A Hardware Implementation of a Viterbi Decoder for a $(3, \frac{2}{3})$ TCM Code

by

Michael Horwitz

B.Sc(Eng) *University of the Witwatersrand*

Submitted to the Department of Electrical Engineering
in fulfilment of the requirements for the degree of

M.Sc(Eng)

at the

UNIVERSITY OF CAPE TOWN

October 1997

Signature of the Author ..... Signed .....
Department of Electrical Engineering
September 29, 1997

Certified by .....
Dr. Robin Braun
Director of the Digital Communications Research Group
Thesis Supervisor

Accepted by .....
Prof. Gerhard de Jager
Acting Head of Department

Declaration

I declare that this thesis is my own unaided work. It is being submitted for the degree of Master of Science in Engineering at the University of Cape Town. It has not been submitted before for any degree or examination.

Signed

.....

(Signature of Candidate)
September 28, 1997

Acknowledgements

There are many people who helped make this thesis a success either by offering some advice, or by encouraging me along the way. I would like to thank them all, and assure them that their help was much appreciated. In particular, I would like to thank the following people:

My parents and my brother for their help, support and encouragement throughout my studies.

My supervisor Dr. Robin Braun for his help and guidance.

Plessey Tellumat Ltd. whose financial support made the work possible. In particular I would like to thank Johan Gericke from Plessey who always offered words of encouragement, even when progress was slow.

Among my fellow students I would like to thank Johan van de Groenendaal for his help, advice, and proof reading of the thesis. Thanks also to Johan Schoonees for his advice and support.

I would also like to thank Daniella Borkum for her patience and support during the final stages of the thesis.

Synopsis

The report details the design of a dedicated Viterbi decoder chip set for an Ungerboek (3,2/3) Trellis Coded Modulation code. It was the specific intention of the thesis to design a system that could be implemented on standard Field Programmable Gate Arrays (FPGA) yet still be able to cope with high bit rates. The focus of the research was to both evaluate and modify the existing VLSI design techniques and to develop new techniques to make this possible.

Trellis Coded Modulation refers to a specific sub-class of convolutional codes that are an example of coded modulation. In coded modulation there is a direct link between the encoding and modulation processes aimed at improving the performance of the code by introducing redundancy in the signal set used to transmit the code. Ungerboek developed a technique for mapping the encoded words onto points in the signal set, called mapping by set partitioning, that maximises the Euclidian distance between adjacent codewords, and hence maximises the minimum distance between any two output sequences in the code.

The Viterbi algorithm is a maximum likelihood decoder for convolutional codes such as TCM. The operation of the Viterbi algorithm is based on using soft decision decoding to produce an estimate of how well the received sequence corresponds with any of the allowed code sequences. The code sequences which most closely matches the received sequence is then decoded to form the output of the decoder.

A central problem in implementing systems using TCM with Viterbi decoding is that although the encoder is a relatively simple device, the decoder is not. The complexity of the Viterbi decoder for any given TCM scheme will be the major drawback in implementing the scheme. As such techniques for reducing the complexity of Viterbi decoders are of interest to developers of communication systems.

The algorithms describing the implementation and operation of the Viterbi algorithm can be categorised into three main layers. The top layer holds the theoretical algorithm itself, in the second layer are the set of algorithms that describe the broad techniques used to manipulate the theoretical algorithm into a form in which it can be implemented, and the third layer of algorithms describe the implementations themselves.

The work contained in this thesis concentrates on the second two layers of algorithms. The algorithms on each layer are split into two main sections in accordance with the sec-

tion of the decoder they describe – they either describe the operation/implementation of the Add–Compare–Select section of the Viterbi Algorithmor they describe the operation/implementation of the Survivor Memory Unit.

The Add–Compare–Select section of the decoder designed in this thesis uses the modulo $2\Delta_{max}$ technique for the comparison, updating and scaling of path metrics. Under the modulo $2\Delta_{max}$ technique the decoder's dynamic range is a critical factor in determining the complexity of the decoder as it sets the size of the path metrics in bits. A new approach, primarily aimed at Ungerboek TCM codes, is introduced that allows the estimate of the decoder's dynamic range to be substantially reduced. This leads to more efficient implementations of the Viterbi decoder.

The Survivor Memory unit was implemented using the R-pointer even and odd algorithms. The R-pointer algorithms are examples of a broader class of algorithms that make up the traceback technique for survivor memory management. A new implementation technique is introduced that allows for efficient implementations of the R-pointer odd family of algorithms. One of the results of the new implementation technique is that the optimum R-pointer algorithm for any given implementation becomes specific to the code used in the implementation. To facilitate the design and comparison of the various algorithms, a single set of equations is presented that describes the operation of all the R-pointer algorithms.

The two improvements mentioned above all deal with the second layer of algorithms. In order to validate the new techniques mentioned above and to evaluate the performance of various designs on FPGAs, multiple designs were produced to decode a $(3, \frac{2}{3})$ Ungerboek convolutional code using an 8–PSK signal set. The are, for the Add–Compare–Select section of the decoder:

- An implementation using 6 bit path metric representations and parallel path metric comparisons.
- An implementation using 5 bits to represent the path metrics, again with parallel path metric comparisons.
- An implementation using 5 bit path metrics with a series path metric comparison unit.

Two designs were implemented for the Survivor Memory unit:

- A design using an 8-pointer even algorithm with a twin–stack bit order reversing circuit.
- A design using a 3-pointer odd algorithm with a single RAM block bit order reversing circuit.

The decoder was built using two XC3064APC84-7 XILINX FPGA for the Add–Compare–Select circuits, and one XC4005APC84-6 XILINX FPGA for the Survivor Memory Unit.

All branch metrics were generated from PROM lookup–tables. As FPGAs are used in the construction of the decoder, it can easily be modified to implement all the designs mentioned above without requiring any physical changes to be made to the board.

A TCM testbed was also built. It comprises of a TCM encoder, a channel, a decoder, a BER tester, and two AWGN sources to generate the noise to be added to the signals present in the channel. As with the decoders, the circuit was built using FPGAs, so it could be re-configured to implement different designs without requiring any physical changes to be made to the board. As such the testbed was configured to perform both BER tests and to test for error event probability in the presence of AWGN.

All designs were tested in the presence of AWGN, where it was confirmed that they are functionally the same. The testbed and the decoder were then re-configured to perform both BER and error event tests on uncoded 4–PSK so that the decoder's coding gain could be measured.

Although the performance of the final implementation proved to be inferior that of similar schemes presented in the literature, it was found that the performance was in line with such schemes when the channel resolution was reduced to the levels used in this project. The reductions in complexity and the favourable performance of the low–complexity designs mean that implementations using higher channel resolution are possible using the same hardware platform.

Speed tests were also performed on the decoder, where it was found that the implementation using serial metric comparisons to compare the 5 bit path metrics along with the 3-pointer odd implementation of the SMU was capable of handling bit rates up to 16Mb/s.

Contents

List of Figures

Chapter 1

Introduction

With the steady advances being made in the miniaturisation of electronic circuits, design engineers have ever increasing high-speed computational power at their fingertips. Tasks that were previously ruled out due to large hardware complexity and excessive circuit sizes are now possible using miniaturised packages.

Communications is one of the areas in which this new technology has had a significant impact. A new industry centred around portable communications devices has been created due to the drop in device size as well as cost. Two problems arise in portable communications: the first is the added pressure placed on the available bandwidth by multiple users, and the second is that the devices are normally battery powered, which means that the transmitter's output power is severely limited, making the devices susceptible to noise.

Implementations of complex digital (as opposed to analog) communication systems that are generally more resistant to both interference and noise have been enabled by the new miniaturised circuits. The individual characteristics of the system, including the bandwidth used by the modulation scheme and its susceptibility to noise, can be tailored by encoding the digital data stream and choosing from among a variety of modulation schemes. The performance of the system will be proportional to the computational effort expended both in the encoding/modulation and decoding/demodulation of the data streams. With the relative drop in cost of circuits capable of performing complex functions, systems that meet the increased demands of the portable communications market have become feasible.

There are many well known digital communications schemes that improve the receiver's ability to detect the correct signal in the presence of noise by making successive symbols transmitted through the channel dependant on one another. Although the encoders in such schemes tend to be relatively simple devices, the decoders are not. Many of these schemes rely on the use of the Viterbi Algorithm to decode the code at the receiver in order to ensure optimal performance.

One of the simplest ways to introduce a dependence between successive symbols in a channel is to use a convolutional code. In convolutional coding redundancy is added to each signal transmitted through the channel in the form of some information about the previous signals transmitted through the channel. The redundancy is then used by the receiver to correct any errors in the received sequence.

Trellis Coded Modulation, [1], is a modification of convolutional coding that allows for greater flexibility in designing codes that are resistant to noise by forming a non-linear link between the code word and the point in the signal set that represents it. One important subset of Trellis Coded Modulation is Ungerboek codes, which is the set of all Trellis Codes generated using an $\frac{m}{m+1}$ convolutional encoder. Ungerboek codes maximise the ratio of coding gain to increase in circuit complexity due to an expansion in the code's signal set.

Combining an Ungerboek code together with Viterbi decoding provides a means of substantially improving the ability of a communications scheme to withstand noise interference. Unfortunately the improvements in performance come at a price. Although the performance of the system will improve with an increase in the complexity of the code, the improvement tends to be small when compared with the growth in size of the circuitry required to implement the Viterbi decoder in the receiver.

Given that the main obstacle in implementing a system using a convolutional code is the Viterbi decoder, the techniques used to implement the hardware have been of interest to researchers for the last few years. In order to keep the complexity of the hardware to a minimum these techniques must exploit both the properties of the Viterbi Algorithm and the convolutional code which is being decoded.

This thesis aims to explore the techniques used to implement the Viterbi Algorithm in hardware through the construction of a Viterbi decoder for a $(3, \frac{2}{3})$ Ungerboek TCM code. The hardware implementation will make use of off-the-shelf *field programmable gate arrays* (FPGA), so, where possible, the existing techniques for manipulating the Viterbi Algorithm into a form suitable for implementation in hardware will be optimised for use in FPGAs.

When designing a Viterbi decoder, the designer will typically have to move through three layers of algorithms. The first layer would encapsulate the theoretical Viterbi Algorithm, as developed by Viterbi himself, [2]. The second layer encapsulates all the algorithms that trim down the original algorithm and present it in a form suitable for implementation in hardware. The third layer would be hardware specific (the first two are not) and would deal with the specifics of the hardware design.

In keeping with the above process, this thesis is laid out in much the same way. Chapter 2 introduces all the background theory required to both fully understand and manipulate the Viterbi Algorithm for use in decoding a TCM code. As this process is also dependant on the specifics of the code, all relevant background theory for TCM is presented as well. Chapter 3 deals with the algorithms that are used to manipulate the Viterbi algorithm into a form in which it can be implemented, and Chapter 4 gives details on the specific

implementations derived for this project.

Chapter 5 outlines the construction of the test rig used to perform both functional and performance evaluations on the decoder. In Chapter 6 the results obtained from these tests are presented and interpreted. Chapter 7 concludes the thesis.

In Appendix A, a proof is outlined that is required for the evaluation of the decoder's dynamic range. Appendix B and C move on to detailing timing diagrams and resource usage for the Add-Compare-Select and Survivor Memory units respectively.

Throughout the thesis any new work, or new interpretations of work presented in the literature, is typeset in an italics font for the purpose of clarity. If the new work spans an entire section, only the section heading is in an italics font – the section body is typeset in the normal font. In such cases the text will make it clear that the work being presented is that of the author.

Chapter 2

Background

In order to obtain an efficient implementation of a Viterbi Algorithm, the operation, properties and structure of the Viterbi Algorithm need to be fully exploited. To a large extent both the properties and the structure of the Viterbi Algorithm are dependant on the code being decoded. As such a thorough understanding of the theory behind both TCM and the Viterbi Algorithm will be required.

2.1 Coded Modulation

There is a theoretical limit to the rate at which data can be transmitted through a bandlimited channel. The maximum number of bits that can be transmitted through a channel per second without error is termed the channel capacity. Shannon's well known theorem states that the theoretical limit for channel capacity in the presence of noise is given by $C = \log_2(1 + \frac{S}{N})$ where C is in bits/T, $\frac{S}{N}$ is the *signal to noise ratio* (SNR), and T denotes the modulation interval, given in seconds.

In digital communication systems the input to the channel will be a set of discrete signals. The output, however, in the presence of *additive white Gaussian noise* (AWGN), will be continuous. Given these criteria, the channel capacity can be derived for a set of well known modulation schemes by assuming that all channel signals are equiprobable [1]. The results are best interpreted graphically, as in Figure 2.1.

As an example, transmission at 2 bits/T is considered. Assuming there is no redundancy in the channel input, one choice of signalling scheme would be 4–PSK. Under this scheme, transmission at a probability of error of 10^{-5} is only possible at a SNR of 12.9 dB. A doubling in the size of the signal set would result in the use of 8–PSK, where – assuming unlimited coding and decoding effort – error free transmission is possible at a SNR of 5.9 dB. Doubling the size of the set yet again will yield an improvement in performance over that of 8–PSK, although the gain will be much smaller. Above 8–PSK only another 1.2 dB can be gained through expansion of the signal set before reaching the theoretical

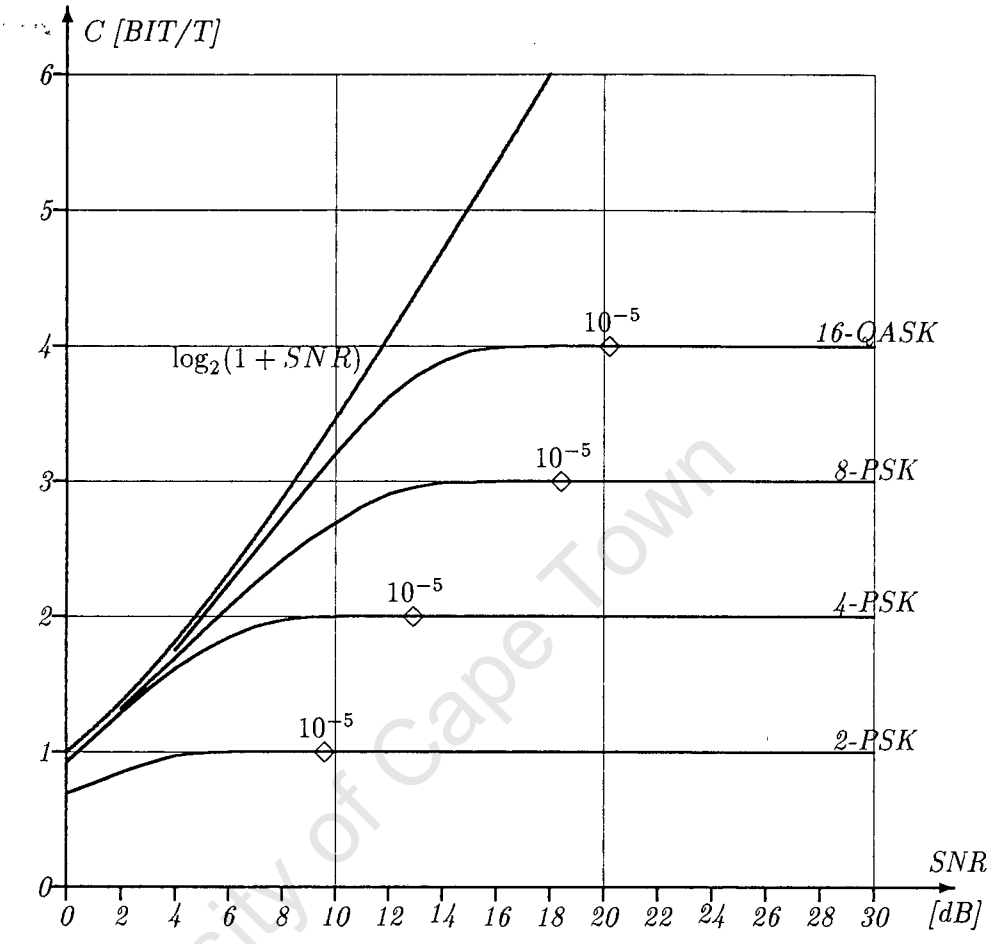


Figure 2.1: Channel capacity, C, of bandlimited AWGN channels for discrete valued input and continuous valued output using two dimensional modulation.

limit for the channel capacity.

Since the transmission rate remains unchanged, adding redundancy to the signal set ensures that the bandwidth remains unchanged. The example given above shows that a significant improvement in performance is possible through doubling the size of the signal set. Further signal set expansion is of little interest since the gains in performance would be small and would require a large increase in the complexity of both the coding and the modulation schemes.

Furthermore, a link is implied between the coding used and the choice of modulation scheme. Encoding and modulation would have to be performed together in the transmitter in order to fully exploit the gains that can be made through signal set expansion. Schemes that fuse the encoding and modulation are known collectively as Coded Modulation.

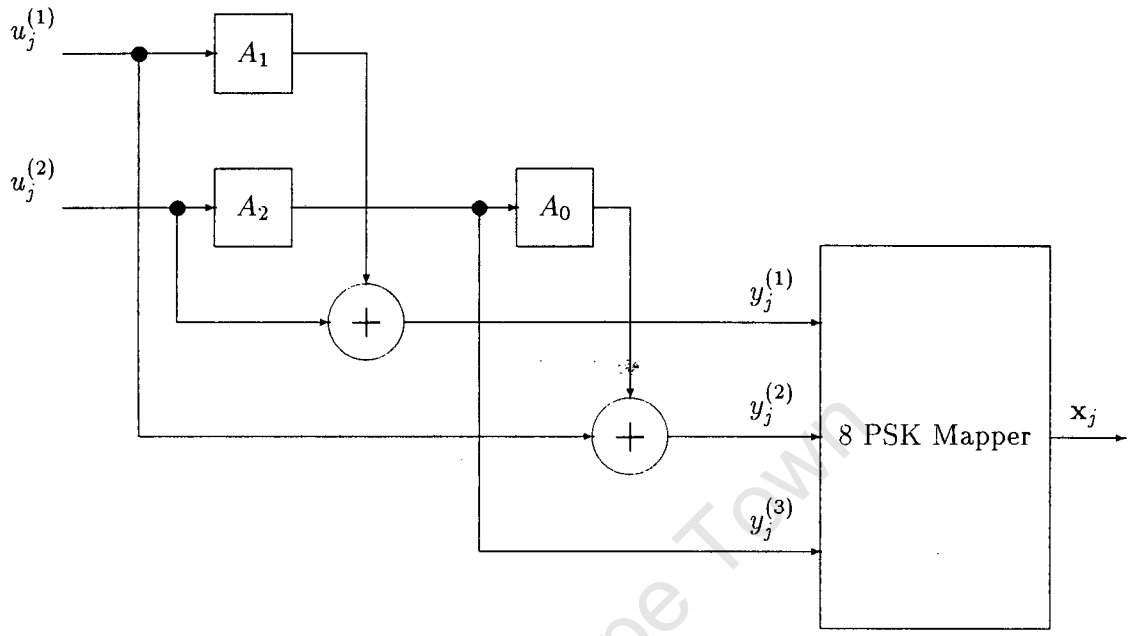


Figure 2.2: A $(3, \frac{2}{3})$ Trellis Coded Modulation Encoder

2.2 Trellis Coded Modulation

The ideas behind coded modulation were first presented by Ungerboek [1]. In order to take advantage of the doubling in the size of the signal set, he proposed the use of a rate $\frac{m}{m+1}$ convolutional encoder. Unlike a standard convolutional encoding system, where the codewords are fed serially to the modulator, he proposed that the codewords be mapped directly onto points in the signal set through the use of some non-linear mapping function. This coded modulation scheme has been expanded into a much broader class of schemes known collectively as *Trellis Coded Modulation* (TCM). The subset of $\frac{m}{m+1}$ codes first presented by Ungerboek are known as Ungerboek codes [3].

By way of example a TCM encoder for a $(3, \frac{2}{3})$ Ungerboek code is shown in Figure 2.2 on page 6 with the corresponding trellis diagram in Figure 2.3 on page 7. The trellis diagram shows all the possible state transitions associated with an input, $\mathbf{u}_j = [u_j^{(1)}, u_j^{(2)}]$, together with the encoder outputs, $\mathbf{y}_j = [y_j^{(1)}, y_j^{(2)}, y_j^{(3)}]$, from any arbitrary time unit j to time unit $j+1$. In the diagram each transition is labelled with $\mathbf{u}_j/\mathbf{y}_j$. Since there are three memory elements in the encoder, the code is said to have a constraint length, ν, of three. The number of states in the code is then $S = 2^\nu$, which would be eight in the example shown. The state at any time j is represented by $\mathbf{s}_j = [A_2, A_1, A_0]$ as shown in Figure 2.2.

In algebraic terms the operation of the TCM encoder can be described by noting that it

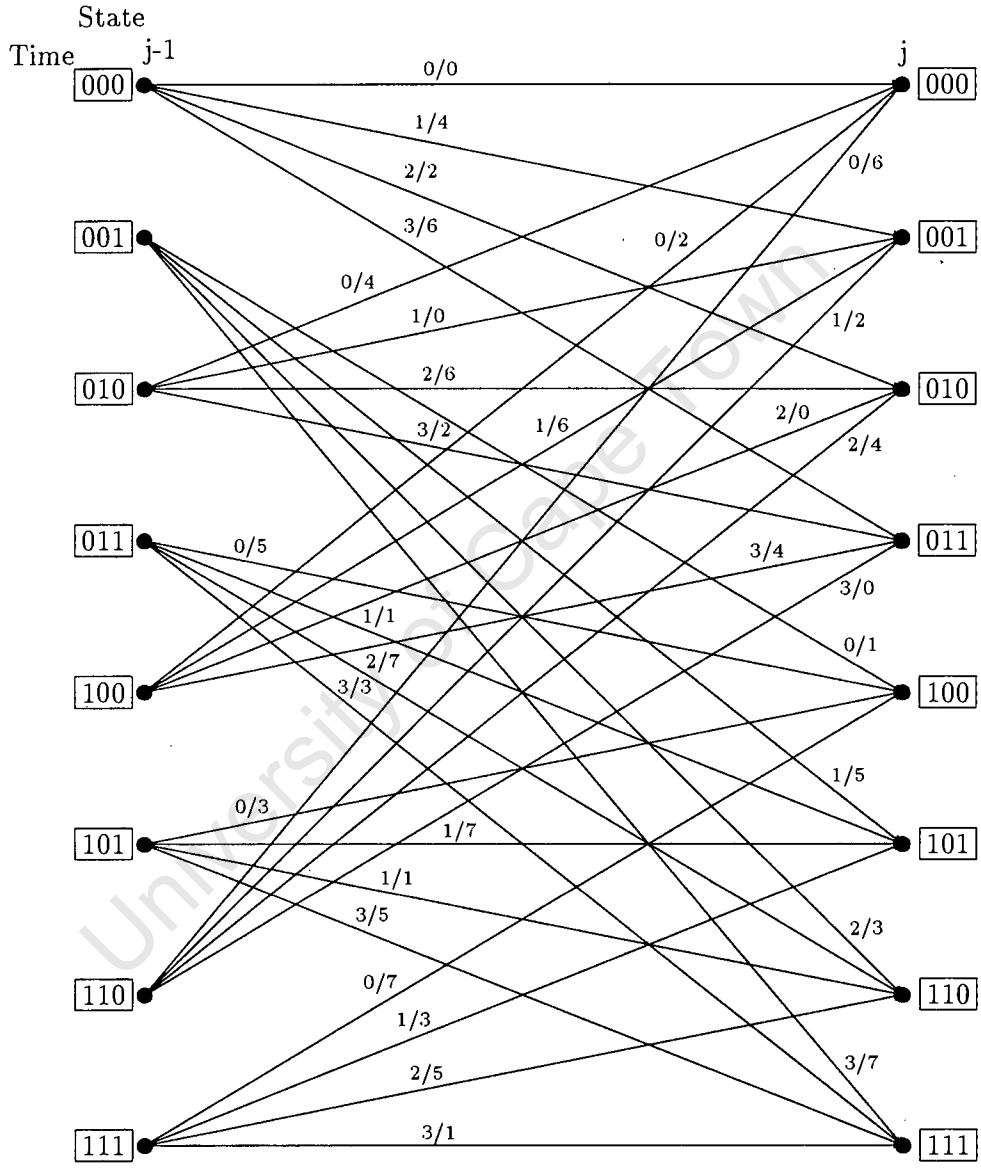


Figure 2.3: Trellis diagram of the coder shown in Figure 2.2

works in much the same way as a convolutional encoder [3]. The input to the encoder would be a binary stream:

$$\mathbf{u}_j = [u_j^{(1)}, \cdots, u_j^{(m)}] \tag{2.1}$$

The output codewords are formed by combining the present input with a certain number of previous inputs:

$$y_j^{(1)} = u_{j-1}^{(1)} \oplus u_j^{(2)} \tag{2.2}$$

$$y_j^{(2)} = u_{j-2}^{(2)} \oplus u_j^{(1)} \tag{2.3}$$

$$y_j^{(3)} = u_{j-1}^{(2)} \tag{2.4}$$

By defining D to be the unit delay operator, matrix algebra can be used to represent the code:

$$\mathbf{y}_j = \mathbf{u}_j \mathbf{G}(D) \tag{2.5}$$

In the case illustrated by equations (2.2) to (2.4), $\mathbf{u}_j = [u_j^{(1)}, u_j^{(2)}]$ and $\mathbf{y}_j = [y_j^{(1)}, y_j^{(2)}, y_j^{(3)}]$. Then:

$$\mathbf{G}(D) = \begin{bmatrix} D & 1 & 0 \\ 1 & D^2 & D \end{bmatrix} \tag{2.6}$$

The output from the mapper is some function $f(\mathbf{y}_j)$, where $\mathbf{y}_j$ is the output from the encoder. It is shown in [3] that the function $f(\cdot)$ must be nonlinear in order for the code to achieve a coding gain over the uncoded case.

2.2.1 Improving Error Performance by Coding and Signal Mapping

A traditional receiver using hard decision decoding would decode each symbol as being the closest point, in terms of Euclidian distance, in the signal set with relation to the received point at the sampling instant. When hard decision decoding is combined with block coding, the received bits would be compared with all the allowed codewords. The codeword which had the shortest Hamming distance between it and the received sequence would then be chosen as the correct output. In this manner the receiver would correct for erroneous decisions made by the hard decision decoder.

Although TCM does have error correcting abilities in line with block codes [4], the system's performance will improve if soft decision decoding is used instead. Under soft decision decoding the receiver will compare the received sequence (rather than the received symbol) with all allowed sequences in the code. This comparison is performed before any decision is made as to which code symbol corresponds to the received signal. The code sequence which is the closest in Euclidian distance will then be chosen as the correct sequence. Although the receiver makes no firm decision, and so cannot be said to have made an error, it does correct offsets in the received sequence from the valid code sequences, and so has "error correcting" abilities.

The performance of a TCM code will improve if the smallest Euclidian distance (d_{free}^2)

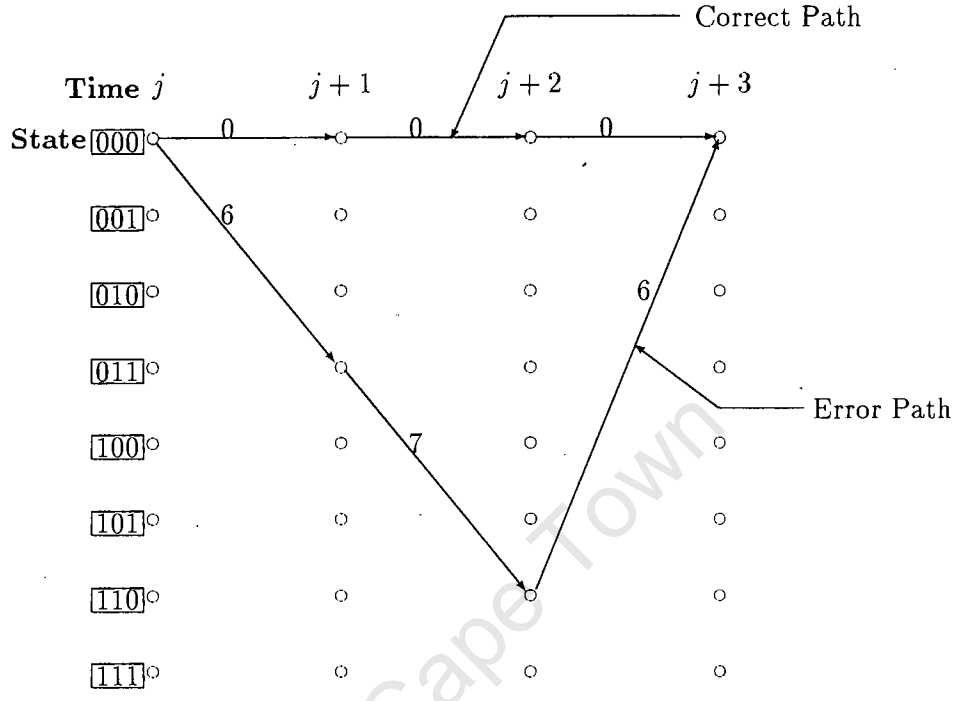


Figure 2.4: Illustration of path propagation through the code trellis.

between any two paths that start and end in the same state is made as large as possible, in the same way that the performance of a block code is improved by increasing the shortest Hamming distance between any two codewords. d^2_{free} is known as the code's free distance. If the output of the encoder, $f(\mathbf{y}_j)$, is traced through time, it will trace out a path through the trellis as depicted in Figure 2.4. Also depicted in Figure 2.4 is an error path L time units long described by $f(\mathbf{y}_j \oplus \mathbf{e}_j)$, where $\mathbf{e}_j$ is an error sequence. The free distance of the code is then described as [5]:

$$d^2_{free} = \min \sum_{j=0}^{L-1} d^2 \left[f(\mathbf{y}_j), f(\mathbf{y}_j \oplus \mathbf{e}_j) \right] \tag{2.7}$$

Where the minimum is taken over:

1. L, the length of the error path,
2. $\mathbf{y}_0, \mathbf{y}_1, \cdots, \mathbf{y}_{L-1}$, the correct vector sequence,
3. $\mathbf{e}_0, \mathbf{e}_1, \cdots, \mathbf{e}_{L-1}$, the error vector sequence.

It must also be noted that, because TCM codes are linear, the error sequence $\mathbf{y}_j \oplus \mathbf{e}_j$

must be a valid code sequence and must begin and end in the same state as the correct sequence, $\mathbf{y}_j$.

TCM achieves an improvement in performance because it introduces a greater separation between adjacent signals ($d^2_{free} > d^2_{min}$). This is in contrast to an uncoded scheme, who's error performance is determined by the minimum Euclidian distance (d^2_{min}) between any two points in the signal set.

The improvement in performance of a TCM scheme over the equivalent uncoded scheme is typically measured in terms of coding gain. Coding gain would refer to the reduction in signal to noise ratio, in dB, of the TCM scheme over the uncoded scheme for the same probability of an error occurring in the received sequence. Since there is an increase in size in the signal set used by the TCM scheme when compared with the uncoded scheme, there may also be an increase in the average energy transmitted per signal. Denoting the average energy per signal as E for the uncoded scheme, and E' for the TCM scheme, the asymptotic coding gain of the TCM scheme is defined as [3]:

$$\gamma = \frac{d^2_{free}/E'}{d^2_{min}/E} \tag{2.8}$$

The encoder shown in Figure 2.2, when used with set partitioning and 8–PSK, will have an asymptotic coding gain of 3.6dB [1].

2.2.2 Set Partitioning

The free distance of a TCM scheme is determined by the way in which the outputs from the encoder are mapped onto the signal set as well as the code itself. A closer examination of Figure 2.4 will reveal that all error paths of length L share the same two properties:

1. At time unit j they diverge from the correct path.

2. At time unit $j + L$ they re-merge with the correct path.

In order to maximise the Euclidian distance between correct and error paths it is therefore sufficient to map the transitions leaving and entering each state (adjacent transitions) in the code trellis to points in the signal set in such a way that the Euclidian distance between them is maximised.

A closer examination of equations (2.2) to (2.4) will reveal that $y_j^{(3)}$ is independent of the inputs to the encoder at time unit j, but is dependant on the current state of the encoder. If the signal set is split into two according to the value of $y_j^{(3)}$, then all the adjacent transitions will be assigned to points in one of the subsets. In order to maximise the Euclidian distance between adjacent transitions it is therefore sufficient to divide the signal set in such a way that the Euclidian distances between the points in the subsets

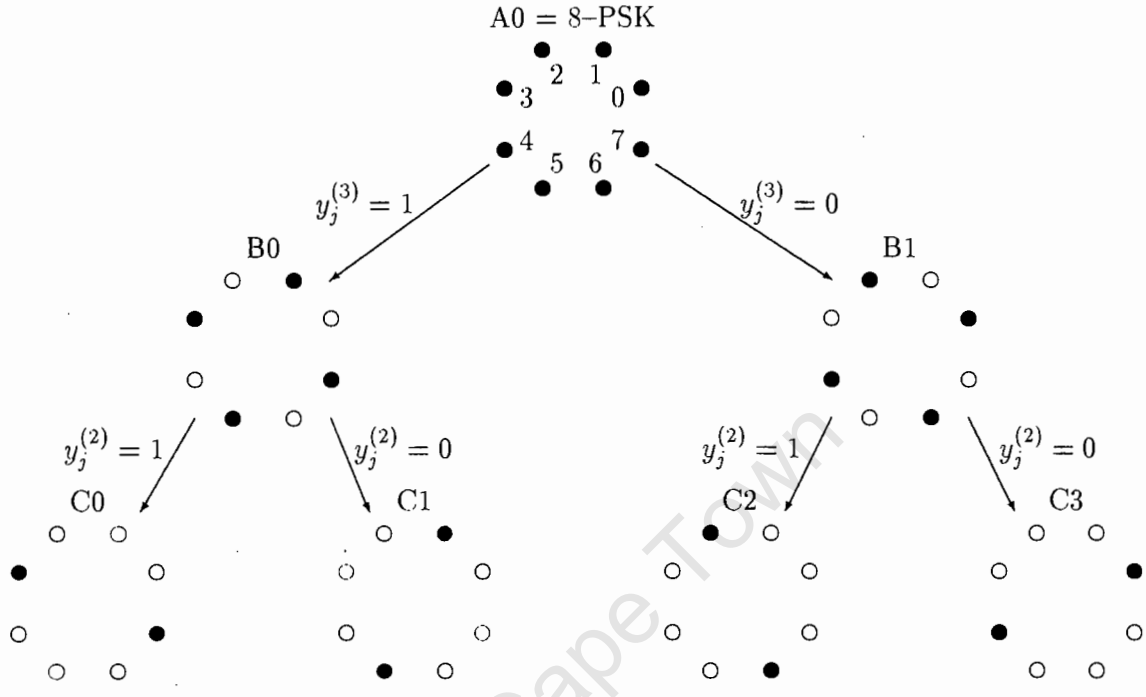


Figure 2.5: Set partitioning for TCM using an 8–PSK signal set

are maximised. This process is known as set partitioning and is illustrated in Figure 2.5 for the encoder shown in Figure 2.2. It was first introduced by Ungerboek [1].

As illustrated in Figure 2.5 each binary symbol in the codeword, $\mathbf{y}_j$, successively selects one of the signal subsets ($y_j^{(1)}$ would select one signal from the four signal subsets labelled C0 to C3). This ensures that adjacent transitions are maximally spaced: 000 is maximally spaced from 100, 010 is maximally spaced from 110, and so on (in this case codewords that differ in their first bit correspond to adjacent transitions).

The technique ensures that transitions leaving each state in the code trellis are maximally spaced by assigning them to either subset B1 or subset B0. In order to maximise the free distance of the code using set partitioning, the transitions entering each state must also be maximally spaced in Euclidian distance. If set partitioning is to ensure that these transitions are also maximally spaced in Euclidian distance, then the codewords assigned to all transitions entering any single state must form a subset corresponding to either subset B1 or subset B0.

This is the case for the encoder shown in Figure 2.2, and is proved here using a new proof developed by the author. The state, $\mathbf{s}_j$, at any time unit j is given by the contents

of the shift registers in the encoder:

$$\mathbf{s}_j = \left[u_{j-1}^{(2)}, u_{j-1}^{(1)}, u_{j-2}^{(2)}\right] \tag{2.9}$$

It is clear from equation (2.9) that all paths entering any single state must be due to the same input to the encoder, $\mathbf{u}_{j-1} = \left[u_{j-1}^{(1)}, u_{j-1}^{(2)}\right]$. *Since each of these paths originates from a separate state, it must be proved that* 2^b *previous states can be defined such that the output of the encoder will form a subset corresponding to a subset after set partitioning. In this case* $b = 2$, *so there are four previous states defined by:*

$$\mathbf{s}_{j-1} = \left[u_{j-2}^{(2)}, u_{j-2}^{(1)}, u_{j-3}^{(2)}\right] \tag{2.10}$$

which each have a single transition ending in the same state, $\mathbf{s}_j$.

Since, from equations (2.9) and (2.10), $u_{j-2}^{(2)}$ *is common to* $\mathbf{s}_{j-1}$ *and* $\mathbf{s}_j$, *and the input to the encoder is the same, the only variables are* $u_{j-2}^{(1)}$ *and* $u_{j-3}^{(2)}$. *By re-working the matrix* $\mathbf{G}(D)$ *to operate on a new input* $\mathbf{p}_{j-1} = \left[u_{j-2}^{(1)}, u_{j-3}^{(2)}\right]$, *and by defining* D^{-1} *to be the inverse of the unit delay operator, the codewords,* $\mathbf{y}_{j-1}$, *assigned to the transitions entering each state are defined by:*

$$\mathbf{y}_{j-1} = \mathbf{p}_{j-1}\mathbf{G}'(D) \tag{2.11}$$

where:

$$\mathbf{G}'(D) = \begin{bmatrix} 1 & D^{-1} & 0 \\ D^{-2} & 1 & D^{-1} \end{bmatrix} \tag{2.12}$$

By inspection of the matrices $\mathbf{G}(D)$ *and* $\mathbf{G}'(D)$, *defined in equations (2.6) and (2.12) respectively, it can be seen that the first two columns in both matrices, and only the first two columns, have entries with a* 1 *in them. Furthermore, the entries containing a* 1 *occur in different rows.*

So in both cases only the first two terms of the resultant codewords will be dependant on either $\mathbf{p}_{j-1}$ *or* $\mathbf{u}_j$ *respectively, and they will be independent of one another. All other terms will be constant over the transition in question. Therefore, since only modulo two addition is used, it is possible to define* 2^b *previous states such that they all have transitions that end in the same state,* $\mathbf{s}_j$. *The codewords associated with these transitions come from one of the signal subsets assigned to the transitions leaving any state within the code trellis (in this particular case, either subset B1 or subset B0 in Figure 2.5)* [1].

