A Reconfigurable Decoder Architecture for Wireless LAN and Cellular Systems

by

Kanu Chadha

A Thesis Submitted
in Partial Fulfillment of the Requirements for the Degree

Master of Science

Approved, Thesis Committee:

Dr. Joseph Cavallaro, Chair
Associate Professor
Electrical and Computer Engineering

Dr. Behnaam Aazhang
Professor
Electrical and Computer Engineering

Dr. Scott Rixner
Assistant Professor
Electrical and Computer Engineering

Houston, Texas
April, 2001
A Reconfigurable Decoder Architecture for Wireless LAN and Cellular Systems

Kanu Chadha

Abstract

The rapid growth of wireless communications has led to the demand for communication devices which can support multiple standards and have the capability of switching from one to another on-the-fly. For instance, a device which could support the Wireless Local Area Network (LAN) and Wideband Code Division Multiple Access (WCDMA) standards would enable seamless communications in the indoor LAN network as well as the outdoor cellular environment. A key challenge involved in the building of such a communications device is the design of a flexible hardware architecture that can dynamically reconfigure itself to run different algorithms as required to support the different standards. In particular, the Viterbi decoding algorithm is used at the receiver of both WCDMA and WLAN systems to decode the convolutionally encoded data. The difference lies in the encoding parameters such as the constraint length, code rate and generator polynomials.

In this thesis, we propose an architecture for a reconfigurable Viterbi decoder that can dynamically adapt to changes in the encoding parameters. The proposed architecture is implemented on a Field Programmable Gate Array (FPGA) device. The proposed architecture can be used to realize a Viterbi decoder which can support constraint lengths from 3 to 9 and rates 1/2 and 1/3.
Acknowledgments

I would like to express my gratitude to my mentors and thesis advisors: Dr. Joseph Cavallaro, Dr. Behnaam Aazhang, and Dr. Scott Rixner for their excellent guidance and encouragement throughout the project.

I am thankful to all my friends at Rice, especially Bryan, Frank, Sridhar, Kiran, SK, Shri, and Mani for their friendly support and encouragement.
Contents

Abstract ii
Acknowledgments iii
List of Illustrations vii
List of Tables viii

1 Introduction 1
1.1 Motivation ................................................. 1
1.2 Contribution ............................................... 2
1.3 Overview .................................................. 4

2 Background 5
2.1 CDMA ....................................................... 5
2.2 WLAN/OFDM .............................................. 9
2.3 Convolution Encoding ..................................... 13
2.4 The Viterbi Decoding Algorithm ....................... 15
2.5 Summary ................................................... 20

3 Architectures for the Viterbi Decoder 22
3.1 Main Functional Units of the Viterbi Decoder .......... 22
   3.1.1 Quantizer ............................................. 24
   3.1.2 The Block Synchronizer ............................ 24
   3.1.3 Branch Metric Unit ................................. 25
3.1.4 Add-Compare-Select Unit ........................................ 26
3.1.5 Survivor Management Unit ....................................... 28
3.2 Architecture Realizations for the Viterbi Decoder ............... 30
  3.2.1 Uniprocessor ..................................................... 30
  3.2.2 Parallel .......................................................... 31
  3.2.3 General Cascade .................................................. 32
  3.2.4 Systolic .......................................................... 32

4 A Reconfigurable Decoder Architecture ............................ 34
  4.1 Previous work ....................................................... 35
  4.2 The Proposed Architecture ....................................... 36
    4.2.1 BMU Core ...................................................... 39
    4.2.2 ACS Core ...................................................... 42
    4.2.3 Survivor Management ......................................... 46
  4.3 Summary ............................................................. 48

5 Prototype Implementation and Results .............................. 50
  5.1 FPGA Development Environment ................................... 50
  5.2 Prototype Implementation ......................................... 51
    5.2.1 System Partition ............................................... 52
  5.3 Implementation Results ............................................ 55
    5.3.1 Area/Timing Analysis ........................................... 56
    5.3.2 Comparison with a DSP implementation ....................... 58

6 Conclusions and Future Work ......................................... 60
  6.1 Conclusions ........................................................ 60
6.2 Future Work ................................................. 61

