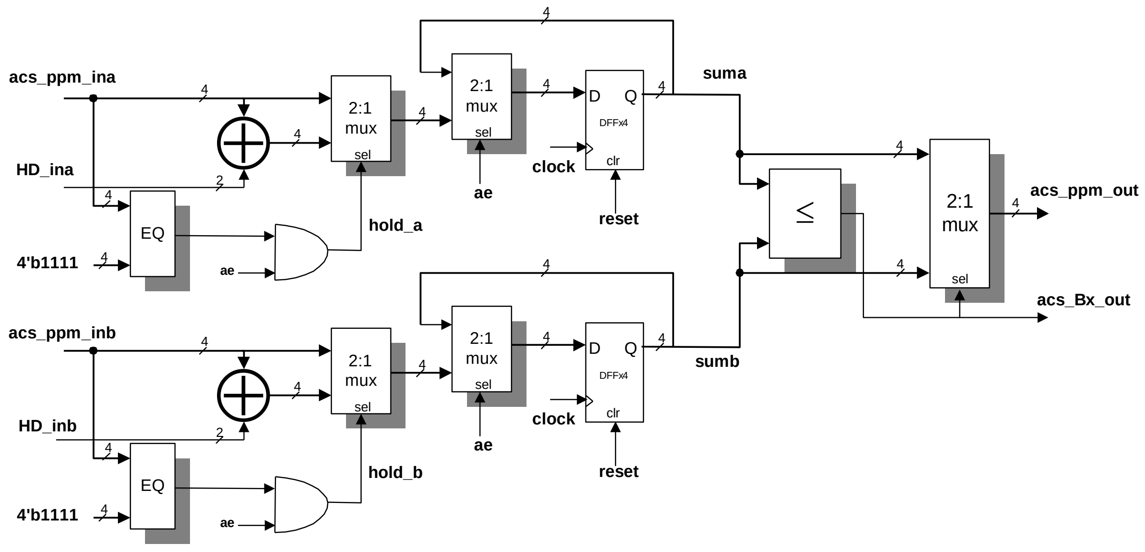


Figure 6-7 (2,1,3) ACS Building Block.

In the (2,1,3) decoder there are 2^M or eight states in a single trellis stage. For this design all eight path metrics and backward labels are computed in parallel by the ACSU block. As the process of producing path metrics and backward labels is identical for all eight states, the ACS block is instantiated eight times to construct the entire ACSU.

6.3.3 (2,1,3) Path Metric Storage Unit

The path metric memory storage is designed to operate as eight parallel 4-bit accumulators, one for each state. During each trellis stage, the stored metric is added to the corresponding incoming path metric from the ACSU block. If the received sequence is synchronized, then after a certain number of trellis stages, the path metric accumulators diverge allowing the Viterbi decoder to distinguish a surviving path. The organization of the path metric accumulators are shown in figure 6-8.

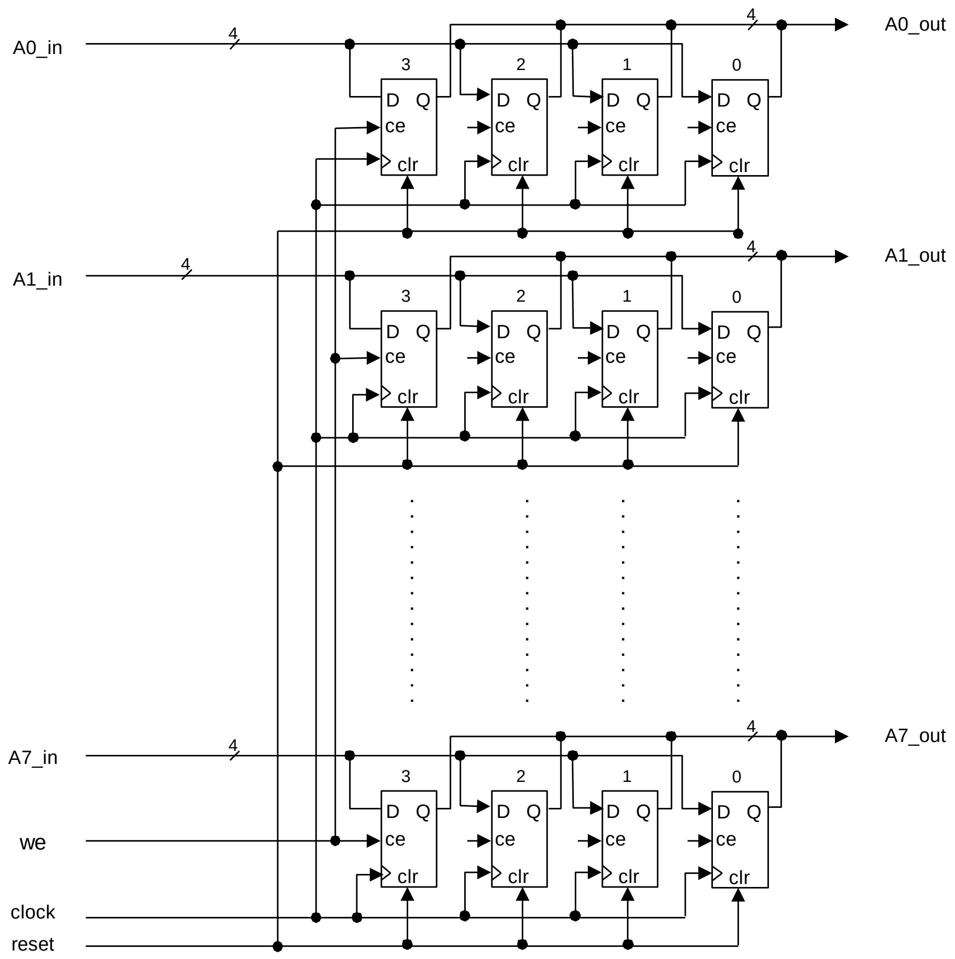


Figure 6-8 (2,1,3) Path Metric Memory and Symbol

6.3.4 (2,1,3) Conventional Path Memory Storage Unit

The conventional path memory is organized as eight parallel shift registers (P0-P7), each of length fifteen, i.e. (2^M x 5m). At every trellis stage eight backward labels are produced by the ACSU, if the write enable (*we*) is asserted these backward labels are shifted into the corresponding path memory shift register with the existing register contents moving one place to the right. A path memory write pointer is incremented each time a set of backward labels are shifted in. The organization of the path memory shift registers are shown in figure 6-9.

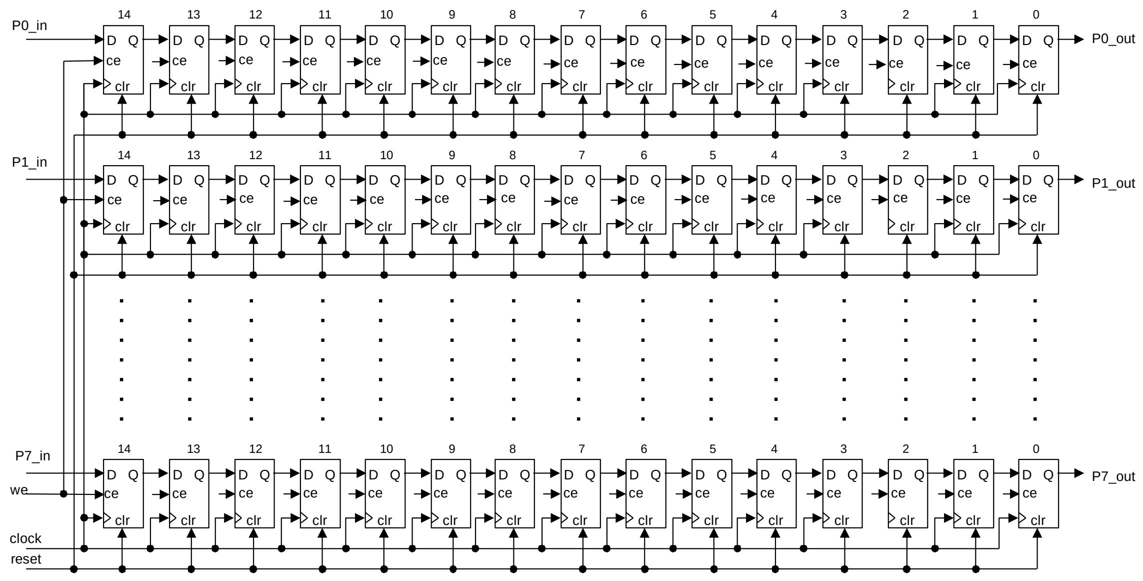


Figure 6-9 (2,1,3) Path Memory.

6.3.5 Conventional (2,1,3) Traceback & Output Decision Unit

Once the path memory is full, the traceback process can begin. From figure 6-3, the traceback and output decision unit consists of the following blocks:

- An 8-way ≤ comparator to select the traceback start state,
- An 8:1 MUX to select a backward label,
- A traceback pointer,
- A traceback register,
- A tri-state enabled output buffer.
- An Out of Sync detector.

The operation is as follows. First, the minimum path metric state is determined. This is determined by taking the eight path metrics as inputs to a ≤ comparator, the comparator outputs the corresponding state number, *min_state*. If the (*be*) signal is asserted, the traceback mapping register is then loaded with this state number, *min_state*. This state number is also used as the select bits for an 8:1 MUX. This MUX takes all eight backward labels from the path memory, and selects the backward label corresponding to the state number. Finally the traceback pointer (*tb_ptr*) is loaded with a value that points to the last entry in the path memory. Traceback then waits for the traceback enable signal (*te*) to be asserted. Once (*te*) is asserted, a new state is traced back every clock cycle, and the traceback pointer is decremented. This operation continues until all backward labels are read from path memory. The traceback mapping circuit is described by the Verilog code below. The equivalent circuit is then shown as part of the output decision block in figure 6-7.

```
  /*=== Traceback Logic ===*/
   always @(posedge clock or posedge reset)
     begin
    if (reset)
       {s1,s2,s3} <= 3'b0;
    else
          begin
             s1 <= (be) ? min_state[0] : (te) ? s2 : s1; // s2 ->s1
             s2 <= (be) ? min_state[1] : (te) ? s3 : s2; // s3 ->s2
             s3 <= (be) ? min_state[2] : (te) ? b1 : s3; // b1 -> s3
          end
     end
```

From the Verilog code above, it can be seen that the traceback register has the same number of registers as the encoding shift register, but the shift register operations work from right to left. That is, the traceback register works in reverse order, i.e. right to left as the decoder attempts to replicate each state transition made by the encoder. Once the traceback operation has completed, the control block asserts the output enable (*oe*) signal to release a decoded symbol. The decoded symbol Dx is equivalent to the *s1* element from the traceback register. The block diagram of the output decision block is shown below in figure 6-10.