Since set partitioning maximises the Euclidian distance between adjacent transitions, it will also maximise the code's free distance. The coding gain will, as a result, be maximised relative to the expansion of the signal set.

[1] An alternative proof, based on Euclidian weights, may be found in [1].

2.2.3 Uniform Codes

Determining the code's free distance must, therefore, involve a thorough search of all possible paths and their associated error paths through the code trellis. In codes of low constraint length this will pose no problem. However, the computational complexity involved in determining the free distance increases with S^2. In order to find optimal TCM schemes, a search must be performed through a wide subclass of codes. Although such a search is of no interest to this particular project, techniques were developed to reduce the computational complexity of the search by exploiting uniformities in the code. These same properties can be exploited to reduce the physical complexity of the decoder.

A TCM scheme is said to be uniform if it satisfies the following criteria [5]:

1. The scheme is v-isometric;
2. The encoded m-tuples

$$\mathbf{k}_j = (y_j^{(1)}, \cdots, y_j^{(v-1)}, y_j^{(v+1)}, \cdots, y_j^{(m+1)})$$

   form an independent sequence;
3. The sequence of $m+1$-tuples $\mathbf{y}_j$ is uniquely determined by the m-tuples $\mathbf{k}_j$.

An isometry refers to a one-to-one distance preserving transformation. In a code that is v-isometric, the subconstellations selected by the vth component of the codeword are related by an isometry. The labelling of the points in the signal subsets are also of importance: the labels of any corresponding points in the two subsets must only differ in their vth component [5].

For the encoder in Figure 2.2, used with 8–PSK, set partitioning guarantees that the signal subsets B0 and B1 are related by an isometry. If the points in the signal set are labelled as shown in Figure 2.5, then the code will be 3-isometric, since the 3rd component of the codeword chooses between subsets B1 and B0 and their corresponding points are appropriately labelled.

The coding scheme also satisfies conditions (2) and (3) since the equations (2.2) to (2.4) guarantee that $y_j^{(1)}$ and $y_j^{(2)}$ form an independent sequence as long as $u_j^{(1)}$ and $u_j^{(2)}$ are independent (they normally are). Furthermore, $y_j^{(3)}$ is uniquely determined by $y_j^{(1)}$ and $y_j^{(2)}$. [2]

When the computational complexity of the calculations required to determine the code's free distance can be reduced, the code is said to have the *Uniform Distance Property* (UDP). If the same reductions in computational complexity can be achieved for the calculation of the upper bound on the error probability for the code, the code is said

[2] It is shown in [5] that all Ungerboek codes are uniform.

to have the *Uniform Error Property* (UEP). A code that is uniform will have both the UDP and the UEP [5].

It is the UDP that is of interest to this project, since techniques similar to the derivation of the free distance can be used to derive the maximum dynamic range of the decoder (see section 3.1.2)

In order for the UDP to hold it must be shown that equality holds in [5]:

$$\min \sum_{j=0}^{L-1} d^2 \left[f(\mathbf{y}_j), f(\mathbf{y}_j \oplus \mathbf{e}_j) \right] \geq$$
$$\min_{e_0, \cdots, e_{L-1}} \sum_{j=0}^{L-1} \min_{\mathbf{y}_j} d^2 \left[f(\mathbf{y}_j), f(\mathbf{y}_j \oplus \mathbf{e}_j) \right] \tag{2.13}$$

The inequality is due to the dependence between $\mathbf{y}_i$'s introduced by the memory in the decoder.

In the case of uniform codes, however, the m-tuples $\mathbf{k}_j$ form an independent sequence (see condition (2) for uniformity on page 13). Therefore it is possible to do all minimisation with respect to this sequence. Although $y_j^{(v)}$ is determined by this sequence, the dependence has no effect on the computation since the scheme is v-isometric.

2.3 The Viterbi Algorithm

The Viterbi Algorithm was first presented by Viterbi [2] in 1967 as an alternative method for decoding convolutional codes. Since the algorithm is optimal in the sense that it performs maximum likelihood estimation of the received sequence, and it lends itself to implementation in hardware, it has become a popular choice in the decoding of a wide range of codes.

2.3.1 Decoding Under TCM

The output of the TCM encoder/modulator can be viewed as a discrete time, finite state Markov process [6]. This sequence will be passed through a memoryless channel in which AWGN will be added to the signal before being observed at the receiver. The receiver must then perform a maximum likelihood *a posteriori* probability estimation of the received sequence.

If the encoder has run from time unit 0 to time unit J, then the state sequence is represented by $\mathbf{s} = (\mathbf{s}_0, \mathbf{s}_1, \cdots, \mathbf{s}_J)$. The process is Markovian in the sense that $P(\mathbf{s}_{j+1}|\mathbf{s}_0, \mathbf{s}_1, \cdots, \mathbf{s}_j)$, the probability of being in state $\mathbf{s}_{j+1}$ given all the previous states up to time j, depends only on the state at time j, $\mathbf{s}_j$ [6]:

$$P(\mathbf{s}_{j+1}|\mathbf{s}_0, \mathbf{s}_1, \cdots, \mathbf{s}_j) = P(\mathbf{s}_{j+1}|\mathbf{s}_j)$$

Furthermore, there is a one-to-one correspondence between the states $\mathbf{s}_j$ and $\mathbf{s}_{j+1}$ and the signal associated with the transition between them, $\mathbf{x}_j = (\mathbf{s}_j, \mathbf{s}_{j+1})$. As shown in section 2.2, there is also a one-to-one correspondence between the signal $\mathbf{x}_j$ and the input to the encoder, $\mathbf{u}_j$.

In TCM schemes the receiver will observe a sequence $\mathbf{z}$, the output of a memoryless AWGN channel where $\mathbf{x}$ is the input to the channel. Since the channel is memoryless, the received symbols, $\mathbf{z}_j$, depend probabilistically only on the transmitted signals, $\mathbf{x}_j$, at time unit j [6]:

$$p(\mathbf{z}|\mathbf{x}) = \prod_{j=0}^{J-1} p(\mathbf{z}_j|\mathbf{x}_j) \tag{2.14}$$

Probability densities, $p(\mathbf{z}_j|\mathbf{x}_j)$, have been used here in place of probability distributions, $P(\mathbf{z}_j|\mathbf{x}_j)$ since although the input to the channel is discrete, the output of the channel will be continuous in the presence of AWGN.

The receiver must find the sequence $\mathbf{x}'$ such that the *a posteriori* probability $p(\mathbf{x}'|\mathbf{z})$ is maximised. Since there is a one-to-one correspondence between the input sequence, $\mathbf{u}$ and the sequence of transmitted signals, $\mathbf{x}$, this is equivalent to minimising the probability of bit error in the received sequence.

2.3.2 The Viterbi Algorithm as an Optimal Decoder for TCM

In order to arrive at the Viterbi Algorithm, it must first be shown that maximizing the *a posteriori* probability is equivalent to determining the shortest path through the code trellis. If a length $\lambda(\cdot)$ is assigned to each transition in the code trellis, then the total length of any path will be given by the sum of the lengths of the individual transitions. So to prove that the shortest path is in fact the optimal solution it must be shown that the sum of the path lengths can be equated to $p(\mathbf{x}'|\mathbf{z})$. This can be done as follows [6]:

$$p(\mathbf{x}'|\mathbf{z}) = \prod_{j=0}^{J-1} p(\mathbf{x}'_j|\mathbf{z}_j)$$

Therefore

$$\ln(p(\mathbf{x}'|\mathbf{z})) = \sum_{j=0}^{J-1} \ln(p(\mathbf{x}'_j|\mathbf{z}_j)) \tag{2.15}$$

As the natural logarithm is a monotonic function, and each path in the code is unique, it is sufficient to maximise the right hand side of equation (2.15). If each transition in the code's trellis is assigned a 'length' inversely proportional to $\ln(p(\mathbf{x}'_j|\mathbf{z}_j))$ then a decoder that finds the path through the code trellis with the minimum 'length' will be a maximum likelihood decoder.

The above approach requires an exhaustive search of all the paths through the code trellis. In theory the search for the correct path should only begin once the entire sequence has

been received. One of the problems associated with this approach is that the number of possible paths through the trellis will grow exponentially with an increase in the length of the received sequence. To reduce the complexity of the decoding algorithm Viterbi noted the following [7]:

Consider any two signal sequences, $\mathbf{x}^1$ and $\mathbf{x}^2$, which trace a path through the code trellis. Both paths end in the same state $\mathbf{s}_j$. Each path has an associated 'length', $\Lambda(\mathbf{x}^1)$ and $\Lambda(\mathbf{x}^2)$ respectively. If $\mathbf{x}^1$ is the most likely path given the received sequence $\mathbf{z}$, then, by definition, $\Lambda(\mathbf{x}^1) < \Lambda(\mathbf{x}^2)$.

The decoder will now advance from time unit j to time unit $j+K$. It is known that the most likely state sequence, $\hat{\mathbf{x}}$, traces a path through the code trellis that ends in state $\mathbf{s}_{j+K}$ and passes through state $\mathbf{s}_j$. Furthermore $\hat{\mathbf{x}}$ has as its initial state sequence up to state $\mathbf{s}_j$ either $\mathbf{x}^1$ or $\mathbf{x}^2$. Since the 'length' of this path is defined as:

$$\Lambda(\hat{\mathbf{x}}) = \Lambda(\mathbf{x}^n) + \sum_{k=1}^{K} \lambda(x_{j+k}) \qquad \text{where } n \in (1,2)$$

then the most likely path must have as its initial state sequence $\mathbf{x}^1$, since this will minimise $\Lambda(\hat{\mathbf{x}})$.

In other words the decoder need only consider the paths with the minimum 'length' entering each state. All other paths can be 'pruned' since they cannot form part of the most likely path. All paths that are not 'pruned' are known collectively as the *survivor paths*. The decoder will therefore need to keep track of 2^{ν} paths, each path ending in a different state at the end of the code trellis.

The algorithm that implements the above processes for determining the most likely state sequence in the code trellis is known as the Viterbi Algorithm. In terms of the decoder shown in Figure 2.2 on page 6 the algorithm would perform the tasks listed in the two sections below. The tasks performed by the algorithm have been split into two main sections, the first being the *Add–Compare–Select* (ACS) unit, and the second being the *Survivor Memory Unit* (SMU). It is in this form that the algorithm lends itself to implementation in hardware.

2.3.3 The Add-Compare-Select Unit

This is the section of the algorithm that performs all the mathematical operations in the decoder. It would perform the following tasks to decode a code generated by the coder presented in Figure 2.2:

Step 1 A *state metric*, $\Lambda(\mathbf{s}_j)$, is assigned to all the states, $\mathbf{s}_j$ in the code trellis. The state metric is the 'length' of the survivor path ending in that state. Since all paths through the trellis must begin at state 00 at time unit $j = 0$, the state metric assigned to state 00 at time unit $j = 0$ is reset to its lowest possible value.

All other state metrics at time unit $j = 0$ are reset to their highest possible value.

Step 2 A noisy symbol, $\mathbf{z}_j$, is received. Each state transition, $\xi_{(\mathbf{s}_{j-1},\mathbf{s}_j)}$, between state $\mathbf{s}_{j-1}$ and state $\mathbf{s}_j$, where $\mathbf{s}$ is the state number and j is the present unit in time, is associated with a channel symbol, $\mathbf{x}'_j$. A *branch metric* is now assigned to all transitions $\xi_{(\mathbf{s}_{j-1},\mathbf{s}_j)}$ defined by:

$$\lambda\left(\xi_{(\mathbf{s}_{j-1},\mathbf{s}_j)}\right) = \|\mathbf{z}_j - \mathbf{x}'_j\|^2$$

where $\|\mathbf{z}_j - \mathbf{x}'_j\|^2$ denotes the Euclidian distance between the received signal and the code signal associated with the state transition in question. Since the channel is memoryless with AWGN, the probability that any given signal $\mathbf{x}'_j$ was transmitted given that $\mathbf{z}_j$ was received, $p(\mathbf{x}'_j|\mathbf{z}_j)$, is inversely proportional to $\lambda\left(\xi_{(\mathbf{s}_{j-1},\mathbf{s}_j)}\right)$.

Step 3 (Add) The branch metric, $\lambda\left(\xi_{(\mathbf{s}_{j-1},\mathbf{s}_j)}\right)$, is added to the state metric, $\Lambda\left(\mathbf{s}_{j-1}\right)$, which corresponds to the state $\mathbf{s}_{j-1}$ associated with the transition $\xi_{(\mathbf{s}_{j-1},\mathbf{s}_j)}$. The new metric formed in this way is known as the *path metric*.

Step 4 (Compare) Since there are four transitions $\xi_{(\mathbf{s}_{j-1},\mathbf{s}_j)}$ entering each state $\mathbf{s}_j$, there will be four path metrics associated with each state $\mathbf{s}_j$. These four path metrics are now compared to determine the minimum path metric.

Step 5 (Select) The minimum of the four path metrics is chosen as the survivor. The survivor becomes the new state metric $\Lambda\left(\mathbf{s}_{j-1}\right)$. A pointer to the previous state $\mathbf{s}_{j-1}$ along the survivor path, $\mathbf{d}_{(\mathbf{s}_{j-1},\mathbf{s}_j)}$ is passed on to the survivor memory unit.

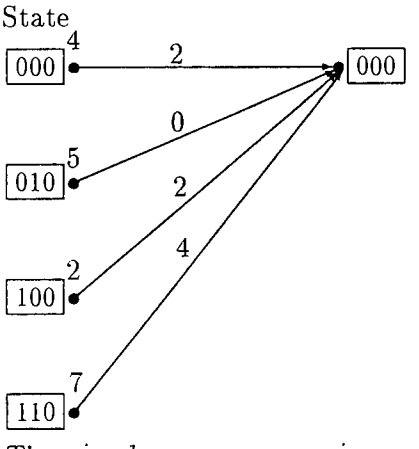
Step 6 Steps 2–5 are repeated until the entire code sequence has been received.

An illustration of the above process for state 0_j is shown in Figure 2.6. The procedure is repeated for all the other states in the code trellis at time unit j.

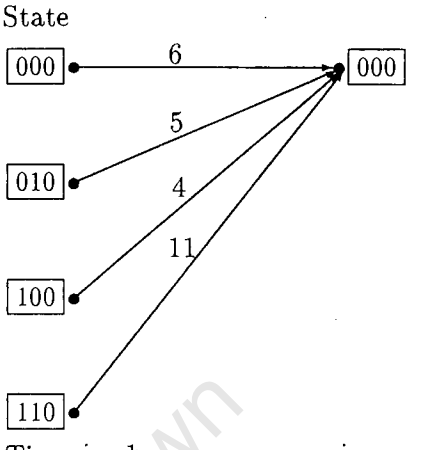
2.3.4 The Survivor Memory Unit

The SMU is responsible for storing and decoding the decisions made by the ACS unit. Until the entire sequence has been received, the SMU simply stores all the survivors. When the entire sequence has been received, the SMU needs to decode the survivors to recover the transmitted data.

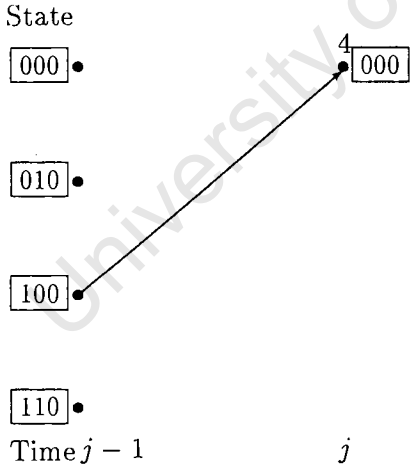
There are two main methods of storing and retrieving the survivors, namely the register exchange method and the traceback method. The register exchange method is generally not used in hardware implementations since it is area inefficient in VLSI implementations [8], [4] and [9]. The traceback method is far more area efficient, but it does result in



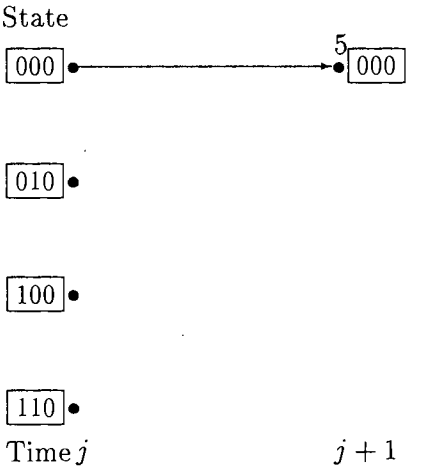
Step 2: Branch metrics are assigned to each transition. They are integers that are proportinal to the Euclidian distance between the received point and the point in the signal set assigned to that transition.



Step 3: The path metrics at time $j-1$ are added to the branch metrics to form the partial path metric.



Step 4 and 5: The partial path metrics are compared and the transition with the minimum metric is chosen as the survivor. The suirvivor's partial path metric becomes the path metric of the state at time j.



Step 6: A pointer to the state at $j-1$ corresponding to the survivor is passed on to the survivor memory unit. Steps 2 to 6 are repeated until the entire sequence has been received.

Figure 2.6: ACS Example

greater latency in the decoder. Since area efficiency was of more concern than latency, the traceback technique was used in the implementation discussed in this document.

In the traceback method the survivor pointers are stored in a memory block that is a copy of the code trellis. Each row in the memory corresponds to (and is hence addressed by) an encoder state, and each column in the memory corresponds to (and is addressed by) a unit in time. Each memory location would then store a pointer to the state at the previous unit in time along the path formed by the survivor entering that state. In order to find the path to be decoded, the survivor memory unit simply has to follow the pointers back from the state at end of the trellis (and hence the memory block) that has the minimum path metric.

Since it is the SMU that will be processing the pointers to the survivors, it is also the SMU that determines the format of the pointers that it will receive from the ACS unit. The pointer format is derived from the operation of the encoder. The state of the encoder is determined by the contents of the shift registers in the encoder, which is simply the past input to the encoder. In order to introduce a dependence between successive symbols, the shift register length will always be greater than the number of inputs to the encoder at any given unit in time. This means that each state in the code trellis contains some information on the previous state along the paths entering that state.

In order to trace back from one state in the trellis to the previous state in the trellis we simply need to store the information that was lost at the encoder when it moved to a new state. In terms of the encoder introduced above, the shift register is of length three and there are two input bits every time unit. That means that we need to store two bits to be able to trace back from one state to the previous state along a given path. In terms of the trellis diagram shown in Figure 2.3 the most significant bit of each state at time unit j is the least significant bit of all the states with transitions leading to that state at time unit $j-1$. In order to move back from any given state at time unit j along a fixed path to a state at time unit $j-1$ the two most significant bits of the state at time unit $j-1$ would have to be stored in the memory segment pointed to by the state at time unit j.

The traceback operation now becomes a relatively simple process:

- A shift register holds a pointer to the current state, $\mathbf{s}_j$, at the current unit in time j.
- To shift one time unit back, the pointer reads the contents of the memory at $(j, \mathbf{s}_j)$.
- A new state, $\mathbf{s}_{j-1}$, is formed in the shift register by shifting in the two bits read from memory in as the most significant bits, and shifting the most significant bit in the old pointer to the least significant bit in the new pointer.
- The counter holding the pointer to the time address is decremented, and the process is repeated until the path has been traced back to the origin.

All through the traceback process the two least significant bits in the pointer at each unit in time are read out as the decoded bit stream. This bitstream now needs to be reversed in order to get the final output.

A useful algebraic description of memory management in Viterbi decoders can be found in [10]. Under this system the pointer to the previous state is described by a vector, where the number of elements in the pointer is equal to the number of states in the code. All the elements are set to zero, except the entry that corresponds to the previous state, which is set to one. If it assumed that $\mathrm{s}_{j-1} = 3$ then a new vector is defined:

$$\underline{\mathrm{s}}_j := (0, 0, 1, 0, 0, 0, 0, 0) \tag{2.16}$$

A vector for the traceback pointer, $\delta(\mathrm{s}_j)$, is defined in exactly the same manner. At any given unit in time there will be S decision vectors, where S is the number of states in the code. These S decision vectors are used to form a square matrix:

$$\Delta_j = \begin{pmatrix} \underline{\delta}(1_j) \\ \underline{\delta}(2_j) \\ \underline{\delta}(3_j) \\ \vdots \\ \underline{\delta}(S_j) \end{pmatrix} \tag{2.17}$$

Traceback would be performed by starting a state s_j and using the pointer stored there to move back in the trellis to state s_{j-1}. Using the algebraic notation a single step in the traceback process is described by:

$$\underline{\mathrm{s}}_{j-1} = \underline{\mathrm{s}}_j \cdot \Delta_j \tag{2.18}$$

The entire traceback process is described by performing matrix multiplication:

$$\underline{\mathrm{s}}_0 = \underline{\mathrm{s}}_j \cdot \Delta_j \cdot \Delta_{j-1} \cdots \Delta_1 \tag{2.19}$$

The algebraic formulation presented above provides a powerful tool for analyzing and developing schemes for implementing SMUs in hardware.

Chapter 3

Implementation Theory

In the previous chapter the Viterbi Algorithm was presented in its theoretical form. Since the algorithm is relatively complex and code specific, there is no single, definitive technique for implementing it in hardware. A few of the broader techniques used to manipulate the algorithm into a form more suitable for hardware implementations will be discussed in this chapter.

3.1 Add–Compare–Select : Practical Considerations

To a large extent the complexity of the Viterbi decoder required to decode a specific convolutional code is set by both the constraint length, ν, of the code and the redundancy in each codeword. The relationships between the decoder's complexity and the properties of the code can be traced hierarchically as follows:

- The constraint length determines the number of states in the code $S = 2^{\nu}$.
- The number of states sets the number of Add-Compare-Select (ACS) circuits that will be required to implement the decoder.
- The redundancy per codeword determines the number of branches entering each state in the code trellis.
- The number of branches entering each state defines the number of path metrics that will need to be compared in each ACS circuit as well as the connectivity between ACS blocks in the decoder.

Therefore the constraint length and codeword redundancy can be used to completely define the number of mathematical operations that need to be performed in the decoder for each received code symbol.

The complexity of each mathematical operation, and as a result the actual complexity of the decoder, is determined by the size of each path metric. In theory all path metrics grow without bound as time stretches to infinity, with the result that infinitely long registers will be needed to store them. However, since survivor paths are those with the minimum path metric, it is the difference between the metrics that are of interest rather than their actual size. A fundamental property of a Viterbi decoder for a code generated by a shift register is that the difference between any two path metrics is bounded [11], [12].

3.1.1 Modulo $2\Delta_{max}$ Path Metric Arithmetic

Given that the dynamic range of the decoder is defined as the maximum difference between any two path metrics, and the actual value of the path metrics is of no interest, the path metrics can be re-scaled to fit within the dynamic range. The most obvious way of accomplishing this would be to re-scale all path metrics by a value equal to the smallest path metric in the decoder. In terms of hardware this would mean comparing all the path metrics from the S states, selecting the lowest one, and subtracting it from all path metrics in the decoder, including itself. Additional delays in the path metric update loop would be unavoidable since the next received symbol cannot be processed until all the path metrics have been re–scaled. This would still apply in cases where the path metrics are only re-scaled every couple of cycles of the ACS unit, since the delay in the path metric update loop would be upper bounded by the delay of the cycle that includes the path metric update process.

Another approach that yields a reduction in both the hardware complexity and the time taken between successive path metric updates is possible [11]. It is known that if it can be shown that $|A - B| \leq \Delta$, then the difference can be evaluated as $(A - B) \bmod 2\Delta$ without ambiguity.

Since it is known that the dynamic range of the path metrics in the decoder is in fact bounded, the above formula can be used to perform the path metric comparisons. Rather than re-scaling the path metrics, they are now allowed to overflow mod 2Δ. The ratio between path metrics is retained even though the actual value of the path metrics are lost.

3.1.2 Deriving the Decoder's Dynamic Range

Although the Modulo $2\Delta_{max}$ approach requires the dynamic range of the decoder to be defined, Hekstra [11] does not formulate a technique of deriving the actual value of the bound, but does prove that such a bound exists. Viterbi [7], however, proves that the dynamic range of a Viterbi decoder for a convolutional code is upper bounded by:

$$\Delta_{max} = \nu \cdot \lambda_{max} \tag{3.1}$$

Where λ_{max} is the maximum branch metric, ν is the code's constraint length and Δ_{max} is the maximum dynamic range between any two path metrics. This result is shown to hold as a maximum limit for all convolutional codes in [7] by noting that since the encoder's shift registers must fill with new bits within ν time units, any state in the code trellis can be reached from a state along the correct path within ν time units as well.

A closer examination of codes with rates greater than $\frac{1}{n}$ shows that this bound is actually too large. If a rate $\frac{b}{n}$ code is being used then the shift registers in the encoder will fill with new bits in round$(\frac{\nu}{b})$ time units (where 'round' denotes rounding up to the nearest integer). Therefore all states can be reached from the correct path (the one with the lowest path metric) in round$(\frac{\nu}{b})$ time units, and the maximum difference between path metrics at any point is defined as:

$$\Delta_{max} = \text{round}(\frac{\nu}{b}) \cdot \lambda_{max} \tag{3.2}$$

Reducing the Estimate of the Dynamic Range in Decoders for TCM

As indicated above, the techniques outlined in the literature for deriving an estimate of the dynamic range of a Viterbi decoder will give results that are too large when applied to codes of rates greater than $\frac{1}{n}$. New techniques are presented here that can substantially reduce the estimate of the dynamic range in such decoders, as well as the dynamic range of decoders for TCM codes that meet certain basic criteria.

In a TCM encoder the dynamic range found in 3.2 will still be too large, if it can be shown that the following properties can be assigned to the code:

1. Given any state $\mathbf{s}_j, \mathbf{s} \in S$, all paths leaving the state at time unit j and ending in a state $\mathbf{g}_{j+m}, \mathbf{g} \in S$ are made up of a unique set of channel symbols.

2. Choosing any path from state $\mathbf{s}_j$ to state $\mathbf{g}_{j+m}$ there is at most one other path starting at state $\mathbf{s}_j$ and ending in any state at time unit $j+m$ that will be separated by the maximum distance, $m \cdot \lambda_{max}$, from the chosen path.

3. The branches leaving any state $\mathbf{s}_j$ are separated by, at most, the maximum Euclidian distance possible given a fixed signal set. Similarly the branches entering each state are separated by, at most, the maximum Euclidian distance. All other transitions between two different pairs of states are not separated by the maximum Euclidian distance, but by some smaller distance.

4. The code is uniform.

Property 1 will be satisfied if set partitioning is used to design the code. Since each branch leaving a state $\mathbf{s}_j$ corresponds to a different input codeword, it must be mapped to a different point in the signal set. In other words the first symbol of any path leaving state $\mathbf{s}_j$ is different from the first symbol of all other paths leaving state $\mathbf{s}_j$ and all paths must be uniquely defined.

Properties 2 and 3 both depend on the signal set used with the code. As long any single point in the signal subset assigned to the branches leaving any state has at most one other point in the signal subset that is spaced from it by λ_{max}, then there can be at most one path separated by $m \cdot \lambda_{max}$ from any other path that also originates from state $\mathbf{s}_j$. When M-ary PSK schemes are used this will always be the case:

- Once the signal subset has been sub-divided through set partitioning, each subset can be further sub-divided into pairs of points that lie opposite one-another across the origin.
- Therefore if any single point is chosen in the signal subset there can only be one other point spaced from it by λ_{max} within the same subset.
- The first branch of all paths leaving from state $\mathbf{s}_j$ must be assigned signals from the same subset, and so there can only be one path separated from any other by $m \cdot \lambda_{max}$.

The first part of property 3 is fairly obvious: branches entering or leaving a state are separated by a maximum of λ_{max} through set partitioning. Depending on the code used, the same may well hold true for all paths entering each state (see section 2.2.2). The second part of property 3 will depend on the modulation scheme used. The property will always be satisfied when the modulation scheme is M-ary PSK since set partitioning would ensure that all branches leaving any single state will be assigned to a signal subset that is maximally spaced. All other subsets would be reached by rotating this first subset by a multiple of $\frac{2\pi}{M}$. This in turn means that branches with symbols assigned to different signal subsets must be separated by less than λ_{max}, which is the case with transitions that do not originate from or end in the same pair of states.

Property 4 is dependant on the code used (see section 2.2.3 for a definition of uniformity and the UDP). As the code is uniform, it must also have the UDP. When these four properties are combined with equations (3.1) and (3.2) the estimate for the maximum dynamic range of the decoder can be substantially reduced.

Since the code is uniform, only the error events from a fixed node (or state) need be considered when defining the operation of the code [3]. So in order to determine the maximum dynamic range of the decoder it is sufficient to simply take the all zero path as the correct path, and consider the dynamic range between this path and all error paths which diverge from the correct path at time unit zero.

The dynamic range between any two paths through the code trellis will be at a maximum if the received sequence corresponds exactly with one of the paths (see Appendix A for a formal proof). An error in the received sequence will increase the path metric of the correct path, and will, by property 3, lower the path metrics of all other paths in the trellis. This will lower the dynamic range of the decoder as a whole. If the errors are severe, then a path other than the correct path will have the lowest path metric. Since the code is assumed to have the UDP (see property 4) this will make no difference to the evaluation of the dynamic range.

Taking note of the above, the properties of the TCM code that allow the estimate of the dynamic range to be reduced are:

- If a state $\mathbf{s}_j$ is chosen that lies along the correct path, then by property 3, of the $2^n - 1$ error paths that branch off from $\mathbf{s}_j$, only one will have an associated path metric that will be λ_{max} greater than the correct path.
- If this path re-merges with the correct path (given that there are parallel transitions in the code) at time unit $j+1$, then the correct path will be chosen as the survivor, and all the other error paths beginning at state $\mathbf{s}_j$ will have path metrics that have grown by less than λ_{max}.
- If the path does not re-merge with the correct path then it must end in a state that does not lie along the correct path.
- Of the 2^n transitions leaving this new state, only those that re-merge with the correct path can lead to a growth of λ_{max} in the error path metric with respect to the correct path metric.
- All other paths will have grown by, at most, some λ_{adj} defined by the Euclidian distance between the received point and a point in an adjacent subconstellation such that λ_{adj} is a maximum and $\lambda_{adj} < \lambda_{max}$.

As a result the maximum dynamic range between all paths beginning in state $\mathbf{s}_j$ and ending in different states at time unit $j + m$ will be given by:

$$\delta_{max} = \lambda_{max} + (m - 1)\lambda_{adj} \tag{3.3}$$

In addition all states within the code trellis can be reached from a single state $\mathbf{s}_j$ within round$(\frac{\nu}{b})$ time units. Setting $m = \text{round}(\frac{\nu}{b})$ in equation (3.3), Δ_{max} can then be re-defined as:

$$\Delta_{max} = \lambda_{max} + (\text{round}(\frac{\nu}{b}) - 1)\lambda_{adj} \tag{3.4}$$

For codes of rate $\frac{b}{n}$ where $b > 1$, the bound described by equation (3.4) will still be too large. In these cases the estimate of the dynamic range can be further reduced by observing that:

- There will be b distinct, unmerged paths originating from state $\mathbf{s}_j$ and ending in all states at time unit $j + m, m = \text{round}(\frac{\nu}{b})$.
- There can only be one path ending in any state $\mathbf{s}'_{j+m}$ that is separated from the correct path by Δ_{max}, where state $\mathbf{s}'_{j+m}$ does not lie on the correct path.
- There must also be $b - 1$ other paths originating from state $\mathbf{s}_j$ that also end at state $\mathbf{s}'_{j+m}$ with smaller path metrics.

- The first transition in all paths originating from the same state must come from the same signal subset.
- As the path with the largest metric has as its first transition the branch with branch metric λ_{max}, the other paths must have as their first transition a branch with a smaller branch metric.

Defining λ_{adj1} as the second largest branch metric associated with a path within any of the signal subsets, the estimate of the dynamic range for decoders for codes with rates of $\frac{b}{n}$, where $b > 1$, can now be reduced to:

$$\Delta_{max} = \lambda_{adj1} + (\text{round}(\frac{\nu}{b}) - 1)\lambda_{adj} \text{ for } b > 1 \tag{3.5}$$

From section 3.1.1 and equation (3.5) or equation (3.4) the size of the path metrics in bits can be derived [14]:

$$\Upsilon = \log_2(\Delta_{max} + \lambda_{max}) + 1 \qquad \text{bits} \tag{3.6}$$

The extra λ_{max} is to account for the growth in dynamic range in the ACS unit when the branch metrics are added to the path metrics prior to any comparisons being made.

It is clear from equation (3.6) that in order to gain a 1 bit saving in the size of the path metrics Δ_{max} will have to be halved. In some cases the new derivations for Δ_{max} introduced in equations (3.4) and (3.5) will yield no additional savings in hardware. In codes with large ν and with $b > 1$, however, savings of 1 bit in the length of the path metric registers are possible.

Tightening the Upper Bound on the Dynamic Range

The approach given above only produces a rough estimate of the dynamic range of a decoder. It is, more precisely, a loose upper bound on the dynamic range of the decoder. In most cases this bound will be sufficiently accurate to determine the number of bits needed to store the path metrics since the dynamic range will need to be rounded up to an integer power of two. In other applications, however, this bound may be too loose to yield any hardware savings when compared to the results from equation (3.2), or may indicate that minor reductions in the upper bound on the dynamic range will yield additional hardware savings.

In the later case techniques similar to those used to derive the code's free distance can be used to tighten up the bound on the decoder's dynamic range. The decoder's dynamic range is defined by:

$$\Delta_{max} = \max_{\mathbf{y},L}\left[\max_{\mathbf{c}}\left(\lambda(\mathbf{y}_{L-1} \oplus \mathbf{c}_{L-1}) + \min_{\mathbf{c}}\sum_{k=0}^{L-2}\lambda(\mathbf{y}_k \oplus \mathbf{c}_k)\right) - \min_{\mathbf{e},\mathbf{e}\neq\mathbf{c}}\sum_{k=0}^{L-1}\lambda(\mathbf{y}_k \oplus \mathbf{e}_k)\right] \tag{3.7}$$

where both $\mathbf{e}$ and $\mathbf{c}$, which denote two error sequences such that $\mathbf{e} \neq \mathbf{c}$, are valid code sequences.

Although equation (3.7) describes the decoder's dynamic range in the general sense, only the maximum dynamic range encountered when performing path metric comparisons in each ACS unit is of interest. In other words limitations must be placed on the error sequences $\mathbf{e}$ and $\mathbf{c}$ such that the paths described by $\mathbf{y} \oplus \mathbf{e}$ and $\mathbf{y} \oplus \mathbf{c}$ both start from, and end in, the same state. This will introduce a dependence between the two error sequences $\mathbf{c}$ and $\mathbf{e}$. In terms of the encoder of Figure 2.2:

1. Since $y_0^{(3)}$ is only dependant on s_0, and both paths start at s_0, then $e_0^{(3)} = c_0^{(3)} = 0$.
2. If the two paths end in different states at time unit $L-2$, then $s_{L-2}^{(3)} = s_{L-1}^{(1)}$ in both cases and $e_{L-1}^{(3)} = c_{L-1}^{(3)}$ to ensure that the two paths end in the same state.

This allows for a small reduction in the complexity of the search.

3.1.3 Branch Metrics and Quantization

The size of the path metrics is determined by the size and allocation of the branch metrics, as was shown in section 3.1.2 . Each transition in the code trellis is assigned to a point in the signal constellation, and hence will be assigned a branch metric by the decoder. The branch metric is an estimate of the likelihood that the channel signal associated with the transition in question forms part of the transmitted sequence given a received signal in the presence of noise. If the received signal is denoted as $\mathbf{z}_j$ then the branch metric assigned to the transition labelled by $\mathbf{x}'_j$ will be given by [6] [1]:

$$\lambda(\mathbf{x}'_j) = -\ln(p(\mathbf{z}_j|\mathbf{x}'_j)) \tag{3.8}$$

In an AWGN channel with discrete input, the output will be continuous. As the decoder is digital, the continuous-valued channel output, $\mathbf{z}_j$, will be sampled and quantized before being processed. The sampled signal, $\hat{\mathbf{z}}_j$, will have a probability distribution rather than a probability density as was the case with $\mathbf{z}_j$. As pointed out by Dunham et al [13] this introduces a mismatch between the decoder and the channel which effectively raises the upper bound on the error probability at the output of the decoder. In order to keep the decoder optimal it is desirable to choose both the branch metrics and the quantization levels such that the decoder models the channel as closely as possible.

Since it is the branch metrics that are of interest as they determine the complexity of the decoder, the branch metrics are typically used to determine the number of levels in the quantizer. There are two central problems in determining appropriate branch metrics for the decoder. The first is that the probability distribution $P(\hat{\mathbf{z}}_j|\mathbf{x}'_j)$ is a real number,

[1] Maximising with respect to $p(\mathbf{z}_j|\mathbf{x}'_j)$ is equivalent to maximising with respect to $p(\mathbf{x}'_j|\mathbf{z}_j)$ [6].

which creates problems when the branch metrics have to be stored and processed as binary numbers. The second problem is that the values of the branch metrics are linked to the signal to noise ratio in the channel. The first problem is solved by noting that the decoder is only interested in the relative difference between the branch metrics, and not their actual value. Integer representations of the branch metrics will suffice as long as the relative sizes of the branch metrics remain more or less unchanged. The second problem either requires the demodulator to estimate the signal to noise ratio in the channel, or it must be assumed that the use of fixed branch metrics across a wide range of signal to noise ratios will have a negligible effect on the performance of the decoder. The later assumption was adopted in this project since it seems to be prevalent in the literature [14], [15].