Bibliography ................................................. 63
Illustrations

2.1 The Physical Layer in a CDMA System .......................... 6
2.2 An OFDM WLAN System ................................. 11
2.3 A $K = 3$, Rate 1/3 Encoder ............................... 13
2.4 The State Transition Graph .................................. 15
2.5 The Trellis Diagram ........................................ 16
2.6 The Viterbi Decoding Algorithm .............................. 19

3.1 A Practical Viterbi Decoder ................................... 23
3.2 One stage of a $K = 3$ encoder - two butterfly elements ........... 27
3.3 An Add-Compare-Select Unit ................................. 27

4.1 The Viterbi Core ................................................ 38
4.2 A Typical Butterfly Computation ................................ 40
4.3 The Trellis for $K = 3$ and $K = 4$ ........................... 41
4.4 The Interconnection Network for $K = 3$ and $K = 4$ (The italicized
    connections change for $K = 3$ and $K = 4$) ............... 45

5.1 The Viterbi Decoder ......................................... 52
5.2 The Viterbi Decoder Implementation .......................... 53
Tables

5.1 Reconfigurable Vs Fixed: Speed/Area ................. 56
5.2 Fixed Implementation: Area/Speed .................. 57
5.3 DSP Benchmarks: WLAN/WCDMA .................. 59
5.4 FPGA Benchmarks: WLAN/WCDMA .................. 59
Chapter 1

Introduction

1.1 Motivation

There has been rapid growth in the field of wireless communications over the last decade. For example, a cellular system has evolved from the traditional analog, voice-only network into an integrated data and digital voice system. Similarly, wireless local area networks have transformed from bulky, costly, niche technologies into high performance systems operating at near-wire speeds. It is natural to expect the convergence of these wireless technologies to produce a seamless, ubiquitous network which provides anytime, anywhere communication to its users. Users of such a network could then potentially always remain connected to the network through a single communications device which has the ability to run different standards for wireless communications. For this to happen there are a lot of challenges to be overcome at the different layers of the network, right from the radio interface to the application level layer. For example, at the radio interface, the device would have to monitor the different RF signals and intelligently switch to the appropriate standard. There are several other issues such as Quality of Service management and network/device cooperation for seamless inter-standard handoff at the other layers and they become all the more exciting because the device is limited in its computational power and size. In this thesis, we have focused on the issues at the lowest layer i.e. the physical layer, specifically targeted towards the implementation of the physical layer communication algorithms.
The problem of supporting multiple standards at the physical layer translates to that of being capable of running complex signal processing algorithms for different wireless standards on a single device. Herein lie the fundamental design choices of computational power, device size and the quality of communication. In other words, the more hardware resources you devote to running computationally intensive signal processing algorithms, the better the bit error rate or the quality of communication you get, but at the same time your device consumes more power and is bulkier. Therefore, if we think of running different algorithms on a single device, appropriating independent hardware resources for each standard would seem wasteful and impractical. On the other hand, reusing the same hardware to handle different communication algorithms is a much more efficient alternative. It is this option of reusable or reconfigurable hardware that we are interested in.

1.2 Contribution

In this thesis, we chose to solve the problem of reconfigurable communications processing in the context of the Viterbi decoding algorithm. The Viterbi algorithm is a very well known technique for decoding of convolution codes and is an essential part of the WCDMA and WLAN receivers. Convolution codes are extremely powerful for correcting random errors or burst errors and the encoders are such that a block of coded bits depends upon some number of previous bits. This inherent memory in convolution codes is the constraint length of the code. It is the constraint length and other parameters such as coding rate and generator polynomials which are different for the decoders that are used in WCDMA and WLAN systems. So, the goal of a reconfigurable Viterbi decoder would be to have the ability to dynamically adapt to changes in the constraint length, code rates and other parameters.
There has been research in the field of convolution codes and Viterbi decoding in general but little work has been done on the aspect of dynamic reconfigurability of the Viterbi decoder. As part of this thesis, we develop a reconfigurable architecture for the Viterbi decoder which can switch between different code configurations on-the-fly. A Digital Signal Processor (DSP) would seem to be the obvious choice for a reconfigurable solution for the decoder. However, We have chosen an FPGA implementation over a DSP based implementation because to achieve the high data rate requirements of WCDMA and WLAN, significant processing needs to be done in hardware. The data rates for WLAN are in the range 6-54 Mbps whereas the most powerful current DSPs can attain maximum decoding rates of just upto 2 Mbps. Moreover, we have used a parallel processing approach to the Viterbi algorithm to achieve the high data rates mentioned above and such a hardware intensive approach does not lend itself well to the uniprocessor architecture of a DSP. We have synthesized and implemented the reconfigurable decoder architecture on a Xilinx XCV800 FPGA. The proposed architecture is capable of supporting upto the 24 Mbps data rate requirements for 802.11a WLAN and switching between code rates 1/2-1/3 and constraint lengths 3-9.