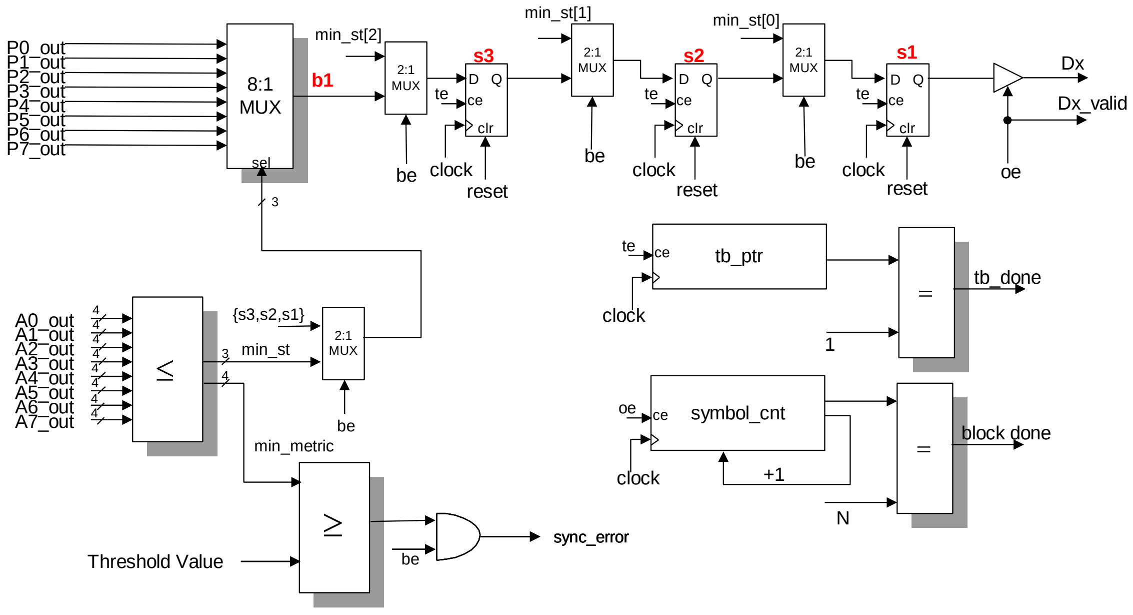


Figure 6-10 (2,1,3) Output Decision & Traceback Unit

6.3.6 (2,1,3) Efficient Path Memory Storage Unit

The hardware design for the efficient path memory unit is organized as eight parallel shift registers (P0-P7), each of length twelve, i.e. (2^M x (5m)-m). The operation of shifting in backward labels is identical to the conventional path memory The efficient path memory shift registers are shown below in figure 6-11.

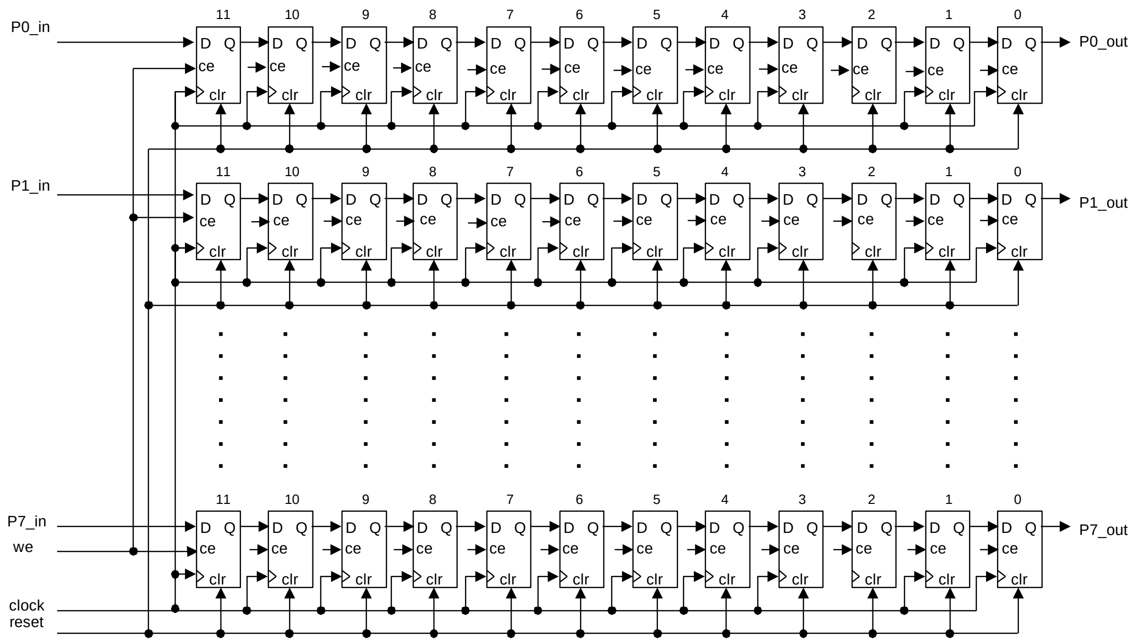


Figure 6-11 Efficient (2,1,3) Path Memory and Symbol

6.3.7 Efficient (2,1,3) Traceback & Output Decision Unit

Similar to the conventional case, the traceback and output decision block contains the same elements, i.e.

- An 8-way ≤ comparator to select the traceback start state,
- An 8:1 MUX to select a backward label,
- A traceback pointer,
- A traceback register,
- A tri-state enabled output buffer,
- An Out of Sync detector.

The operation of tracing back is identical to the conventional case. However, the traceback operations terminate earlier than the conventional case because of the smaller memory requirements. In addition, the output decision circuit is wired differently than the conventional case in order to take advantage of the efficient traceback architecture.

In the efficient implementation, the output decision is wired to the *s3* register instead of the *s1* register in the conventional case, as shown below in figure 6-12.

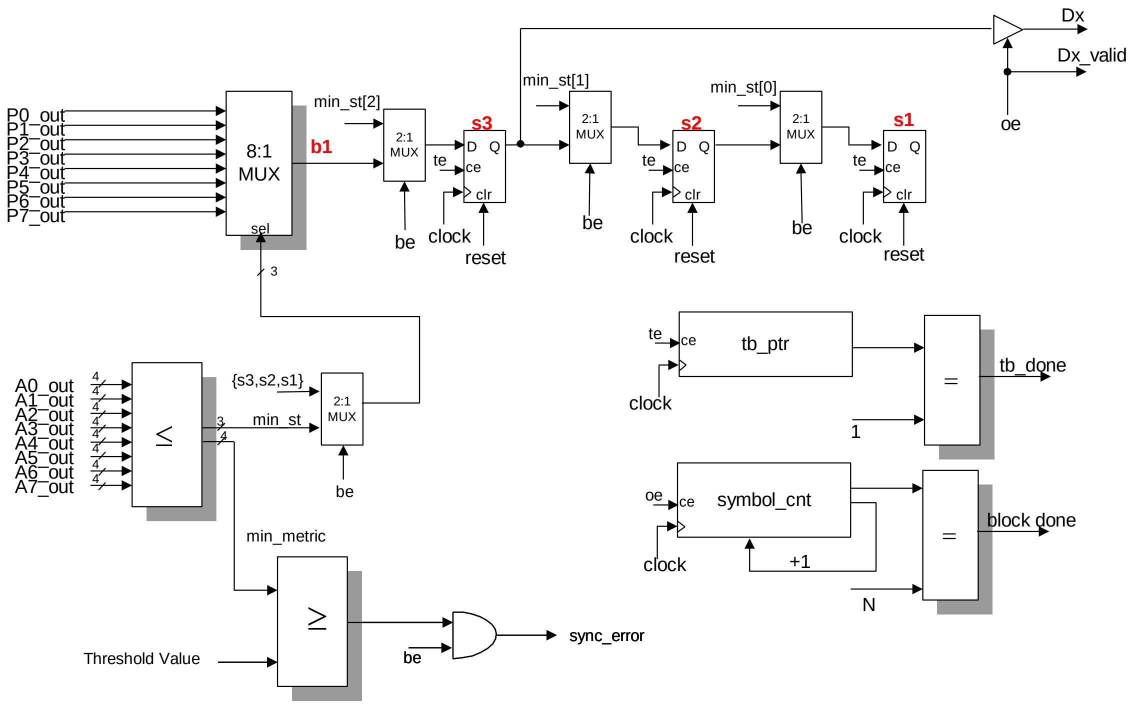


Figure 6-12 Efficient (2,1,3) Output Decision & Traceback Unit

6.4 Hardware Design (3,2,2) Traceback Decoders

Similar to the previous decoders, the (3,2,2) hardware traceback decoders work on the assumption that all pre-processing activity on the received sequence has been carried out correctly. The block diagram for both hardware based (3,2,2) traceback decoders, and the associated top-level symbol is shown below in figure 6-13, followed by the signal descriptions in table 6-3.

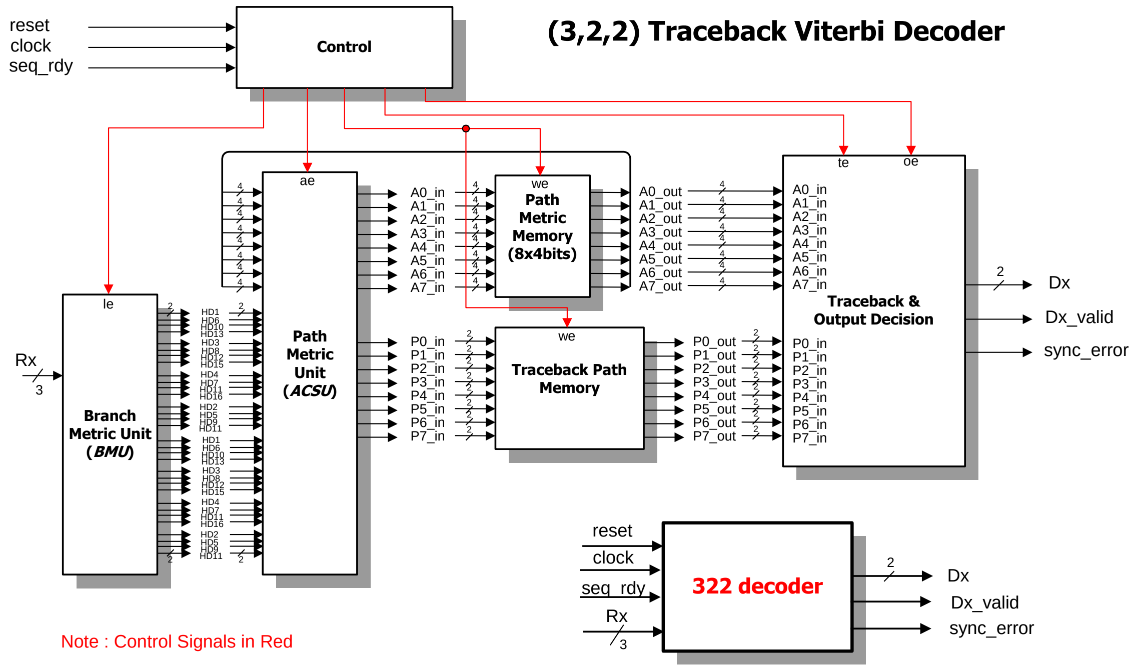


Figure 6-13 (3,2,2) H/W Viterbi Decoder.

Signal	Direction	Description
Rx[2:0]	input	A 3-bit digital sequence representing the n-bit received sequence for the current trellis stage.
seq_rdy	input	A pulse to indicate that a new received sequence is ready to be decoded.
clock	input	Chip level clock input.
reset	input	Chip level reset input.
Dx[1:0]	output	The 2-bit decoded symbol output. A valid decoded symbol is qualified by the assertion of the *Dx_Valid* signal. Otherwise the signal remains in the tri-state condition.
Dx_Valid	output	An output signal used to qualify a valid decoded symbol is available.
seq_error	output	An output signal to indicate that the decoder is out of sync with the received sequence.