There are numerous schemes in the literature for assigning branch metrics in Viterbi decoders for general convolutional codes [15], [13], [16]. In these implementations it was assumed that the codewords are fed to the modulator as a serial bit stream. In the demodulator each received signal would be allocated two metrics, one giving the probability that a binary one was received, and the other giving the probability that a binary zero was received. The branch metrics for the codewords assigned to the transitions in the code trellis would then be formed by adding the probabilities that the received bits correspond to the respective bits in each of the codewords.

The simplest implementation of the above involves assigning a linear scale to each of the quantization intervals [15]. In Clark [16], however, it is shown that with a small loss in performance it is possible to reduce the number of bits required to represent the branch metrics by "zero padding" around the quantization intervals corresponding to the binary digit under scrutiny. This approach is based on the assumption that if the received signal is very close to the quantization interval assigned to either a binary one or a binary zero, then it has a very high probability of being either a binary one or a binary zero. This approach is taken one step further by Dunham et al [13] who show that it is in fact optimal in terms of matching the decoder to the channel. They attribute the loss in performance suggested by Clark not to the "zero padding", but to the reduction in the number of bits used to represent the non-zero branch metrics. They do, however, back up Clark's claim that Viterbi decoders are relatively insensitive to changes to the number of bits used to represent the branch metrics.

Although the above mentioned techniques cannot be directly applied to TCM schemes since under TCM the codewords are mapped directly into channel signals, the general principles will still apply. For two dimensional signalling in AWGN of variance σ, the probability that a channel symbol $\mathbf{x}'_j$ was transmitted given that a noisy symbol $\mathbf{z}_j$ was received is given by the probability density:

$$p(\mathbf{z}_j|\mathbf{x}'_j) = \exp\left[-|\mathbf{z}_j - \mathbf{x}'_j|^2/2\sigma^2\right] \cdot \left(2\pi\sigma^2\right)^{-1} \tag{3.9}$$

Equation (3.9) assumes that the decoder is working directly on the continuous valued variable $\mathbf{z}$. If the signal is passed through a linear quantizer with quantization intervals of

size h to yield the input signal to the decoder $\hat{\mathbf{z}}$, then the resultant probability distribution will be:

$$P(\hat{\mathbf{z}}_j|\mathbf{x}'_j) = \int_{\hat{z}_j-\frac{h}{2}}^{\hat{z}_j+\frac{h}{2}} \exp\left[-|\mathbf{z}-\mathbf{x}'_j|^2/2\sigma^2\right] \cdot \left(2\pi\sigma^2\right)^{-1} \, d\mathbf{z} \tag{3.10}$$

Substituting equation (3.10) in place of $p(\mathbf{z}_j|\mathbf{x}'_j)$ in equation (3.8) and scaling all results to integer numbers will yield the desired branch metrics.

The number of quantization levels that will be needed in the receiver will follow from the number of bits used for the branch metrics. Although making the quantization intervals finer and then "zero padding" to keep the branch metrics the same size is likely to yield improvements in performance, the gains will be small in relation to the added complexity in the circuitry. Typically the number of quantization levels are chosen so that the branch metrics can just be accommodated with a little bit of leeway for "zero padding".

3.2 Survivor Memory : Practical Considerations

3.2.1 Truncation Depth in Viterbi Decoders

Maximum likelihood detection of TCM sequences implies that the entire received sequence is compared with all legitimate code sequences, and the code sequence with the shortest Euclidian distance from the received sequence is then chosen as the most likely of the transmitted sequences. In practical terms this would mean that the entire transmitted sequence would have to have been received before decoding begins. This would not present a problem if our original bit stream was sent as packets, but when using a continuous bit stream it would mean receiving an infinitely long sequence before getting any output from the decoder.

In practice this problem is circumvented by noting that error paths in the Viterbi decoder will merge with the correct path in a probabilistic fashion. It is shown in [7] that the probability of an error path exceeding a length M before re-merging with the correct path can be made smaller than the union bound on the error performance of the Viterbi decoder by making the length M sufficiently large. M is called the decoder's truncation depth, and corresponds to the number of decisions that are stored before decoding begins. For a rate $\frac{2}{3}$ code the truncation depth needs to be about 10 times the constraint length, ν.

3.2.2 Survivor Memory Management

General Issues

As a graphical representation of the above process, assume a code trellis is plotted as the symbols are received, and all paths that do not end at one of the S states at the right hand end of the trellis are "pruned" from the trellis. After M symbols have been

received the trellis would resemble a tree lying on its side, with the "branches" at the right and the "trunk" at the left, as illustrated in Figure 3.1 [14]. If the SMU were to initiate S traceback pointers and started tracing back along the paths ending at each of the S states at the right hand end of the trellis, a point would be reached where all S pointers would be moving along the same path.

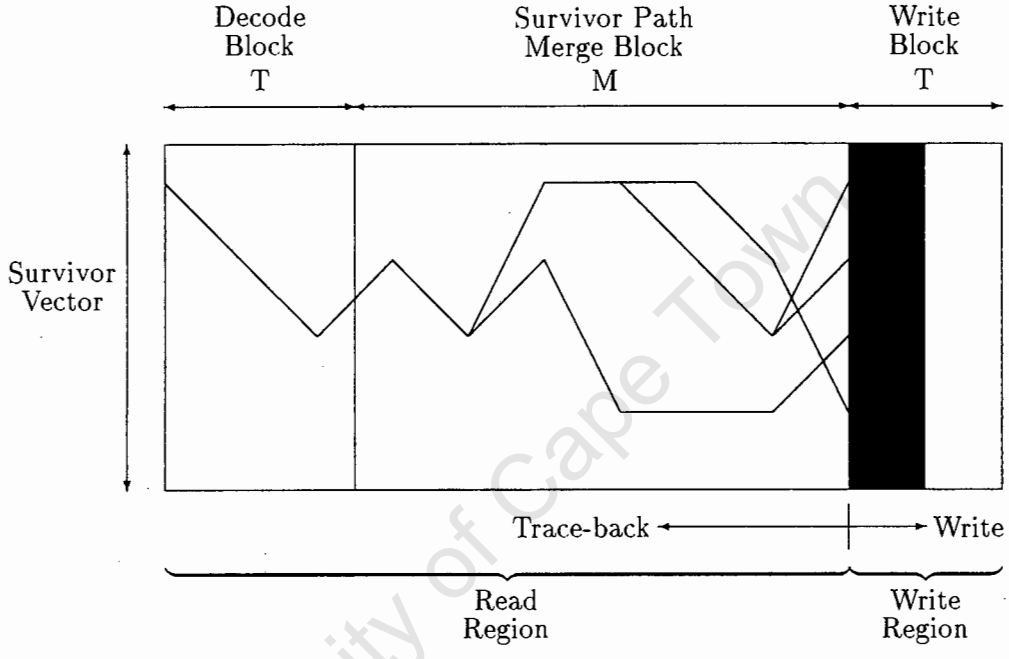


Figure 3.1: SMU Trellis "Tree"

In terms of the algebraic notation presented in Section 2.3.4, the $\underline{s}_0$ arrived at in equation (2.19) would be the same no matter which $\underline{s}_j$ were used to initiate the trace-back as long as $j \geq M$. This obviates the need for finding the state with the minimum path metric before starting the traceback, and therefore saves a substantial amount of hardware.

Since the path being traced back may not be the path with the minimum path metric, the entire path can no longer be decoded. Instead, as each symbol is received, only the transitions M time units back (or more) along the path are decoded. In terms of the algebraic notation introduced in section 2.3.4 the first three steps after M symbols are received would be:

$$\underline{s}_0 = \underline{s}_M \cdot \Delta_M \cdot \Delta_{M-1} \cdots \Delta_1$$

$$\underline{s}_1 = \underline{s}_{M+1} \cdot \Delta_{M+1} \cdot \Delta_M \cdots \Delta_2$$

$$\underline{s}_2 = \underline{s}_{M+2} \cdot \Delta_{M+2} \cdot \Delta_{M+1} \cdots \Delta_3$$

As can be seen, after $\underline{s}_0$ has been decoded, the decision matrix Δ_1 is no longer needed. The space occupied in memory by Δ_1 can therefore be overwritten by Δ_{M+1} in time unit

$M + 1$. This creates a "sliding window" effect in the survivor memory, and allows the decoder to produce an output bit stream in a continuous fashion after M symbols have been received.

The k-Pointer Algorithms

Closer examination of the previous three equations will show that M matrix multiplications are performed at each stage in order to decode the output. In terms of an SMU implemented in hardware, this would mean performing M memory read operations for each single write operation. Although read operations can normally be performed substantially faster than write operations [8], the value of M is usually large, and the SMU would become the limiting factor in terms of the operating speed of the decoder.

The only way to overcome this problem is to use multiple read pointers. Numerous schemes have been proposed [4], [8], [9] and [14]. Although [9] introduced the k-pointer method developed further in [8], the paper goes on to recommend using shift registers to implement the SMU in much the same configuration as proposed in [4]. In a shift register implementation the SMU is made up of $N = 2^{\nu} \cdot b$ shift registers each $2M$ long. As new survivors are generated by the ACS unit, they are shifted into the registers and all the previous survivors are moved along. The M read pointers perform one read operation for each write operation. A single read pointer, starting its traceback at stage M, will only reach the end of the memory at stage $2M$ and hence the requirement for shift registers $2M$ long.

The above approach seems counter intuitive. The traceback method was first introduced in [17] where one of the reasons given for its inception was to do away with the need for shifting large amounts of data in the SMU (this was a requirement when using the register exchange method). This is especially true when ν and/or b are large.

The alternative approach uses RAM modules to implement the SMU and uses the RAM address to move through the memory, rather than physically moving the data around. The algorithms associated with this approach are collectively known as the k-pointer algorithms. As with the shift register method the total memory is N rows wide, and typically $2M$ columns long.

Three different operations need to be performed on the RAM. Referring to Figure 3.1, these operations are described in terms of the workings of the decoder [8].

Writing New Data (WR) As the survivor vector is generated by the ACS unit it needs to be written into the memory. Each individual survivor is written into a memory position that corresponds to its state. The write pointer moves forward (from the left to the right in Figure 3.1) through the memory as the ACS unit moves on to each new stage in the code trellis. Since the survivor vector is N bits wide, N bits need to be written to memory before the write pointer can move forward one stage.

Traceback Read (TR) Once M (or more) survivor vectors have been written to memory, the SMU needs to start tracing a path back to the origin of the trellis in order to decode the output bits. It is one of two read operations performed by the SMU, but in this case no decoding is done on the bits read from memory. At each stage a pointer, b bits wide, is read from the survivor vector at the position determined by the current state. This pointer is combined with the current state to obtain the previous state along the path being re-traced.

Decode Read (DR) This operation is identical to the TR operation, except that the pointers read from memory are now decoded and passed on to the bit order reversing circuits. Once these pointers have been reversed, they form the output to the decoder. The decode read operation frees up memory space for use by the write operation.

Since the memory size in hardware is limited, the write operation must fill the space freed up by the decode read operation. In order to avoid a collision between the decode read pointer and the write pointer there must be an average of one decode read operation for every survivor vector written into memory.

The overhead incurred by having to trace back at least M stages before a decode read operation can occur can be reduced by waiting $M + T$ stages before commencing a traceback front. As all paths merge M stages back, T stages can now be decoded once the traceback read operation has run over M stages. Now $(k-1) = \frac{M}{T}$, where k is the number of read operations that need to be performed for each column write operation. The decode read operation will produce the output bits in reverse order, so some form of bit order reversing circuitry will be required.

The physical memory structure presented in Figure 3.1 is derived from the above principles. The memory is split into three distinct regions according to the operation being performed in that region. In order to get an average of one decode read operation for each column write operation, the physical size of the area occupied by the decode read and write operations is made equal. Since the pointers move through the memory in a cyclic fashion, the area of memory occupied by the traceback read pointer must be made up of blocks equal to T.

Having established a basic memory structure there are three main algorithms presented in the literature [8], plus hybrids of the three. Two of the three algorithms presented in [8] are generalisations of algorithms used in other implementations in the literature [18], [9]. It will be shown that a tightened mathematical description of the hybrid algorithms encompasses the descriptions of these two algorithms as well. The third algorithm was originally presented as being distinct from the previous two. Once again it will be shown that the description of the third algorithm is encompassed in the tighter description of one of the hybrid algorithms.

Once the global definitions for the two families of algorithms are obtained, a new way of managing read and write operations in the SMU will be introduced that simplifies the

implementation of one of these families. The relative merits of each family of algorithms will then be discussed.

The k-Pointer Even Algorithm In the k-pointer even algorithm, k read pointers (including decode read pointers) are used. The memory is made up of $2k$ memory blocks, each block being $\frac{M}{k-1}$ columns long. The write pointer moves from left to right through the memory and the read pointers move from right to left. One write, one decode and $k-1$ traceback read operations are performed in parallel. The structure is illustrated in Figure 3.2 for $k = 3$.

There are two fundamental measures of the performance of the various k-pointer algorithms. The first is the number of memory blocks required by the algorithm and the second is the latency in the algorithm. In [8] the latency is defined as the time taken from a column being written into memory and the same column being subjected to a decode read. Under this definition the latency varies within each memory block depending on which column is under scrutiny. A more useful definition is proposed here: the latency is the time taken from the very first column write until the first output is received from the SMU. Under this new definition, the latency is the "delay" in the decoder due to the SMU.

In the k-pointer even algorithm the first bit will be read out when the write pointer reaches the end of the memory. Since each memory block is $\frac{M}{k-1}$ columns long, and there are $2k$ memory blocks, the overall latency under the new definition will be $\frac{2kM}{k-1}$ (this is equivalent to the latency of the first column in the block under the old definition).

As previously mentioned, the output is produced in reverse order. Feygin et al, [8], list three main schemes to perform the bit order reversing:

1. Two **LIFO stacks** can be used to perform the bit order reversing. Each stack will be $T = \frac{M}{k-1}$ columns long, and b rows wide. As the data from the decode read operation is written into one stack, the data from the previous block decode read operation is popped from the other stack in reverse order.

2. One **single port RAM block** can be used to perform the bit order reversing by interleaving the read and write processes. The RAM block would need to be $T = \frac{M}{k-1}$ columns long, and b bits wide. The write operation would be able to overwrite the space freed up by the read operation in exactly the same way as the write operation in the SMU overwrites the space freed up by the decode read operation.

3. One **dual-port RAM block** can also be used to perform the bit order reversing. The second port would do away with the need to interleave the read and write operations. Operation would otherwise be functionally the same as the single port RAM block technique.

4. *A* new scheme will be presented in section 3.2.2 that constitutes a fourth method for implementing bit order reversing circuits in SMUs.

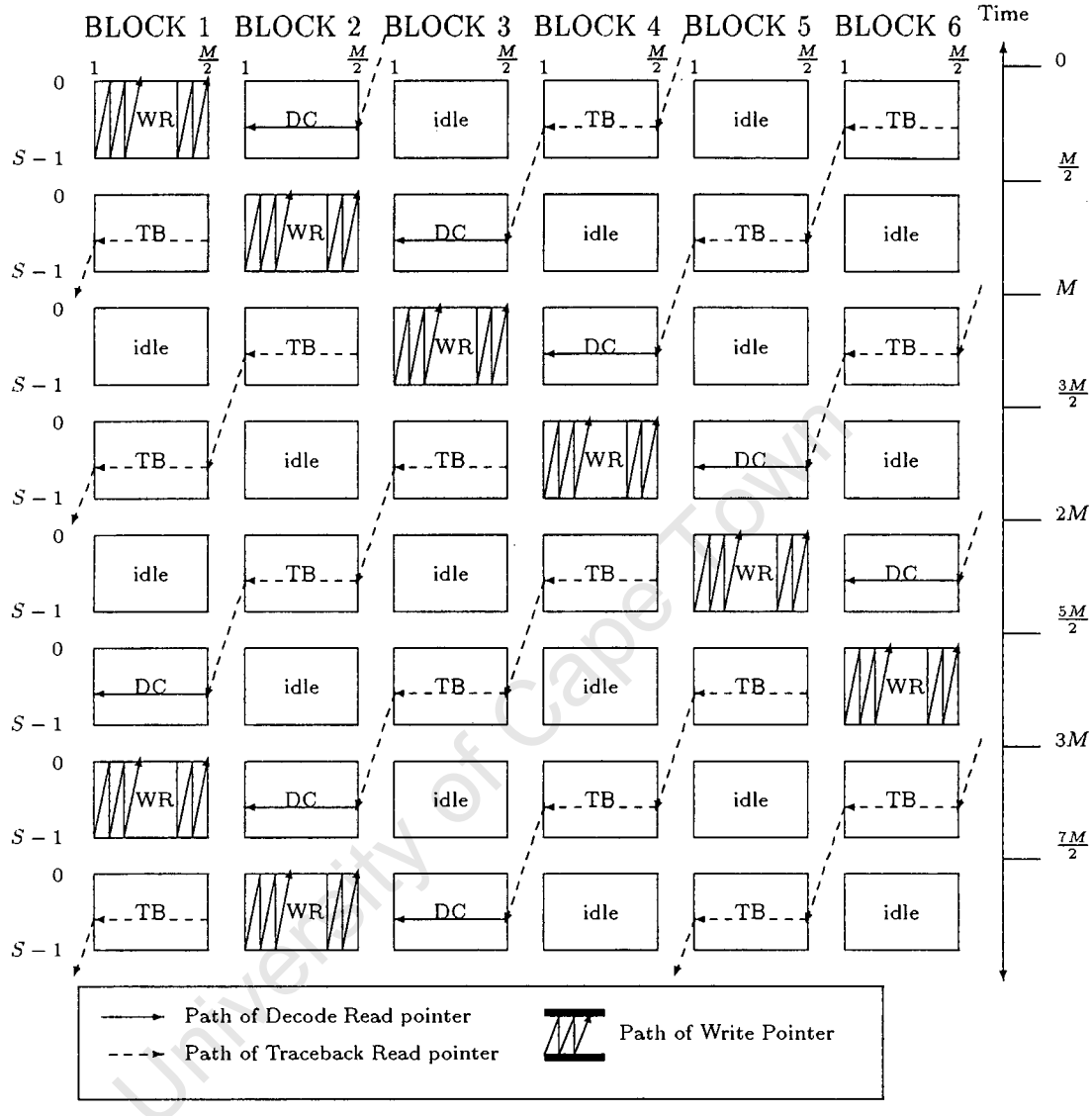


Figure 3.2: Structure and workings of the $k = 3$ even algorithm

The k-Pointer Odd Algorithm Like the k-pointer even algorithm, the k-pointer odd algorithm still uses k read pointers. However, by interleaving the decode read and the write pointer, the algorithm only requires $2k - 1$ memory blocks. Each memory block is still $\frac{M}{k-1}$ columns long. The latency of the algorithm is $\frac{2kM}{k-1}$.

In the k-pointer even algorithm, $\frac{M}{k-1}$ blocks were cleared by the decode read pointer before being overwritten by the write pointer. Since the memory is freed up as the pointer is read from memory by the decode read operation, it can immediately be overwritten by the write operation. In other words, the decode read pointer and the write pointer always

point to the same column in the same memory block. As the read and write operations must take place in opposite directions, successive write fronts through any single memory block must take place in opposite directions. The k-pointer odd algorithm for k=3 is illustrated in Figure 3.3.

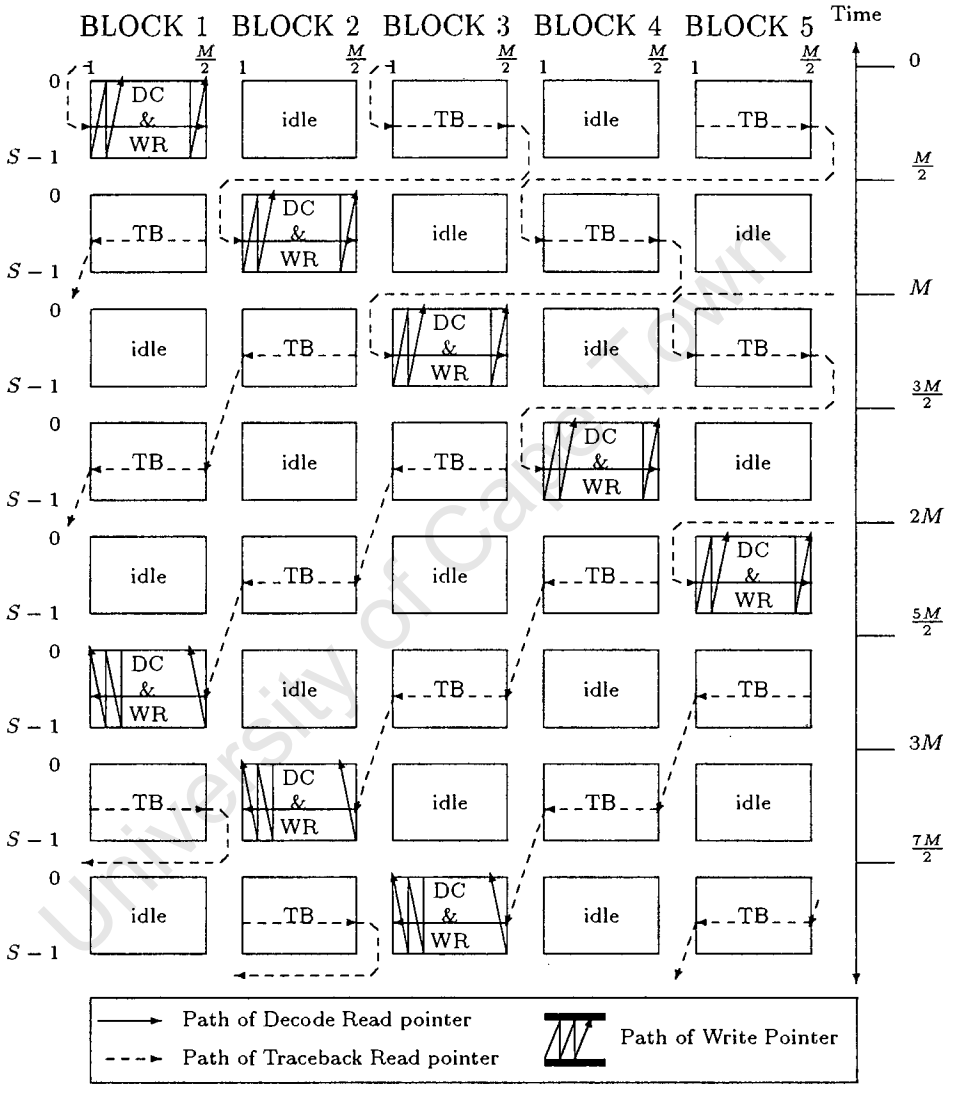


Figure 3.3: Structure and workings of the $k = 3$ odd algorithm.

As with the k-pointer even algorithm, the output bits from the decode read operation have to be reversed using either a LIFO stack structure or a RAM block. The fourth option mentioned above is to perform the bit order reversing in a RAM block in exactly the same way as the decode read and write operations are interleaved in the other RAM blocks in the SMU. Specific implementation issues will be discussed when the algorithms

are compared in section 3.2.2.

The One-Pointer Algorithm The one-pointer algorithm was originally presented in [8] as being distinct from the two algorithms presented above. It will be shown, however, that the one-pointer algorithm can be considered a special case of a more general R-pointer even algorithm.

Under the one-pointer algorithm the read operations are accelerated, so that k read operations are performed for each column write operation. This means that only $k+1$ memory blocks will be needed, each $\frac{M}{k-1}$ columns long. The total latency of the algorithm will be $\frac{(k+1)M}{k-1}$ since k blocks are traced back in the time it takes the write operation to fill one block and at least $\frac{3M}{k-1}$ column write operations need to be performed before traceback can begin. The one-pointer algorithm is illustrated in Figure 3.4.

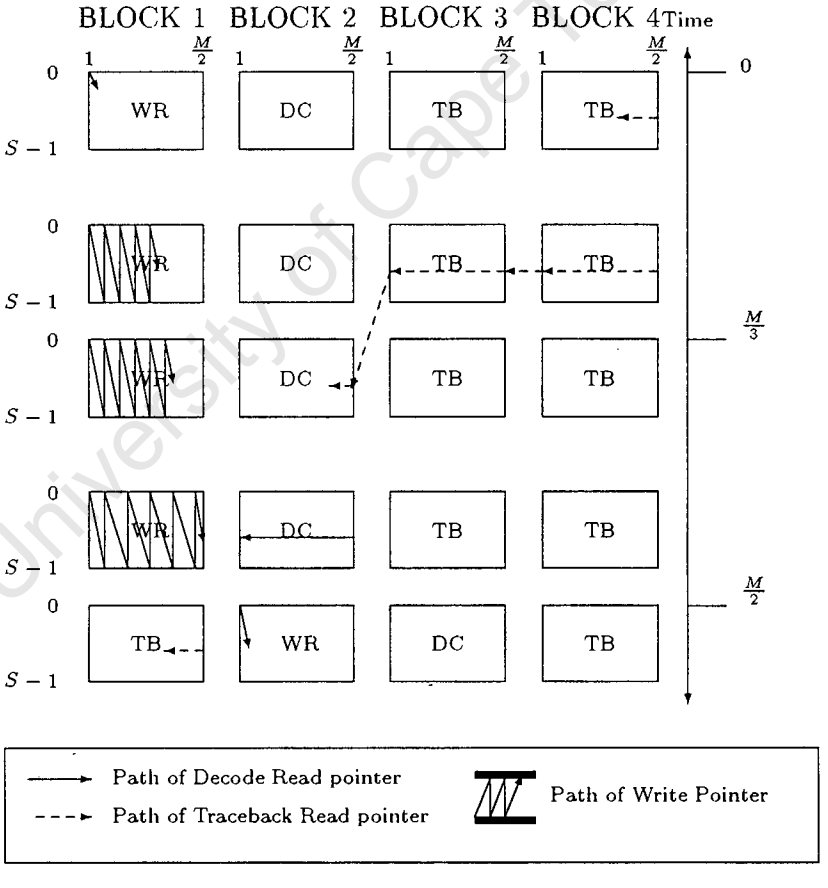


Figure 3.4: Structure and workings of the one-pointer algorithm

In the one-pointer algorithm the decode read operation produces output in bursts. These

bursts can be eliminated by the bit order reversing circuit by making provision for writing the bits to the circuit at a higher rate than they are read out. Under the one-pointer even algorithm only the two stack LIFO bit order reversing circuit can be used, since the stacks can be filled and emptied at different rates.

Feygin et al, [8], also point to the existence of hybrid algorithms, formed by combining the ideas contained within the one pointer algorithm with those contained in the k-pointer even algorithm. Essentially the even hybrid algorithms involve speeding up the read pointer in relation to the write pointer.

The Two-Pointer Odd Algorithm Also presented in [8] is a hybrid algorithm of the k-pointer odd and one pointer algorithms. As with the even hybrid algorithms, the read operations are sped up with respect to the write operations in the k-pointer odd algorithm. However, because the decode read pointer and the write pointer move together through the same memory block, it is not possible to reduce the number of read pointers below two. As a result $k-1$ read operations are now performed for each column write/decode read operation.

The two-pointer odd algorithm is then the k-pointer odd equivalent of the one-pointer algorithm. Each memory block is still $\frac{M}{k-1}$ columns long, and because $k-1$ read operations are being performed for each write operation there will be a total of $k+1$ memory blocks. The latency of the algorithm will, however, increase to $\frac{(k+2)M}{k-1}$.

As before, the output from the decode read operation is in reverse order. Unlike the one-pointer algorithm, there is one output in the two-pointer odd algorithm for each column write operation. The two-pointer odd algorithm therefore requires no burst elimination and can use any of the bit order reversing circuits mentioned in the k-pointer even algorithm.

Generalised R-Pointer Algorithms It is proposed here that a more intuitive description of the algorithms can be gained from tightening the mathematical descriptions of the hybrid algorithms to form two new R-pointer families of algorithms. R, in this case, would refer to the number of physical read pointers, rather than the number of read operations performed in the SMU per column write operation, k, as was used to describe the k-pointer algorithms.

The distinction between the two families of algorithms would be based on the way the decode read and write operations are handled in the SMU. Once this definition is adopted it becomes clear that the k-pointer even algorithm and the one-pointer algorithm are simply special cases of a broader family of algorithms that have their decode read and write pointer operating in separate memory blocks within the SMU. This broader family of algorithms will be referred to as the R-pointer even algorithms.

In much the same way the k-pointer odd algorithm and the two-pointer odd algorithm can be thought of as special cases of a broader family of algorithms that have as their

distinguishing feature the fact that the write and decode read pointers operate on the same column in memory. This family of algorithms will be referred to as the R-pointer odd family of algorithms.

In order to form the descriptions of the two broader family of algorithms, the *traceback recursion rate* (TRR), first introduced in [14], will need to be introduced into the mathematical descriptions of the algorithms. The TRR of the SMU is defined as the number of series read operations that need to be performed for each iteration of the ACS unit.

The two R-pointer families of algorithms are now defined together:

- Of the $2^{\nu} \cdot b$ bits making up the survivor vector produced by the ACS unit each cycle, 2^{x} bits are written into memory simultaneously($x \leq \nu + \log_2 b$). As a result the SMU will have to perform $2^{\nu - x} \cdot b$ write operations per cycle of the ACS unit.
- For a TRR of η, the algorithm will contain $R = \frac{k}{\eta}$ read pointers, including decode read pointers. Here k is any integer such that $1 \leq k \leq M + 1$. In the odd family of algorithms, η is now allowed to be a rational number since one read pointer is slowed down to operate at the same speed as the column write operation [2]. η would be calculated for the odd family of algorithms as:

$$\eta = \frac{(\text{No of series traceback reads performed per column write}) + 1}{R}$$

- The average number of series read operations that need to be performed for each write operation will be given by:

$$\Gamma = \frac{\eta}{2^{\nu - x} \cdot b} \tag{3.11}$$

- M column write operations (ACS cycles) need to occur before traceback can begin. In addition a limit is imposed in that the memory must be made up of blocks of equal size. The length of each memory block is defined as:

$$T = \frac{M}{k - 1} \tag{3.12}$$

- The total number of memory blocks is given by:

$$B_{even} = k \cdot \left(\frac{\eta + 1}{\eta} \right) \tag{3.13}$$

$$B_{odd} = R \cdot (\eta + 1) - 1 \tag{3.14}$$

[2] This is in contrast to Feygin et al who treat η (which they call k_1) as being strictly an integer. The net result is that the equations they give for the total number of memory blocks and the latency of odd hybrid algorithms yield results that are too large.

where equation (3.13) would apply to the even family of algorithms and equation (3.14) would apply to the odd family of algorithms. The odd algorithm will require less memory blocks than the equivalent even algorithm since the write and decode read pointers are combined.

- The latency for the resulting algorithm (both even and odd) will be:

$$L = M \cdot \left(\frac{R \cdot (\eta + 1)}{k - 1} \right) \tag{3.15}$$

The one-pointer algorithm, the k-pointer even algorithm as well as all the hybrids of the two are now included in the definition of the R-pointer even family of algorithms. The k-pointer odd algorithm as well as the hybrids of the k-pointer odd and one-pointer algorithms are encapsulated in the definition of the R-pointer odd family of algorithms. An examination of the equations describing the number of memory blocks that will be required by each algorithm will reveal that the one-pointer algorithms is a special case of the R-pointer even family of algorithms, and does not form part of the R-pointer odd family of algorithms.

Comparison of the R-Pointer Algorithms

The first choice facing any designer will be which family of algorithms to use in the design. Once the family of algorithms is chosen the various criteria can be traded off versus one another to reach an optimum trade-off between speed, complexity and latency. The two families of algorithms are compared in [8] where the conclusion was reached that the even family of algorithms is preferable in terms of ease of implementation.

This was attributed to two properties of the odd family of algorithms:

1. As the decode read and the write pointer occupy the same block of memory, the write operation has to be stalled so that the bits can be read out of memory by the decode read operation before being overwritten.
2. All operations performed in a SMU designed using the odd family of algorithms would need to be able to operate in both from left to right and from right to left through each memory block (see Figure 3.3).

The effects these two properties have when trying to implement the algorithm are:

- Property one means that, at worst, the counter which controls the write operations would need to be able to count $2^{\nu} \cdot b + 1$ cycles for each column write; at best it means that the write counter would need to count two intervals for an increment of η in the read column counter.

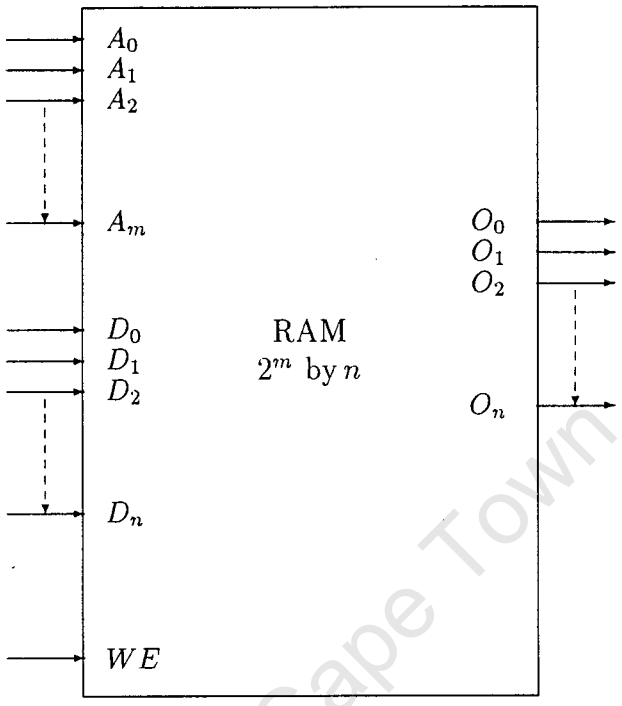


Figure 3.5: Diagram of a typical RAM block.

- Hardware can be limited by noting that although $2^{\nu} \cdot b$ bits are written to memory, only b bits need to be simultaneously read from memory.
- The low-high transition of the write enable destroys the data at location A in memory, so the same pulse can be used to drive the latches on the memory output to store the data being decoded.
- The latches themselves are already present in the SMU in the form of the register that stores the read pointers.

In applications where multiple write operations are performed before a whole column write is completed, the decode read can be made to wait until the write pointer is pointing to the row(s) of interest. The data would then only be latched out when the decode pointer matches the write pointer.

The decode read pointer may be pointing to any of the 2^{ν} states, so the data can be read out anywhere inside the $2^{nu-x} \cdot b$ clock cycles of the column write operation. The new data will, however, always be available at the end of the column write operation and can be latched into the bit order reversing circuit at that point. The bit order reversing circuit can then be used to make sure that the data appears in a synchronous fashion at the output of the decoder.

As a direct result of the integration discussed above it is no longer necessary to "stall" the write pointer. The counters that handle the column write and traceback read pointers can now be made the same size. Effectively this does away with the negative effects associated with property one discussed above.

The need for bi-directional counters will also fall away if the counters are no longer associated with either a read or write pointer, but are rather associated with either the forward or reverse direction. Each memory block then simply has to choose one of the counters, independently of whether a read or write operation is being performed inside the block. In the case of the even algorithm this choice would have been between the read and write pointers respectively, so there is no difference in complexity between the two algorithms.

The above only holds for an η of one. Should the design require an η of greater than one, an alternative would be to note that as long as the counters are some integer power of two, the reverse direction simply involves inverting the counter output. Having separate, bi-directional counters would be impossible, since there are times when two read pointers would be moving in the opposite direction. A one bit pointer can then be associated with each memory block, and would determine the direction of travel of the pointer through that block.

There are also other advantages to the R-pointer odd family of algorithms:

- The pointer that keeps track of the memory block being written to is also the pointer to the memory block in which the decode read operation is taking place. This will save some hardware when compared with the equivalent R-pointer even algorithm where the SMU has to keep track of the memory block in front of the write pointer.
- Output is always generated at the same rate as the column write operation. Even if the SMU has to perform 2^{ν} write operations per column write, output will be available at the end of the column write operation, so no burst elimination will ever be necessary.

The last point effectively means that any odd algorithm will always be able to use the simplest possible bit order reversing circuit. In applying the techniques discussed above to schedule the read and write operations to the single port RAM implementation mentioned above, a fourth option is presented for implementing the bit order reversing circuit that is simpler to implement in hardware than the three techniques proposed earlier.

There are many cases where the advantages listed above will make the odd family of algorithms more attractive to a SMU designer than their even algorithm equivalents. This certainly holds true for all cases where algorithms with an η of one are being considered, since the odd algorithm will always require one less memory block than its even counterpart, and memory block sizes for both algorithms will be the same.

When η is greater than one the odd algorithms may well loose this advantage. In these cases the memory block lengths required by each algorithm will differ, and the most efficient algorithm will be the one that yields memory block sizes that are closest to an integer power of two. Making block sizes an integer power of two will simplify the design of the address counters in the SMU.

The difficulty introduced by no longer having a single, optimum algorithm for all implementations is offset by the new, universal description of all the algorithms. The trade-offs involved in the design of any particular SMU can quickly be evaluated from the equations describing each of the two families of algorithms.

Chapter 4

FPGA Implementation

The project specifications were to build a low cost, high speed Viterbi decoder for the Ungerboek $(3, \frac{1}{2})$ code presented in section 2.2 capable of a throughput rate in excess of 16 Mbit/s. Using these requirements along with the algorithms presented in chapter 3, it is possible to draw up a rough outline of the decoder and choose a suitable platform for an implementation of the decoder.

4.1 Add-Compare-Select Implementation

There is a high degree of parallelism in a Viterbi decoder in that the operations performed on each state are the same, and must be performed once for each state with each received signal. A single ACS circuit performs all the calculation required for any single state in the code. A number of different implementations can be arrived at by changing both the number of ACS circuits in the decoder as well as the way they are arranged with respect to one another. The most straightforward implementation requires S ACS circuits operating in parallel, one for each state in the code. This implementation is used as the benchmark in assessing the speed and complexity of other implementations of the Viterbi Algorithm, and is referred to as the "state-parallel" implementation [19].