Thus, We have broken the problem of reconfigurable communications hardware into three parts. First, understanding the communication algorithms to be employed at the physical layer and identifying some commonality between them which can be exploited at the hardware level. Second, designing a flexible hardware architecture which is based on the common parts identified above. And third, making the hardware dynamically reconfigurable i.e. it should be capable of switching from one algorithm to another reasonably fast.
1.3 Overview

The rest of the thesis is organized as follows: In chapter 2, we shall first briefly study the WLAN and the WCDMA systems and build a better understanding of the various algorithms that are used at the physical layer of these two standards. We then describe convolution coding and Viterbi decoding in detail and look at the specific codes that are used in the WCDMA and the WLAN systems.

Chapter 3 describes the Viterbi algorithm from an implementation perspective. Some well known architectures for the Viterbi decoder are presented and the typical area-time tradeoffs associated with different implementations examined.

In Chapter 4, we present the new architecture that we have proposed for a re-configurable Viterbi decoder. The detailed design of the various components of the decoder is described.

In Chapter 5, we give the simulation results and performance measures for the implementation of the Viterbi decoder on the FPGA and compare them with a DSP implementation. Finally, we conclude with some directions for future work in chapter 6.
Chapter 2

Background

A typical communication system consists of a transmitter, a channel and a receiver at the physical layer. There are many users modulating their bit sequences and sending the modulated bits across the channel using multiple access techniques. The signal from the channel is then processed at the receiver to recover the information bits. CDMA or Code Division Multiple Access is one such multiple access technique being proposed for use in Third Generation wireless cellular systems. Similarly, OFDM or Orthogonal Frequency Division Multiplexing is a special form of modulation that is being adopted in the 802.11b standard for high data rate wireless LANs. In the following sections we shall describe these two technologies, understand the basic concepts of Viterbi decoding, and motivate the need for a dynamically reconfigurable Viterbi decoder.

2.1 CDMA

Code Division Multiple access is a relatively new multiple access technology that has revolutionized the cellular concept by its ability to provide superior capacity and performance over traditional multiple access technologies like frequency and time division multiplexing. CDMA is basically a variant of spread spectrum, a form of digital communication techniques that has been used in military communications for years. The core principle behind spread spectrum is the use of noise like carrier waves,
and as the name implies, bandwidths much wider than required for simple point to point communication at the same data rate.

![Diagram of CDMA system](image)

**Figure 2.1** The Physical Layer in a CDMA System

In CDMA, multiple access to different users is provided by assigning a signature waveform or “code” to each user. Direct Sequence Spread Spectrum CDMA (DSSS-CDMA) is one flavor of CDMA in which each user’s “code” is a unique pseudo random spreading sequence. As Figure 2.1 shows, the transmission chain begins with encoding of the input bit sequence. First, source coding is done where in the information is compressed so as to get rid of the redundancy in the data. Then channel coding is done to combat wireless channels which are highly susceptible to errors due to interference from other users, thermal noise, fading and other effects. Convolution coding, which shall be described in detail later in this chapter is a prime example of a channel code. Channel coding essentially adds redundancy to data in a controlled manner so that
the effects of the channel can be undone at the receiver to retrieve the information bits. Channel coding is followed by spreading, which means multiplication of the coded data with the user’s spreading sequence.