Table 6-3 Top level Signal Description for the (3,2,2) Decoder.

6.4.1 (3,2,2) Branch Metric Unit

The design of the BMU is similar to the (2,1,3) case, however the input symbol R_x is 3-bits for (3,2,2) implementation, which effectively increases the number of BMU computations. At every trellis decoding stage the (3,2,2) BMU receives a 3-bit symbol R_x and computes $2^{(M+k)}$ or 32 Hamming distance branch metrics one for each trellis branch in the current trellis stage. The 32 outputs of the BMU, (HD1 to HD32) correspond to the trellis branch labels in table 6-4 below.

	S_{x+1} (s_{22},s_{21},s_{11})							
S_x(s_{22},s_{21},s_{11})	000	001	010	011	100	101	110	111
000	HD1	HD2	HD3	HD4	-	-	-	-
001	HD5	HD6	HD7	HD8	-	-	-	-
010	-	-	-	-	HD9	HD10	HD11	HD12
011	-	-	-	-	HD13	HD14	HD15	HD16
100	HD17	HD18	HD19	HD20	-	-	-	-
101	HD21	HD22	HD23	HD24	-	-	-	-
110	-	-	-	-	HD25	HD26	HD27	HD28
111	-	-	-	-	HD29	HD30	HD31	HD32

Table 6-4 (3,2,2) BMU output assignment table

For the (3,2,2) trellis, there can only be four valid Hamming distances of 0, 1, 2, 3. For example some (3,2,2) Hamming distances are computed as follows, $\delta(000,000) = 0$, $\delta(111,111) = 0$, $\delta(000,001) = 1$, $\delta(000,010) = 1$, $\delta(010,001) = 2$, $\delta(101,010) = 2$, $\delta(110,001) = 3$, $\delta(101,010) = 3$.

The circuit that computes Hamming distance branch metrics can be described by the following Verilog below. The actual synthesized circuit and symbol for this building block are then shown in figure 6-14.

```
module bm322(HD,Rx,Vx);

output [1:0] HD;
input  [2:0] Rx;
input  [2:0] Vx;

assign HD[1] = ((Rx[0]^Vx[0]) & (Rx[1]^Vx[1]) |
                (Rx[0]^Vx[0]) & (Rx[2]^Vx[2]) |
                (Rx[1]^Vx[1]) & (Rx[1]^Vx[1]) |
                (Rx[0]^Vx[0]) & (Rx[1]^Vx[1]) & (Rx[2]^Vx[2]));

assign HD[0] = ((Rx[0]^Vx[0]) ^ (Rx[1]^Vx[1])) ^ (Rx[2]^Vx[2]);

endmodule
```

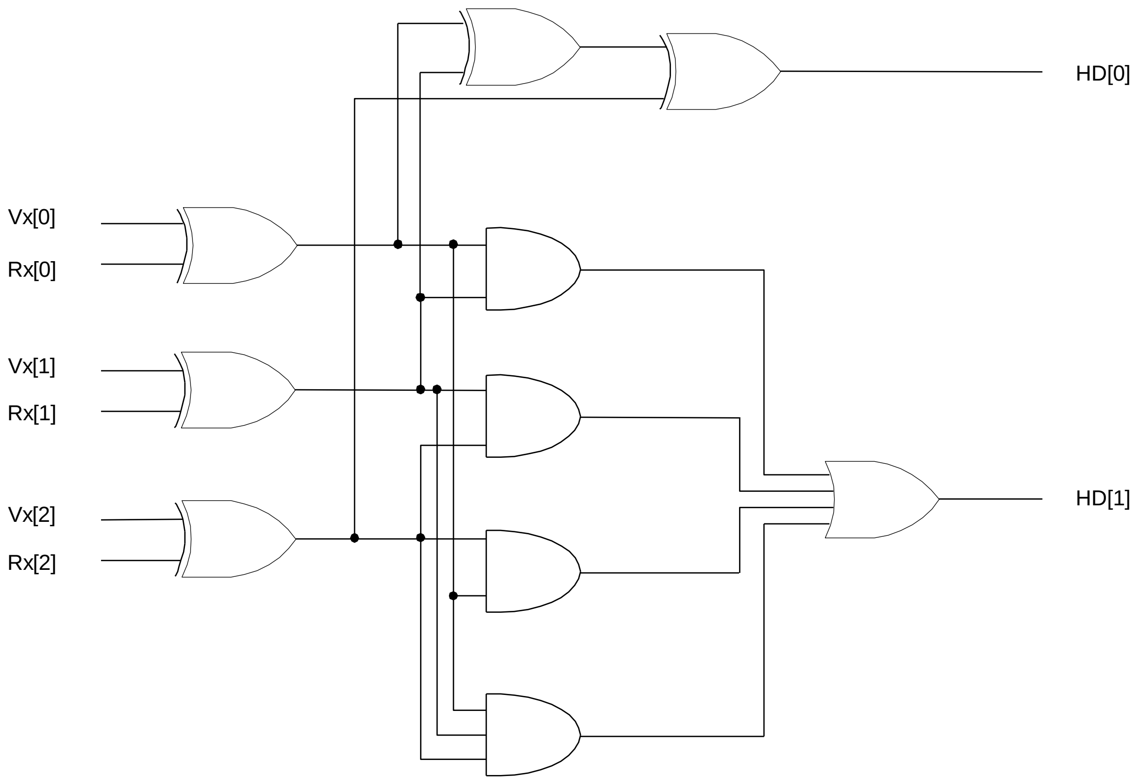


Figure 6-14 (3,2,2) HD Branch Metric Building Block

As in the (2,1,3) case, a single instance of the branch metric circuit can be used a number of times throughout the trellis stage as the branch labels are repeated. For the (3,2,2) case, the building block is instantiated eight times in order to compute each of the eight possible branch labels {000-111} for the entire BMU, as shown in figure 6-15. The outputs of the instantiated blocks are then registered when the load enable signal (*le*) is asserted to load and hold the branch metrics for the current trellis stage.

The registered outputs of the branch metric blocks are then shared by four outputs each, because in a (3,2,2) trellis stage there is always four branches with the same branch label. The entire (322) BMU is shown in figure 6-15.

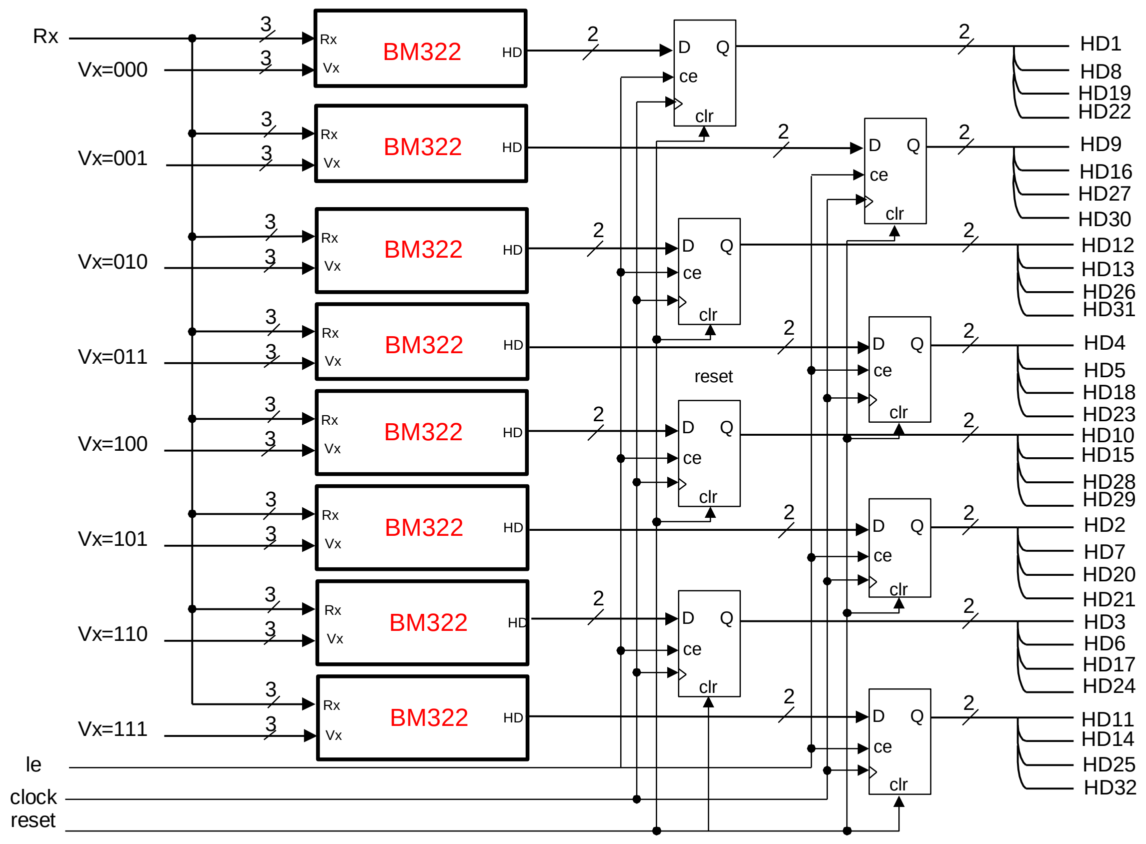


Figure 6-15 (3,2,2) BMU

6.4.2 (3,2,2) Add Compare Select Unit (ACSU)

Similar to the (2,1,3) case a new path metric and backward label are produced by the Add-Compare-Select (ACS) operation on every clock cycle. However the (3,2,2) ACS block has to select between four paths instead of two which makes the (3,2,2) ACS more complex. The (3,2,2) ACS block works as follows.

The range of metrics that can be computed by the each state path metric ACS is (0000_2-1111_2) or 0 – 15 in decimal. At the start of each pseudo block, state (000), is initialized to (0000_2), and the remaining seven path metric memories are initialized to their maximum value (1111_2). As the algorithm proceeds, if all four incoming path metrics equals (1111_2), the outgoing path metric of the ACS block is also (1111_2) to prevent overflow problems in the ACS, otherwise if one or more incoming path metrics is less than (1111_2), the ACS block determines the minimum path metric.

If the add enable signal (*ae*) is asserted each incoming path metric and the corresponding branch metric are added together. All sums from the Add operation are then fed into a ≤ comparator, with the output of the comparator generating the surviving backward label output. The comparator output is also used as the select signal for the 4:1 path metric MUX. This Verilog code to describe (3,2,2) ACS circuit is similar to the (2,1,3) version except there are four paths instead of two. The equivalent circuit for the (3,2,2) ACS building block is shown in figure 6-16.