Since all S ACS circuits in the decoder are identical, it is also possible to "share" ACS circuits between states in the code [20], [19] . These implementations use between 1 and $S - 1$ ACS circuits operating in parallel with some sort of routing structure and path metric storage between the ACS circuits. As pointed out in [20], these implementations typically achieve a linear reduction in speed for a linear reduction in circuit complexity. Another advantage in these implementations is the path metrics do not need to be updated before the shared ACS circuit can move on to processing a new state. This allows for a certain amount of pipelining within the ACS circuit.

The opposite approach involves speeding up the ACS unit by increasing the parallelism in the code [14], [21]. In both cases the speed-up within the ACS unit is achieved by

concatenating two or more stages in the code trellis. Although this approach does not increase the number of states in the trellis, it doubles the number of connections between states in the trellis for every two stages concatenated together. As a result there is an exponential increase in circuit complexity for a linear increase in speed.

Although the above approaches offer some attractive features in the implementation of the Viterbi Algorithm, the state-parallel implementation was chosen for this project. The architectures that offered increased throughput were rejected as it was thought from previous work done by members the Digital Communications Research Group that the complexity of the state-parallel approach represented the limit in terms of feasibility with off-the-shelf programmable hardware [22].

The slower, low complexity implementations also have their drawbacks. Although there are algorithms available for designing the ACS units, [10], [19], algorithms for implementing the corresponding SMUs do not exist. The complexity of the SMUs could significantly offset the reductions in complexity of the ACS unit since the survivor vectors are now generated over several time units.

Furthermore an aim of the project was to form a benchmark for the research group for use in assessing the complexity involved in implementing any particular Viterbi decoder. In this manner the viability of coding and modulation schemes that rely on the presence of a Viterbi decoder in the receiver may be assessed. Since the state-parallel approach is used as the point of reference in assessing the complexity of Viterbi decoders, it would make sense to use this implementation as the benchmark for the research group.

The state-parallel implementation is a high-speed implementation, so it will also meet the speed requirements laid down in the project specifications. A state-parallel implementation of the Ungerboek $(3, \frac{2}{3})$ code will require eight ACS circuits operating in parallel, with each ACS circuit processing four path metrics.

4.1.1 Branch Metric Generation Unit

Without first determining a rough outline for the complexity of the circuit, it is impossible to determine the most suitable platform in which to implement the decoder. As was mentioned in section 3.1 the complexity of the ACS unit depends to a large extent on the complexity of the branch metrics used in the implementation. At the same time the quantization levels need to be set, since the design of the branch metrics and the quantizer are linked.

The quantizers need to digitize two analog signals. One corresponds to the "I" channel, and the other corresponds to the "Q" channel. Splitting the constellation along the I and Q axes, it is clear that the quantizer will require a minimum of four levels to be able to distinguish between the constellation points. Using four quantization levels for each signal dimension is therefore analogous to hard decision decoding the signal. Doubling the number of quantization intervals to eight will allow the receiver to distinguish two points between each of the adjacent signal constellation points. Doubling the number of

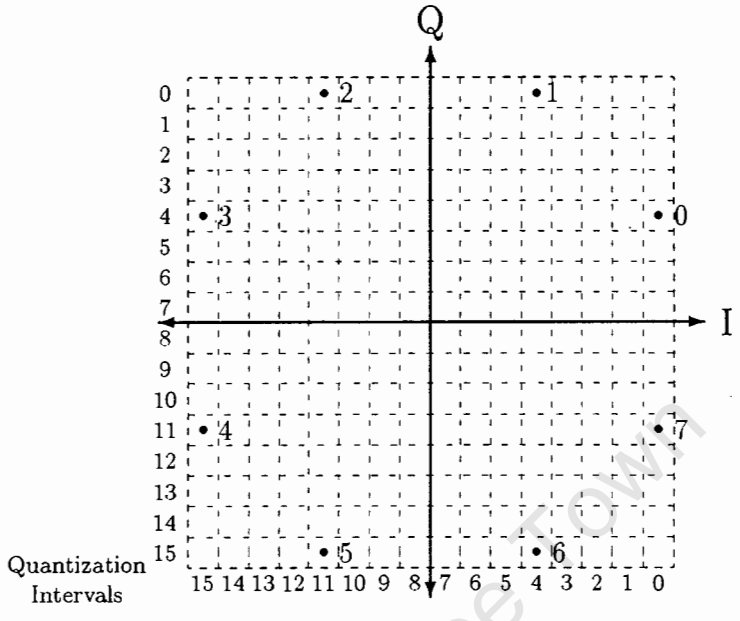


Figure 4.1: Illustration of I and Q quantization levels showing the relative placement of the signal constellation points.

quantization levels again will allow the receiver to distinguish at least four points between adjacent constellation points. It was felt that the latter was sufficient to constitute soft decision decoding without adding excessive complexity to the circuit. An illustration of the resulting "grid", showing the placement of the points from the 8–PSK signal set can be found in Figure 4.1.

The quantizer will pass eight bits on to the *branch metric generation unit* (BMGU). Four bits will correspond to the I channel and four bits to the Q channel. The simplest way of generating the branch metrics is to form a look up table addressed by these eight bits. For this particular implementation it was decided that the branch metric look up table would be implemented in PROMs externally to the rest of the decoder to allow for flexibility in the allocation of branch metrics in the final design. PROMs were chosen in preference to EPROMs to meet the speed requirements laid down in the project specifications.

The decoder requires eight branch metrics to be generated for each iteration of the ACS unit. If three bits are used per branch metric, then the BMGU would have to generate 24 bits per iteration of the ACS unit. Therefore, three 8-bit PROM's or EPROM's would be needed in the circuit. In addition, the platform that is chosen for the decoder will require at least 24 inputs for the branch metrics. Using equation (3.1) to get an estimate of the hardware complexity showed that with a maximum branch metric size of seven, six bits would be required to represent the path metrics. Since four path metrics are compared in each ACS unit, and there are eight ACS units, a total of 192 bits would

have to be processed in parallel. It was felt that this represented the limit in terms of hardware complexity with off–the–shelf hardware.

Although it is possible to get a smaller estimate of the dynamic range of the decoder using equation (3.7), and therefore reduce the size of the path metrics, the initial design was performed using the existing techniques for finding the size of the decoder's dynamic range. A second design using the smaller path metrics was generated at a later date. The aim was to prove that equation (3.7) does generate the correct dynamic range for the decoder, and to measure the total reduction in hardware resources used due to the resultant saving in the number of bits used to represent the path metrics.

4.1.2 The Implementation Platform

Having defined the size of the branch metrics a rough estimate of the complexity of the decoder can be drawn up. As stated in the previous section, six bit path metrics need to be used in the ACS unit if the branch metrics are represented using three bits. A rough gate count can now be drawn up:

1. There are eight states in the code, so the state-parallel implementation will requires eight ACS circuits operating in parallel. Each ACS circuit will need to be able to store one path metric. If a single latch is used for each bit, a total of 48 latches will be needed. Since each latch will need to be reset to a different value depending upon which state it is assigned to, 48 gates will be required to reset the latches to their proper values.

2. As each signal is received, branch metrics are generated by the PROM's and fed to the ACS unit. Each ACS circuit requires four three bit branch metrics. Sufficient routing will have to exist in order to route the branch metrics to their respective ACS circuit. Each branch metric is added to a path metric. Assuming that one gate is required per bit in the path metric to perform the addition, a total of 192 gates will be required.

3. The four path metrics need to be compared within each ACS circuit. In order to perform the four way comparison, six subtractions will have to be performed. Since the decoder need only know the sign of the answer in each subtraction, only the carry chains need to be implemented. Implementing the comparison process will require 288 gates assuming that one gate is required per bit to be processed.

4. Finally the smallest path metric must be selected as the survivor in each ACS circuit. Again it will be assumed that one gate is required per bit to perform this operation. The ACS unit as a whole will require another 288 gates to select the survivors.

All in all a rough estimate of the hardware requirements comes out as approximately 816 gates. Since this does not include any of the timing or reset circuitry, some leeway will

have to be included for additional hardware. The above estimate, however, will serve to choose the implementation platform.

It was decided that the low speed, sequential type implementation was unsuitable due to the speed constraints laid down in the project specification, so the use of microprocessors is virtually ruled out. Although one approach would involved the use of a single microprocessor for each ACS circuit, speed would still be compromised due to the microprocessors inability to process large numbers of bits in parallel. Furthermore, the complexity of such an implementation would be excessively high.

The operations performed in the decoders are extremely simple: addition for the add operation, subtraction for the compare operation and multiplexing for the select operation. Even RISC microprocessors offer a much wider range of operations than would be needed by the decoder. For high speed implementations the platform must support a high degree of parallelism while performing simple operations on all the bits being processed.

Based on the above criteria, programmable logic was chosen as the most suitable platform. The development tools available within the department where for the XILINX range of FPGA's, so these where selected to implement the ACS unit. The choice then lay in which family of devices would be most suited to the requirements of the ACS unit.

At the time the project was started only three device families were available: the XC2000 series, the XC3000 series and the XC4000 series. The XC2000 devices are the oldest of the XILINX devices, and do not offer a high gate count. As such they were rejected. The XC3000 devices offer a much larger gate count, but without added features such as RAM which are available in the XC4000 family of devices. It was felt that the gate count offered by the XC3000 devices was sufficiently high to enable an implementation of the ACS unit. Coupled with lower cost and higher speed, this feature led to them being chosen as the platform on which to implement the ACS unit.

The largest device available within the XC3000 family at the start of the project was the XC3195 with 484 Configurable Logic Blocks (CLB) [23]. This device, however, was new to the XILINX range and therefore expensive. The next size down was the XC3090/XC3190 with 320 CLBs. The XC3100 family of devices simply represents a faster version of the XC3000. Since the XC3100 devices are more expensive than the equivalent XC3000 devices, only the XC3090 was considered for this implementation along with two smaller devices, the XC3064 (224 CLBs) and the XC3042 (144 CLBs). Furthermore the XC3000 family of devices would give a good indication of the throughput rate that can be achieved with the cheapest devices on the market.

A closer investigation of the XILINX design tools available within the department showed them to be incomplete and outdated. An updated set of tools were ordered – the XILINX Pro-Flow series, version 5.1.1, comprising the Viewlogic schematic capture and design simulation packages, and the XILINX XACT design conversion and implementation tools. Since this introduced a delay to the start of the project a decision was taken to order the devices ahead of time to avoid additional delays once the designs were

completed. A small stock of devices were ordered, being made up of two XC3090 devices, three XC3064 devices and three XC3042 devices. These were deemed to be sufficient to implement the decoder.

4.2 Schematic Design: The Add-Compare-Select Unit

The ACS circuits were grouped by following the procedure suggested in [14] where the states are first grouped according to the branch metrics they require. This will simplify the layout of the schematic, and will facilitate splitting the ACS unit across multiple devices should it be required. States 000 to 011 in Figure 2.3 all use the same set of four branch metrics. Likewise states 100 to 111 all use the second set of four branch metrics. Ultimately this should also simplify the routing of the circuit on the FPGA device.

An interconnection graph is then plotted between the states, as illustrated in Figure 4.2(a). The two "states" in the interconnection graph are defined by the first bit of the current state, s_j. Having defined the interconnection graph, the ACS units are organised in a ring topology that keeps routing between the groups of states to a minimum as shown in Figure 4.2(b). This was done purely to ensure that should the ACS unit be divided across multiple devices, routing between the devices would also be kept to a minimum.

From there on the implementation is straightforward. Each ACS circuit is split up into three sections: the add section where the branch and path metrics are added, the compare section where the path metrics are compared, and the select section where the smallest path metric is chosen as the survivor. A general block diagram of the ACS circuit can be found in Figure 4.3.

The "add" section of each ACS unit will be made up of four addition blocks, one for each path entering that particular ACS unit. The most efficient implementation of the add blocks was using the macros supplied by XILINX. Two four bit adders were used to add the three bit branch metric to the six bit path metrics. The XILINX design tools automatically trim all unused gates from the design.

4.2.1 Comparing the Path Metrics

Once the branch and path metrics have been added together, the smallest path metric must be selected as the survivor. There are two possible implementations. The first involves comparing all the path metrics to one another, and then selecting the minimum path metric as the survivor, as is illustrated in Figure 4.4 and implemented in [14]. The second implementation requires the minimum path metric to be selected after each path metric comparison and then only compared with the next path metric, as illustrated in Figure 4.5.

The advantages and disadvantages of the two implementations are the same encountered

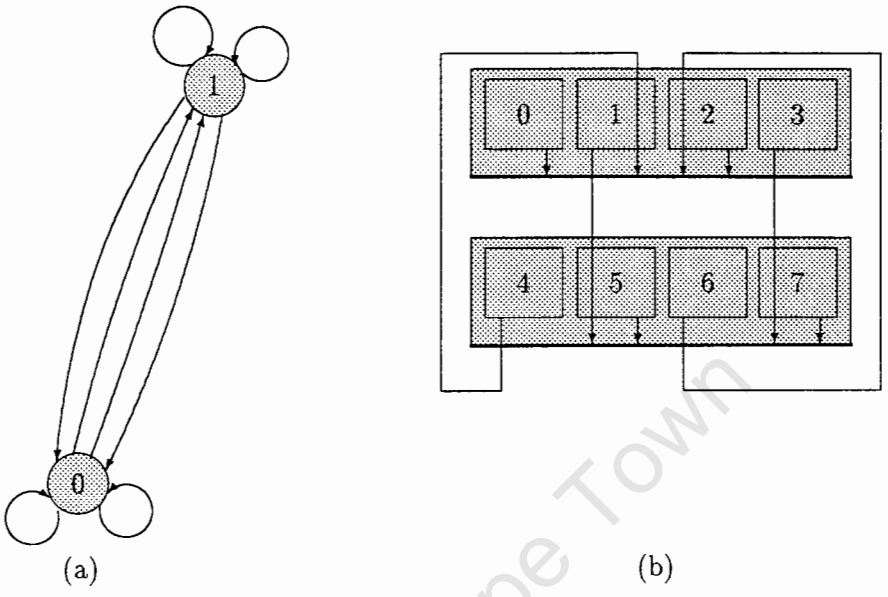


Figure 4.2: Routing topology for the schematic of the ACS unit: (a) Routing diagram and (b) layout of ACS units.

in designing the ACS unit as a whole. The first implementation has a high degree of parallelism at the price of an increase in the number of gates required for the implementation. In order to compare all four path metrics to one another, six subtractions need to be performed, followed by six 4-to-1 multiplexers to select the minimum path metric.

In the second implementation, however, only three subtractions are needed. In addition the selection of the minimum path metric is simplified since each stage only requires six 2-to-1 multiplexers. A loss in operating speed is associated with the second implementation due to the fact that the three comparisons need to be performed in series, as opposed to the six comparisons in the first implementation which are all performed in parallel.

Although, in a typical high-speed ASIC implementation, the fully parallel approach will guarantee the highest operating speed, the same is not always true of FPGA's. The added strain placed on the routing resources by the increased parallelism may well yield final operating speeds lower than the equivalent serial implementation. In light of the above both implementations of the compare section where investigated.

Fully Parallel Path Metric Comparison

In order to compare the four path metrics, six subtractions must be performed. The compare section is illustrated in Figure 4.4. As illustrated, the most significant bit of

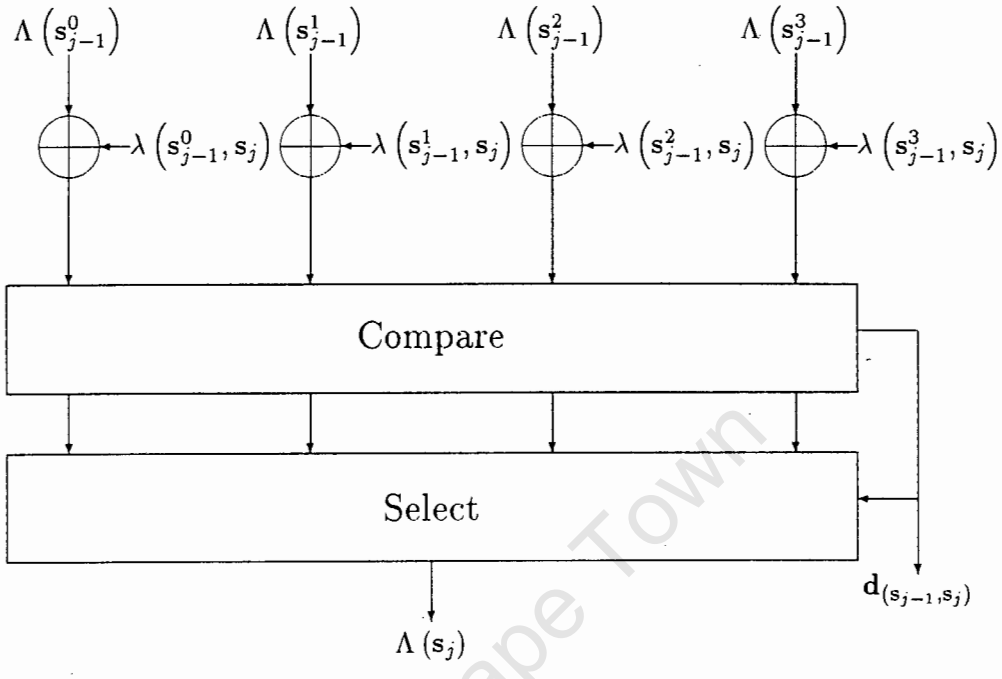


Figure 4.3: General block diagram of an ACS circuit.

MSB A	MSB B	A - B	Min.
0	0	0	A
0	0	1	B
0	1	0	B
0	1	1	A
1	0	0	B
1	0	1	A
1	1	0	A
1	1	1	B

Note: A - B denotes the comparison of the 5 LSBs of A and B. A "1" denotes B less than A

Table 4.1: Truth table for path metric comparisons.

the path metric is processed separately from the other five bits. The most significant bit is treated as the "overflow" indicator. As the path metrics are compared mod 2Δ, and the dynamic range between any two path metrics cannot exceed Δ, the result of the comparison must be as given in table 4.1.

As indicated in Figure 4.4, only the carry section of the adder need be implemented, since all that is of interest is whether the result is a positive or negative number (i.e.

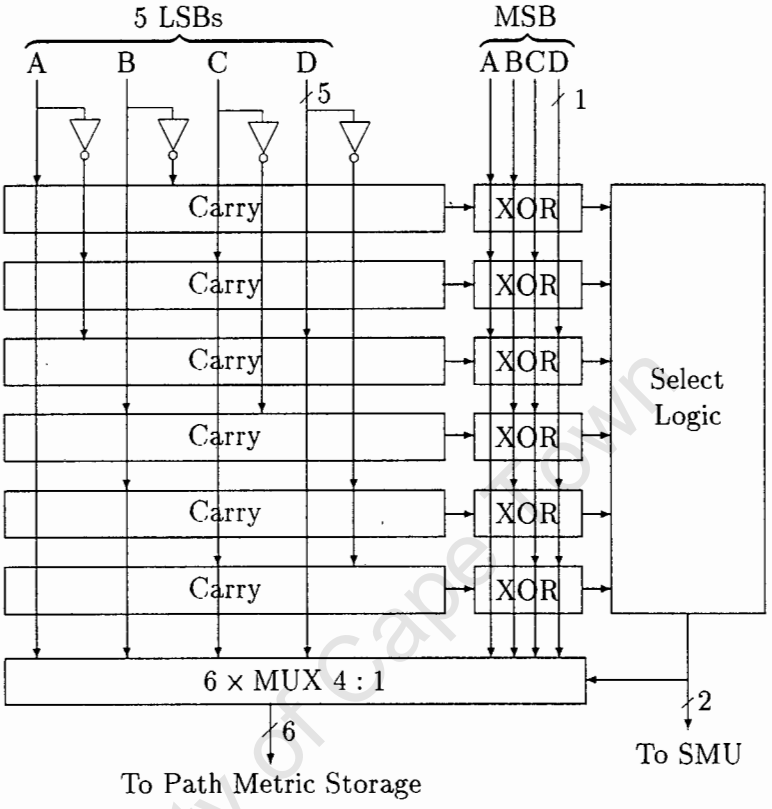


Figure 4.4: The parallel implementation of the path metric comparison section of the ACS unit.

whether A>B or A<B). The results of the subtraction are exclusive-ored with the most significant bits of both path metrics to form the final result. The result of each of the six comparisons is passed on to the select logic which chooses the minimum path metric entering that state. The truth tables for the select logic are included in Appendix B.

The two bits produced by the select logic are used to select the minimum path metric using six 4:1 multiplexers and are simultaneously passed on the SMU as pointers to the survivors entering the state associated with that particular ACS circuit.

Serial Path Metric Comparisons

Unlike the parallel implementation, the serial implementation selects the minimum path metric after each comparison. The resultant minimum path metrics are then compared once more to find the overall minimum path metric. This allows the number of sub-

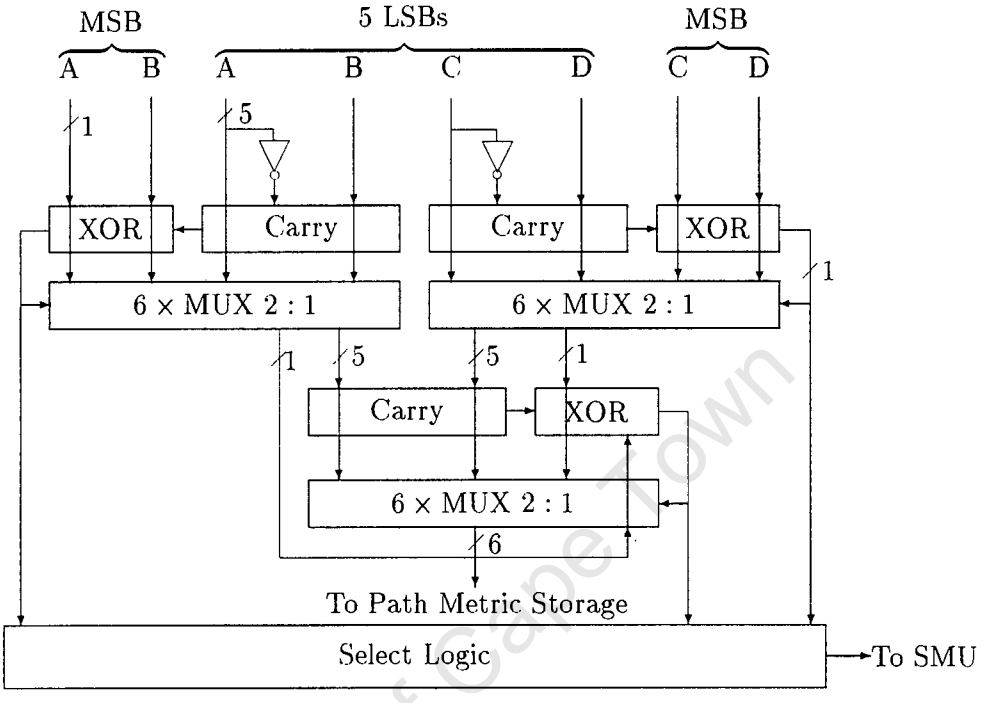


Figure 4.5: The serial implementation of the path metric comparison section of the ACS unit.

tractions to be reduced from six to three, with a corresponding saving in hardware. As previously mentioned there is a cost in terms of the added processing delay, since the final comparison is delayed until all the other comparisons have been performed. The result is that the select unit is effectively merged with the compare unit in the serial implementation of the path metric compare unit.

The serial implementation is illustrated in Figure 4.5. In the serial implementation the select logic is no longer responsible for selecting the minimum path metric. It simply processes the decisions made by each compare unit into a format that can be handled by the SMU. The truth tables for the select logic can be found in Appendix B.

There is no difference in the way the path metrics are compared in the serial and the parallel implementation of the compare unit.

4.2.2 Path Metric Storage

Once the minimum path metrics have been selected, they need to be stored and re-routed to the ACS circuits to be processed in the next time unit. Using XC3000 devices

means that there are only flip-flops available for storage purposes. The flip-flops in these devices are available with an asynchronous reset, or a synchronous preset or reset. The synchronous preset and reset is implemented by adding gates onto the input of the flip-flop.

Unfortunately only the path metric for state zero is reset to zero. All other path metrics need to be preset to their largest value. In terms of the current implementation this will require the use of flip-flops with synchronous presets. Since this adds to the complexity of the circuitry, the number of synchronous presets needs to be kept to a minimum.

Only the storage unit for the path metric for state zero is implemented using flip-flops with asynchronous resets. In all the other states only the two most significant bits of each path metric are reset to zero and the next most significant bit is preset to one. All the other bits in the path metrics are allowed to remain in their original state. In other words only two flip-flops with an asynchronous reset and one flip-flop with a synchronous preset are used along with three flip-flops that have no reset state, to store all path metrics other than the path metric for state zero. This approach reduces the circuit complexity and frees up routing resources while guaranteeing that the path metrics are initialised to a value larger than the metric assigned to state zero. It also ensures that the maximum allowed dynamic range will not be exceeded within the first cycle of the ACS unit after a reset operation.

4.2.3 Clocking and Reset Circuitry

FPGAs work best when implementing synchronous circuits [23]. As such all inputs to the FPGA from the external circuit are latched. In addition all outputs from the FPGA are latched as well. In the case of the inputs this serves to eliminate glitches in the reset signal due to noise on the circuit boards, and ensures that the inputs do not change during any single cycle of the ACS unit.

There is some additional circuitry in the ACS unit for clocking and reset purposes. A global clock is supplied on the board that must drive both the ACS and the SMU circuits. Since the SMU needs several clock pulses per cycle of the ACS unit (the exact number depends on the implementation, see section 4.4.1), the clock is divided down inside the ACS circuits using a synchronous binary counter.

As previously mentioned, the reset pulse is latched into the ACS circuit as are all inputs to – and outputs from – the circuit. It therefore takes several cycles once a reset has been initiated before any valid data is available on the output pins, so the ACS circuit must pass a delayed reset pulse on to the SMU. Furthermore, as mentioned in section 4.2.2, some presets in the circuit are synchronous. To accommodate these presets the reset pulse needs to be held high for at least one clock cycle. A timing diagram showing the various reset pulses is included in Appendix B.

4.2.4 Comparison of Implementation Options

Having completed the schematics of the ACS circuits, they were debugged using the Viewlogic functional simulation tool. Test vectors for debugging purposes where generated using a PASCAL program which mimicked the operation of the circuit. The same test vectors where used for both the functional and the timing simulations.

The placing and routing of the final designs was left to the automatic tools supplied by XILINX. At this stage it was found that the initial design was too complex to fit onto a single XC3000 device. As such the ACS unit was split across two FPGA devices (the XC3064APC84-7 was sufficiently large), each device handling four ACS circuits. The FPGA device housing the ACS circuits for states 000 through 011 was made responsible for generating the reset pulse for the SMU.

Once the ACS circuits were placed and routed, the Viewlogic timing simulation tools were used to verify that the design was capable of meeting the timing constraints laid down in the project specifications. In the case of the ACS circuits this was achieved relatively easily by placing individual timing constraints on various paths through the circuit in the schematics.

Once it was verified that the circuit was capable of meeting the time constraints, a program was written to evaluate equation (3.7) for the code being decoded given the branch metric lookup table being used. It was found that the initial estimate of the decoder's dynamic range was, as expected, too large. The new dynamic range required only five bits to represent the path metrics in the decoder as opposed to six in the original design. The result was a 10% saving in resources on the FPGA device, and the new design met the timing constraints with greater ease. The program listing used to generate the new dynamic range, along with the output from the XILINX design tools showing the resource usage of each of the designs, may be found in Appendix B.

In contrast to the designs mentioned above, the implementation using serial compare sections in the ACS circuits along with five bit representations of the path metrics achieved a reduction of 32% in the use of FPGA resources when compared to the original design. When compared with the five bit, parallel compare implementation, a saving of 22% is achieved. The rooting tools did, however, struggle to route the series implementation within the required timing constraints. The time delay analyzer showed the additional delay to be in the region of three nanoseconds, which was sufficient to push the timing constraints close to the imposed limits. That the increase is so small is thought to be due to the lower demand placed on the routing resources of the FPGA device due to the reduced use of the available logic resources.

4.3 Survivor Memory Unit Implementation

In the case of the SMU, the design choices are more limited. Following the arguments laid out in chapter 3, the SMU has to use the traceback algorithm in cases where there is

a limit on the available hardware resources. Although there are a number of algorithms available other than those outlined in section 3.2.2, these algorithms seem to offer the best compromise in terms of hardware complexity and latency of the SMU. Previous attempts by members of the group to implement other algorithms, [22], along with a successful implementation of an algorithm very similar to those outlined in section 3.2.2, [14], back up this claim.

Having chosen to use one of the R-pointer algorithms, there are still many design choices to be made. Most of these relate to the complexity/latency trade-off in the SMU. As was done with the ACS unit, multiple designs were envisaged for the SMU. The aim was to get a clear idea of what is possible using off-the-shelf hardware.

4.3.1 The Implementation Platform

Initially an attempt was made to design the SMU using a small amount of hardware and several two-port RAM chips. The complexity of the control circuitry required in the SMU quickly ruled out this approach. Given that FPGAs were already being used to implement the ACS unit, an attempt was made at migrating the control circuitry onto one of the XC3000 series FPGAs already available, and to then use the FPGA to drive external two-port RAM chips. An initial analysis showed that this approach, although feasible, proved to be too slow to meet the project specifications due to the amount of time required to read data into – and write data from – the FPGA devices. With the control circuitry and RAM blocks implemented on separate devices, a large number of such operations would need to be performed.

The next logical step was to migrate the whole design onto a single FPGA device. The SMU requires a number of RAM blocks, so the XC3000 series devices bought to implement the ACS unit cannot be used. At the time the project was started, the only family of XILINX devices that had built-in RAM modules was the XC4000 series of devices. As such a design was developed for a XC4000 device.

There are a number of general features that apply to all the designs for the SMU. A rate $\frac{2}{3}$ code requires a truncation depth of at least ten times the constraint length. In this particular case the constraint length is $\nu = 3$, so a truncation depth of $M \geq 30$ will be required. The truncation depth can be altered as indicated by the inequality to adjust the size of the RAM blocks that will need to be used in the SMU.

Another central issue in the design of the SMU is the number of bits that will be written to memory simultaneously. The project specifications require a high speed decoder, so the approach used in [14] was adopted here: all the bits making up a single survivor vector are written into memory simultaneously. Slowing down the write operation could make the SMU the bottleneck in the decoder.

Given that a column write can take place within a single clock cycle, it will be difficult to speed up the read pointers with respect to the write pointer. All designs are therefore further limited to a TRR of one.

4.4 Schematic Design: The Add-Compare-Select Unit

4.4.1 The 8-Pointer Even Implementation

The first proposed design for the SMU, as with the ACS unit, was chosen to be the most complex. Once the initial design is complete, a device can be chosen that will be big enough to hold both the present design and all future designs. In the case of the SMU the complexity will increase with a decrease in latency, so the most complex design will be the one with the lowest latency.

The RAM block size can be derived from the equations presented in section 3.2.2. Making the memory block sizes smaller decreases the latency of the SMU, but leads to a corresponding increase in the complexity of the circuit. Low-complexity implementations also limit the block sizes to integer powers of two in order to simplify the design of address counters. It was felt that the smallest practical RAM block size would be $T = 4$, which requires the truncation depth to be decreased to 28. The new truncation depth should not lead to any degradation in the performance of the decoder since it is so close to the suggested truncation depth of $M = 30$.

Part of the project also involved comparing the complexities of the various R-pointer algorithms. From the arguments put forward in section 3.2.2 it was assumed that the R-pointer even algorithm would have the higher complexity, particularly in implementations where the TRR is one. The initial design was therefore based on this algorithm. Following the remainder of the design equations, it was calculated that $B_{even} = 16$ RAM blocks would be needed, and that the overall latency of the design would be $L = 64$.

Unfortunately the RAM blocks in the XC4000 series of devices come in blocks of 16×1. The use of block sizes of four columns would require a total of 256 RAM blocks, which can only be handled by the larger devices in the XC4000 family. To reduce the number of RAM blocks, pairs of blocks will be fused into a single RAM block 8 columns long. This halves the number of RAM blocks needed to implement the SMU, but it does mean that read and write operations will have to be interleaved within the RAM blocks. The survivor memory will therefore be made up of eight 8×16 RAM blocks.

The design progresses by splitting the implementation into three main sections: the logic required to implement the RAM blocks, the logic required to do the bit order reversal, and the general control logic required to drive the SMU.

The RAM Block Logic

Since the RAM blocks form the core of the SMU, this is the logical point from which to start any design. The algorithm presents these blocks as consisting of a simple RAM module, but this is not quite the case. Each RAM block needs to be able to select one of two possible addresses – either the one corresponding to the write pointer, or the one corresponding to the read pointer. The read and write operations also need to be

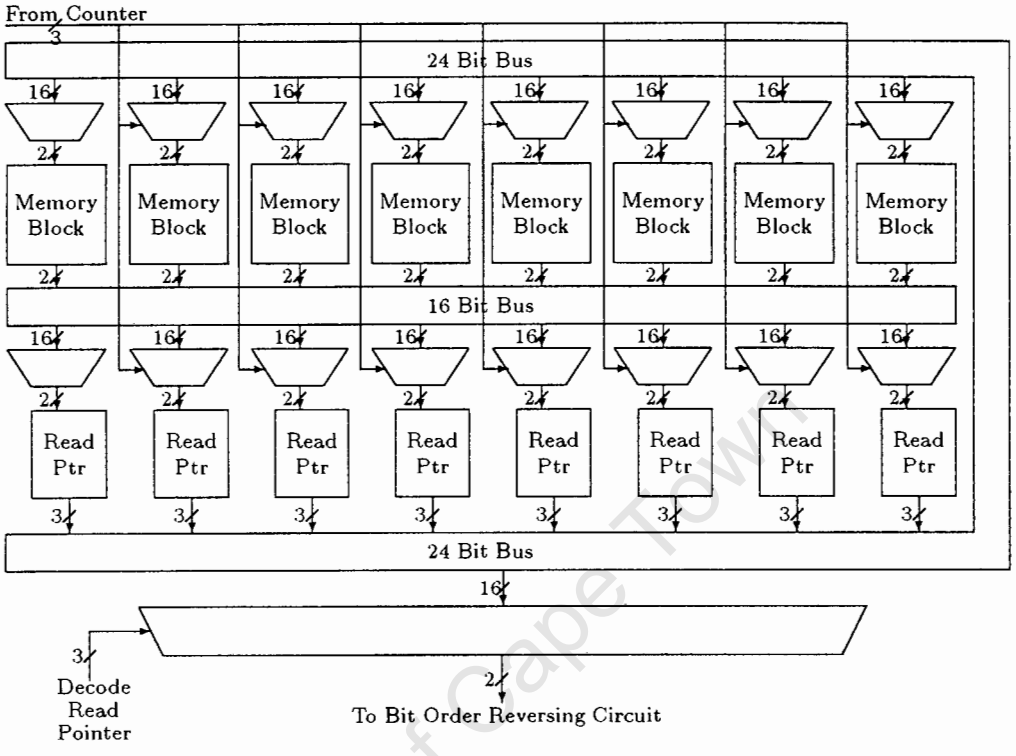


Figure 4.6: The separate memory block and read pointer implementation.

controlled within each RAM block – in this case 16 bits are written into each memory location, but only 2 bits need to be read out.

The pointers which manage the read operation can be handled in several different ways. Either each read pointer needs to be completely separate from the RAM blocks, or each RAM block needs to be assigned a dedicated read pointer. In the former implementation each pointer would have to be coupled with a multiplexer to allow it to choose from which block it will receive data as well as an additional multiplexer having to be built into each memory block so that it can choose which read pointer will be operating on it.

In a conventional implementation, the second option will effectively double the number of read pointers in the SMU, some of which will stand idle (or, alternately, be redundant) during a complete block write operation. A flag will be required to signal the end of a block read, at which point the current pointer will pass its information on to the next memory block in the chain. In this way the read fronts will propagate through the entire memory. As such each memory block needs a multiplexer to select whether data is read from its own pointer, or from the read pointer attached to the previous memory block in the chain. The flag described above will be used to drive this multiplexer. The two options are illustrated in Figure 4.6 and Figure 4.7 respectively.

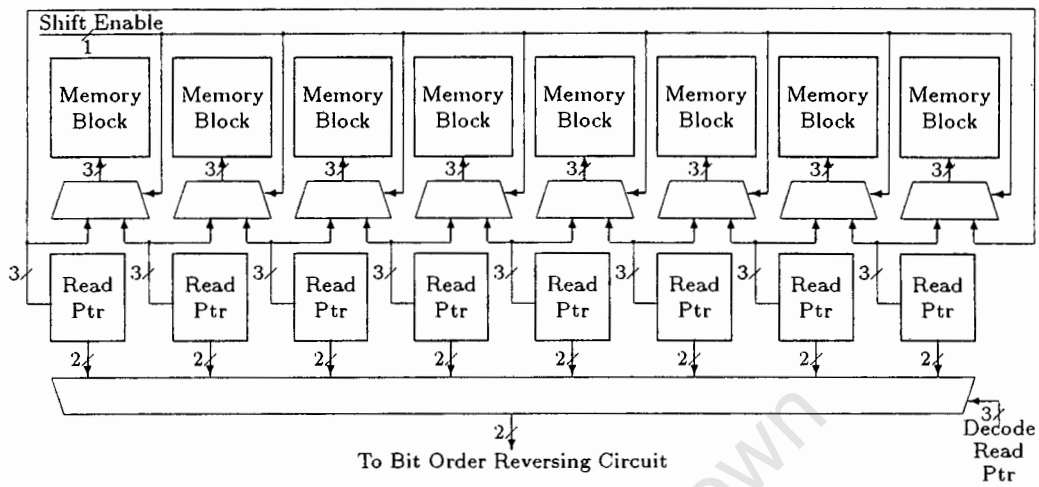


Figure 4.7: The joined read block and memory pointer implementation.