A spreading sequence typically consists of a combination of +1/-1’s which is chosen such that it satisfies some unique auto/cross correlation properties. The length of the spreading sequence is fixed and is known as the “spreading gain”. In the time domain, each bit is spread into a number of chips equal to the spreading gain and correspondingly, the frequency spectrum of the signal also expands by the same factor. This expansion in the frequency domain makes DS-CDMA signals resistant to narrow band interfering noise. The unique auto/cross correlation properties of spreading sequences are such that as far as possible, the spreading sequences for different users are orthogonal to each other. This property of the user sequences is used at the receiver in the detection process. After the spreading, the chips so obtained from the spread bits are modulated by the RF carrier using a digital modulation technique such as binary phase shift keying. This intermediate signal, composed of superimposed chips of different users entering the channel is the baseband signal. This composite signal then suffers from channel effects like attenuation, multi-path and fading. Multi-path is the result of superposition of signals bouncing off various obstacles, suffering varying attenuations and delays and being superimposed at the receiver. Fading is the attenuation of different paths with time. It results from the relative motion of the transmitter and receiver (Doppler effect) or of the movement of the reflectors in the path of the radio signals.

The front-end of the receiver is the demodulator and chip matched filter. The demodulator retrieves the baseband signal from the radio signal coming from the channel for further processing. The continuous time signal from the channel is converted to a discrete time signal by sampling the output of a filter matched to the chip
waveform. The chip matched filtering is an integration and dump of the baseband signal correlated with the chip waveform, over a bit duration, with the output held at the end of the chip waveform. The next step before we are able to detect user data bits is to estimate the channel parameters. The parameters we are interested in are the delay, magnitude and phase change for all the users. These are required to correctly find the bit boundaries for detection. The initial part of the channel estimate involves an acquisition phase, where the first estimate of the channel is obtained.

The rest of the transmission involves tracking, where the channel parameters are continuously updated. Tracking is required for a channel with a high Doppler spread. The detector in a typical multi-user detection scheme has the task of picking out the data bits for a particular user out of the composite signal. It essentially involves solving the following equation 2.1 for the original bits $b_i$, given the received information vector $r_i$. The matrices $A$ and $W$ are obtained from the channel estimation process and $A$ captures the effect of the channel delays while $W$ represents the attenuation factors introduced by the channel.

$$r_i = AWb_i + n_i$$

(2.1)

Where $r_i$ is a discrete observation vector formed at bit $i$ by sampling the received signal and collecting $N$ chip matched filter outputs. $A$ is the $N$ by $2K_{user}$ matrix of signal vectors, which depends on spreading codes and delays of each of the users; $W$ is the $2K_{user}$ by $2K_{user}$ diagonal matrix of complex amplitudes; $b_i$ is the $2K_{user}$ by 1 vector of the $K_{user}$ users’ previous and current data bits; $n_i$ is the Gaussian noise; $N$ is the spreading gain and $K_{user}$ the total number of users. There are several algorithms that can be used for this purpose, a simple one being the code matched filter. The code matched filter takes the output of the chip matched filter and matches it to the spreading sequence of the desired user. As equation 2.2 shows, the chip matched filter
output $r_i$ is processed to give $d_i$, an estimate of the detected bits.

$$d_i = S^T r_i$$ \hspace{1cm} (2.2)

Where $S$ is the $N$ by $K_{user}$ spreading matrix. Since the spreading sequences have been chosen such that the cross-correlation of different spreading sequences is small, we are able to detect the users by this code matched filtering. This is followed by the channel and source decoding clocks. The channel decoder extracts the bit stream before source decoding and corrects some of the errors based on the property of the codes used. The source decoder uncompresses the bits from the channel decoder. This corresponds to the original bit stream sent across the transmission system.

### 2.2 WLAN/OFDM

Coded Orthogonal Frequency Division Multiplexing is a modulation technique that is being adopted as the standard for the 802.11a high data rate wireless LAN. It uses a multi-carrier approach and is considered a robust method to overcome the hostile effects of a wireless channel such as multi-path.