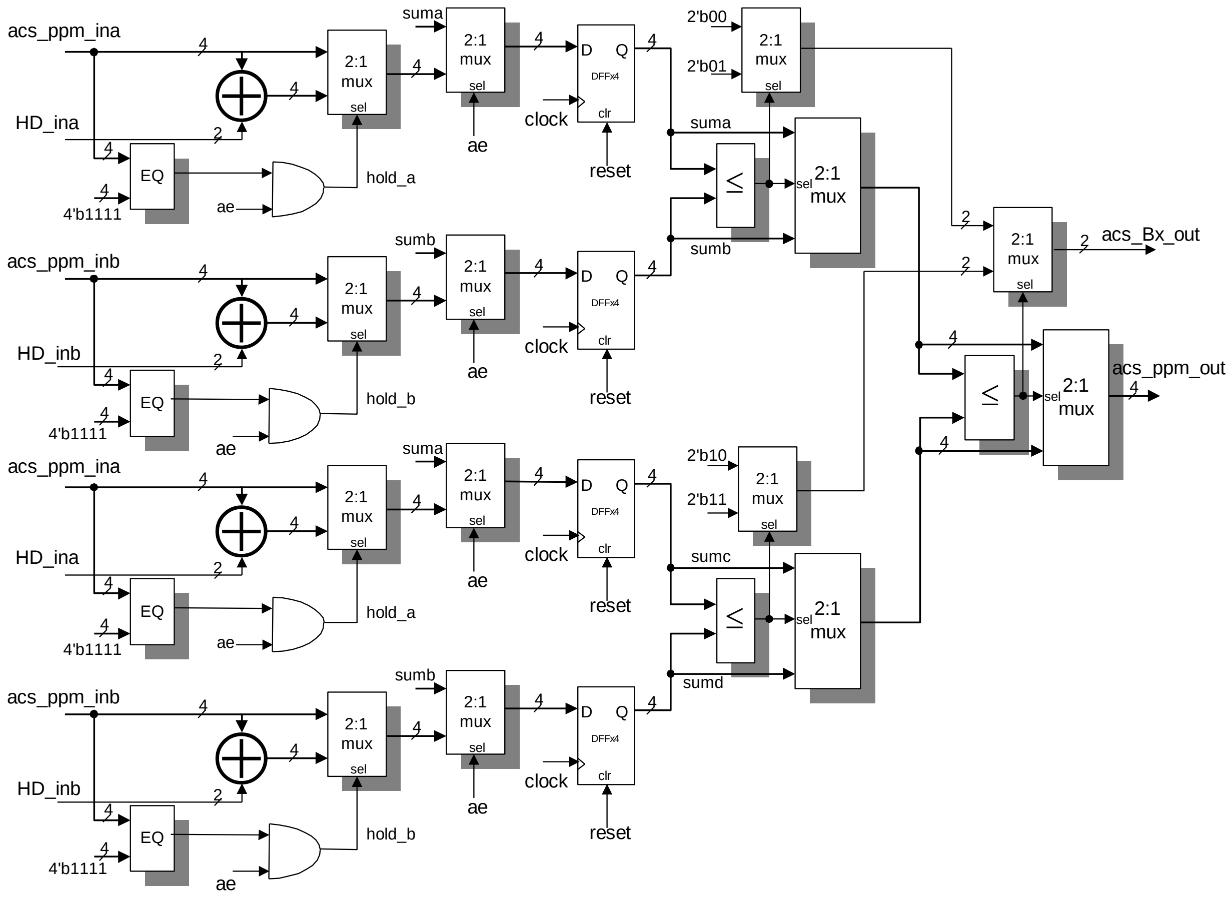


Figure 6-16 (3,2,2) ACS Building Block.

In the (3,2,2) decoder there are 2^M or eight states in a single trellis stage. For this design all eight path metrics and path labels are computed in parallel, by the ACSU block. As the process of producing path metrics and path labels is identical for all eight states, the ACS block is instantiated eight times to construct the ACSU.

6.4.3 (3,2,2) Conventional Path Memory Storage Unit

The (3,2,2) conventional path memory is organized into sixteen parallel shift registers (P0[1],P0[0]-P7[1],P7[0]), each of length ten, i.e. (2^M x 5m). At every trellis stage eight 2-bit backward labels are produced by the ACSU. If the write enable (*we*) signal is asserted these backward labels are shifted into the corresponding path memory shift register with the existing register contents moving one place to the right. A path memory write pointer is incremented each time a set of backward labels are shifted in. The organization of the path memory shift registers is shown in figure 6-17.

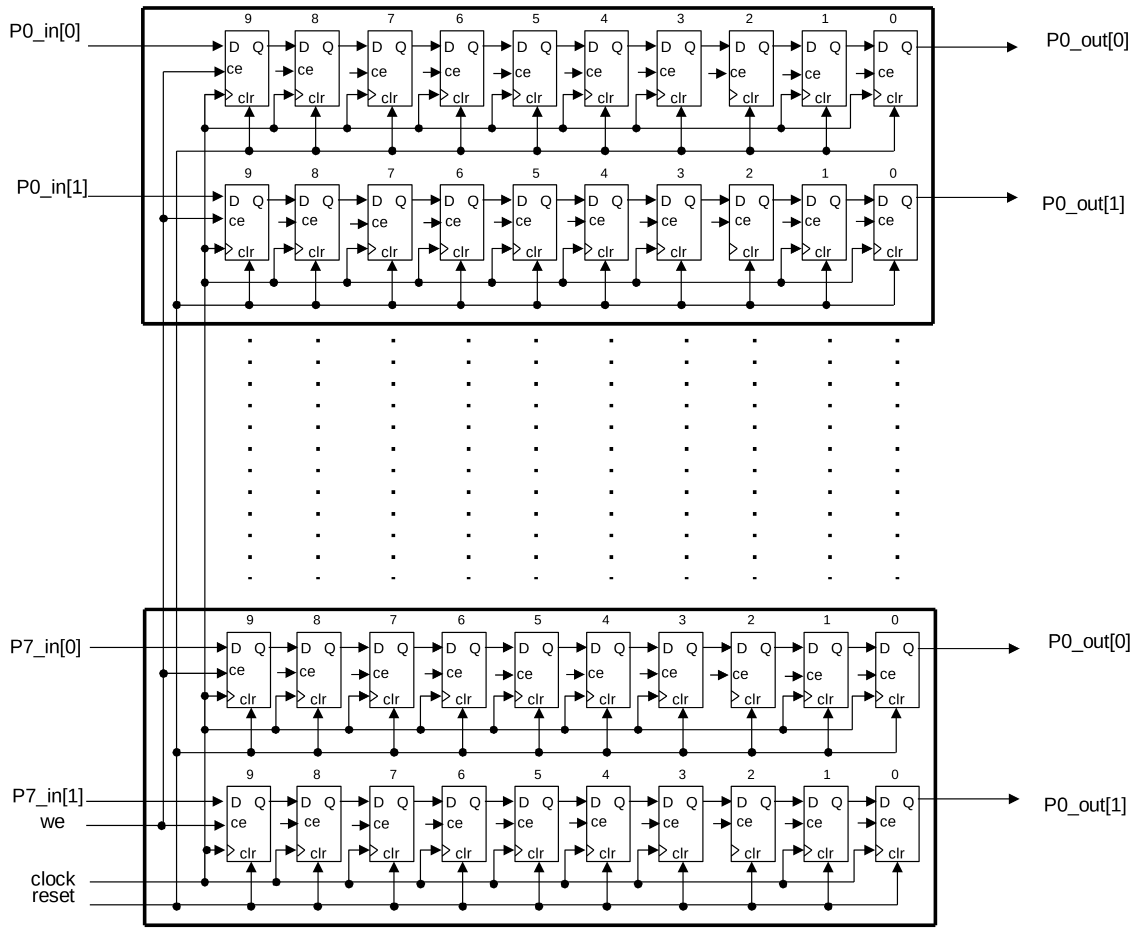


Figure 6-17 (3,2,2) Path Memory.

6.4.5 Conventional (3,2,2) Traceback & Output Decision Unit

Similar to the (2,1,3) case, this block contains similar elements as follows:

- An 8-way ≤ comparator to select the traceback start state,
- An 8:1 MUX to select a backward label,
- A traceback pointer,
- A traceback register,
- A tri-state enabled output buffer,
- Out of Sync Detector

The operation of selecting the minimum path metric state and loading the traceback register is identical to the (2,1,3) case. For every clock cycle in the traceback operation, the backward label bits *{b2,b1}* are read from path memory and shifted into the traceback register elements *s22*, and *s11* respectively. Once the traceback pointer terminates, the control block asserts the output enable (*oe*) signal to the tri-state output to release a decoded symbol from the *s21* and *s11* of the traceback register elements. The block diagram of the output decision block is shown below in figure 6-18.

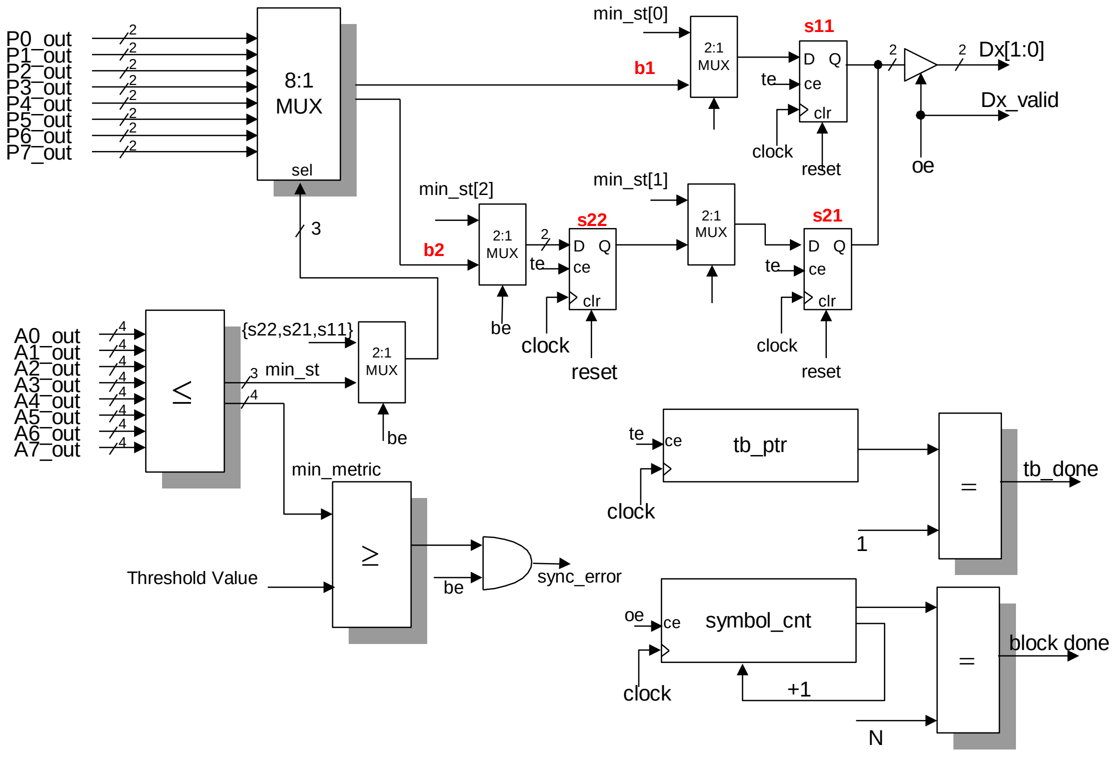


Figure 6-18 (3,2,2) Output Decision & Traceback Unit

6.4.6 (3,2,2) Efficient Path Memory Storage Unit

The (3,2,2) efficient path memory is organized into sixteen parallel shift registers (P0[1],P0[0]-P7[1],P7[0]), each of length nine and eight, as shown in figure 6-19. Each 2-bit path label is organized as two parallel shift registers of length eight and nine. The operation of shifting in backward labels is identical to the conventional case.