In this particular implementation the RAM blocks have been grouped in pairs. As a result both the implementation options discussed above will use the same number of read pointers (as will be seen in Figure 4.6 and Figure 4.7). From the illustrations it is clear that the option presented in Figure 4.7 is the one requiring the lowest complexity, and was chosen for use in this design.

The general block diagram of a combined read pointer and RAM block is illustrated in Figure 4.8. Due to the interleaving of the read and write operations within each memory block, the memory address will change twice within any given ACS cycle. This means that data is only read from each block over half a cycle of the ACS unit. Sixteen bits of data will be read simultaneously from memory during any read operation, so the data needs to be passed through a multiplexer before being stored in the latches housing the read pointer. It is the read pointer that, in turn, drives the select inputs on the multiplexer, thereby creating a loop. In order for the data to have enough time to propagate through the multiplexer it must be present at the output of the RAM over a full cycle of the ACS unit, so it needs to be latched at the RAM output. As there is a latch available within the FPGA on the output of each memory block, this places no additional demands on the FPGA's resources.

A timing diagram showing the timing of the various signals within each memory block may be found in Appendix C.

The Bit Order Reversing Circuit

Again, as was done with the rest of the circuit, the implementation thought to yield the maximum complexity was used in the initial design. In the case of the bit order

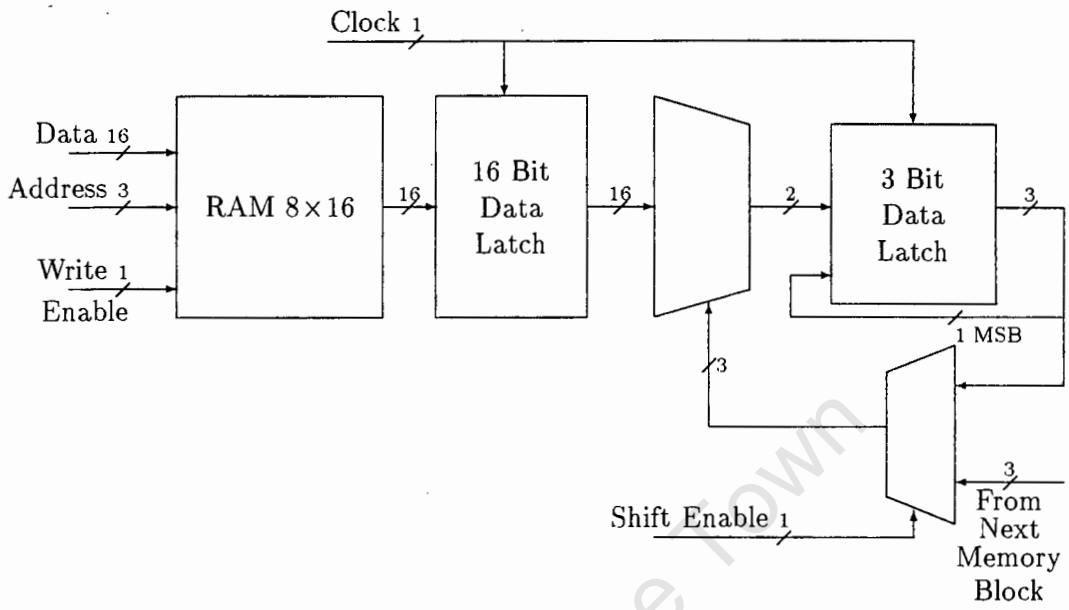


Figure 4.8: Block diagram of the memory blocks in the 8-pointer even implementation.

reversing circuit this meant using the two-stack structure proposed in [8]. As such four bi-directional shift registers were set up, each shift register being four bits long. The shift registers work in pairs to reverse the two bits that are read out simultaneously from RAM for each decode read operation.

The bi-directional shift registers were implemented using a modified version of the XILINX macro of the 74HC194 CMOS device. The 74HC194 is a 4-bit bi-directional shift register with parallel data in, parallel data out, and has an active-Low asynchronous clear. The original macro was trimmed down by hand: the shift-right input was removed, the ability to load data in parallel was removed, and the active-Low clear was changed to an active-High clear. This was done primarily because the XILINX layout tools had difficulty in trying to remove the inverter added in the macro to force the active-Low clear.

As explained in [8], as data is shifted into one pair of shift registers, it is shifted out of the other pair of registers. The single bit that determines the direction in which the registers shift data is taken from the address counter for the RAM. A multiplexer is added on the output of the shift registers which selects data from one of the pairs of registers as output for the decoder which is driven by the same bit that determines the direction of the shift registers.

The only additional circuitry required by the bit order reversing circuitry is a multiplexer to select the data from the memory block performing the decode read operation. As with the multiplexers in the memory block themselves, there is a latch on the output of the multiplexer in the bit order reversing circuit. The latch helps alleviate timing delays from

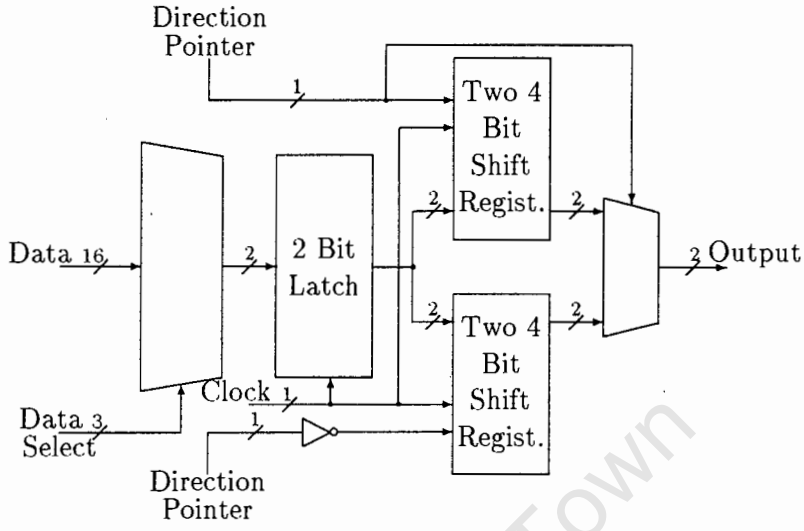


Figure 4.9: Block diagram of the twin stack bit order reversing circuit.

the memory blocks to the bit order reversing circuits. The bit order reversing circuit is illustrated in block diagram form in Figure 4.9.

General Control Circuitry

In addition to the memory blocks and the bit order reversing circuit, some control circuitry is required to generate all the control signals that ensure all the blocks interact in a synchronous fashion. The basis of the control circuitry is an 8 bit binary counter which generates all the clock pulses as well as the read and write pointers for the SMU.

As previously mentioned, read and write operations need to be interleaved. This means that the SMU will have to go through at least two full clock cycles for each cycle of the ACS unit. It was found, however, that the process of writing data into memory requires at least two clock cycles:

- At the beginning of the first clock cycle the address and data change simultaneously. Allowances must be made for both propagation delays and set-up time requirements before the write enable input can be taken high.
- At the beginning of the second clock cycle the write enable pulse is taken high, and is taken low again one half clock cycle later. This was done to allow for the required hold times before either the address or the data is changed.

A timing diagram showing the scheduling of the various waveforms can be found in Appendix C.

The counter is organised as follows:

- The two least significant bits (bits 0 and 1) of the counter are used for clocking purposes.
- The next three bits (bits 2 to 4) are used as the address bits for the read and write operations.
- The three most significant bits (bits 5 to 7) are used to generate the pointers to both the block in which the write pointer is operating, as well as the block in which the decode read operation is taking place.

Due to the timing constraints on the circuit, it was found that the propagation delay of the least significant bit through the carry chain to the logic generating the most significant bits was too long. As such a latch was placed between the carry out of the 5 least significant bits and the carry in to the three most significant bits. There will therefore be a single clock cycle delay in generating the three most significant bits. This delay is balanced by the corresponding delay in data being read from memory.

As previously explained, the read and write operations need to be interleaved. To reduce circuit complexity, the read and write addresses are generated at the counter and are fed to all the memory blocks in the SMU. The read address is generated by simply inverting the write address. As such the write address is passed through three XNOR gates along with bit 1 from the counter before going on to the rest of the memory. In this way the address generated by the counter will be inverted during the first two clock cycles in order to perform the traceback and decode read operations, and will be non-inverted for the second two clock cycles to perform the write operation.

All additional control circuitry is used to generate the write enable pulse, the shift enable pulse, the waveform to control the direction of the bit order reversing circuit and the address to properly select the data from the decode read operation. The write enable pulse is created by gating the clock along with the two least significant bits of the counter. Although this practice is frowned upon in the XILINX design guides (see [24] and [23]) due to the resulting unpredictable waveform delays, it was the only way to generate the write enable pulse without extending the number of clock cycles over which the write operation takes place. In this case all simulations indicated that the design was working properly.

Once the write enable pulse has been generated it needs to be gated before being passed on to the memory blocks to ensure that the write operation takes place only in the memory block indicated by the write-pointer. This was realised using a decoder to decode the write-pointer, and using the write enable pulse to drive the output enable on the decoder.

The decode read pointer can be generated from the write pointer by noting that the memory block in which the decode read pointer will operate will always be one ahead of the write pointer. The read operation in the memory block, however, is stalled when compared to the write operation to accommodate all the propagation delays in the circuit.

In order to ensure that the decode read pointer points to the correct block, some logic circuitry was added to the SMU. Truth tables for this circuit can be found in Appendix C.

The remainder of the control circuitry – the shift enable and the waveform to control the direction of the bit order reversing circuitry – is relatively straightforward. Bit 4 of the counter changes sign whenever a block read is completed, so it can be used to determine the direction in which the two stack structure should operate. It does, however, need to be delayed to take into account the pipelining in the decode read operation. The shift enable pulse is generated from bits 1 to 4 of the counter. The equation for the shift enable can be found in Appendix C.

Resetting the SMU simply involves clearing the counter to the all zero state. As with the ACS unit, the reset pulse is latched into the FPGA device to ensure that no asynchronous resets occur due to noise on the board. Some additional circuitry was added in the bit order reversing circuit to ensure that the output remained low until valid data was available at the output of the decoder. A timing diagram showing the various reset pulses is included at the end of Appendix C.

4.4.2 The 3-Pointer Odd Implementation

Unlike the 8-pointer even implementation, the 3-pointer odd implementation was designed to minimise the use of FPGA resources as much as possible. The design was started by noting that the FPGA's RAM modules came in blocks of 16 columns each. Using $T = 16$ as the memory block size in equation (3.12) along with a truncation depth of $M = 32$ as discussed above, shows the required number of read pointer to be three. From equation (3.14), five memory blocks will be needed for the implementation. When compared with the eight blocks required by the previous implementation, this should represent a significant saving in hardware. The hardware saving will, however, be offset by an increase in the latency of the implementation to $L = 96$.

The new design was built up using the old one as a skeleton, so many features are common to both implementations. As before the design was split up into three major sections: the RAM blocks, the bit order reversing circuitry and the general control circuitry. Each section of the design will be discussed separately.

The RAM Block Logic

Fundamentally the operation of the RAM block have not changed. As was done with the 8-pointer even implementation, each memory block will be coupled to its own read pointer. During any complete block read, a number of these read pointers will be redundant. The alternative, which is to keep the read pointers separate from the memory blocks would require greater circuit complexity (see section 4.4.1).

The number of columns in each RAM block has, however, been increased from eight to sixteen, so an extra bit will be required to address the RAM module. Furthermore, an

examination of Figure 3.3 will show that successive write fronts through any particular block will take place in the opposite direction. This also applies to read fronts, so at any one time there may be read pointers operating in separate regions of the SMU memory that will be moving in different directions to one another.

The net effect of the above is that each memory block needs to be assigned not only either a read or a write operation, but also a direction – either left or right. As such a single bit is passed to each memory block to determine the direction of the operation, be it read or write, that takes place inside that memory block. The value of this bit will determine whether or not the address being fed to the memory block gets inverted.

The operation of the rest of the memory block is identical to the implementation discussed in section 4.4.1. The structure of the memory blocks used in the 3-pointer odd implementation is illustrated, in block diagram form, in Figure 4.10. As with the 8-pointer even method, a diagram showing the scheduling of the signals entering each memory block may be found in Appendix C.

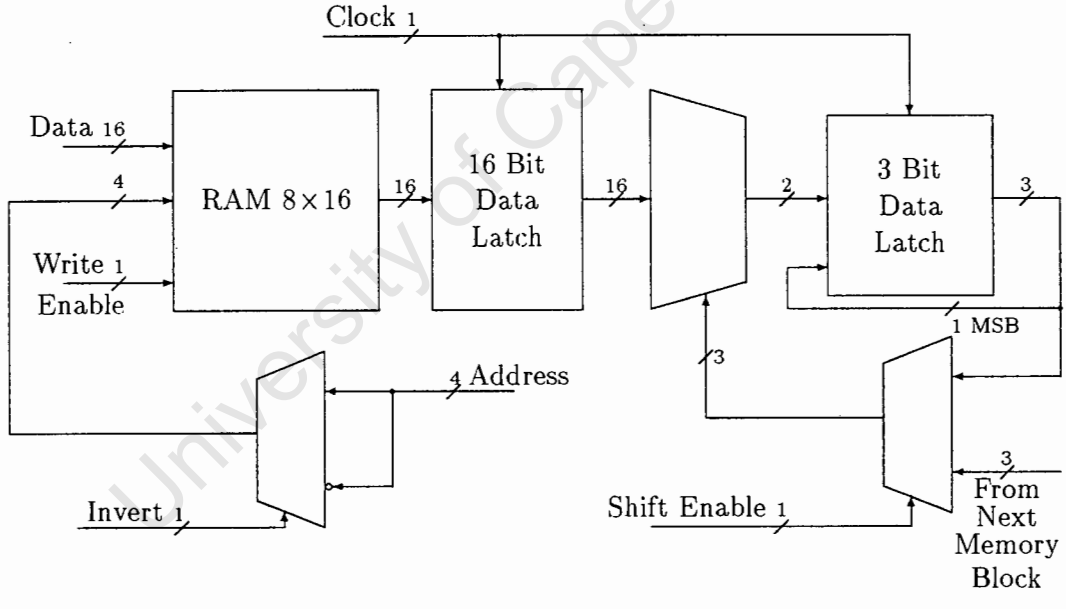


Figure 4.10: A block diagram of the structure of the memory blocks used in the 3-pointer odd implementation.

The Bit Order Reversing Circuit

Here again the circuit was re-designed to minimise the FPGA resources used. The hardware in the bit order reversing circuit can be reduced by noting that the same techniques used to schedule read and write operations in the block performing the decode read operation can be used in the bit order reversing circuit (see section 3.2.2).

In this case the bit order reversing circuit will need to be made up of a single 16×2 RAM block. As new bits are written in to the memory the old bits are read out simultaneously in reverse order. The read/write front therefore needs to continuously change direction at the end of every block read/write. Although in the case of the traceback read operation this was done by inverting the counter output, the same approach cannot be used in the bit order reversing circuit due to the pipelining in the decode read process. Some additional logic is required to generate the correct address as well as a single toggle flip-flop to control the direction of the read/write front.

There are two possible methods of generating the required address. Either a second bi-directional counter can be used, or the address can be generated from the counter used to house the read/write pointers for the rest of the SMU. The basic circuit in each case will be of roughly the same complexity – all logic circuits are implemented on the FPGA in blocks of functions of up to five variables. In the first case, however, some additional circuitry will be required to properly synchronise the counter with the rest of the circuit. As a result it was felt that the second option was the more attractive in terms of hardware complexity. The equations giving the mapping of the address bits may be found in Appendix C. The output of the mapping circuitry was latched so that the address remains stable over a complete read/write cycle. As the FPGA has a dedicated latch for each logic block, the additional latches do not increase the number of FPGA resources used.

The decode read operation produces data for the bit order reversing circuit just before the write enable goes high. As a result a separate write enable pulse needs to be generated in order to accommodate the required data set up and hold times in the RAM making up the bit order reversing circuit. This will require some additional logic, and, again, the relevant equation may be found in Appendix C along with a timing diagram showing the relative timing of the various signals within the bit order reversing circuit.

The rest of the circuit works in the same way as the bit order reversing circuit used in the 8-pointer implementation discussed in section 4.4.1. It must be noted, however, that the multiplexers used to choose the data being written into the reversing circuit will be much simpler since the number of memory blocks has been reduced from eight to five. A block diagram of the bit order reversing circuit used in the 3-pointer odd implementation can be found in Figure 4.11.

General Control Circuitry

The control circuitry in the 3-pointer odd implementation is similar in nature to that employed in the 8-pointer even implementation. The circuit as a whole is again controlled by an eight bit binary counter. In the case of the odd algorithm, however, the read and write operations no longer need to be interleaved. The two clock cycle requirement for the write operation remains, so two system clock cycles will correspond to a single ACS unit cycle in this case, as opposed to the four clock cycles under the 8-pointer even implementation. The net result is an easing in the time constraints on the propagation

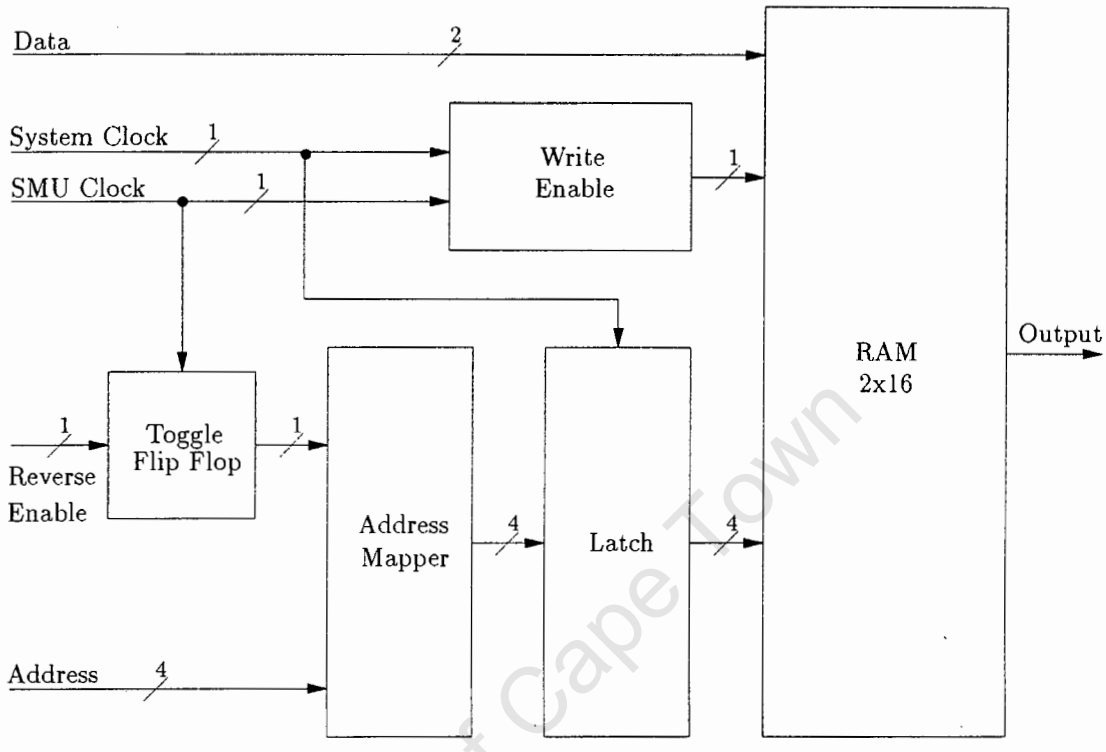


Figure 4.11: Block diagram of the bit order reversing circuit used in the 3-pointer odd implementation of the SMU.

delays within the counter, so the pipelining used in the 8-pointer even implementation is no longer required.

The output bits from the counter are also assigned different tasks:

- Bit 0 (the least significant bit) remains a clock signal.
- Bit 1 is incorporated with bits 2 to 4 to form the longer address vector required by the memory blocks.
- Bits 5 to 7 remain as the write pointer.

In this case there are five, not eight, blocks that need to be counted. As such the finite state machine which generates bits 5 though 7 needs to be re-designed to count modulo 4. The equations for the required logic may be found in Appendix C. Although a typical implementation of such a finite state machine would have a higher gate count than the equivalent sized binary counter, no additional resources will be used on the FPGA since all logic is implemented on blocks that can implement functions of up to 5 variables.

All control pulses in the SMU are derived from the counter bits. The circuit used to generate the write enable pulse was substantially modified from the one used in the 8-pointer even method to make it less sensitive to delays between the various clock pulses and to eliminate all glitches in the write enable waveform. As the path delays for the write enable pulse were always found to be the limiting factor in any attempt to increase the performance of the design, this approach should yield a far more robust implementation. Details of both the circuit used and the timing diagrams may be found in Appendix C.

The write enable pulse is still gated through a decoder before being passed on to each memory block. The decoder takes the write pointer as its input, and uses the write enable pulse as its "enable" input. In this way only the write enable input of the memory block corresponding to the contents of the write pointer is taken high.

The circuitry generating the shift enable pulse remains largely unchanged. In this particular implementation, however, the shift enable pulse is used as a clock enable on the latches that store the memory block operation direction vector as well as being used to drive the multiplexers in each memory block that determine which read pointer is used to generate the current state address. Since these two operations do not take place in the same clock cycle, the shift enable pulse has to be passed through two latches which act as a delay before it is used to drive the multiplexers in the memory blocks. As before, the equation describing the circuitry used to generate the shift enable pulse is given in Appendix C.

The bit-order reversing circuit also needs a reverse enable signal to change the direction of the address counter. It has to be a separate signal from the shift enable signal to account for the pipelining in the traceback read operation. The reverse enable pulse is used as a clock enable on the toggle flip-flop in the bit order reversing circuit. The equation describing the circuitry which generates the reverse enable signal may be found in Appendix C.

Unlike the 8-pointer even implementation, separate decode read and write pointer are no longer required. The write pointer does, however, need to be delayed by one clock cycle before being passed on to the bit order reversing circuit in order to take account of the pipelining in the decode read operation. The decode read pointer is still used to drive the multiplexer in the bit order reversing circuit which selects the output from one of the five memory blocks.

The only additional circuitry is a circuit that holds the decoder output low until valid data is available. This particular circuit had to be modified from the one used in the 8-pointer method to take into account the larger latency of the memory, and the different counter used in the 3-pointer odd implementation.

Reset Circuitry

In both implementations the reset circuitry is extremely simple. Only the global counters need to be reset. The initial state of the registers holding the read pointers is of no

concern since a traceback front can be started from any state and will still yield the same response.

The reset to the SMU is generated from the ACS unit to ensure that all the circuits are properly synchronised. As with the ACS unit, the reset is latched on the input to the SMU to prevent noise spikes on the board resetting the circuit. A timing diagram showing the relative timing of the reset waveforms within the ACS unit and the SMU may be found in Appendix C.

4.4.3 Comparison of Implementation Options

As with the ACS unit, each design for the SMU was debugged using the Viewlogic simulation tools with input vectors generated by a PASCAL program that mimicked the operation of the circuit. The PASCAL program was also used to compare the output of the various stages in the FPGA design against a known reference to assist in the debugging process. Once more the same test vectors were used in both the functional and the timing simulation.

The placing and routing was handled entirely by the automatic tools provided with the XILINX software. Although it is possible to improve the performance of any large design by placing and routing all components manually, the automatic tools managed to generated designs that met the timing specifications of the project for both implementations presented above, so the extra effort required to manually place and routing the design was thought to be superfluous.

The initial 8-pointer even design required 185 *Configurable Logic Blocks* (CLB) in order to be implemented. The XC4005 device has 196 CLBs available, so it would be just big enough to hold the design. The XILINX Programmable Logic Data Book, [23], recommends that an attempt should be made to migrate all designs made for XC4000 series devices onto the equivalent XC4000A device on the basis that the XC4000A devices are cheaper since they have fewer available routing resources. The design was migrated across to the XC4005A device, and was re-worked until the project's timing specifications were met. When the orders where placed for the XC4005APC84-6 devices to be used in the design, it was discovered that the XC4005PC84-6 device was in fact cheaper. The difference in price between the two devices was still marginal, so the implementation on the XC4005APC84-6 device went ahead.

In comparison to the 8-pointer even implementation, the 3-pointer odd implementation used only 155 CLBs, or 79% of the XC4005A device. This represents a saving of 17% when compared to the 8-pointer even implementation. The comparison is useful for assessing the possible hardware savings associated with a corresponding increase in the latency of the SMU. The 3-pointer odd implementation can also be used to estimate the additional hardware complexity required by the equivalent even algorithm. A 3-pointer even implementation would require six memory blocks. Each memory block will need 8 CLBs to implement the RAM alone, and roughly another two CLBs for the associated

control logic (a conservative estimate), which pushes the device usage up to 165 CLBs, or 84% of the available resources. This estimate will be on the conservative side since no account is taken of the extra control logic or routing resources required to manage the extra memory block. The odd algorithms can, therefore, offer significant hardware savings when compared to the equivalent even algorithm.

The tables detailing the FPGA resources used by each implementation may be found in Appendix C.

4.5 Board Layout and PROM Programming

The three FPGA's were arranged together on a PCB along with the three PROMs required to generate the branch metrics. All three FPGAs are permanently wired to configure themselves in slave serial mode, where FPGA device configuration is controlled by a PC through a series download cable. To simplify the configuration process, the three FPGA's are organised in a daisy-chain structure whereby the first FPGA in the daisy-chain will configure itself before allowing the configuration bit stream to be passed on to the next FGPA in the chain. The three open drain DONE outputs, one from each FPGA, are tied together to form a single, global indicator that the configuration process has been successfully completed.

The FPGA's can accept several different types of resets. The first, asserted by an active Low level on the $\overline{\text{PROGRAM}}$ input in the XC4000 device or a High to Low level transition on the DONE pin in the XC3000 devices, will make the FPGA clear its internal configuration memory and wait to be re-configured. The second, asserted by a Low level on the $\overline{\text{RESET}}$ pin in the XC3000 series devices and by a programmable level (either Low or High) on any input configured to be connected to the internal global reset network, asserts a global reset of all storage elements used in the design on the FPGA (RAM blocks excluded). The final reset is the one built into the design, which has already been discussed in the preceding sections. In the board layout it was decided that only the reset required in the design would be used – the devices can be forced to re-configure themselves by temporarily disconnecting the power supply from the board and the global reset was not used because it would not perform any useful function in this particular design.

Each PROM, on the other hand, was arranged to be as close as possible to the FPGA which will use the branch metrics the PROM generates. Since the mapping of input address to output bit is arbitrary, the exact position of each output bit within the output word on each PROM was changed to facilitate routing on the board.

The information to be programmed into each PROM was generated using a PASCAL programme that calculated the relative Euclidian distance between each point in the grid illustrated in Figure 4.1 and a given point in the signal constellation. Branch metric allocation was done using a pattern similar to the one used in [13] where the points in the space adjacent to the constellation point are assigned a branch metric of zero, and

all points outside the space are assigned a branch metric that increases exponentially with an increase in Euclidian distance from the constellation point. This was achieved by describing a normal distribution around each point in the signal constellation, where the mean of the distribution lies on the constellation point in question. Changing the variance of the distribution will change both the rate at which the branch metrics increase outside the "zero region" and the size of the "zero region". By offsetting the distribution mean from the signal constellation point, the size of the "zero region" can be changed without changing the rate at which the branch metrics increase outside the "zero region". The branch metrics generated by the programme where visually inspected, and the mean offset and variance where changed until it was felt that the pattern of branch metrics gave a reasonable weighting to all points in the signal set.

Chapter 5

TCM Testbed

Once the design of the decoder was completed, a suitable test rig had to be assembled to verify that the decoder met the project specifications, and to gauge the decoder's performance in the presence of AWGN. Starting with a basic block diagram of a TCM communication system coupled with a BER tester, presented here in Figure 5.1, the operation of the test rig can be properly defined. From the diagram it is clear that the test rig has to be able to perform all tasks that lie outside the shaded box in the figure. The tasks can be listed as:

- Generate a pseudo-random bit stream to be used as input to the system.
- Encode the pseudo-random bit stream using the Ungerboek $(3, \frac{2}{3})$ encoder presented in Figure 2.2.
- Map the encoder output to the required points in the 8–PSK signal set and do all necessary modulation.
- Provide a physical channel in which noise can be added to the encoded signal.
- Generate the AWGN to be added to the signal.
- Demodulate the noisy signal generated by the above processes.
- Derive a digital representation of the received signal and pass it on to the branch metric generation unit of the decoder.
- Generate a delayed copy of the pseudo-random binary sequence fed to the encoder.
- Compare the delayed binary bit stream with the output bit stream of the Viterbi decoder.
- Count all the errors in the decoded sequence.

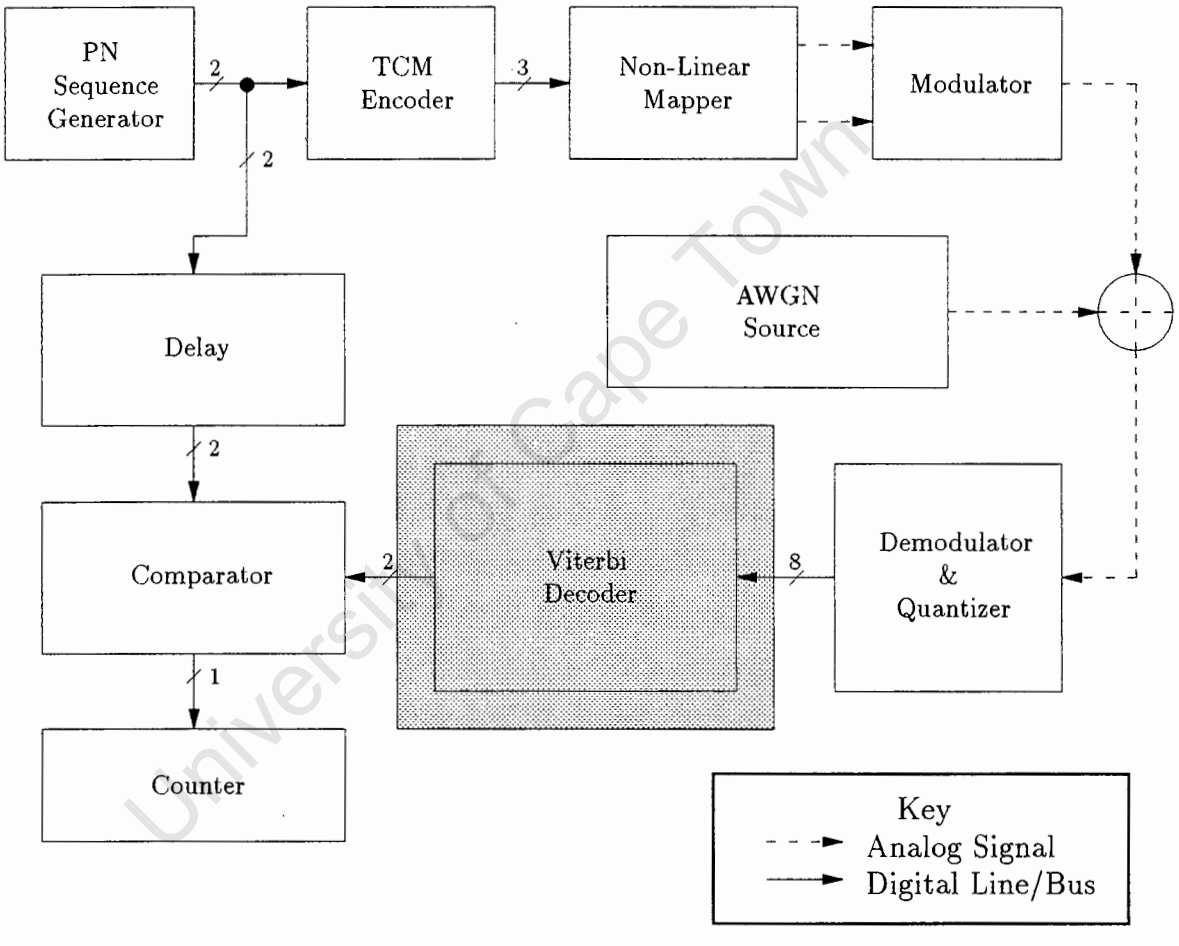


Figure 5.1: Diagram of a basic TCM communications system coupled with a BER tester.

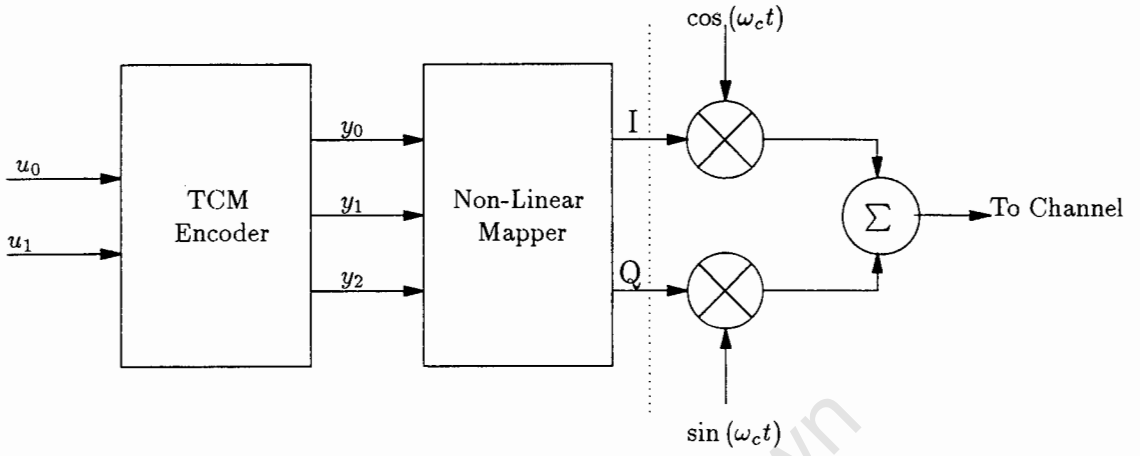


Figure 5.2: A block diagram of a TCM transmitter.

5.1 Degree of Abstraction

In order to be able to properly quantify the performance of the Viterbi decoder in an AWGN transmission channel, it is important that all other forms of interference are removed from the system. To ensure that carrier phase offsets and errors in symbol timing recovery do not interfere with the results, perfect carrier and symbol timing synchronisation have to be guaranteed. Therefore the carrier and symbol clock would need to be fed directly from the transmitter to the receiver.

The presence of the carrier becomes superfluous to the simulation once perfect carrier synchronisation is assumed between the transmitter and the receiver. Removing the sections of the circuit in Figure 5.2 and Figure 5.3 that are responsible for modulating the baseband I and Q signals onto, and demodulating them from a carrier will have no effect on the results of the simulation. The previous statement will hold true as long as it is noted that the I and Q components of bandlimited AWGN are un-correlated with one another [25], and so un-correlated AWGN must be added individually to the I and Q components of the signal in the testbed circuit. Furthermore the mean square value of both the I and Q noise components will be equal to the mean square value of the noise in the channel [25].

It is sufficient, for the purposes of simulation, to simply add un-correlated noise to the I and Q voltage levels and pass these signals straight to the quantizers in the receiver. Therefore all circuitry to the right of the dotted line in Figure 5.2, and to the left of the dotted line Figure 5.3, is unnecessary for the simulation. Instead, the system will operate as illustrated in Figure 5.4 where "AWGN Source 1" and "AWGN Source 2" produce AWGN waveforms that are un-correlated with one another.

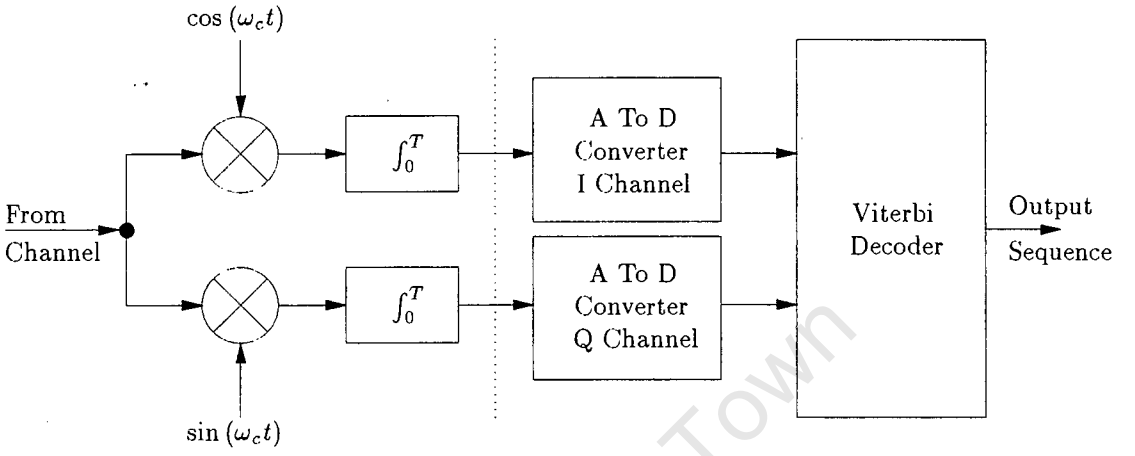


Figure 5.3: A block diagram of an MPSK receiver.

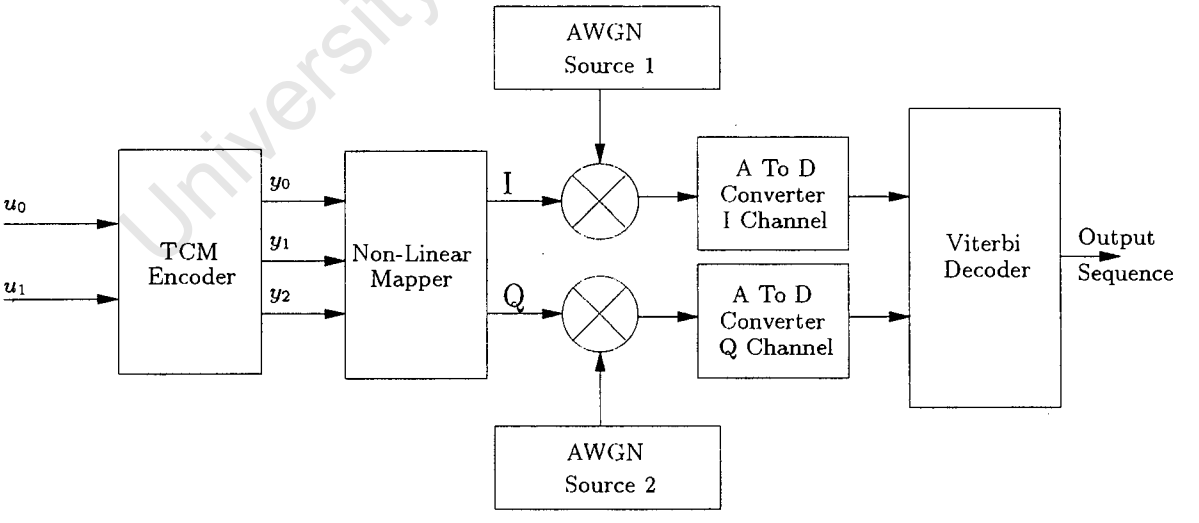


Figure 5.4: A block diagram of the simplified TCM transmitter/receiver

5.2 Hardware Emulation

An investigation was conducted into the feasibility of building a testbench out of equipment available in the department coupled with circuits built by past group members. Unfortunately the speed requirements laid down in the project specifications essentially ruled out this approach as all available equipment was unable to generate throughput at the required rate. At the very least the TCM encoder/modulator and demodulator section would have to be built from scratch to perform tests on the operating speed of the decoder.

Although there were a number of AWGN sources available within the group, none of them were capable of generating noise with a sufficiently high bandwidth to cater for the increased speed requirements of the project. Although it would have been technically feasible to perform the *bit error rate*(BER) tests at a lower clock rate and perform the operating speed tests with no noise present in the channel, it was felt that building a noise source to generate noise with a wider bandwidth did not represent a significant increase in the complexity of the circuit already being built.

Similarly, no BER checker was available to test for bit errors. Again it would have been possible to perform tests at low clock rates using a PC, but it would have been impossible to check for error free operation at high clock rates using this technique. Furthermore any BER checker used for the project would have to be able to generate a pseudo-random bit stream at 18 Mbits/s, and test for errors at a similar rate. No such device was available at the time the project was being completed.

Therefore a complete test circuit had to be built. As there were a number of FPGA's still available from the stock that was originally purchased at the beginning of the project, it was possible to implement all the digital circuitry needed in the testbed on one FPGA device. The FPGA devices are capable of implementing complex digital circuits with relative ease, so there was scope to implement a reasonably sophisticated testbench.

5.3 FPGA Circuitry

The design is effectively split into two sections: the first section covers all the digital circuitry which was implemented using one of the XC3000 series devices already in stock, and the second section covers all the analog circuitry including the digital to analog and analog to digital converters.

5.3.1 The Pseudo-Random Bit Stream Generator

The starting point in the design was the *pseudo-random binary sequence* (PRBS) generator required to generate the bit stream to be used as an input to the encoder. PRBS sequences are generated using a shift register m bits long with a tap taken from the n^{th}

bit. Bit n and bit m are fed back to the input of the shift register through an exclusive-OR gate [26]. If the contents of the shift register after each clock pulse is taken to be the state of the shift register, then the shift register will cycle through a number of states bar the all zero state.

The all zero state is specifically excluded since the feedback taps would always produce a binary zero at the input to the shift register under such conditions, and the shift register would never change state. If the position of the feedback tap is carefully chosen then the shift register can be induced to cycle through the maximum number of possible states, $2^m - 1$ once the all zero state has been excluded. Such a PRBS generator is said to be a maximal-length PRBS. A table giving the shift register lengths and tap positions for various maximal-length PRBS generators with shift register lengths of up to 39 can be found in [26]. A second table is available in [27] which gives the tap positions for maximal-length PRBS generators with shift register lengths of up to 168 bits along with techniques to reduce resource usage when using XILINX devices to implement the shift registers.

Although a maximal-length PRBS goes through all its states in a pre-defined sequence, an output bit stream taken from tap m in the shift register will appear to be random over a fixed time period. The observation will remain true as long as the time period over which the sequence is observed is shorter than the time it takes the shift registers to cycle through all of its states. The longer the shift register used to implement the PRBS generator, the longer the period over which its output can be taken to be random.

For BER testing purposes the PRBS generator must be capable of providing a sequence that is random over a sufficiently long period of time in order to obtain meaningful results. In other words, given a particular signal to noise ratio, x number of errors will be measured over a run of y bits. The BER is calculated as being $\frac{x}{y}$, where y must be made large enough to ensure that all similar runs using the same signal to noise ratio would yield approximately the same BER. In this particular case it is relatively easy to implement a long shift register because an FPGA is being used. As such the longest PRBS generator presented in [26] with a shift register of length 39 was chosen. This PRBS generator is capable of producing a bit sequence of length 549 755 813 887 bits before the sequence starts repeating itself. At a clock rate of 17.5 MHz, the run will last just over 8 hours, which was assumed to be long enough to generate statistically valid results over a wide range of signal–to–noise ratios.

5.3.2 TCM Encoder and Signal Mapper

The two most significant bits of the shift register in the PRBS generator are latched once every two clock cycles to form the two bit input to the TCM encoder. The implementation of the encoder is straight forward – the circuit is illustrated in Figure 2.2. The output of the encoder is fed to two non-linear mappers: one for the I channel and one for the Q channel.

The non-linear mappers were implemented in the form of a ROM lookup table. Although ROM blocks are not available on any of the XC3000 devices that were being used to build the testbed, a small ROM block can be implemented by hard-wiring the inputs of a multiplexer to either the High or the Low voltage level (see the application notes in [23]). The select inputs on the multiplexer are used as the ROM address.

The output of each ROM lookup table will be used to select one of four voltage levels which represent the eight points in the signal constellation (see Figure 4.1). The details on how each of the four voltage levels on the I and Q channels were chosen will be presented later on in section 5.4.1. For the sake of the ROM lookup tables it suffices to note that a four bit output is sufficient to completely define the correct four voltage levels to a resolution of eight bits. Each non-linear mapper will generate a four bit output that will be used to drive the eight bit input of a digital to analog converter.

To ensure that the output of the encoder only changes at the end of each clock cycle, the output of the ROM lookup tables are latched before being passed on to the output pins of the FPGA device.

5.3.3 The Time Delay Circuit

A delayed version of the PRBS bit stream used as input to the encoder is required so that the output bit stream from the Viterbi decoder can be checked for errors. In a standard BER checker the delay introduced by the circuit would be overcome by generating a second, identical PRBS, that would have to lock onto the output of the decoder. Normally this can only be done in the absence of noise, so the BER checker would have to have some way of holding off the noise in the channel until the second PRBS generator can be locked onto the delayed output of the first PRBS generator.

There are a number of problems with the above approach when applied to a system using convolutional coding. The whole premise of the approach outlined above is that the delay due to the circuit is unknown, so by definition the BER checker has no knowledge of the latency of the decoder. Once the two PRBSs are in sync, noise will be enabled in the channel. The BER checker will immediately begin to count the number of bits coming out of the decoder, and calculate the percentage of those bits that are in error. An initial stream of bits from the decoder, however, will still be as a result of operating through a noiseless channel due to the large latency in the decoder. The resultant inaccuracy introduced in the results can be reduced by increasing the run length of the simulation. Otherwise, if the delay in the circuit is known, more accurate results can be obtained by pre-programming the second PRBS to start up at the correct time.

The BER checker is also being implemented on a XILINX FPGA, which can easily be re-programmed to cope with design modifications yet still maintain the same pin-out. As such the FPGA housing the BER can simply be re-programmed via a PC download cable to cater for the variable latencies of each implementation of the Viterbi decoder. The result will be greater confidence in the accuracy of the BER results over all run

lengths.

There are two possible approaches to implementing a delay within the BER checker. The first, and most obvious, is to use two shift registers, where each shift register is made as long as the required delay. In cases where the latency of the circuit is short this approach would perhaps require lower hardware complexity. In circuits using Viterbi decoders, however, the latency can be expected to be large. Furthermore there is a relatively large difference in the latencies of the implementations discussed in Chapter 4. Large changes in FPGA resource requirements between different designs being implemented on the same device would make it difficult to route all the designs within the required timing constraints.

A second implementation was derived by using a second, identical, PRBS generator in the BER checker, and initialising it once the required delay had passed. The delay circuit will therefore consist of a PRBS generator, a counter and some logic circuitry which generates a reset for the PRBS generator once the counter has counted up to the required delay. Variable delays can be catered for by simply changing the logic circuitry, which will result in only small changes in the demand placed on FPGA resources and, as a result, all designs should route without difficulty.

5.3.4 The Comparator and Counters

Once a second PRBS has been generated in sync with the decoder's output, the two bit streams need to be compared so that the number of errors in the encoder output can be counted. The output from the decoder is first latched into the FPGA to ensure that it does not change during a clock cycle of the BER checker. The bits are compared using two exclusive-OR gates whose output will go high only if the two bits on its input do not match each other.

There can be a maximum of two bit errors within any single ACS clock cycle. As such the outputs of the exclusive-OR gates need to be fed to a counter one at a time so that they can be counted. This was done by using the high speed clock in the testbed which drives the PRBS, to drive a two bit multiplexer connected to the output of the exclusive-OR gates. The output of the multiplexer connects through to a clock enable on a counter. Once every half ACS clock cycle the counter is clocked. If the enable is high the counter output will be incremented. A sixteen bit counter was used to count bit errors.

Simultaneously a second thirty two bit counter is used to count the total number of bits received from the decoder. Both counters will need to be synchronised with one another in order to ensure accurate results. The reset pulse generated in the delay circuit to synchronise the two PRBS generators can also be used to hold the bit counters reset until there is valid output from the decoder.

A second design was implemented to count for event errors in the received sequence. An event error occurs when a symbol is decoded that does not form part of the correct sequence following a state that is part of the correct sequence [1]. So to count for event

errors the testbed needs to determine the trellis path corresponding to the output of the decoder and compare it to the path described by the output of the delayed PRBS generator.

To do this both the bit stream from the second PRBS generator and the output from the decoder are fed into two encoders. The output codewords of the encoder are compared to check for a symbol that is not part of the correct sequence through the code trellis. As soon as this happens, an event error has started. To prevent the counters from counting a string of falsely decoded symbols, the counter is disabled until the two paths re-merge. The point at which the paths re-merge can be determined from a comparison of the bits fed to the encoder, and from a comparison of the codeword outputs of the encoders. Once the two pairs match each other, the decoder is following the correct path through the code trellis.

In the error event implementation, the event counter and error event counter are clocked from the symbol clock as opposed to the bit clock used in the BER implementation. The error event implementation only required marginal additional resources to be implemented in the same FPGA.

One extra feature was added to both implementations: a single external input which controls the point at which the simulation stops running. The input is generated by means of a jumper on the circuit board. If the jumper is in place, the input will be held low. If the jumper is removed, the input will go high. The jumper voltage is used as an input to the FPGA to reset the circuit into two different modes of operation:

1. If a reset is applied to the testbed FPGA and the jumper is in place (the input is held low) then the testbed FPGA will carry on generating bits regardless of whether or not the counters overflow. The readout from the bit error and bit counters will be frozen as soon as the jumper is removed. Once the jumper is removed the circuit will lock, so replacing the jumper will not allow the BER tests to proceed. This was done to ensure that the results are not affected by jitter on the input of the FPGA due to the jumper being replaced and removed. A BER test can be re-started by applying a reset pulse to the FPGA.

2. If a reset is applied to the testbed and the jumper is not in place, the bit error and bits received counters will be frozen automatically as soon as any one of the two counters overflows. This feature allows for unsupervised operation of the testbed, as well as operation at signal–to–noise ratios that result in a high probability of error from the decoder. In the later case the feature will allow the BER tests to be run for the maximum length of time without the counters overflowing. If the jumper is replaced during a BER test, the counters will be instantly frozen, and the jumper will be locked out of the circuit. Once more BER testing can be resumed by applying a reset pulse to the FPGA housing the BER tester.

5.3.5 Reset Circuitry

On application of an external reset the testbed will have to reset the first PRBS generator and reset the counter which generates the reset for the second PRBS generator. Once again the external reset pulse is latched onto the device to prevent noise on the board resetting the circuit. The testbed also needs to delay the reset before passing it on to the Viterbi Decoder to allow the signals enough time to propagate through the channel before the decoder starts operating on the received sequence.

The reset pulse is also delayed internally to the FPGA so that the latches with synchronous presets can be properly initialised.

5.3.6 Implementation Results

The final design using shift registers to delay the bit sequence fed to the encoder required 155 CLBs, which is 69% of a XC3064APC84-7 device. The design was just too big to fit on a XC3042APC84-7 device, so the XC3064APC84-7 was the smallest device available that can still implement the design.

In comparison, the design using the second PRBS generator required only 116 CLBs making it small enough to fit on a XC3042A device. Unfortunately, by this stage, the boards had already been built to hold the XC3064A device.

5.4 Analog Circuitry

The FPGA used on the testbed is only capable of producing digital output. In terms of the simulation, the digital output must be converted to an analog voltage level to which noise is added in the channel. The noisy signal then has to be turned back into a digital form to allow it to be processed by the decoder.

5.4.1 Digital to Analog and Analog to Digital Converters

The heart of the analog section of the testbed are the two *digital to analog converters* (DAC), and the two *analog to digital converters* (ADC). There is a pair of converters (one DAC, and one ADC) for each of the I and Q channels. Each DAC and ADC is presented here as a pair, since the design of the one affects the design of the other.

The critical factor in choosing both the DAC and the ADC was their speed of operation. The project requires data rates of 17 MBits/s or more, which translates to a symbol rate of 8.5 MSymbols/s. Both the DACs and the ADCs have to be capable of conversion times of under 114ns in order for tests to be performed on the operating limits of the decoder and still be confident that the transmitter/receiver pair is operating correctly.

Finding a DAC capable of settling times of less than 110ns proved to be fairly easy. The

DAC08 is capable of settling times of as low as 85ns if its output is properly configured. Even more attractive was the fact that the device was both readily available and low in cost. The DAC08 was therefore chosen for use in the project.

As the DAC08 has a current output there are a range of choices as to the output configuration used to convert the current to a voltage. For this project, however, the choices are severely limited by the requirement that the DAC08 operate at maximum speed. The output capacitance of 15pF requires that the load resistance be kept below 500Ω to prevent the RC time constant at the output from dominating the settling time.

Initial tests were performed with the output of the DAC08 directly coupled to the input of the adder (in this case a LM6361 operational amplifier configured as an adder) through a small resistor, where it was found that the capacitance on the output of the DAC08 and the stray capacitance present on the input of the LM6361 where high enough to make the circuit unstable. To reduce the load capacitance and the load resistance seen by the DAC08, its output was buffered using a 2N3904 npn BJT configured as a common base amplifier as suggested in the DAC08 data sheet for a settling time measurement circuit. The only modification made to the circuit found in the data sheet was that the collector pin of the transistor was tied directly to the input of the summing amplifier. In this way the value of the collector resistor is set to zero and the current gain goes as close to unity as is possible.

All other settings on the DACs relate to the input reference current and full-scale adjustment circuits. For the former the data sheet on the DAC08 recommends that the reference current be kept between 2mA and 4mA for optimum operation. The output current is also a function of the binary inputs and the reference current, so the choice of reference current determines the full-scale output current. The equation used to derive the output current is:

$$I_{out} = \frac{\mathbf{B}}{256} \times I_{ref}$$

Where I_{ref} is the reference current and $\mathbf{B}$ is the 8 bit input vector ($\mathbf{B}$ ranges between 0 and 255, so the full-scale current can be derived by setting $\mathbf{B} = 255$ in the equation above). Choosing the reference current to be too small will yield constellation points that are close together, and the noise generated in the circuit due to the DAC and the summing amplifier will become significant in terms of the signal to noise ratio. The larger the reference current is made, the longer the required output settling time will be. Since the data sheet itself measures the settling time to be 85ns with a reference current of 2mA, 2mA was chosen as the reference current for the DAC.

Having chosen the reference current, the output current corresponding to the various constellation points can be derived. Calculating the I and Q currents and translating them back into digital words ($\mathbf{B}$ in the equation presented above) the decimal values of $\mathbf{B}$ and their binary equivalents are:

except that the position of the flip-flops with synchronous presets within each shift register is different in each PRBS. As a result each PRBS will start up in a different state, and the output sequences will be un-correlated with one another.

This time the FPGA is clocked using a 10MHz crystal connected to its crystal clock inputs. As a result the two output PRBS's will run at a rate of 10 Mbits/s. The output spectrum of the two PRBS waveforms is flat up to approximately 10% of the clock frequency, so AWGN can be generated with a maximum bandwidth of 1MHz.

The two PRBS's fitted with ease onto one XC3030APC84-7 device. Should it be required, the FPGA can always be re-programmed to generate longer PRBS sequences.

5.5.2 Analog Circuitry

The AWGN waveform, once added to either the I or Q waveform, should make that waveform blur uniformly in the both the positive and negative directions. As such the AWGN waveforms must be generated from a bi-polar waveform, so the output from the FPGA, which is uni-polar, will have to be level shifted before passing on through the low pass filters. Due to the high frequency of the output waveform, a simple, single BJT level shifting circuit is used on each PRBS to shift the output voltage range down by 2.5V. The net result is a bi-polar PRBS waveform between 2.5V and -2.5V.

The PRBS waveform then has to be filtered to generate the wanted AWGN waveform. As previously mentioned, the power spectrum of the PRBS is flat up to approximately 10% of the frequency of the clock driving the PRBS generator. Allowance must be made, however, for filter roll-off in setting the noise bandwidth, so typically the PRBS is filtered between 1% and 5% of the clock frequency, depending on the type of filter used. In this case a fourth order Butterworth filter was chosen to filter the PRBS waveform at 5% of the clock frequency to get the noise bandwidth as high as possible. Increasing the noise bandwidth will allow the bandwidth of the TCM system to be increased as well. As a result the bit-rate can be pushed up and the simulation run times will be reduced.

Setting the cutt-off point of the filters at 5% of the clock frequency limits the noise to a bandwidth of 500KHz. To calculate the noise power allowed to pass through the filter, the filter's equivalent noise bandwidth needs to be assessed. In the case of a fourth order, low pass Butterworth filter, the noise equivalent bandwidth is calculated using [25]:

$$B_N = \frac{1}{2\pi}\int_0^\infty \frac{1}{1+\frac{\omega}{\omega_c}}d\omega \qquad \text{Hz}$$

Where ω_c is the filter's cutt-off frequency in radians/s. In this case the noise equivalent bandwidth will be 513KHz. As can be seen the noise power is limited to a bandwidth well below the 1MHz limit, so the use of the 500KHz cutt-off frequency in the filters is justified.

At the output of the final filter, the noise is passed through a variable resistor connected

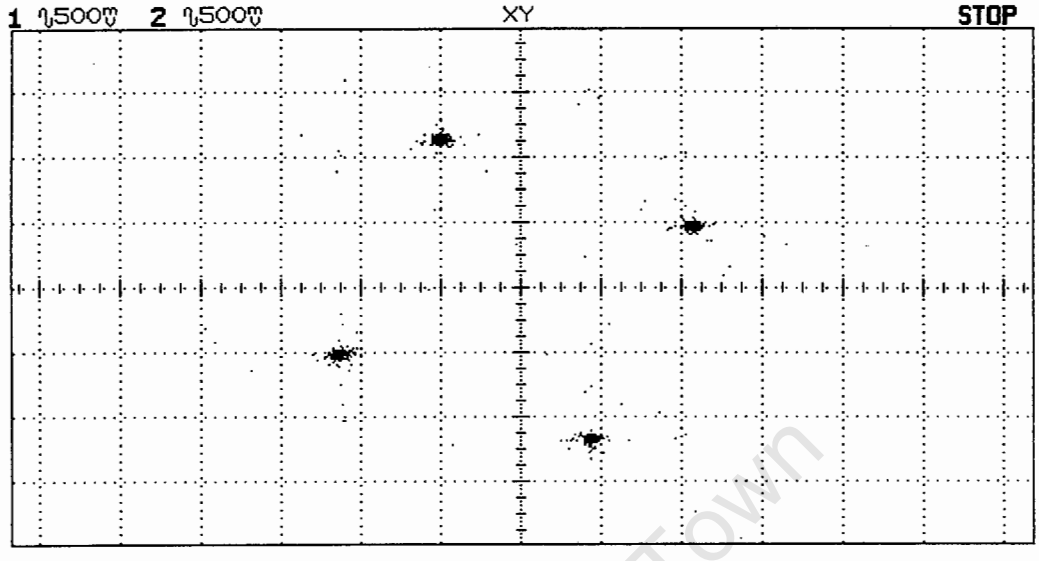


Figure 5.5: Screen capture showing the 4–PSK signal constellation at a throughput rate of 200Mb/s.

to ground. By adjusting the tap on the resistor the noise power can be set. In this way tests can be performed at various signal to noise ratios.

5.6 Uncoded 4–PSK

The asymptotic coding gain of the $(3, \frac{2}{3})$ Ungerboek code is measured relative to uncoded 4–PSK. In order to properly estimate the gain in receiver sensitivity due to coding in the transmitter, the system needs to be tested against a practical 4–PSK scheme.

As both the testbed and the decoder were built using FPGAs, they were simply re-programmed to generate and detect 4–PSK respectively. In the case of the transmitter, the binary source is then mapped directly into I and Q voltages. In order to get an accurate estimate of the coding gain, the 4–PSK signal set that was used was simply a subset of the 8–PSK signal set used for the Ungerboek code. A screen capture from an oscilloscope is included in Figure 5.5 for a system clock frequency (and therefore bit rate) of 200KHz.

In the case of the receiver, the two FPGAs housing the ACS circuits were re-programmed to determine which side of the decision boundaries the received signal is on. One FPGA was programmed to handle the decision boundary between the codeword pair 00 and 10, and codeword pair 01 and 11, and the other FPGA was programmed to handle the decision boundary between the codeword pair 00 and 10, and codeword pair 01 and 11. The FPGA that previously housed the SMU was re-programmed to interpret the output from each of the other two FPGAs and pass the result on to the testbed circuit for

comparison with the original bit stream.

As an illustration of the complexity of a TCM receiver, the resource usage in this case can be compared to the most efficient Viterbi decoder implementation (ACS units with 5 bit path metrics using series metric comparisons, and the 3-pointer odd implementation of the SMU):

- The difference in resource usage between the two implementations of the testbed was small – 110 CLBs for the 4–PSK system as opposed to 116 CLBs for the 8–PSK system.
- The FPGAs housing the ACS units required 133 CLBs each to implement the Viterbi decoder. For the 4–PSK implementation they required 7 CLBs.
- The FPGA housing the SMU requires 1 CLB when used in the 4–PSK receiver as opposed to 152 CLBs in the implementation of the Viterbi decoder.

The Viterbi decoder therefore represents a sizeable increase in the complexity of the receiver.

Chapter 6

Results

Two sets of basic tests were performed on the final designs for the decoder aside from the functional tests to ensure the hardware was working properly. The first set were speed tests, where an attempt was made to gauge the maximum achievable bit rate of each design of the decoder. The second set of tests were performance tests, aimed at assessing the decoder's performance in the presence of AWGN.

6.1 Speed Tests

The design specifications required the encoder to operate at bit rates over 16 Mb/s (17.5 Mb/s was used as the upper limit in this case). In order to generate this bit rate in the decoder, different clock rates will have to be generated for the two SMU designs (the SMUs determine the ratio of the system clock to bit rate in the design):

- The 8-pointer even design needs four system clock cycles for every two bits fed through the channel. As such the system clock will need to be able to run at 35MHz to get a bit rate of 17.5Mb/s.
- The 3-pointer odd design needs two system clock cycles for every two bits of output from the decoder. In order to generate a bit rate of 17.5Mb/s in a system using the 3-pointer odd algorithm, the system clock will need to run at a rate of 17.5MHz.

6.1.1 Clock Generation Circuitry

Unfortunately there were no function generators available within the department capable of generating clock signals at such high frequencies. Although it would have been feasible to put together two crystal oscillators to generate the two clock frequencies mentioned above, the resultant oscillators would have capable of operation at only one frequency.

For test purposes a variable rate function generator was needed to enable approximate measures to be made of the clock frequency at which the circuit stops operating correctly.

There was, however, a signal generator in the department capable of generating a $200mV_{p-p}$ sine wave from 100KHz up to 110MHz. To convert the output of the signal generator to a 5V square wave, it was fed through three 74HCU04 un-buffered inverters configured as a single high gain amplifier. The output from the amplifier was buffered through a 74HC240 inverting buffer in order to cope with the fan–out required by the decoder and testbed circuits.

Tests on the circuit showed it to be capable of generating a 5V square wave at frequencies up to about 30MHz. Although the 8-pointer even design cannot be tested up to the full operating speed using such a system clock, there was greater confidence in the ability of the 3-pointer odd design to meet the project specifications. As such the proposed method of generating the clock waveform was considered to be acceptable for use in the speed tests.

6.1.2 Speed Test Results and Interpretation

Initial test proved to be disappointing - the noise generated in the analog circuitry caused the decoder to fail at throughput rates as low as 4Mb/s. Various approaches were used to reduce noise in the circuit (see below), and as much as a fourfold increase in operating speeds was achieved.

The results for the various designs were:

- The 3-pointer odd SMU design coupled with ACS units using 5 bit path metrics and parallel path metric comparisons, stopped operating correctly when the system clock reached 16.4MHz. Allowing for a small safety margin it was found to operate reliably for all frequencies up to 16.0MHz, which corresponds to a throughput rate of 16.0Mb/s.
- The same SMU design with 6 bit path metrics and parallel compare stopped operating correctly when the system clock reached 8.7MHz. When the clock frequency was reduced to 8.5MHz, the decoder operated without error. At a system clock frequency of 8.5MHz, the encoder is being fed bits at a rate of 8.5Mb/s.
- When the decoder was made up of the 3-pointer odd SMU and ACS units using 5 bit path metrics with series path metric comparisons, the design began to fail when the system clock exceeded 16.6MHz. Of all the designs, this one was the most stable at high clock frequencies. The limit on operating speed was reached when the system clock was at 16MHz, or a bit rate of 16Mb/s.
- The 8-pointer even SMU design with ACS units using 6 bit path metric representations and parallel path metric comparisons started to make errors when the system clock reached 16.2MHz. Stable operation was again at around 16MHz, or a bit rate of 8.1Mb/s.

Although the results mentioned above do just meet the project criteria, they fall short of the predictions made during the FPGA simulations. The cause of the decoder's failure at high clock rates needs to be examined before any conclusions can be reached about the reliability of the FPGA simulation software. The first check must be on the input to the decoder – if the input becomes corrupted at high system clock frequencies then the decoder cannot be expected to function correctly. Included in Figure 6.1, Figure 6.2 and Figure 6.3 are screen captures from a digital storage oscilloscope showing the signal constellation at system clock rates of 100KHz, 2MHz and 16MHz respectively.

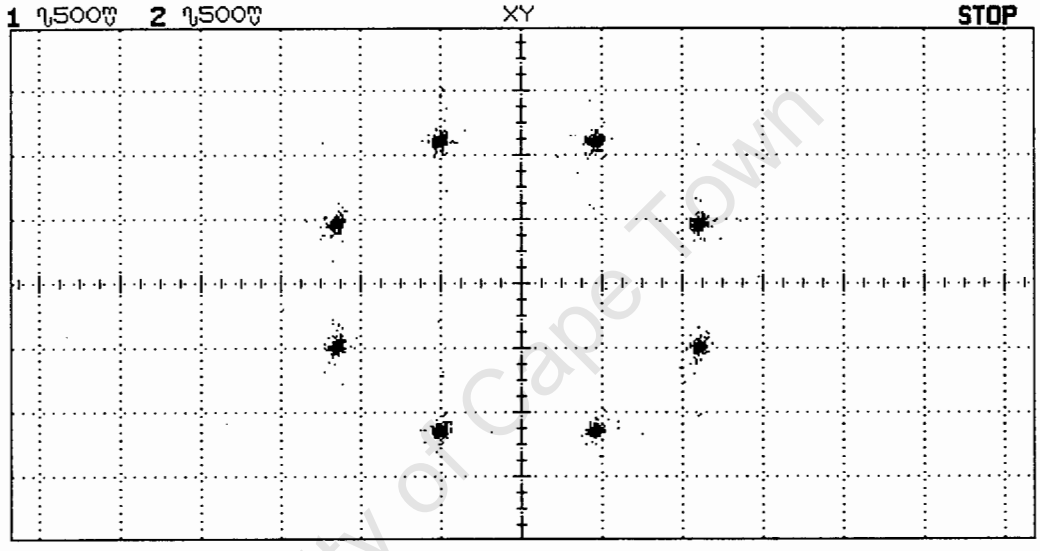


Figure 6.1: Screen capture of the 8–PSK constellation at 100KHz with no noise added.

It is clear from these diagrams that as the system clock increases, so the noise in the circuit also increases. The following steps were introduced to minimise noise in the circuit and obtain the results given above:

- Additional decoupling capacitors were added in the circuit. This had a marked effect on the operating range of the decoder, particularly when decoupling was added in the circuit used to generate the system clock waveform.
- Each part of the circuit was given its own power supply. Initially this was only done for the testbed and the decoder, with the decoder sharing a power supply with the system clock circuitry. A moderate increase in the operating speed was obtained as a result.
- The clock circuit was then given its own, separate power supply. A large increase in the decoder's maximum operating frequency was obtained as a result.
- All power leads for the circuits were joined straight to the power supplies rather than through leads with crocodile clips. Although there was no dramatic increase

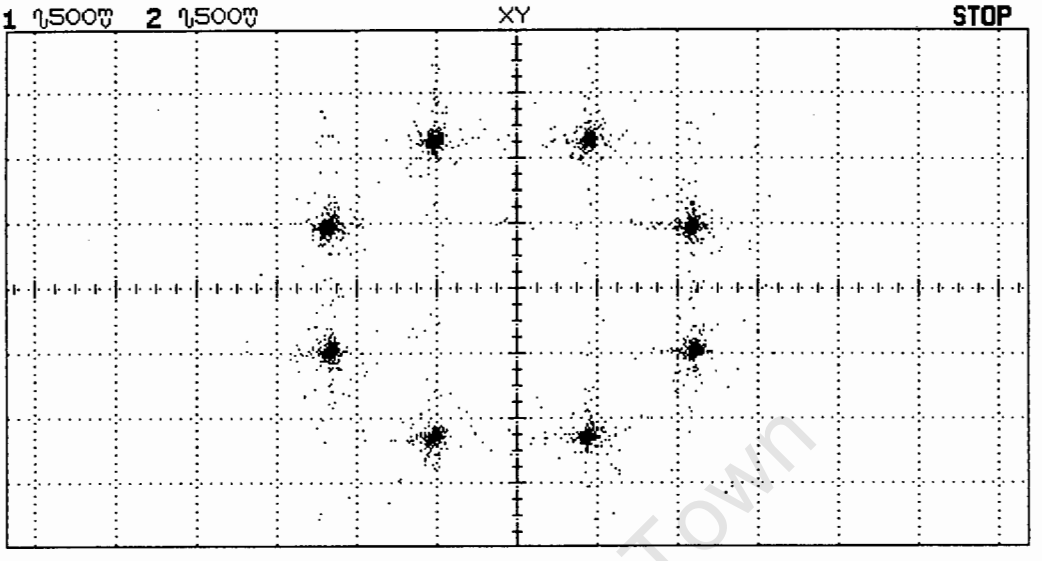


Figure 6.2: Screen capture of the 8–PSK constellation at 2MHz with no noise added.

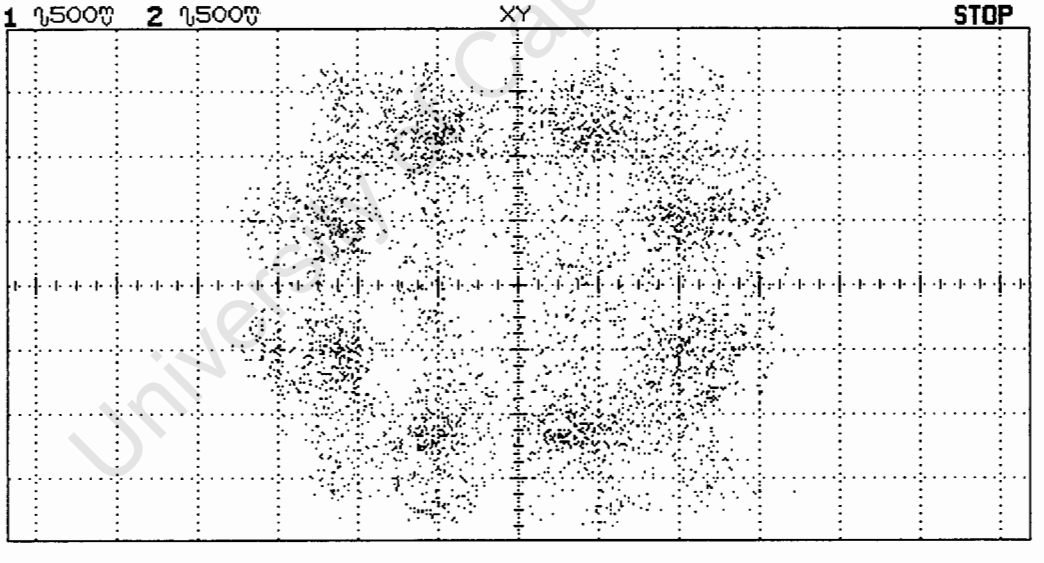


Figure 6.3: Screen capture of the 8–PSK constellation at 16MHz with no noise added.

in operating speed, the circuit became noticeably more stable a high system clock rates.

All indications are that the decoder failed due to noise in the circuit as opposed to failure of the FPGA devices to operate correctly when subjected to high clock rates. At the required 17.5Mb/s bit rate, the decoder was still generating output and all indications were that the testbed FPGA was still operating correctly. The signal constellation, when

viewed on an oscilloscope, was, however, totally un-recognisable.

Failure of the decoders using 6 bit path metric representations to match the speeds of the of the equivalent 5 bit implementations can be attributed to the increased power requirements of the FPGA adding noise to the circuit. Better board design and increased capacitive decoupling, particularly around the FPGAs, should enable all designs to run correctly at the clock frequencies promised in the simulations.

6.2 Bit Error Rate Tests

The bit error rate will need to serve two functions:

1. To evaluate the performance of the decoder, and, in particular, to evaluate the performance of the decoder relative to an uncoded 4–PSK system running on the same testbed.

2. To verify that the decoders using 5 bits to store the path metrics have the same performance as the designs using 6 bits representations of the path metrics.

The first item listed above will serve to both measure the implementation losses involved in the design, as well as the coding gain achieved in practice over the uncoded case. The second item is included to verify that equation (3.7) does in fact accurately determine the decoder's dynamic range.

There are many different measures of the performance of Viterbi decoders. The most common, based on the measure of TCM systems being the code's free Euclidian distance, is the error event probability, measured as the first symbol in error after a symbol that does form part of the correct sequence [1]. In the case of 4–PSK this corresponds to all symbols in error, since the symbols in the channel can form any possible sequence given the number of symbols in the code.

Although the above is an intuitive measure of the performance of a TCM system based on the Euclidian distance, it is of little interest to a communications engineer. The critical factor in the measurement of a system's performance is the bit error rate, or the probability of incorrectly decoding any bit transmitted through a noisy channel. As such it was this measure that was primarily used to evaluate the performance of the designs used in this project.

A second test was conducted on the decoder using 5 bit path metric representations and the 3-pointer odd SMU to determine its error event probability in the presence of AWGN. The results from this test can then be compared to those obtained by Ungerboek to evaluate the system's performance.

6.2.1 Test Procedure

The tests are to be performed over a channel with AWGN of power $\overline{n^2(t)}$, where $n(t)$ is the noise voltage of the band–limited noise at the receiver. If the noise is narrowband, then the noise voltages in the orthogonal components of the signal can be expressed in terms of the total noise voltage [25]:

$$n(t) = n_I(t)\cos(\omega_0 t) + n_Q(t)\sin(\omega_0 t) \tag{6.1}$$

Where $n(t)$ is the noise voltage in the channel, and $n_I(t)$ and $n_Q(t)$ are the noise voltages in the I and Q components of the signal respectively.

The noise power in the orthogonal components of the signal are derived from equation (6.1) as being [25]:

$$\overline{n^2(t)} = \frac{1}{2}\overline{n_I^2(t)} + \frac{1}{2}\overline{n_Q^2(t)} \tag{6.2}$$

And:

$$\overline{n^2(t)} = \overline{n_I^2(t)} = \overline{n_Q^2(t)}$$

When testing the decoder, however, un-correlated noise is added directly to the I and the Q channels. The noise that is added corresponds to the two terms on the right hand side of equation (6.1). Defining σ^2 to be the variance of the noise in each dimension of the signal set, the total noise power can be derived in terms of σ^2 for two dimensional signalling from equation (6.2):

$$\overline{n^2(t)} = 2\sigma^2 \tag{6.3}$$

The above result can then be applied in the derivation of the signal–to–noise ratio in the receiver. Assuming that the received samples are of the form [1]:

$$z_n = a_n + w_n$$

where z_n is the received sample at time $t = nT$, T is the modulation interval, a_n is a complex-valued discrete channel signal and w_n is an independent normally distributed noise sample with zero mean and variance σ^2 along each dimension of the signal set. From the equations above, it is clear that the average signal–to–noise ratio is [1]:

$$\frac{S}{N} = \frac{E\{|a_n^2|\}}{2\sigma^2}$$

where S denotes the average signal power, N denotes the average noise power ($N = \overline{n^2(t)}$) and $E\{\cdot\}$ denotes the expectation operator.