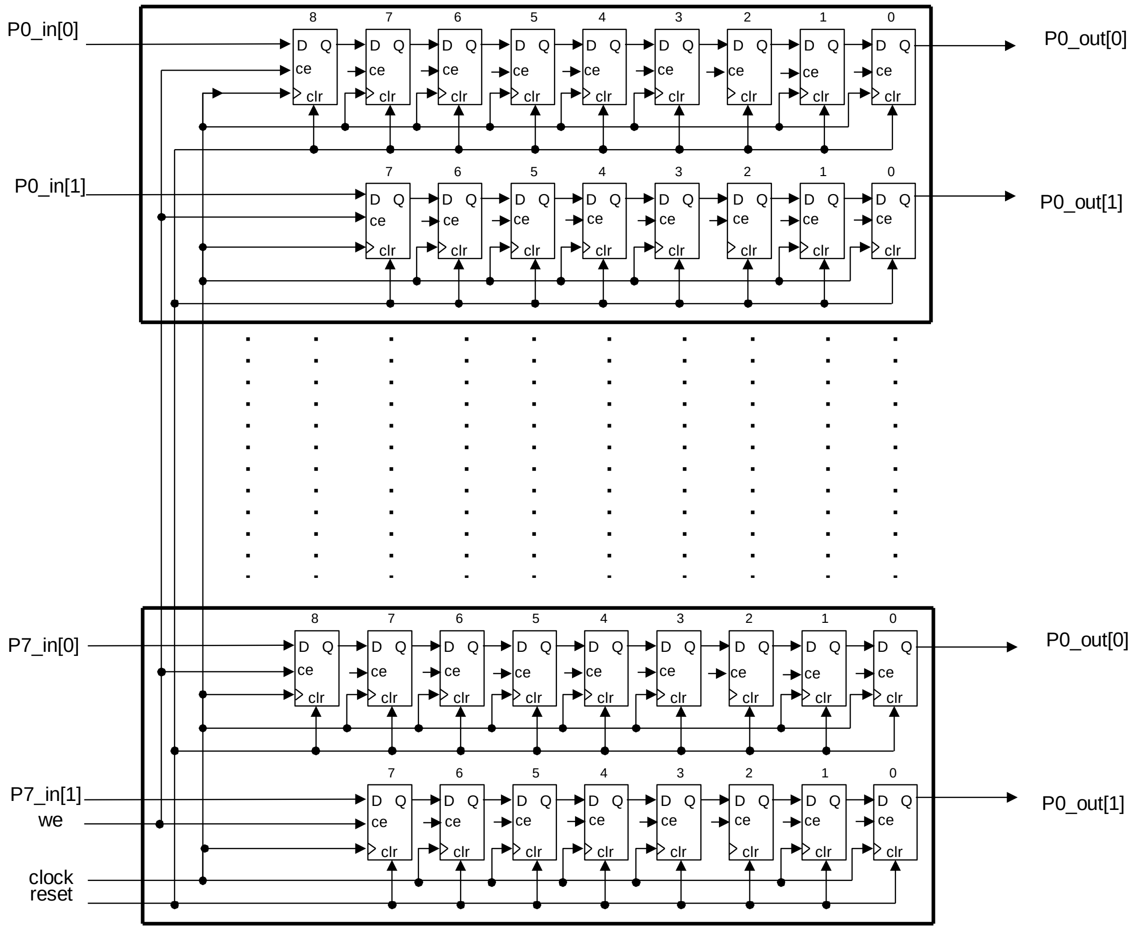


Figure 6-19 (3,2,2) Efficient Path Memory.

6.4.7 Efficient (3,2,2) Traceback & Output Decision Unit

Like previous implementations, this block contains the following elements:

- An 8-way ≤ comparator to select the traceback start state,
- An 8:1 MUX to select a backward label,

- A traceback pointer,
- A traceback register,
- A tri-state enabled output buffer,
- Out of Sync detector.

The operation of tracing back is identical to the conventional case. However, the traceback operations terminate earlier than the conventional case because of the smaller memory requirements. In addition, the output decision circuit is wired differently than the conventional case in order to take advantage of the efficient traceback architecture.

In the efficient implementation, the most significant bit is wired to s22 instead of *s21*, and least significant bit is wired to *b1* instead of the *s11* in the conventional case. The output decision circuit is shown in figure 6-20.

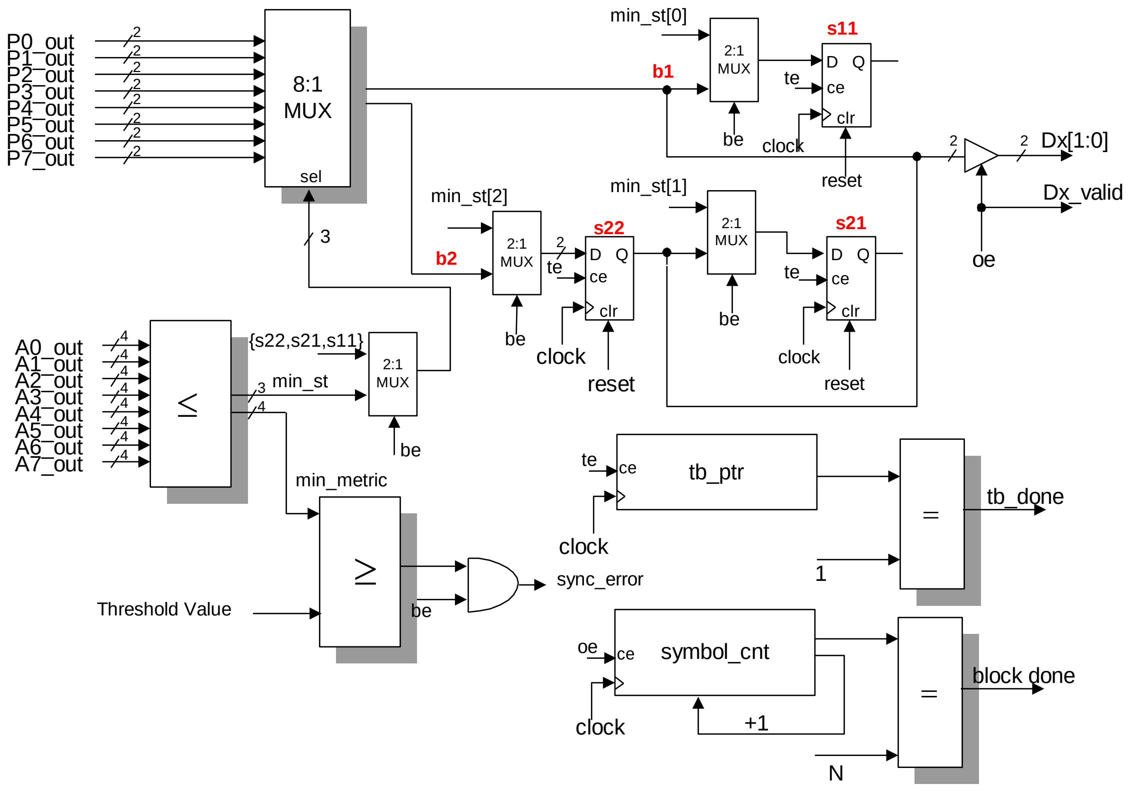


Figure 6-20 (3,2,2) Efficient Traceback & Output Decision Unit

6.5 Hardware Design - Control Unit

The control unit for the Viterbi decoder is designed to control the entire data path from calculating branch metrics in the BMU to the traceback process to releasing decoded symbols. The control unit moves data in a pipelined fashion through the decoder by enabling and disabling control signals in sequence to the appropriate blocks. The hardware design for the sequencer is carried out by the use a finite state machine FSM. A FSM is any circuit specifically designed to sequence through specific patterns of states in a predetermined sequential manner. A FSM can be represented by a state table or a state diagram. A state diagram is a graphical representation of a state machines sequential operation. A FSM is used in the control block of the Viterbi decoder to enable various control signals throughout the decoding process. Typically these control signals are for the BMU (*le*), ACSU (*ae*), and storage blocks (*we*), and are used to clock data into a bank of flip flops and hold the data for one trellis stage. The state diagram for the entire Viterbi decoder is shown below in figure 6-21.

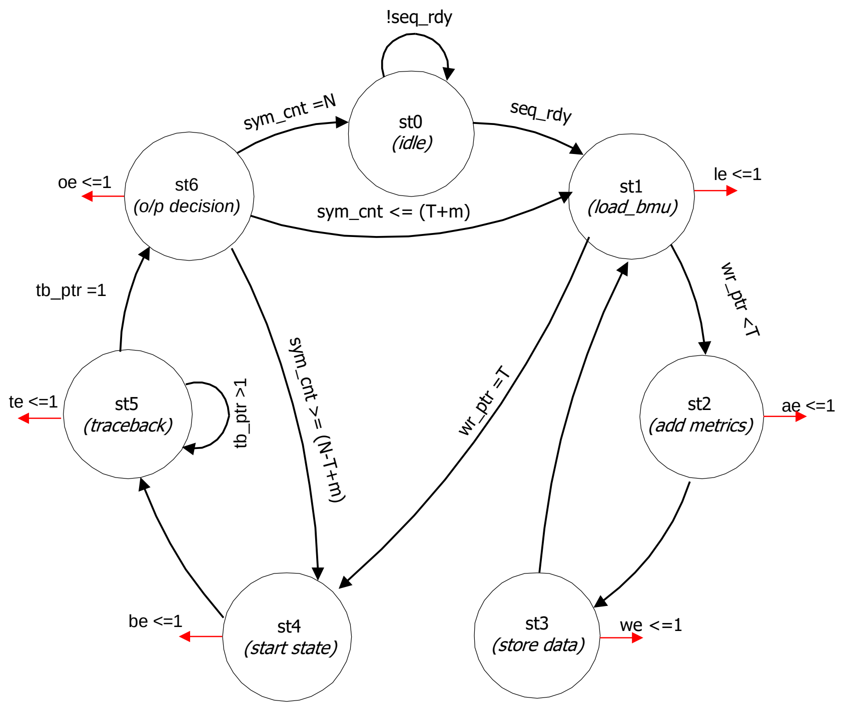


Figure 6-21 Viterbi Decoder State Diagram

The sequences of states (st0 – st6) in the state diagram above is represented by the following Verilog code. The equivalent timing diagram for all decoders up to and including the first decoded symbol are shown in figure 6-22 through 6-25.

```
/*=== Viterbi Decoder State Machine ===*/
always @ (c_st or seq_rdy or pm_full or tb_done or Rx_done or blk_done)
   case (1)   // synopsys parallel_case
   c_st[0]:   n_st <= (seq_rdy) ? st1 : st0;                      // Idle State
   c_st[1]:   n_st <= (~pm_full) ? st2 : st4;                     // Load_HD State
   c_st[2]:   n_st <= st3;                                        // Add_HD State
   c_st[3]:   n_st <= st1;                                        // Store Data
   c_st[4]:   n_st <= st5;                                        // Find MinState
   c_st[5]:   n_st <= (~tb_done) ? st5 : st6;                     // Perform Traceback
   c_st[6]:   n_st <= (~Rx_done) ? st1 : (~blk_done) ? st4 :st0;  // Release Symbol
   default:   n_st <= st0;
   endcase
```