Typically the performance of digital modulation schemes is not measured in terms of $\frac{S}{N}$, but rather in terms of $\frac{E_s}{\eta}$. In the second measure E_s is the average symbol energy and

η is the single sided noise spectral density. The two ratios are related by [25]:

$$\frac{S}{N} = \frac{E_s}{\eta} TB$$

where T is the modulation interval and B is the equivalent noise bandwidth of the system. If T is inverted and multiplied by two to obtain R_b, the system's bit rate, the ratio $\frac{R_b}{B}$ is defined as the bandwidth efficiency of the system [28]. When testing 8–PSK codes, Ungerboek assumed a bandwidth efficiency of 2 bps/Hz [1]. In order to allow comparisons to be made between the results obtained here and those obtained by Ungerboek, the same ratio was used when performing the BER tests. Setting $R_b = 2B$ and noting that there are two bits per modulation interval, T, gives $T = \frac{1}{B}$. Applying this result above:

$$\frac{S}{N} = \frac{E_s}{\eta}$$

in this case.

As the noise–equivalent bandwidth is 513KHz, all tests were done at a bit rate of 1026Kb/s. This includes both the 8–PSK and 4–PSK simulations. The average symbol power was calculated using:

$$S = \frac{1}{M} \sum_{i=0}^{M-1} \left(I_i^2 + Q_i^2 \right)$$

Where M denotes the number of signals in the code's signal set, and I_i and Q_i are the I and Q voltages associated with the i^{th} signal in the signal set. η is measured directly from the circuit using a RMS volt meter.

A screen capture from a digital oscilloscope showing the signal constellation at a SNR of 10dB is included in Figure 6.4.

6.2.2 Derivation of Theoretical Bounds on 4–PSK

The Ungerboek $(3, \frac{2}{3})$ TCM code achieves its coding gain by signal set expansion of the 4–PSK signal set. As such all coding gains achieved by the system are measured relative to the performance of 4–PSK.

In order to properly quantify the performance of the decoder designed for this project, its performance needs to be measured against a practical 4–PSK system operating over the same channel. As a measure of the implementation as a whole, the performance of the 4–PSK system can be compared to its theoretical limit. The symbol error probability, P_e, of a 4–PSK system is given by [29]:

$$P_e = \text{erfc}\left(\sqrt{\frac{E_s}{2\eta}}\right) - \frac{1}{4}\text{erfc}^2\left(\sqrt{\frac{E_s}{2\eta}}\right) \tag{6.4}$$

Where $\frac{E_s}{\eta}$ is the ratio of average symbol energy to noise spectral density and is calculated

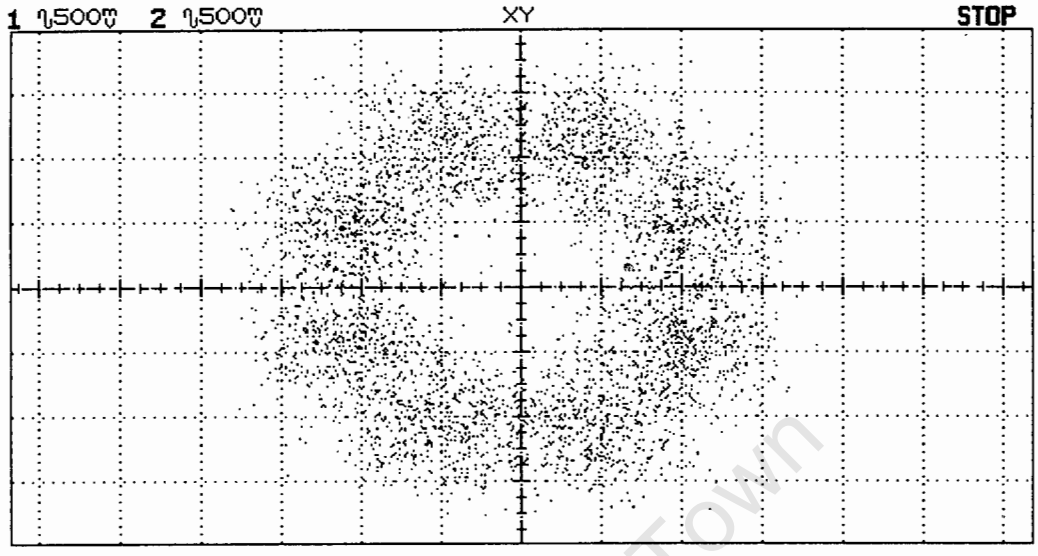


Figure 6.4: Screen capture of the 8–PSK constellation at 1.026MHz with 10dB of noise.

in the same way as for the 8–PSK system. erfc(·) refers to the Gaussian integral function. The bit error rate can also be derived from equation (6.4) by simply dividing P_e by two [25].

6.2.3 Bit Error Rate Test Results

Graphs of the results of both the bit error rate tests and the error event tests are included in Figure 6.5 and Figure 6.6 respectively.

Analysis of Results: Coding Gain

From the results it is clear that the coding gain of the TCM scheme presented here over uncoded 4–PSK is substantially lower than the theoretical *asymptotic coding gain* (ACG) for the $(3, \frac{2}{3})$ Ungerboek TCM code of 3.6 dB. An examination of the simulation results presented in [1] show that the coding gain of any TCM scheme only approaches the ACG at high signal to noise ratios. This is borne out in the performance results presented here: at low signal to noise ratios the performance of the TCM scheme is in fact worse than the 4–PSK scheme. As the signal to noise ratio increases, so the TCM scheme begins to out-perform the 4–PSK scheme and the coding gain begins to approach the ACG.

The coding gain obtained here is, however, about 1 dB lower than the coding gain of simulations of similar schemes obtained in [1] at all signal to noise ratios. In [30], however, simulations were performed using the same Ungerboek code to determine the

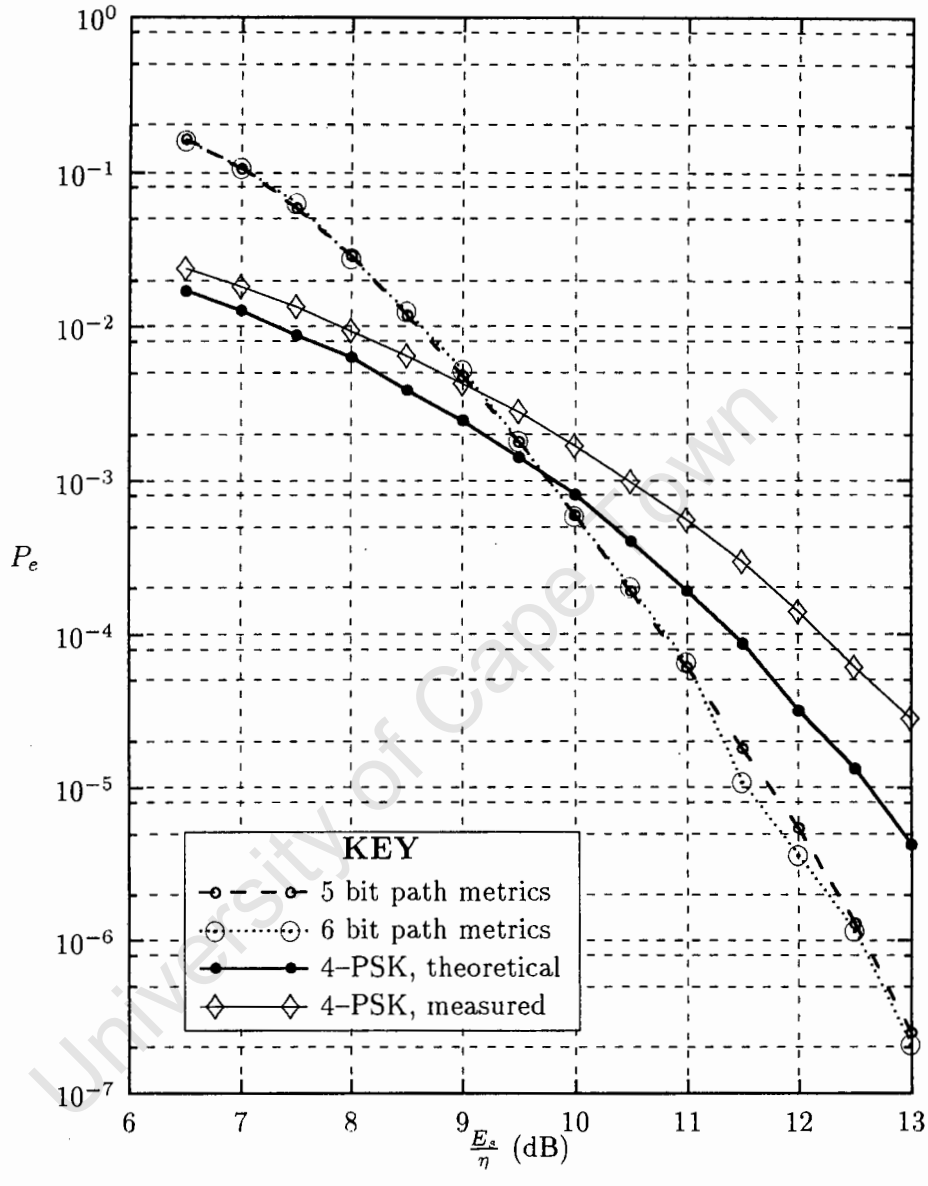


Figure 6.5: Graph showing results of BER tests for 4–PSK and coded 8–PSK.

effect of reducing the channel resolution in the receiver, in terms of the number bits used to represent the I and Q components of the received signal, on the performance of the TCM scheme at a signal to noise ratio of 9.5 dB. The results showed that the TCM scheme would perform as indicated by the simulation results in [1] if the channel resolution was above 6 bits. Below six bits the performance dropped off fairly rapidly with a 1 dB loss in performance for a channel resolution of 4 bits. As such the performance results

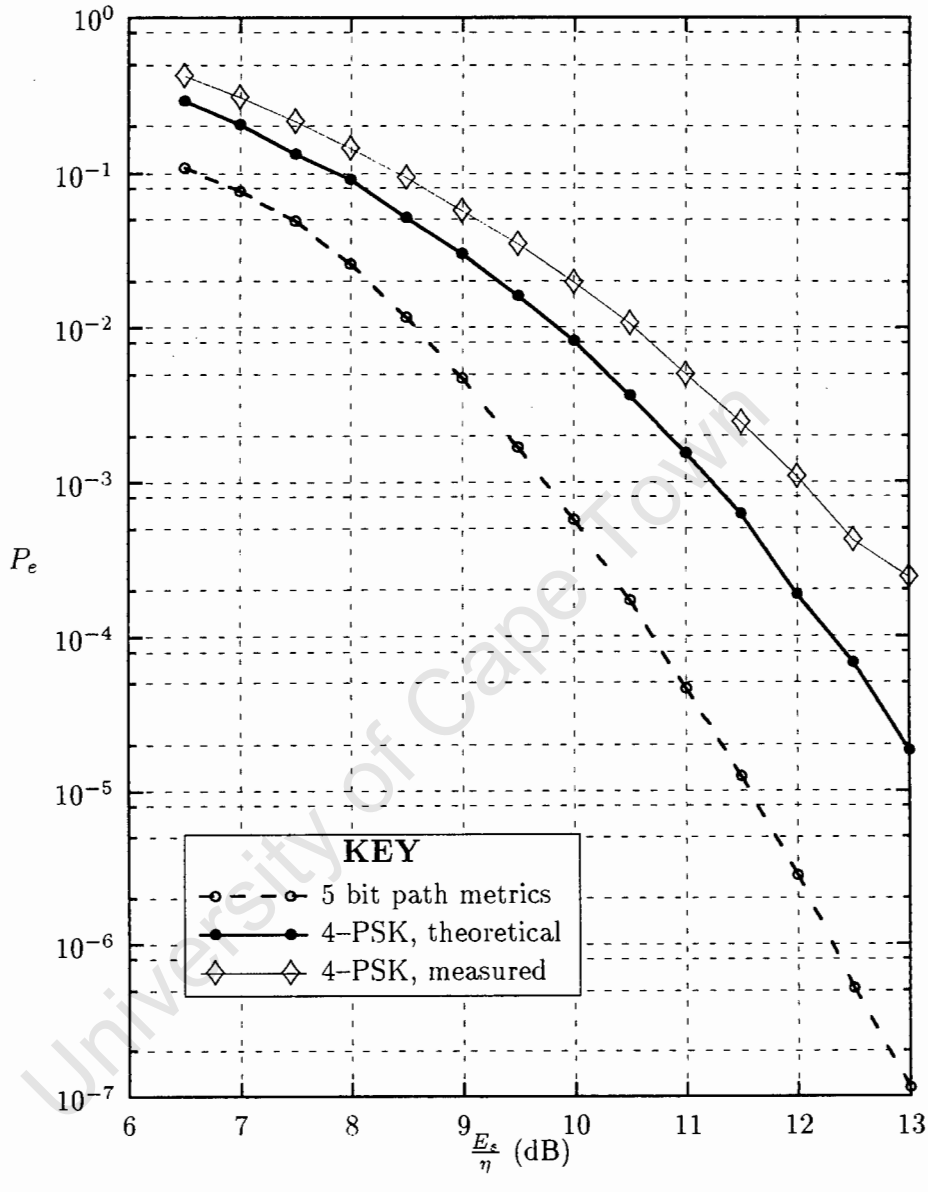


Figure 6.6: Graph showing results of error event tests for 4–PSK and coded 8–PSK.

obtained here using 4 bit channel resolution in each dimension match the predictions on the performance of such schemes presented in the literature.

The effects of low channel resolution are also clearly visible in the results presented for 4–PSK. The performance of any practical 4–PSK scheme will be dependant on the receiver's ability to resolve the position of the decision boundaries within the signal space. In cases where coarse channel resolution is used, the 4–PSK receiver will assume that

the decision boundary lies between two quantization intervals. In reality, however, the decision boundary will typically lie inside a specific quantization interval. The net effect is to shift the decision boundary in favour of a pair of signals in the signal set.

The effects of coarse channel resolution can best be understood by considering any pair of signals, A and B, in the signal set. Assuming that the coarse channel resolution shifts the decision boundary in favour of A (further away from A), the margin–over–noise of signal A will be increased. At the same time the margin–over–noise of signal B will be reduced. The probability of receiving any particular point in the signal space given that signal A was transmitted is described by a normal distribution of variance σ^2 centred on A. The area under the normal distribution curve that lies on the other side of the decision boundary – from A towards B – corresponds to the probability that A will be incorrectly decoded as B. Shifting the decision boundary will effectively reduce the size of this area, or will reduce $P(B/A)$, the probability of receiving B given that A was transmitted. At the same time, however, there is an associated increase in $P(A/B)$.

As the distribution is described by an exponential curve, the increase in $P(A/B)$ will be greater than the corresponding reduction in $P(B/A)$, so the overall probability of error will increase. The effect described above will become marked at higher signal to noise ratios, since the noise variance, σ^2, is reduced and the ratio of the increase in $P(A/B)$ to the $P(B/A)$ will increase as a result.

In most practical systems the effect described above can be made negligible by designing the receiver in such a way that the decision boundary lies as close as possible to the boundary between two quantization intervals. In the simulations presented here, however, this was not possible since the decision boundaries were rotated by $\frac{\pi}{8}$ from the I and Q axes respectively. The performance of the receiver is therefore affected by the receiver's ability to approximate the two decision boundaries as straight lines described by pairs of quantized I and Q voltages. The low resolution used during simulations of the 4–PSK system should, therefore, mean that the receiver performance relative to the theoretical case will get worse as the signal to noise ratio is increased. This effect is clearly evident in the results.

Increasing the channel resolution would improve the approximation of the decision boundaries in the receiver. As a result the effects of channel resolution on the error performance of the receiver will become less marked.

Of further interest is that there appears to be no degradation in performance due to the use of three bit branch metrics with four bit channel resolution, or of the use of fixed branch metrics over a range of signal to noise ratios. As previously mentioned, the results at a SNR of 9.5 dB match those presented in [30] where the Euclidian distance between the received points and the trellis transition was explicitly calculated and used as the branch metric.

Analysis of Results: Viterbi Decoder Dynamic Range

The two curves presented in Figure 6.5 for the probability of bit error in the decoders using five bit and six bit path metrics lie directly on top of one another. Therefore the decoder's dynamic range is properly defined by equation (3.7), and the reduction in the number of bits used to describe the path metrics leads to no degradation in the performance of the decoder.

Analysis of Results: Bit Error Probability Versus Error Event Probability

As previously mentioned the error event probability, described in the case of TCM as the first false channel symbol decoded after a state that still belongs to the correct path through the code trellis [1], is the measure most often used to compare TCM systems and calculate the coding gains associated with any particular TCM scheme over the corresponding uncoded case.

In practice, however, it is the bit error probability that is the most important measure of performance of any particular modulation scheme. From the results presented above (see Figure 6.5 and Figure 6.6), and the results presented in [1], it is clear that the use of error event probability provides an overly optimistic measure of the improvement in performance of a TCM scheme over the uncoded case.

The derivations of both the upper and lower bounds on the bit error performance of a TCM system do, however, rely on the derivation of the code's free distance [3]. The free distance of the TCM code is also used in the derivation of the both the upper and lower bounds on error event probability [3], so the error event probability and the bit error probability are linked. As such error event probability is a useful measure when comparing the performance of TCM schemes.

Chapter 7

Conclusion

Several different designs for Viterbi decoders were presented in this thesis with the aim of determining design techniques that are suited to FPGA implementations of Viterbi decoders. All designs were functionally equivalent in terms of their performance in the presence of AWGN, yet the decoder using 5 bit path metric representations with the 3-pointer odd SMU clearly outperformed the other designs in terms of speed of operation and reduction in complexity. The reasons for the improved performance are discussed below.

Most of the broad techniques presented in the literature led to efficient implementation of the functions making up the Viterbi algorithm, leading to the conclusion that techniques used for the design of VLSI decoders can be equally well applied to the design of FPGA decoders. There was, however, one noticeable exception to the above – the success of the implementation using serial path metric comparisons showed that increasing parallelism in FPGA designs does not necessarily lead to an increase in the speed of operation of the final implementation. This can be directly attributed to the greater parallelism placing an increased burden not only on the logic resources in the FPGA, but also on the routing resources. As a result the delays introduced through the additional routing may well offset any gains made from the increase in parallelism in the design.

There were, however, still improvements to be made over the design techniques presented in the literature. These improvements apply to both VLSI and FPGA designs. These applied to three areas of the design:

- The first relates to branch metrics and will be discussed in more detail when conclusions are drawn about the performance of the decoder.
- Following on from the previous point it was shown that given a set of branch metrics, equation (3.7) generates a more accurate estimate of the decoder's dynamic range than the methods presented in the literature.
- A new implementation technique was presented that lead to efficient hardware implementations of the R-pointer odd algorithm.

In this thesis it was shown that the improvement in estimating the dynamic range of the decoder leads to a reduction in the number of bits used to represent the path metrics in the decoder with a corresponding decrease in the complexity of the hardware implementation of the decoder. The results of the performance tests clearly show that the new dynamic range is in fact correct, and no loss in performance is incurred through the use of five bits (as opposed to six) to represent the path metrics.

The derivation of the new dynamic range relies on the exploitation of the uniformity and symmetry of the TCM code used for the project. As such the results cannot be held to be true for all convolutional codes, or even all TCM codes. Ungerboek TCM codes, however, are always uniform, so hardware savings should always be possible when implementing such codes. Therefore Ungerboek codes not only provide optimum mapping of $\frac{m}{m+1}$ convolutional codes to their signal set through set partitioning, but can also have associated Viterbi decoders of reduced complexity when compared with other non-uniform TCM or convolutional codes of the same constraint length.

Conversely the derivation of a more efficient implementation of the R-pointer odd family of algorithms for the specific code presented here does not point to the odd family of algorithms yielding more efficient implementations in all cases. The new description of the algorithms, the R-pointer even and R-pointer odd algorithms, does, however, provide a more intuitive and uniform framework for the derivation and comparison of possible hardware implementations of SMUs. From both the designs presented in this thesis, and the equations describing the hardware requirements of the various R-pointer algorithms, it is apparent that the new implementation technique used in conjunction with the R-pointer odd family of algorithms does lead to more efficient implementations in all cases where the TRR is set to one.

In terms of performance, all the designs presented here fell short of the performance of similar systems presented in the literature. The drop in performance, however, could be directly linked to the use of low channel resolution in the receiver. Increasing channel resolution also increases the complexity of the decoder. The four bit channel resolution was held to correspond to the limit in terms of hardware complexity for the devices being used for the more complex designs such as the decoder using six bit path metrics with the 8-pointer even SMU. The positive performance of the lower complexity designs, particularly the design using 5 bit path metrics and the 3-pointer odd SMU, indicates that there is scope to increase the channel resolution using the same hardware platform.

Of specific interest in the performance results was that there seemed to be no degradation due to the use of fixed branch metrics over all signal to noise ratios or the use of three bit branch metrics with four bits of channel resolution. The assumptions made in the choice of branch metrics are therefore held to be valid, namely that the decoder is relatively insensitive the resolution chosen for the branch metrics, or to the exact placement of the branch metrics. The decoder is, however, sensitive the channel resolution used in the receiver.

Along with the techniques used to reduce the required hardware, the reduction in com-

plexity afforded by the use of lower resolution on the branch metrics makes implementations with higher channel resolutions both possible and practical on the hardware platforms used in this thesis.

It is also evident from the performance results that although the error event probability may provide a useful measure to compare TCM schemes, it does not provide an accurate measure of the performance of TCM relative to either the uncoded case or other modulation schemes. The most accurate measure, and the one of interest to the designers of communication equipment, is the probability of bit error in the receiver.

The speed tests showed that Viterbi decoders can be implemented using FPGAs and still handle a high bit rate. Although the speed tests showed that the decoder only just met the project requirements, the 5 bit path metric implementation using serial path metric comparison and the 3-pointer odd SMU operated at a bit rate of 16Mb/s, which is close to the 17.5Mb/s originally designed for, and found to be the limit in the FPGA software simulations. All results point to the failure of the devices to operate any faster being due to excessive noise appearing in the channel at high clock rates. Furthermore all devices used in the implementations had the lowest speed grade in their range. Substituting devices with higher speed grades should easily get the design to meet the 17.5Mb/s bit rate if it is found that the FPGAs failed at the higher clock rates.

Overall the thesis provides an indication of the hardware complexity required to implement Viterbi decoders for any specific coding schemes. The design techniques presented in this thesis along with the actual implementation of the Viterbi decoder for a $(3, \frac{2}{3})$ Ungerboek TCM code offer a benchmark for members of the Digital Modulation Research Group to evaluate the complexity involved in implementing Viterbi decoders in practical communication systems. As such the project can be held to be a success.

Appendix A

Dynamic Range Issues

When calculating the maximum dynamic range of a Viterbi decoder it is important to gauge the effects of noise on the dynamic range. It is important that the dynamic range of the decoder will not be exceeded in the presence of noise. A proof, developed by the author, is presented here that shows that the dynamic range of the decoder will be at a maximum in the absence of noise.

Theorem 1: The decoder's dynamic range will be at a maximum when no noise is present in the channel.

Proof: Consider any two fixed points, A, B, in two dimensional Euclidian space. These two points are taken to be analogous to two points in the signal constellation. A point representing the received signal, C, is also plotted within the same space. The branch metric assigned to points A and B would be directly proportional to the Euclidian distance between point C and point A or B respectively. Denoting the branch metric assigned to point A as $\lambda(A)$ and the branch metric assigned to point B as $\lambda(B)$, the distance between the two branch metrics would be $|\lambda A - \lambda(B)|$.

The dynamic range of the decoder is then the maximum difference between any two path metrics in the decoder. Since the path metrics are defined as the sum of the branch metrics assigned to the transitions in the path, maximising the dynamic range of the decoder is equivalent to maximising the difference between the branch metrics making up the path.

Consider the case where the branch metric $\lambda(B)$ is fixed. This corresponds to the point C describing a circle of radius $\lambda(B)$ around point B of radius $\lambda(B)$. Referring to Figure A.1 the size of $\lambda(A)$ is given by:

$$\lambda(A) = \lambda(B)^2 + D^2 - 2\lambda(B)D\cos(\phi_B) \tag{A.1}$$

Where D denotes the Euclidian distance between points A and B. The difference in the

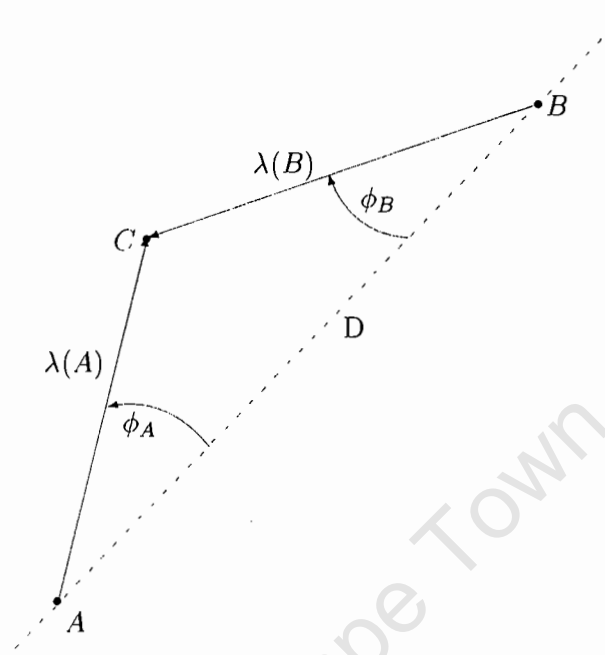


Figure A.1: Illustration of points A, B and C in a two dimensional Euclidian space.

two branch metrics is given by:

$$\Delta_{branch} = \lambda(A) - \lambda(B) \tag{A.2}$$

Substituting equation (A.1) into equation (A.2), differentiating Δ_{branch} with respect to ϕ_B, and setting the result equal to 0 to find the maximum:

$$\phi_B = n\pi \qquad \text{where } n \text{ is any integer} \tag{A.3}$$

n even will correspond to a minima, and n odd will correspond to a maxima. Therefore the difference between the two branch metrics will be at a maximum when ϕ_B is π radians. Setting $\lambda(B)$ equal to zero, this corresponds to point C lying on top of point B. In other words the difference in the branch metrics will be at a maximum when the received signal corresponds to one of the signals assigned to the branches in question. Since the dynamic range between any two paths in the decoder is simply the sum of the differences in the branch metrics making up the two paths, the dynamic range between the two paths will be at a maximum when the received signal sequence corresponds exactly with the signal sequence assigned to one of the paths. This will be the case in the absence of noise. As the maximum dynamic range of the decoder is defined as the maximum difference between any two path metrics, the decoder's dynamic range will be at a maximum in the absence of noise, and the theorem is proved. □

Appendix B

Timing Diagrams and Truth Tables: ACS Circuits

B.1 Truth Tables and Timing Diagrams

The truth table to generate the survivor vector for any ACS unit using the parallel compare implementation may be found in table B.1. Here the six subtractors are labelled from 1 to 6, and perform the comparisons listed in the table. The four paths entering each state are labelled with the letters A, B, C, and D, where A would represent the top-most path entering the state in the trellis diagram in Figure 3.1. The two equations for the bits making up the survivor vector are:

$$O_1 = \bar{2}4\bar{6} + \bar{3}56$$
$$O_0 = 56\bar{3} + 1\bar{4}\bar{5}$$

Min. Met.	1 A-B	2 C-A	3 D-A	4 B-C	5 B-D	6 C-D	Sel. Vect. O_1O_0
A	0	1	1	X	X	X	00
B	1	X	X	0	0	X	01
C	X	0	X	1	X	0	10
D	X	X	0	X	1	1	11

Note: A - B denotes the comparison of path metrics A and B. A "1" denotes B less than A, an "X" denotes the "don't care" state.

Table B.1: Truth table for survivor vector generation in the parallel compare implementation.

In the case of the serial implementation, however, there are only three bits that need to be

Min. Met.	R_{AB} B-A	R_{CD} D-C	R_{ABCD} $R_{CD} - R_{AB}$	Sel. Vect. O_1O_0
A	1	X	1	00
B	0	X	1	01
C	X	1	0	10
D	X	0	0	11

Note: A - B denotes the comparison of path metrics A and B. A "1" denotes B less than A, an "X" denotes the "don't care" state.

Table B.2: Truth table for survivor vector generation in the series compare implementation.

compared. The truth table for generating the survivor vector is illustrated in table B.2. In the table R_{AB} denotes the minimum path metric chosen after the comparison between path metrics A and B. The equations for the survivors are:

$$\begin{aligned} O_1 &= \overline{R_{ABCD}} \\ O_2 &= \overline{R_{AB}} R_{CD} + \overline{R_{CD}}\,\overline{R_{ABCD}} \end{aligned}$$

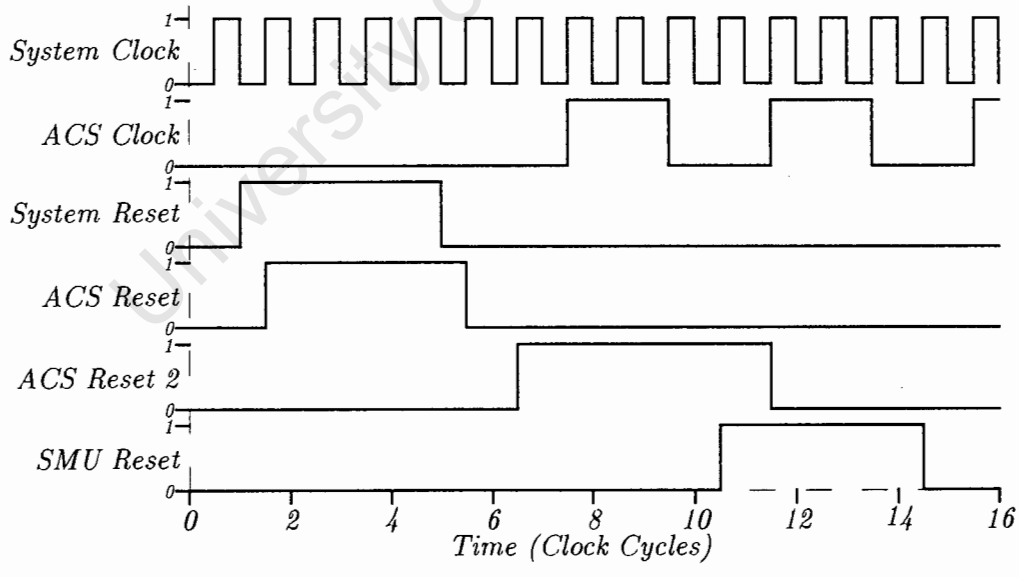


Figure B.1: Timing diagram showing timing of reset pulses inside the ACS circuit.

There are a number of reset circuits within the ACS units, as illustrated in Figure B.1. The "system clock" is the externally applied clock, whereas the "ACS clock" is the

divided down clock used to drive the latches in the ACS circuit. "ACS reset 1" is simply a latched version of the externally applied reset pulse. It is responsible for resetting the counter that drives the internal clock. This helps achieve synchronisation between the various circuits making up the decoder. The "ACS reset 2" signal is the one responsible for performing the synchronous presets. As can be seen in the timing diagram, the counter starts operating as soon as "ACS reset 1" goes low. The last signal, the "SMU reset", is an output of the ACS circuit. It is used to provide a reset to the SMU just before the first data is present at the output of the ACS circuitry.

B.2 Resource Usage

Included below are the tables showing the internal resource usage of each of the design options used in the project. The tables are grouped in pairs, since two devices were needed to implement each ACS unit. The first table listed refers to the FPGA device used to implement the ACS circuits for states 000 to 011 and to generate the reset pulse for the SMU. The second table refers to the FPGA device used to implement the ACS circuits for states 100 to 111. In all cases the FPGA device being used is the XC3064APC84-7.

For the initial design, using parallel compare and six bit path metrics, the resource usage is:

```
Design Statistics and Device Utilization
----------------------------------------

Partitioned Design Utilization Using Part 3064APC84-7

                          No. Used   Max Available   % Used
-------------------------  -------  -------------  ------
Occupied CLBs                  208            224     92%
-------------------------  -------  -------------  ------
Bonded I/O Pins:                47             70     67%
CLB Function Generators: (*)   302            448     67%
CLB Flip Flops:                 18            448      4%
IOB Input Flip Flops:           13            120     10%
IOB Output Flip Flops:          21            120     17%
3-State Buffers:                 0            480      0%
3-State Longlines:               0             32      0%
-------------------------  -------  -------------  ------
(*) Each base F or FGM function counts as two
```

Partitioned Design Utilization Using Part 3064APC84-7

	No. Used	Max Available	% Used
Occupied CLBs	205	224	91%
Bonded I/O Pins:	46	70	65%
CLB Function Generators: (*)	301	448	67%
CLB Flip Flops:	17	448	3%
IOB Input Flip Flops:	13	120	10%
IOB Output Flip Flops:	20	120	16%
3-State Buffers:	0	480	0%
3-State Longlines:	0	32	0%

(*) Each base F or FGM function counts as two

As can be seen, the complexity added by the circuitry required to generate the reset pulse for the SMU is only marginal.

In comparison, the resource usage by the design using five bit path metrics and parallel compare sections is:

Design Statistics and Device Utilization

Partitioned Design Utilization Using Part 3064APC84-7

	No. Used	Max Available	% Used
Occupied CLBs	184	224	82%
Bonded I/O Pins:	43	70	61%
CLB Function Generators: (*)	256	448	57%
CLB Flip Flops:	16	448	3%
IOB Input Flip Flops:	13	120	10%
IOB Output Flip Flops:	19	120	15%
3-State Buffers:	0	480	0%
3-State Longlines:	0	32	0%

(*) Each base F or FGM function counts as two

```
Partitioned Design Utilization Using Part 3064APC84-7

                            No. Used   Max Available   % Used
---------------------------  -------  -------------  ------
Occupied CLBs                   182         224         81%
---------------------------  -------  -------------  ------
Bonded I/O Pins:                 42          70         60%
CLB Function Generators: (*)    255         448         56%
CLB Flip Flops:                  15         448          3%
IOB Input Flip Flops:            13         120         10%
IOB Output Flip Flops:           18         120         15%
3-State Buffers:                  0         480          0%
3-State Longlines:                0          32          0%
---------------------------  -------  -------------  ------
(*) Each base F or FGM function counts as two
```

There is a 10% saving in the number of resources required on both FPGA devices due to the one bit reduction in the number of bits used to represent the path metrics.

The third implementation used five bits to represent the path metrics, and had series compare sections. The resource usage for this implementation was:

```
Design Statistics and Device Utilization
----------------------------------------

Partitioned Design Utilization Using Part 3064APC84-7

                            No. Used   Max Available   % Used
---------------------------  -------  -------------  ------
Occupied CLBs                   134         224         59%
---------------------------  -------  -------------  ------
Bonded I/O Pins:                 43          70         61%
CLB Function Generators: (*)    231         448         51%
CLB Flip Flops:                  16         448          3%
IOB Input Flip Flops:            13         120         10%
IOB Output Flip Flops:           19         120         15%
3-State Buffers:                  0         480          0%
3-State Longlines:                0          32          0%
---------------------------  -------  -------------  ------
(*) Each base F or FGM function counts as two
```

```
Partitioned Design Utilization Using Part 3064APC84-7

                            No. Used   Max Available   % Used
---------------------------  -------  -------------  ------
Occupied CLBs                   133         224         59%
---------------------------  -------  -------------  ------
Bonded I/O Pins:                 42          70         60%
CLB Function Generators: (*)    231         448         51%
CLB Flip Flops:                  15         448          3%
IOB Input Flip Flops:            13         120         10%
IOB Output Flip Flops:           18         120         15%
3-State Buffers:                  0         480          0%
3-State Longlines:                0          32          0%
---------------------------  -------  -------------  ------
(*) Each base F or FGM function counts as two
```

Using series compare sections yields a reduction in used FPGA resources in the order of 22%. In these cases, however, the design tools had greater difficulty in routing the design to meet the timing constraints. The extra delays incurred by the series implementation were in the order of three to four nanoseconds. That the extra delays are so small can be attributed to the lower demand on FPGA resources making the design easier to route.