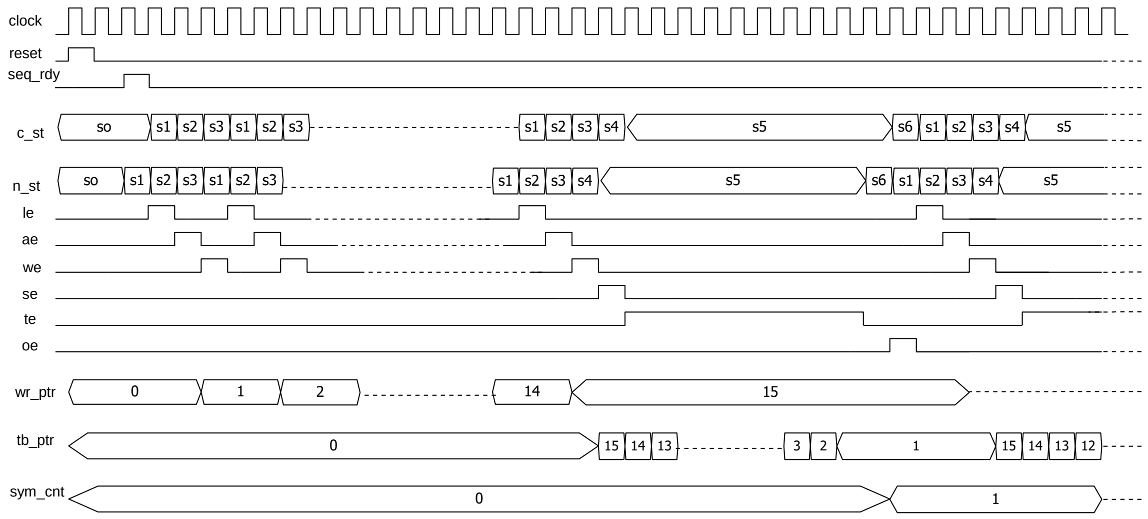


Figure 6-22 Conventional (2,1,3) Decoder Timing.

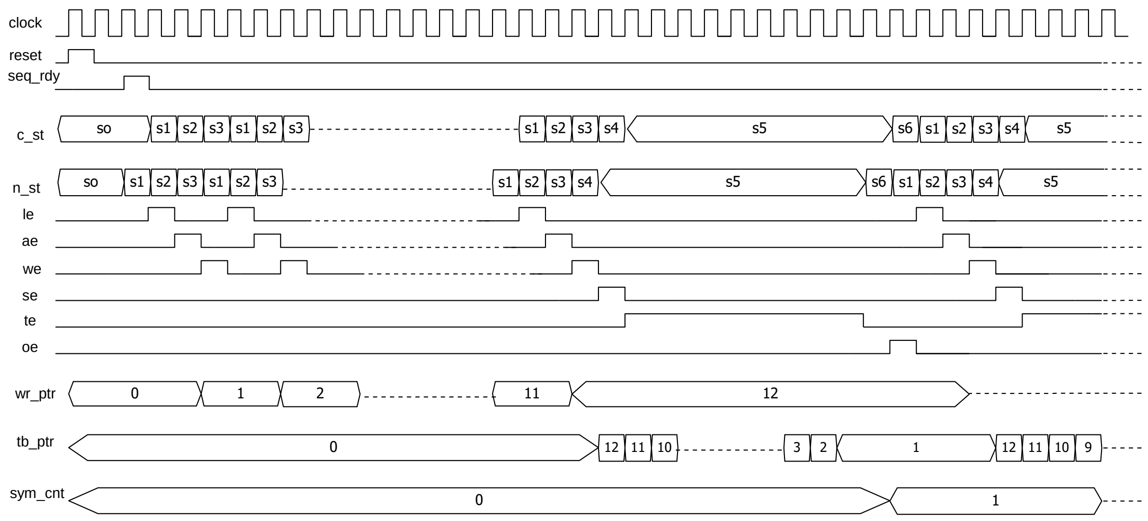


Figure 6-23 Efficient (2,1,3) Decoder Timing.

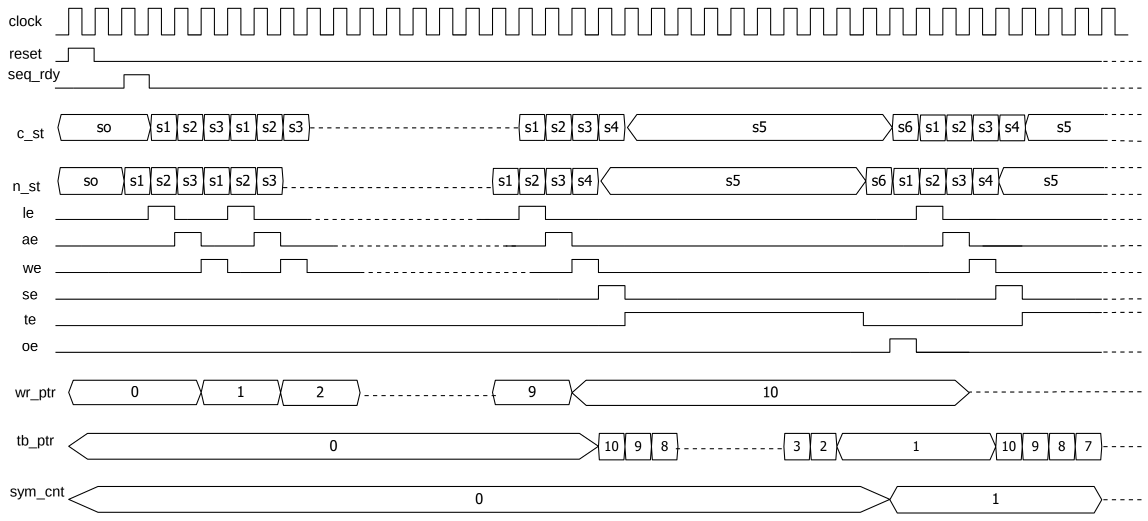


Figure 6-24 Conventional (3,2,2) Decoder Timing.

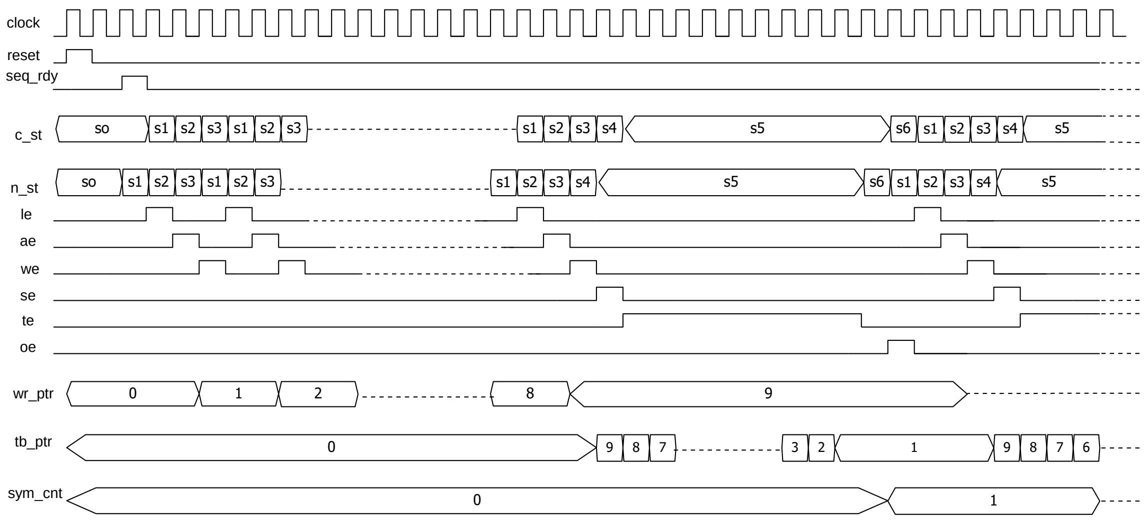


Figure 6-25 Efficient (3,2,2) Decoder Timing.

6.6 Hardware Design Process

This section details the hardware design process using the commonly used ASIC/FPGA Verilog Hardware Description Language (HDL). Specifically this project is targeted to the popular Xilinx Virtex series FPGA, however the Verilog code for the designs can be ported easily to other FPGA/ASIC synthesis processes.

There are multiple FPGA vendors on the market and each vendor provides a unique FPGA architecture, however a lot of similarities can be drawn between each vendor. Section 6.6.1 provides an overview of FPGA architectures concentrating on the Xilinx Virtex FPGA. This is followed by section 6.6.2 which discusses the HDL design and verification process for the project. The FPGA implementation results are then discussed in section 6.6.3, which provides details of the actual hardware savings gained for the (2,1,3) efficient based designs, and section 6.6.4 which details the (3,2,2) decoder savings.

6.6.1 Overview of Xilinx Virtex FPGA Architecture

An FPGA is a programmable device with an internal array of logic blocks, surrounded on the periphery by programmable input/output blocks. The entire array is connected with programmable interconnect. There are two types of FPGA architectures. One is coarse-grained, and the other is fine-grained. Coarse-grained architectures consist of ‘large’ logic blocks, and typically contain two or more programmable look-up tables and two or more flip-flops. In this type of architecture, typically there are four-input look-up tables which can be used to generate any 4-variable combinatorial function. In some cases, this same four-input look-up table (LUT) can also be configured to operate as a 16x1 ROM or 16x1 RAM. Fine-grained architectures consist of relatively ‘simple’ logic blocks, and are constructed using either a two-input logic function or a 4-to-1 multiplexer and a flip-flop. If the design is written in Verilog HDL, an FPGA synthesis tool is used to map the logic implied in the Verilog code to the internal FPGA logic blocks.

The Xilinx Virtex family has a course-grained architecture. In the Virtex FPGA family the basic logic building block is called configurable logic block (CLB), [16]. Each CLB has two identical slices, and each slice contains two 4-input look up tables

(LUTs), two D-type flip-flops with dedicated clock enable, set or reset, and fast carry-in and carry-out signal paths. The 4-input LUTs in a CLB can perform any 4 variable Boolean function or can be configured as a 16x1 Ram. Around the periphery of the Virtex FPGA are configurable input/output blocks (IOB). Each Virtex IOB includes configurable termination (pull-up/pull-down), slew rate control, and flip-flops that can used to register I/Os. Other features of the Virtex FPGA include an on-chip oscillator and dedicated low-skew networks that can be used for clocks and other fast global signals, internal tri-state signals and busses. Figure 6-26 illustrates the Xilinx Virtex architecture features.

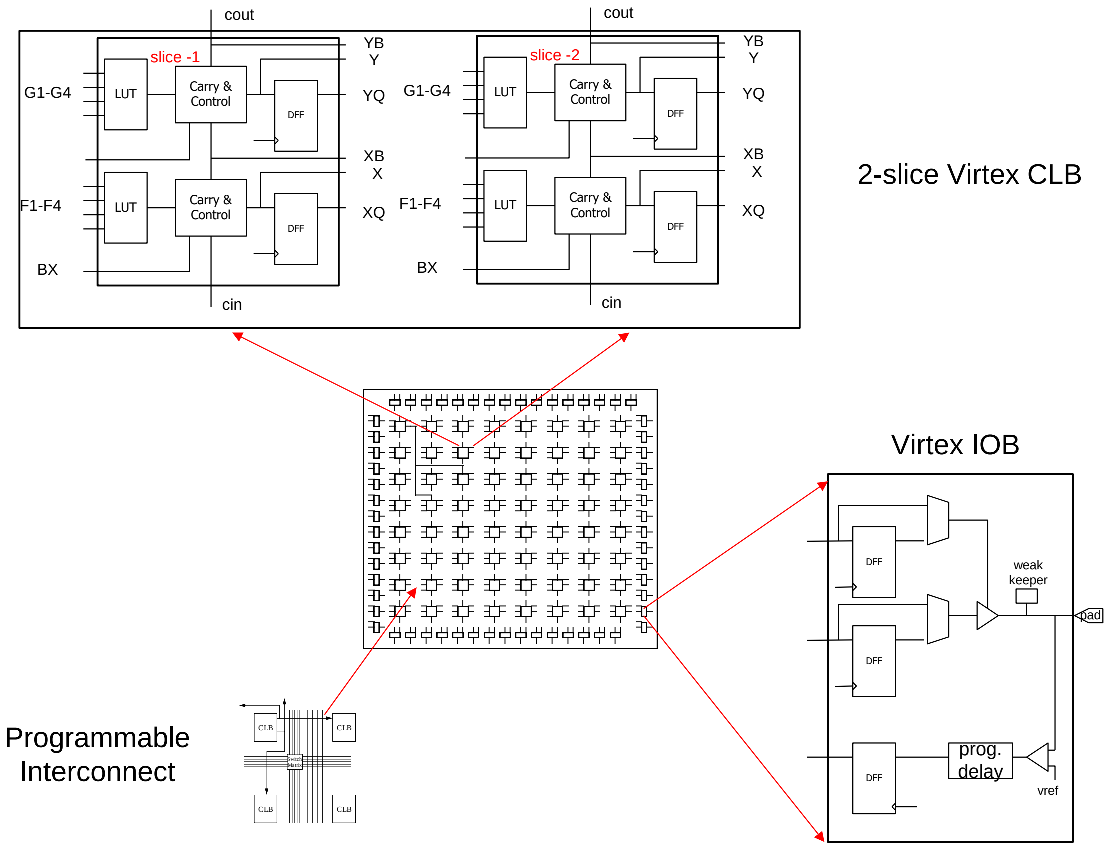


Figure 6-26 Xilinx Virtex Architecture

The target device used for the Viterbi decoders is the Virtex XCV50. This device has 1536 flip-flops, and 1536 LUTs in 768 slices.

6.6.2 HDL Design Flow

An FPGA design can be captured in a number of different methods. A schematic entry tool can be used to capture a design or more commonly with a hardware description language (HDL) such as Verilog, VHDL, or ABEL. For this project, Verilog HDL was used to capture the designs. The design flow using Verilog is shown below in figure 6-27.

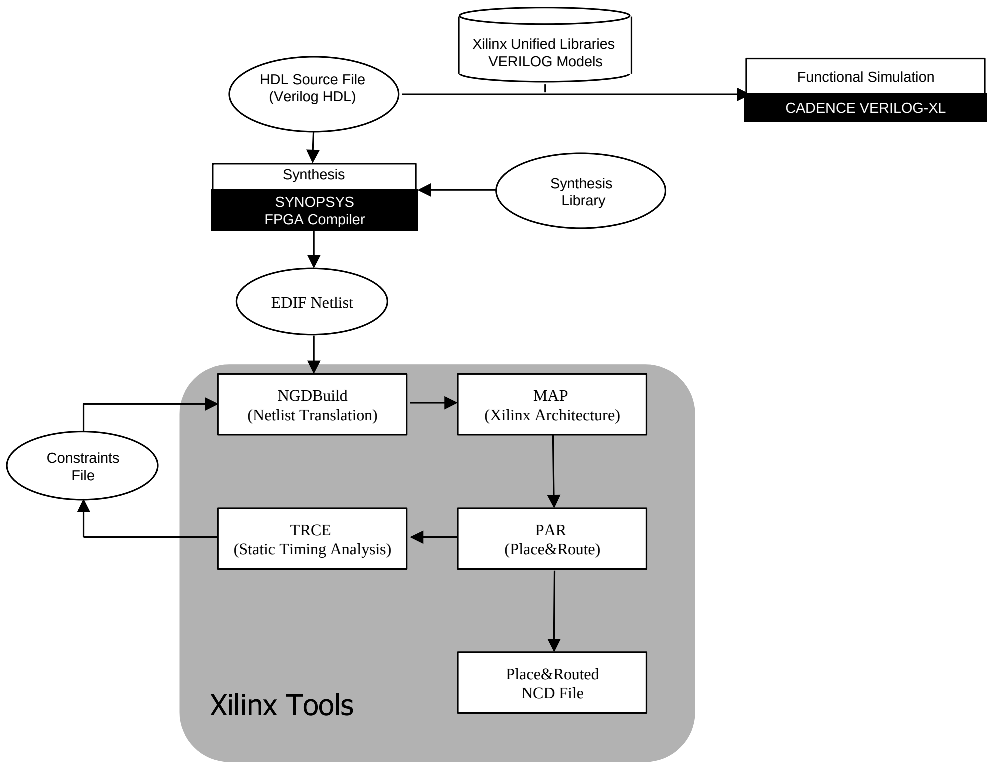


Figure 6-27 Verilog Design Flow for Viterbi Decoders

Initially, the Cadence Verilog-XL tool was used to check the Verilog code syntax. Once all syntax errors are corrected, the next step is to verify the functionality of the design. Typically the verification step is carried out with a Verilog based test-bench program. A test-bench is typically a wrapper around the design and used to apply stimulus to the design and monitor responses. For this project a test bench was written to stimulate the decoders under test with known convolutional coded test sequences.

The decoded sequences are then verified against known convolutional decoded sequences as shown in figure 6-28.

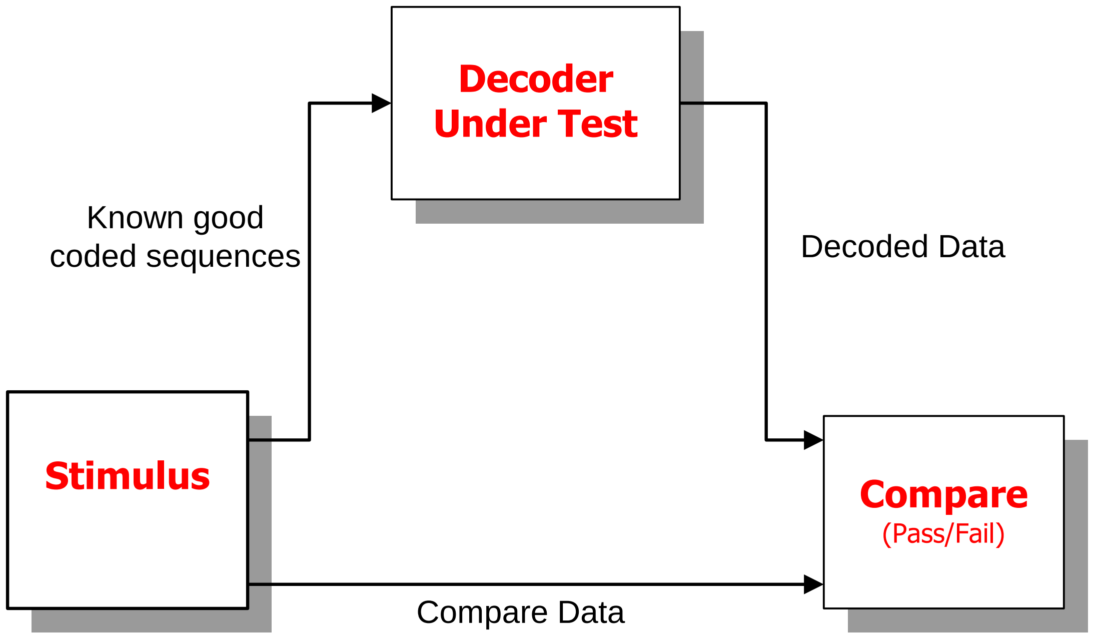


Figure 6-28 Verilog HDL Test Bench Environment

Once the Verilog designs simulate correctly, the code is passed through the synthesis step to produce gates. The synthesis step used the Synopsys FPGA Compiler, which takes the simulated Verilog code and generates an EDIF netlist.

The Xilinx backend tools are then used to produce a functional design in an FPGA.

6.7 Hardware Implementation Results

This section provides the logic utilization for the conventional and efficient decoders as implemented in a Xilinx XCV50 FPGA. Section 6.7.3 then details the performance improvements of the efficient designs.

6.7.1 (2,1,3) Decoder Implementation Results

As outlined in section 6.6.2, each CLB in a Xilinx XCV50 FPGA has two identical slices, where the main component in each slice are two 4-input look up tables (LUTs), and two D-type flip-flops. The actual synthesized circuits for both the conventional and efficient designs will thus be reported in terms of the number of LUTS and DFFs used per module. As the decoder designs are modularized, some of the building blocks are identical between both implementations. Table 6-5 provides a logic utilization comparison between both (2,1,3) implementations.

Module	Conventional Design		Efficient Design	
	LUTs	DFFs	LUTs	DFFs
BMU	9	32	9	32
ACSU	184	64	184	64
Path Memory	0	120	0	96
Path Metric Memory	0	32	0	32
state_sel	99	19	99	19
Control, Traceback, & Output Decision Logic	200	34	157	34
TOTAL	492	301	449	277

Table 6-5 (2,1,3) Utilization Comparison.

6.7.2 (3,2,2) Decoder Implementation Results

Similar to the (2,1,3) case, the actual synthesized circuits for both the conventional and efficient designs is reported in terms of the number of LUTS and DFFs used per module. Table 6-6 provides a logic utilization comparison between both (3,2,2) implementations

Module	Conventional Design		Efficient Design	
	LUTs	DFFs	LUTs	DFFs
BMU	16	62	16	62
ACSU	448	128	448	128
Path Memory	0	160	0	136
Path Metric Memory	0	32	0	32
state_sel	99	19	99	19
Control, Traceback, & Output Decision Logic	207	34	216	38
TOTAL	770	435	779	415

Table 6-6 (3,2,2) Utilization Comparison.

6.7.3 Decoding Performance

This section discusses some metrics for evaluating the decoding performance of the Viterbi decoders in this project. It is demonstrated that Viterbi decoder implementations using the novel path memory have a faster decoding time.

The sequence of operations and timing diagrams for all decoder implementations were detailed in section 6.5. The decoders remains in the idle state until a new sequence is ready to be decoded, denoted by the assertion of "*seq_rdy*". Once asserted, the decoder performs the following operations for each trellis stage until the path memory is full:

- 1 clock cycle to compute the branch metrics.
- 1 clock cycle to compute the path metrics, and backward labels
- 1 clock cycle to store the path metrics and backward labels

These three operations are repeated for each of the T stages until the path memory is full. Once the path memory is full, it takes 1 clock cycle to register the minimum path metric state, followed by T clock cycles for traceback. Once the traceback pointer reaches the beginning of path memory the first decoded symbol is then determined from the traceback mapping register. Therefore the number of clock cycles to produce the first symbol amounts to 3T + (T+1) clock cycles.