B.3 Program Code

The program used to calculate the decoder's dynamic range was written using Turbo Pascal version 6. The resultant code is:

```
{*********************************************************************}
{* Program to calculate dynamic range of a Viterbi Decoder          *}
{*********************************************************************}
PROGRAM DynRng {INPUT;OUTPUT};

USES Crt, Dos;

CONST
NoState=7;      {*Largest state number in the code                    *}
Sig = 11;       {*Constant relating mean deviation for Branch Metrics*}
MeanOff = -1;   {*Mean offset for calculating Branch Metrics         *}
StpRng = 4;     {*Exit after StpRng steps if dyn. rng. does not grow *}
```

```
{* Array showing mapping of constellation pts to co-ordinates        *}
{                                        0     1       2       3       }
PMap : ARRAY[0..7,1..2] OF INTEGER = ((8,4),(4,8), (-3,8), (-7,4),
{                                        4       5       6       7    }
                                    (-7,-3), (-3,-7), (4,-7), (8,-3));
{* Array showing mapping of codewords to constellation pts.           *}
{                                  0 1 2 3 4 5 6 7}
TMap : ARRAY[0..7] OF INTEGER = (0,1,2,3,4,5,6,7);


TYPE
Path_Mets=ARRAY[0..NoState] OF INTEGER; {*Stores Path Metrics         *}
Pth_Con=ARRAY[0..NoState] OF BOOLEAN;   {*Determines connectivity     *}


{*********************************************************************}
{* Procedure to initialise all arrays to their starting values.      *}
{*********************************************************************}
PROCEDURE Initialise (InitState: INTEGER;
                      VAR Metrics:Path_Mets; VAR Connect1:Pth_Con);

VAR
  I: INTEGER;

BEGIN
  FOR I:= 0 TO NoState DO
   BEGIN
    Metrics[I]:=0;
    Connect1[I]:=FALSE;
   END;
  Connect1[InitState]:=TRUE;
END;

{*********************************************************************}
{* Function returns branch metric between any two code words.        *}
{*********************************************************************}
FUNCTION GetBranch (Trans1, Trans2: INTEGER):INTEGER;

VAR
 Spce: REAL;
 GDist:INTEGER;


BEGIN
```

```
{* Find distance between two codewords                                    *}
 Spce := SQRT(SQR(PMap[TMap[Trans1],1]-PMap[TMap[Trans2],1])+
         SQR(PMap[TMap[Trans1],2]-PMap[TMap[Trans2],2]))-MeanOff;
 Spce:=1/(SQRT(2*pi)*Sig)*EXP(-1*SQR(Spce)/(2*SQR(sig)));
{* Assign Branch Metric                                                   *}
 GDist :=ROUND(10*(1-sig/(0.4)*Spce));
 IF GDist>7 THEN GDist :=7;
 GetBranch := GDist;
END;

{************************************************************************}
{* Procedure is called recursively to determine dynamic range.          *}
{************************************************************************}
PROCEDURE FindRng (State, Count:INTEGER; PthMetr:Path_Mets;
                   Connect:Pth_Con; VAR Rng: INTEGER);

VAR
I, J, K, NewRng, Temp, NewState, Trans1, Trans2, State2,
Branch, MaxPth, MinPth:Integer;
Connect2: Pth_Con;
NewPth: Path_Mets;

BEGIN
{* From each state there are four transitions                          *}
 For I := 0 TO 3 DO
  BEGIN
{* Initialise connectivity matrix                                      *}
   FOR J := 0 TO 7 DO
    BEGIN
     Connect2[J] := FALSE;
     NewPth[J] := 0;
    END;
{* Calculate codeword being transmitted (i.e. correct path)            *}
   Trans1 := (I SHL 1) XOR State;
{* Calculate next state along path                                     *}
   State2 := ((State SHL 2) AND 7) OR I;
{* Loop until dyn. rng. stops increasing                               *}
   WHILE (Count < StpRng) DO
    BEGIN
{* Initialise range between paths entering a state to zero             *}
      NewRng :=0;
{* Cycle through all 8 states                                          *}
      For J:= 0 TO 7 DO
       BEGIN
```

```
{* Initialise maximum path to 0                                          *}
       MaxPth:=0;
{* Run through all 4 pths entering state J                               *}
       FOR K:= 0 TO 3 DO
        BEGIN
{* Calculate previous state along path being considered                   *}
         Temp := ((J SHR 2) OR ((K SHL 1) AND 7));
{* Only proceed if there is a valid path ending at the previous state*}
         IF (Connect[Temp]) THEN
          BEGIN
{* Include the next state as part of a valid path                        *}
           Connect2[J] := TRUE;
{* Calculate codeword assigned to branch K                               *}
           Trans2 :=(((J AND 3) SHL 1) XOR Temp);
           Trans2 :=((Trans2 AND 1) OR ((Trans2 AND 2) SHL 1) OR
                    ((Trans2 AND 4) SHR 1));
{* Get branch metric                                                     *}
           Branch := GetBranch(Trans1, Trans2);
{* Add to path metric                                                    *}
           Temp:= Branch+PthMetr[Temp];
{* Compare total metric to minimum and maximum metrics entering          *}
{* state J                                                              *}
           IF (K=0) THEN MinPth := Temp
                  ELSE IF (Temp<MinPth) THEN MinPth := Temp;
           IF (Temp>MaxPth) THEN MaxPth := Temp;
          END;
         END;
{* Choose survivor                                                       *}
        NewPth[J] := MinPth;
{* Check for maximum dynamic range at the J states                       *}
        IF (NewRng < (MaxPth -MinPth))
          THEN NewRng := MaxPth -MinPth;
      END;
{* Check to see whether overall dynamic range has been exceeded. If      *}
{* not, increment counter.                                               *}
      IF (NewRng > Rng) THEN Rng := NewRng
                        ELSE Count := Count +1;
{* Call procedure recursively.                                           *}
      FindRng(State2, Count, NewPth, Connect2, Rng);
   END;
 END;
END;
```

```
{*************************************************************************}
{*                          MAIN PROGRAM                                 *}
{*************************************************************************}


VAR
InitState, DinRng, Count: INTEGER;
Metrics:Path_Mets;
Connect1: Pth_Con;

BEGIN
{* Clear the screen and initialise dynamic range                         *}
 ClrScr;
 DinRng :=0;
{* Run through all 8 initial states                                      *}
 FOR InitState := 0 TO NoState DO
  BEGIN
   Count := 0;
{* Initialise all arrays                                                 *}
   Initialise(InitState, Metrics, Connect1);
{* Trace all paths beginning at state InitState                          *}
   FindRng(InitState, Count, Metrics, Connect1, DinRng);
  END;
{* Write the dynamic range to the screen                                 *}
  WRITELN(DinRng)
END.
```

Appendix C

Timing Diagrams and Truth Tables: Survivor Memory Unit

There were two separate implementations of the SMU. The timing diagrams, along with the equations for various functions in the SMU, are grouped according to the design to which they apply.

C.1 The 8-Pointer Even Implementation

The timing diagram for the operations within the memory blocks are shown in Figure C.1. The delays shown are for the final placed and routed design. All delays are measured with reference to the edge on the system clock which drives the change in the associated waveform. The waveforms are all plotted at their source – the delay shown between the clock edge and the change in the address waveform does not include the delay from the counter to the actual RAM modules. Similarly, in the case of the write enable waveform, the delay shown does not include the delay from the circuit that generates the write enable to the actual RAM modules. In the case of the write enable waveform, the waveform still has to be gated before it will reach the RAM module. The delay shown for the input data is the delay due to the input buffer – again, the delay from the input buffer to each RAM module is not included. All delays represent the worst case delays. Signals may propagate faster – but never slower – than shown.

A more general timing diagram showing the relationship between the waveforms entering each memory block and the shift enable waveform is in Figure C.2. The numbers in the address waveform indicate the column within each RAM module on which the operation is being performed. Within any ACS cycle (measured from the beginning of new data appearing in the SMU) two operations need to be performed within each memory block. The traceback read operation is performed first – in this way more time is allowed for the new data to propagate to the input of the RAM modules, making these paths non-critical

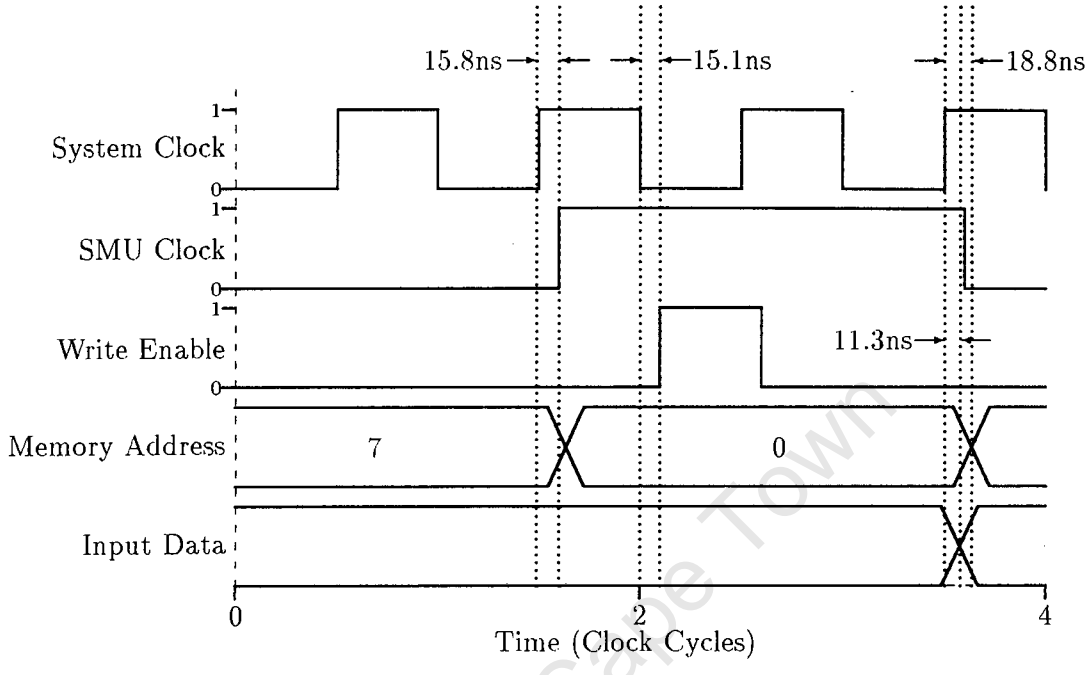


Figure C.1: Timing diagram for waveforms entering memory blocks in the 8-pointer even SMU implementation.

from the point of view of timing. As will be noted, the write front moves from left to right through the memory, whereas the read front moves in the opposite direction. The diagram shows one complete block read/write cycle.

Once the data has been read from memory it is latched before being passed on to the decode read circuit. The pointer which determines in which memory block the decode read operation takes place must take this into account. The equations governing the three bits that make up the pointer are [1]:

$$\begin{aligned} D_2 &= (A_4 \oplus A_7)\, A_5 A_6 + A_7 \overline{(A_5 A_6)} \\ D_1 &= A_4 A_5 \oplus A_6 \\ D_0 &= A_4 \oplus A_5 \end{aligned}$$

Where A_n is bit n from the counter, and D_n is the n^{th} bit of the address passed on to the multiplexer in the bit order reversing circuitry. The equation for the write enable pulse is:

$$\text{Write Enable} = \overline{A}_0 \overline{Clock} A_1$$

[1] The bits are numbered with a subscript of 0 through N. 0 denotes the least significant bit, whereas N denotes the most significant bit.

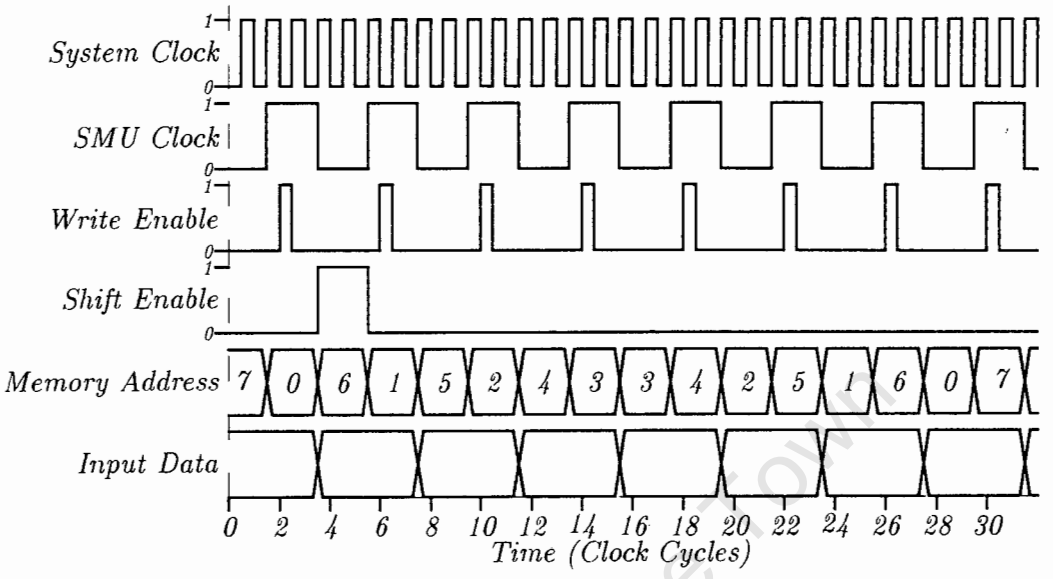


Figure C.2: General timing diagram showing relationship between shift enable and other waveforms in the memory blocks.

Where *Clock* denotes the system clock waveform. The equation for the shift enable pulse is:

$$\text{Shift Enable} = \overline{A_4 + A_3 + A_1 + \overline{A}_2}$$

Where, as before, A_n denotes the n^{th} bit from the counter.

C.2 The 3-Pointer Odd Implementation

In the 3-pointer odd algorithm the timing of the signals entering the bit order reversing circuit, as well as the timing of the signals entering the memory blocks, is critical. As such a general timing diagram showing the sequence of all the signals entering the memory blocks is illustrated in Figure C.3. This is followed by the equations used to generate the various signals in the bit order reversing circuit. Lastly timing diagrams for the critical signals in the memory blocks and the bit order reversing circuit are presented in Figure C.6 and Figure C.4 respectively.

Figure C.3 spans one complete block write operation. All waveforms shown are periodic with a period equal to the span of the diagram. The "Reverse Enable" pulse is used as a clock enable on a shift register which stores a vector, "Reverse Vector", which dictates the direction of propagation of all read and write fronts through each of the five memory blocks. The "Shift Enable" pulse is asserted only at the end of a complete block read,

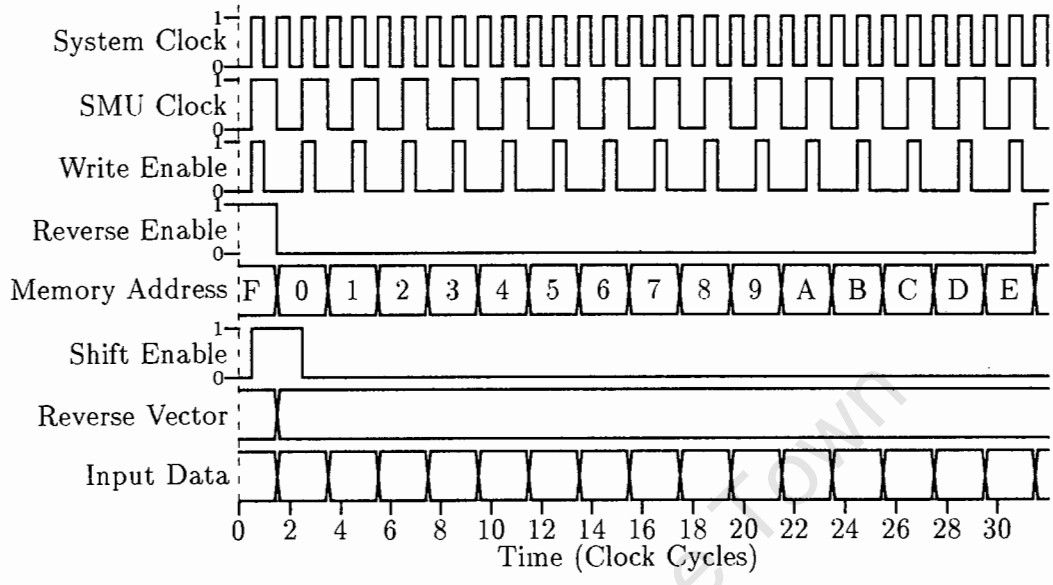


Figure C.3: Timing diagram showing the sequence of all waveforms entering the memory blocks in the 3-pointer odd implementation.

and allows the contents of the pointers assigned to each memory block to be passed to the next memory block in the chain (see Figure 4.7).

The bit order reversing circuit requires equivalents of many of the signals used in the memory blocks which are illustrated in Figure C.3. Only two signals were used to generate the write enable pulse: the system clock, FC, and the SMU clock, SC. The later clock signal runs at half the rate of the system clock, and is generated internally in the SMU using a counter. As such the edges of the two clocks will not be perfectly synchronised (see Figure C.4), and feedback had to be used to ensure that the write enable pulse is generated over the correct portion of the clock cycle. The equations for the write enable pulse are:

$$\begin{aligned} B &= (FC + SC) \cdot (SC + B) \\ \text{Write Enable} &= \overline{SC} \cdot FC \cdot B \end{aligned}$$

The address used for the RAM block in the bit order reversing circuit needs to take into account the pipelining in the decode read operation. The equation for the new address bits are:

$$\begin{aligned} RA_0 &= SFT \oplus A_1 \\ RA_1 &= SFT \oplus A_2 \\ RA_2 &= A_2 \end{aligned}$$

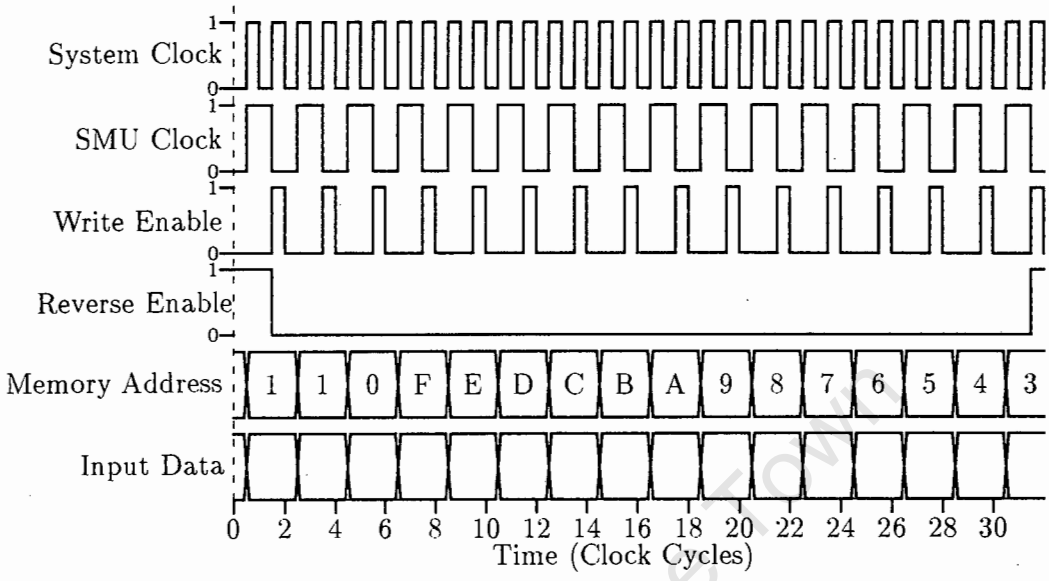


Figure C.4: Timing diagram showing the sequence of all waveforms in the bit order reversing circuit of the 3-pointer odd implementation.

$$RA_3 = (SFT \oplus A_4)A_3 + \overline{A}_3 A_4$$

Where the boolean variable SFT determines the whether the address will increment or decrement. SFT is generated by a toggle flip-flop which toggles every time the following equation is true:

$$\text{Toggle} = A_1\overline{A}_2\overline{A}_3\overline{A}_4$$

The signals described above are shown in Figure C.4 where their sequence is shown relative to the system clock. The diagram spans one complete block read/write cycle. The memory address starts at 0, goes to 1, and remains at 1 over the next clock cycle due to the assertion of the reverse enable pulse. The direction of the read/write operations are then reversed and the address sequence is: $0, F, E, \cdots, 3, 2, 2, 3, \cdots, F, 0, 1, 1, 0, \cdots$. In the diagram, "Input Data" refers to the data coming from the decode read operation, and is shown at the input of the RAM module.

Unlike the signals which are fed to the memory blocks, the signals in the bit order reversing circuit have a low fan-out. The delays shown in Figure C.5 will be close to the true delays in the circuit even though path delays to the actual RAM module are not included. As with the timing diagram for the signals entering the memory blocks, all delays are shown between each waveform and the clock edge that generates a change in that particular waveform. Again, all timing delays are worse case delays: signals may propagate faster on the actual device.

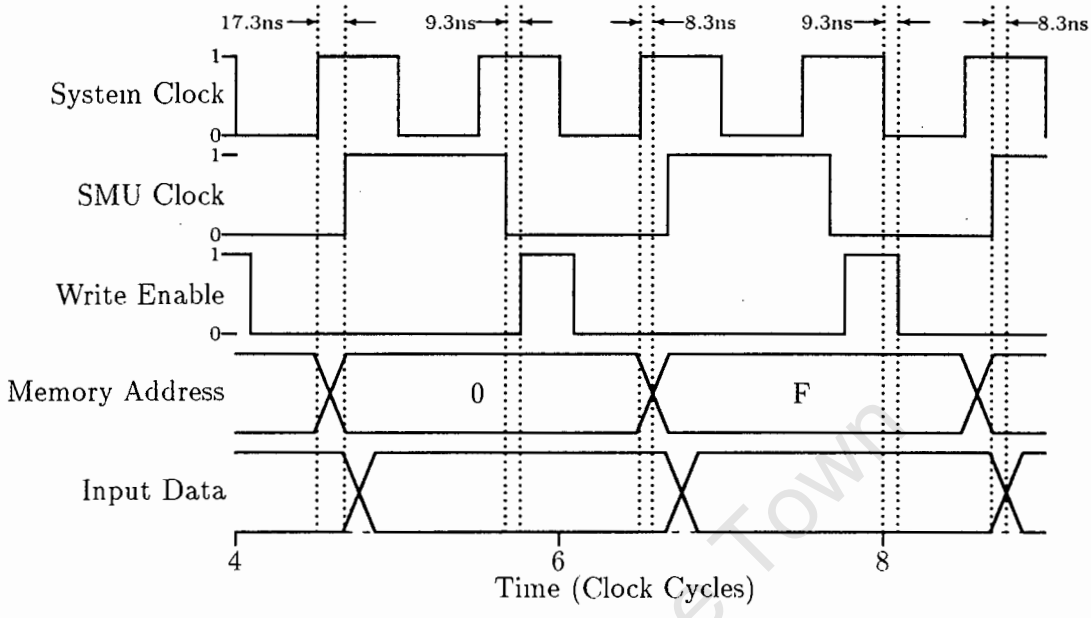


Figure C.5: Timing diagram for all critical waveforms in the bit order reversing circuit of the 3-pointer odd implementation.

The equivalent timing diagram for the critical waveforms entering each memory block in the SMU can be found in Figure C.6. All the signals shown need to be routed to all five of the memory blocks. As a result there will be variable delays between the circuits that generate the waveforms and the memory blocks in which they are used. Only the delays between the clock edges and the output of the logic circuit that generate the waveforms are shown, since these will be the most significant delay in the system. Once again, the write enable pulse has to be gated before it is passed on to each memory block, so allowances must be made for a reasonably large delay before the write enable pulse reaches the memory block under scrutiny. The delays shown are for the final placed and routed design, where simulation was used to verify that the circuit met the required timing specifications.

The implementation requires five memory blocks to make up the SMU, so the write pointer has to count modulo 5. The counter designed to house the write pointer was built as a finite state machine made up of toggle flip-flops, whose state table is shown in table C.1. The equations describing the inputs to three flip-flops are shown in equations (C.1) to (C.3), where $T_0 \cdots T_2$ describe the inputs to the three toggle flip-flops whose corresponding outputs, $Q_2Q_1Q_0$, form the write pointer. C_{in} is a count enable input to the counter. In this particular case C_{in} would go high if bits 0 through 4 of the 8 bit counter were all high.

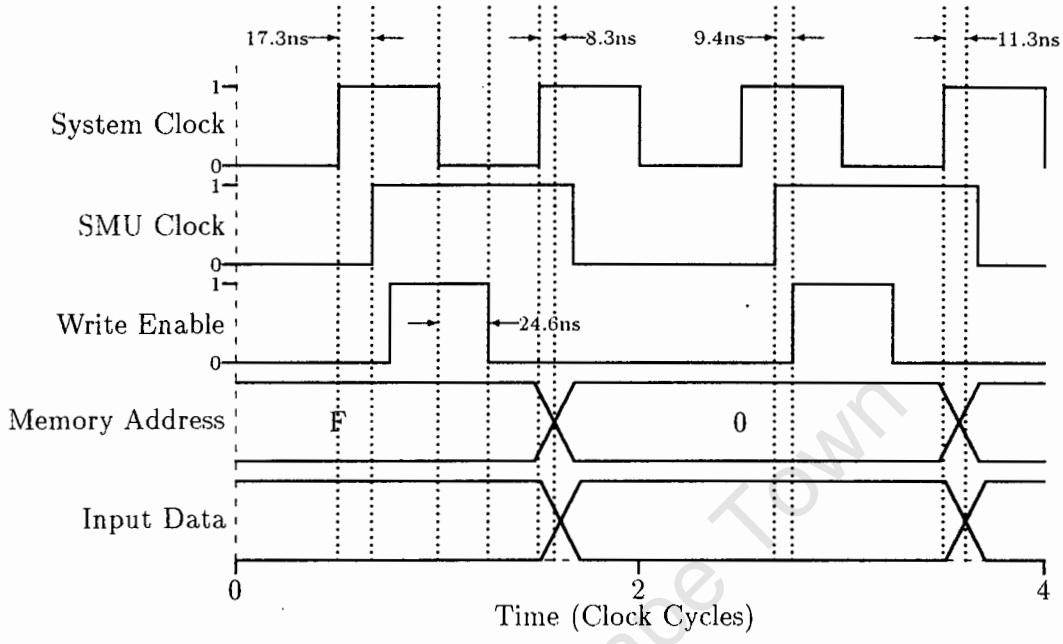


Figure C.6: Timing diagram for all critical waveforms used in the memory blocks in the 3-pointer odd implementation.

$$T_0 = C_{in}\overline{Q}_2 \quad \text{(C.1)}$$

$$T_1 = C_{in}Q_0\overline{Q}_2 \quad \text{(C.2)}$$

$$T_2 = (Q_0Q_1 + Q_2)\,C_{in} \quad \text{(C.3)}$$

All the control signals used in the SMU are formed from the output of the 8 bit counter. Perhaps the most critical of these control signals is the write enable waveform which controls the time at which new data is written into the RAM within each memory block.

T	Q_t	Q_{t+1}
0	0	0
0	1	1
1	0	1
1	1	0

Table C.1: State table for toggle flip-flop.

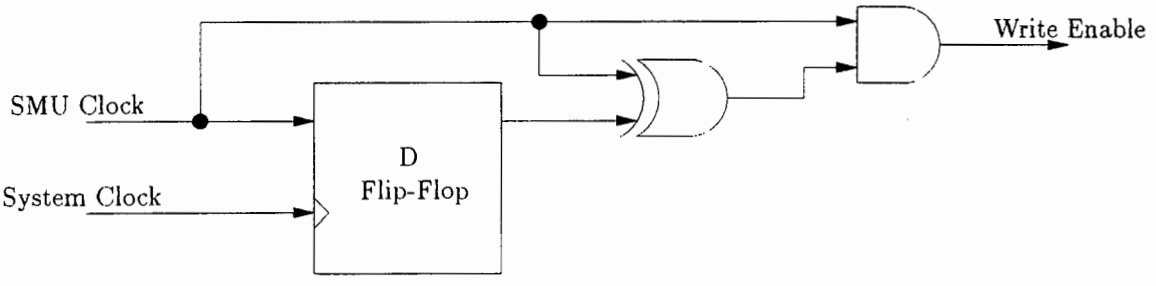


Figure C.7: Circuit used to generate the write enable waveform in the 3-pointer odd implementation.

Due to the requirement for both setup and hold times, the write enable pulse has to be generated from half a cycle of the system clock. The correct half cycle is chosen by using the least significant bit of the counter. The problem here is that in the final design there will be variable propagation delays between the two signals. To ensure that there are no glitches in the write enable pulse the circuit illustrated in Figure C.7 was used. The write enable pulse needs to be generated during the first half of the positive half cycle of the internal SMU clock (see Figure C.6). The write enable pulse is sent high when the SMU clock signal goes high, and is bought low again when the system clock latches the SMU clock into a D flip-flop. Due to the delay in the flip-flop and the logic following the flip-flop, the write enable pulse can be guaranteed to be wider than the required 7ns. Since the write enable pulse is effectively generated from only the SMU clock, it can be guaranteed to be glitch free.

The timing of the shift enable pulse is far less critical. It is generated using bits 1 through 4 (bit 0 is the least significant bit) of the 8 bit counter, and is described by the equation:

$$\text{Shift Enable} = A_4 A_3 A_2 A_1$$

The shift enable pulse is used to drive the clock enable on the shift register which houses the vector that determines the direction of the operations taking place in each of the memory blocks. It is delayed by an additional two clock cycles through two flip-flops before being used to drive the multiplexer in each memory block which selects the read pointer that determines the contents to be read from memory.

The reverse enable pulse for the bit-order reversing circuit is generated by the following equation:

$$\text{Reverse Enable} = A_1 \overline{A}_2 \overline{A}_3 \overline{A}_4$$

Where the reverse enable pulse is used as a clock enable on the toggle flip-flop which determines whether the memory address in the bit order reversing circuit will be incremented or decremented each SMU clock cycle.

C.3 Resource Usage

Both SMU implementations were realised using the XC4005APC84-6 XILINX FPGA device. The first table presented below lists the resources used for the 8-pointer even implementation, and the second table lists the resources used for the 3-pointer odd implementation.

The 8-Pointer Even Implementation

Partitioned Design Utilization Using Part 4005APC84-6

	No. Used	Max Available	% Used
Occupied CLBs	185	196	94%
Packed CLBs	93	196	47%
Bonded I/O Pins:	20	61	32%
F and G Function Generators:	158	392	40%
H Function Generators:	38	196	19%
CLB Flip Flops:	186	392	47%
IOB Input Flip Flops:	17	112	15%
IOB Output Flip Flops:	2	112	1%
Memory Write Controls:	64	196	32%
3-State Buffers:	0	448	0%
3-State Half Longlines:	0	56	0%
Edge Decode Inputs:	0	168	0%
Edge Decode Half Longlines:	0	32	0%

The 3-Pointer Odd Implementation

Partitioned Design Utilization Using Part 4005APC84-6

	No. Used	Max Available	% Used
Occupied CLBs	155	196	79%
Packed CLBs	61	196	31%
Bonded I/O Pins:	20	61	32%
F and G Function Generators:	122	392	31%
H Function Generators:	24	196	12%
CLB Flip Flops:	122	392	31%
IOB Input Flip Flops:	17	112	15%

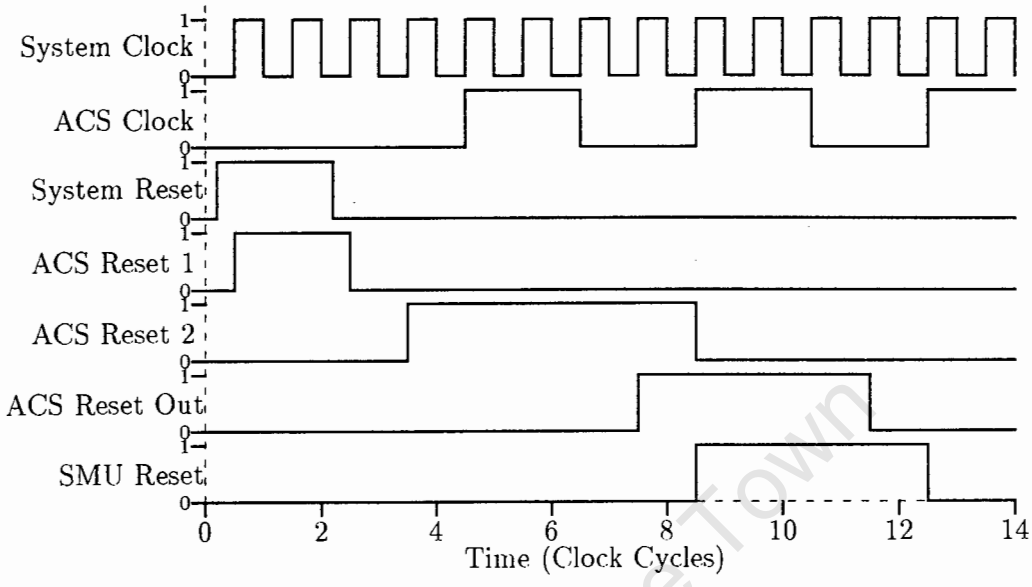


Figure C.8: A timing diagram showing the relationships between the various reset pulses within the Viterbi decoder.

IOB Output Flip Flops:	2	112	1%
Memory Write Controls:	41	196	20%
3-State Buffers:	0	448	0%
3-State Half Longlines:	0	56	0%
Edge Decode Inputs:	0	168	0%
Edge Decode Half Longlines:	0	32	0%

C.4 Reset Waveforms

Once implemented the Viterbi decoder will span three separate FPGA devices. To ensure proper operation of the decoder these three devices need to be properly synchronised with one another. As such the timing of the reset pulses fed to each unit is important. A timing diagram of the various reset waveforms for both the ACS unit and the SMU is shown in Figure C.8.

In the diagram, the first two waveforms are the system clock and the ACS clock respectively. The system clock is common to all the devices making up the decoder. The ACS clock is the internal clock of the ACS unit. It is formed by dividing down the system clock using a binary counter. "System Reset" refers to the externally applied reset signal. "ACS Reset 1" is a latched version of the system reset waveform. It is used to reset the counter within the ACS unit. "ACS Reset 2" is a delayed reset pulse (it stretches over at

least one ACS clock cycle) used to generate the synchronous presets in the latches used to store the path metrics. "ACS Reset Out" is the reset output from the ACS unit used to drive the reset input on the SMU. "SMU Reset" shows the ACS reset latched into the SMU, where it is used to reset the 8 bit counter.

Bibliography

[1] G. Ungerboeck, "Channel coding with multilevel/phase signals," *IEEE Transactions on Information Theory*, vol. IT-28, pp. 56–67, January 1982.

[2] A. J Viterbi, "Error bounds for convolutional codes and an asymptotically optimum decoding algorithm," *IEEE Transactions on Information Theory*, vol. IT-13, pp. 260–269, April 1967.

[3] E. Biglieri, D. Divsalar, P. J. Mclane, and M. K. Simon, *Introduction to Trellis-Coded Modulation with Applications*, MacMillan, New York, 1991.

[4] T. K. Truong, M. Shih, I. S. Reed, and E. H. Satorius, "A VLSI design for a trace-back Viterbi decoder," *IEEE Transactions on Communications*, vol. 40, no. 3, pp. 616–624, March 1992.

[5] E. Biglieri and P. J. McLane, "Uniform distance and error probability properties of TCM schemes," *IEEE Transactions on Communications*, vol. 39, no. 1, pp. 41–52, January 1991.

[6] G. D. Forney Jr., "The Viterbi algorithm," *Procedures of the IEEE*, vol. 61, no. 3, pp. 268–278, March 1973.

[7] A. J. Viterbi, "Convolutional codes and their performance in communication systems," *IEEE Transactions on Communications*, vol. COM-19, no. 5, pp. 751–772, October 1971.

[8] G. Feygin and P. G. Gulak, "Architectural tradeoffs for survivor sequence memory management in Viterbi decoders," *IEEE Transactions on Communications*, vol. 41, no. 3, pp. 425–429, March 1993.

[9] R. Cypher and C. B. Shung, "Generalized trace back techniques for survivor memory management in the Viterbi algorithm," *GLOBECOM*, pp. 1318–1322, 1990.

[10] G. Fettweiss, "Algebraic survivor memory management design for Viterbi detectors," *IEEE Transactions on Communications*, vol. 43, no. 9, pp. 2458–2463, September 1995.

[11] A. P. Hekstra, "An alternative to metric rescaling in Viterbi decoders," *IEEE Transactions on Communications*, vol. 37, no. 11, pp. 1220–1222, November 1989.

[12] A. J. Viterbi and J. K. Omura, *Principles of Digital Communication and Coding*, McGraw-Hill, New York, 1979.

[13] J. G. Dunham and K. H. Tzou, "Performance bounds for convolutional codes with digital Viterbi decoders in Gaussian noise," *IEEE Transactions on Communications*, vol. COM-31, no. 10, pp. 1124–1132, October 1983.

[14] P. J. Black and T. H. Meng, "A 140-Mb/s, 32-state, radix-4 Viterbi decoder," *IEEE Journal of Solid-State Circuits*, vol. 27, no. 12, pp. 1877–1885, December 1992.

[15] J. A. Heller and I. W. Jacobs, "Viterbi decoding for satellite and space communication," *IEEE Transactions on Communication Technology*, vol. COM-19, no. 5, pp. 835–848, October 1971.

[16] G. Clark and J. B. Cain, *Error-Correction Coding for Digital Communications*, Plenum, New York, 1981.

[17] C. M. Rader, "Memory management in a Viterbi decoder," *IEEE Transactions on Communications*, vol. COM-29, no. 9, pp. 1399–1401, September 1981.

[18] H. A. Bustamante, I. Kang, C. Nguyen, and R. E. Peile, "Stanford Telecom design of a convolutional decoder," in *MILCOM 89*, Boston MA, October 1989, pp. 171–178.

[19] C. B. Shung, H. Ling, R. Cypher, P. H. Siegel, and H. K. Thapar, "Area-efficient architectures for the Viterbi Algorithm–Part I: Theory," *IEEE Transactions on Communications*, vol. 41, no. 4, pp. 636–644, April 1993, and "Area-efficient architectures for the Viterbi Algorithm–Part II: Applications," *IEEE Transactions on Communications*, vol. 41, no. 5, pp. 802–807, May 1993.

[20] G. Feygin, P. G. Gulak, and P. Chow, "A multiprocessor architecture for Viterbi Decoders with linear speedup," *IEEE Transactions on Signal Processing*, vol. 41, no. 9, pp. 2907–2917, September 1993.

[21] G. Fettweis and H. Meyr, "Parallel Viterbi algorithm implementation; breaking the ACS-bottleneck," *IEEE Transactions on Communications*, vol. 37, no. 8, pp. 785–789, August 1989.

[22] P. J. Golda, "Hardware implementation of a Viterbi decoder," Tech. Rep., Undergraduate Thesis, University of Cape Town, 1993.

[23] XILINX Inc., *The Programmable Logic Data Book*, 1994.

[24] XILINX Inc., *XACT Libraries Guide*, 1994.

[25] F. G. Stremler, *Introduction to Communication Systems*, Addison–Wesley, USA, third edition, 1990.

[26] P. Horowitz and W. Hill, *The Art of Electronics*, Cambridge University Press, Cambridge, second edition, 1989.

[27] Peter Alfke, "Efficient shift registers, LFSR counters, and long pseudo-random sequence generators," Application Note XAPP 052, XILINX Inc., July 1996.

[28] S. Haykin, *An introduction to analog & digital communications*, John Wiley & Sons, USA, 1989.

[29] S. Haykin, *Digital Communications*, John Wiley & Sons, 1988.

[30] A. Benzakein, "Design and testing of a real time simulation for Trellis Coded Modulation," MSc thesis, Department of Electrical Engineering, University of Cape Town, 1995.