After the first decoded symbol is produced, the oldest path memory contents are shifted out as the next set of backward labels are shifted in. The corresponding path metric contents are also updated. The path memory is now full again. The state with the minimum path metric is registered and traceback recommences. Therefore the number of clock cycles to produce each of the remaining N-1 symbols is (T+4) clock cycles per symbol:

- 1 clock cycle to compute the branch metrics
- 1 clock cycle to compute the path metrics, and backward labels
- 1 clock cycle to store the path metrics, and backward labels
- 1 clock cycle to register the minimum path metric state
- T clock cycles to perform traceback

An expression for the total number of clock cycles is now derived. Given the truncated path memory length T, the total number of clock cycles, *C* performed by a conventional Viterbi decoder to decode an N-bit sequence can be determined as:

$$C = 3\mathrm{T} + (\mathrm{T}+1) + \left(\prod_{\mathrm{i}=1}^{\mathrm{N-1}} (\mathrm{T}+4) \right)$$

For the conventional (2,1,3) decoder this sequence of operations are repeated for all fifteen stages of path memory. Therefore the total number of clock cycles taken to fill path memory is 3T or 45. Once the path memory is full, the decoder takes one clock cycle to register the minimum metric state, and finally fifteen clock cycles to traceback through the path memory. Therefore in total the decoder takes (45+1+15) = 61 clock cycles before the first decoded symbol is released. A calculation for the total number of clock cycles consumed to decode an entire 225 symbols is now illustrated for the conventional (2,1,3) decoder:

$$C_{b213} = 3\mathrm{T} + (\mathrm{T}+1) + \left(\prod_{\mathrm{i}=1}^{\mathrm{N-1}} (\mathrm{T}+4) \right)$$
$$C_{b213} = (61) + (224(19))$$
$$C_{b213} = 4{,}317$$

For the (2,1,3) efficient backward label decoder, there are twelve path memory stages. The total number of clock cycles *C* consumed by an efficient Viterbi decoder to decode same sequence is:

$$C_{e213} = 3\mathrm{T} + (\mathrm{T}+1) + \left(\prod_{\mathrm{i}=1}^{\mathrm{N-1}} (\mathrm{T}+4) \right)$$
$$C_{e213} = (49) + (224(16))$$
$$C_{e213} = 3633$$

This is a percentage gain in decoding performance of approximately 16% for the (2,1,3) efficient decoder.

Similarly, the conventional (3,2,2) backward label decoder has a truncated path memory of ten stages. During every path memory write, k = 2 bits are written to two of these shift registers.A calculation for the total number of clock cycles consumed to produce an entire decoded sequence of 150 symbols is now illustrated for the conventional (3,2,2) decoder:

$$C_{b322} = 3\mathrm{T} + (\mathrm{T}+1) + \left(\prod_{\mathrm{i}=1}^{\mathrm{N-1}} \left(\mathrm{T}+4\right) \right)$$
$$C_{b322} = (41) + \left(149(14)\right)$$
$$C_{b322} = 2{,}127$$

For the (3,2,2) efficient backward label decoder, there are a maximum of nine stages of path memory, and the total number of clock cycles consumed to decode the same sequence is:

$$C_{e322} = 3\mathrm{T} + (\mathrm{T}+1) + \left(\prod_{\mathrm{i}=1}^{\mathrm{N-1}} \left(\mathrm{T}+4\right) \right)$$
$$C_{e322} = (37) + \left(149(13)\right)$$
$$C_{e322} = 1{,}974$$

This is a percentage gain in decoding performance of approximately 7%.

7. Conclusions

A realization of a novel path memory implementation for use in traceback Viterbi decoders is demonstrated. It is shown that by implementing this novel technique in hardware based Viterbi decoders yields a 20% saving in storage requirements for all (n, 1,m) codes and certain (n,k,m) codes, where each of the k parallel shift registers are of the same length. For (n,k,m) codes where each of k parallel shift registers are of different lengths the results are typically less than 20%. The (3,2,2) decoder in this project produces 15% savings in path memory storage requirements.

A modified traceback algorithm is presented to manage the novel path memory unit. In fact this modification is in the output decision circuit, and merely selects the decoded symbol from earlier states in the traceback register. A number of comparative FPGA based designs provided further confidence in the novel technique. The novel decoders provide a faster decoding time. A metric for measuring decoding performance is presented, and it is shown that gains in decoding performance are improved significantly. The novel (2,1,3) decoder has a performance gain of 16%, with the novel (3,2,2) decoder having a performance gain of 7%.

Finally from studying traceback architectures described in the literature, the storage and performance gains for the novel path memory technique can be easily incorporated into existing architectures with minimal effort for significant gain.

This project has not looked at the impact of the novel technique for punctured convolutional codes. Further work is desirable to evaluate these applications.

Appendix A.1 Definition of Terms

An (n, k, m) non-systematic convolutional encoder is an application of k parallel serial-in parallel-out shift registers.

k: Number of bits in input symbol.

n: Number of bits in output symbol.

m: Encoder memory, defined as the length of the longest of it's k shift registers, m = max(m_j).

M: Total encoder memory, M= $\sum_{j=1}^{k} m_j$

g: Generator Polynomial matrix which describes the connections between the k shift registers and the modulo-2 adders.

U: Input sequence.

V: Convolutional encoded sequence.

R: Received sequence.

U': Decoded sequence.

N: Length of input sequence, U.

U_x: k-bit input to convolutional encoder at stage *x*. $U_x = (u_k, u_{k-1}, \dots u_1)$.

V_x: n-bit symbol from convolutional encoded sequence at stage *x* $V_x = \{v_n, v_{n-1}, \dots v_1\}$.

R_x: n-bit symbol received from channel at stage *x*. $R_x = \{r_n, r_{n-1}, \dots r_1\}$.

F_x: Forward label, $F_x = (f_{k,x}, f_{k-1,x}, f_{1,x}) = (f_k, f_{k-1}, f_1)_x = (u_k, u_{k-1}, u_1)_x$

B_x: Backward label, $B_x = (b_{k,x}, b_{k-1,x}, b_{1,x}) = (b_k, b_{k-1}, b_1)_x = (S_{k,mk}, S_{k-1,mk-1}, \dots S_{1,m1})_x$.

S_x: State of encoder memory at stage *x*. $S_x = (S_{j,i})_x \quad j = k \dots 1 \quad i = m_j \dots 1$.

$\delta(R_x, V_x)$: Hamming distance between branch label V_x and received branch label R_x

T: Truncation length of path memory, related to the traceback depth d.

$S_{x-1} = f(S_x, B_x)$: Traceback mapping function.

$C = 3T + (T+1) + \left(\prod_{i=1}^{N-1}(T+4)\right)$: Formula for computing the number of clock cycles in a Viterbi decoder with a path memory of length T.

Appendix A.2 Glossary of Terms

1. ACS: Add-Compare-Select Block. Used in a Viterbi decoder to compute partial path metrics and path labels of surviving paths.
2. ACSU: Add-Compare-Select Unit. Overall partial path metric and path label computation unit, and contains 2^M ACS blocks
3. BMU: Branch Metric Unit. Used in a Viterbi decoder to compute a set of branch metrics for each trellis stage.
4. CLB: Configurable Logic Block. The basic logic block used in the Xilinx XC4000 FPGA architecture.
5. FPGA: Field Programmable Gate Array
6. HDL: Hardware Description Language. A high level language used to design digital logic circuits. There are many standards, notably Verilog, VHDL, and ABEL
7. IOB: Input Output Block. The basic I/O logic block used in Xilinx FPGA architectures.
8. SMU: Survivor Memory Unit. Another expression used to describe the path memory unit in a Viterbi decoder.
9. SYNTHESIS: The process of taking a logic design captured with a schematic or a HDL and translating it to a logic circuit
10. VERILOG: A hardware description language used for capturing large digital designs.

Appendix B. Source Code

The source code for the MATLAB and Verilog based designs are too large to include in an appendix. Instead the source code is a compressed archive. All Verilog files have a .v suffix, with all MATLAB files have a .m suffix. A read-me file is also included documenting the files.

Bibliography

[1] A.J. Viterbi, *Error Bounds for Convolutional Codes and an Asymptotically Optimum Decoding Algorithm*. IEEE Transactions on Information Theory, Vol. IT-13, April 1967, pp. 260-269.

[2] G.D. Forney Jr. *The Viterbi Algorithm*. Proc. IEEE Vol. 61, 1973, pp. 268-278.

[3] J.K. Omura, *On the Viterbi Decoding Algorithm*. IEEE Transactions on Information Theory, Vol. IT-15, 1969, pp. 177-179.

[4] M. O'Droma, et al, *More Efficient ASIC Implementation of digital radio CODECS*. I.E.E.E. 1997

[5] M. O'Droma, et al, *Memory Savings in Viterbi decoders for (n,1,m) convolutional codes*. I.E.E 1997.

[6] M. O'Droma, et al, *VLSI Design of a of GSM channel decoder*. Report, University of Limerick and Teltec Irl., Ireland, July 1993.

[7] S. Wicker, *Error Control Systems for Digital Communication and Storage.* Prentice-Hall Inc., 1995

[8] T. Truong, et al, *A VLSI Design for a Trace-Back Viterbi Decoder* IEEE Transactions on Communications, Vol. 40, No.3, March 1992, pp. 617-624.

[9] R. Cypher, et al, *Generalised Trace-Back Techniques for Survivor Memory Management in the Viterbi Algorithm*. Journal of VLSI signal processing, 5, Jan 1993, pp85-94.

[10] M. Horwitz, et al, *A Generalised Design Technique for Traceback Survivor Memory Management in the Viterbi Decoders*. Proc. I.E.E.E, 1997

[11] S. Jung, et al, *A New Survivor Memory Management Method in Viterbi Decoders*. Proc. I.E.E.E, Vol. 1 1996, pp.126-130.

[12] C. Rader. *Memory Management in a Viterbi Decoder*. IEEE Transactions on Communications, Vol. 29, no.9, Sept 1981, pp. 1399-1401.

[13] G. Feygin, et al, *Survivor Sequence Memory Management in a Viterbi Decoder.* Proc. IEEE, Vol. 5, June 1991, pp. 2967-2970.

[14] B. Sklar, *Digital Communications: Fundamentals and Applications*. Prentice-Hall Inc., Englewood Cliffs, N.J., 1988

[15] Lin and Costello, *Error Control Coding: Fundamentals and Applications*. Prentice-Hall Inc., Englewood Cliffs, N.J., 1983

[16] Xilinx, *The Programmable Logic Data Book*, 2000

[17] World Patents: 004396397#85-231796/38; 004404918#85-231796/38; 004507556#86-010900/02; 004615631#86-118975/18; 007246970#87-243977/35; 007294747#87-291754/41; 007559253#88-193185/28; 007506667#88-140600/20; 007731046#88-364978/51; 007731047#88-364979/51; 007849089#89-114201/15; 007851916#89-117028/16; 007959916#89-225028/31; 008124360#90-011361/02; 008124361#90-011361/01; 008397351#90-284352/38; 008463423#90-350423/47; 008510742#91-023819/04; 008843925#91-347940/48; 008890210#92-017479/03; 009216122#92-340544/41; 009251008#92-378425/46; 009303177#92-430586/52; 009442189#93-135706/17.