The basic principle of multi-carrier modulation schemes is to multiplex the original high data stream into a number of parallel lower data rate streams and then modulate each of these lower data rate streams with a different frequency. The resultant signals are transmitted together in the same band. The separation between the different frequencies is crucial and that is what imparts the “orthogonal” nature to this frequency division multiplexing scheme. The frequencies are chosen such that the null of one frequency is exactly at the peak of an adjacent frequency. Note that even though this would make the spectra of the carriers overlap, it ensures that the signals at the different frequencies are recovered at the receiver without mutual interference. To achieve this, it would require a large number of oscillators each locked
to precise frequencies for the orthogonality of the carriers to hold. This is expensive and cumbersome to implement in practical systems.

However, what makes OFDM viable is the advances in Digital Signal Processing and particularly the Fast Fourier Transform (FFT). Using the IFFT and the FFT, one can digitally modulate/demodulate the data over a large number of carriers. Another important feature of an OFDM system is the length of the FFT transform. The set of sub-carriers generated during one transform is called an OFDM symbol. The duration of the OFDM symbol or the length of the FFT is related to one of the main reasons for using OFDM which is: the reduction of multi-path or echo effects. The longer the length of the FFT, the longer the echoes that can be handled. For instance, in a single frequency network, if the distance between transmitters is of the order of 80 km, then delays of up to 250 us have to be treated. If the symbol length is made much larger than the maximum echo, these low bit rate signals shall be unaffected by the multi-path channel. Thus, if the signal bandwidth for an OFDM system is 7.5 MHz, and we choose the larger symbol duration as 1 ms, then the number of carriers required would be 7500. This kind of OFDM system with a large number of carriers is used in the European standard for Digital Video Broadcasting (DVB-T). On the other hand, for an indoor office LAN, where delay spreads are of the order of hundreds of ns and bandwidth in the range of 20 – 25 MHz, a 64 point FFT would be sufficient.

Figure 2.2 shows the various components of an OFDM transmitter and receiver. Data enters the transmit chain and is convolutionally encoded. The data is then passed through an interleaver. The interleaver decorrelates the data and spreads adjacent data over many tones. The data is then passed to a modulator, which is commonly Phase Shift Keyed (PSK) or Quadrature Amplitude Modulation (QAM) depending on the type of communication system. Wired OFDM systems such as
digital subscriber loop (xDSL) tend to use higher order modulation schemes whereas wireless systems typically employ Binary PSK or Quaternary PSK. The modulator thus sequentially modulates a set of tones in the OFDM spectrum. The output of the modulator is then converted from serial to parallel form and fed into the IFFT. Complex frequency domain data is thus transformed into the time domain. The time varying data is then cyclically extended with a cyclic prefix to reduce inter symbol interference between successive OFDM symbols. The cyclic extension of an OFDM symbol is commonly referred to as the guard interval. This works as long as the guard interval is greater than the maximum channel delay spread. However, we do lose some part of the signal bandwidth in transmitting the cyclic prefix.

Taking the cyclic prefix into account, the signal model for OFDM transmission over a multi-path channel becomes very simple. The transmitted symbols $X_{l,k}$ at
time slot \( l \) and sub-carrier \( k \) are solely affected by a factor \( H_{l,k} \) which is the channel transfer function (Fourier transform of the channel impulse response) at that sub-carrier frequency and and by additional white Gaussian noise. As shown in the equation 2.3 below, the influence of the channel can be easily removed by dividing with the channel transfer function.

\[
Y_{l,k} = H_{l,k} * X_{l,k} + N_{l,k}
\]  

(2.3)

At the receiver, this is done in the equalizer block which follows the removal of the cyclic prefix and the FFT. The FFT brings the data back to the frequency domain and then the equalizer retrieves the input data symbols by dividing with the channel coefficients. To help in estimation of the channel transfer function, embedded in the OFDM symbol are training symbols. The channel transfer function is obtained by interpolation between the coefficient values obtained from the training symbols. However, this simple equalization scheme might not work for those frequencies where the channel transfer function is relatively small because dividing by a small value shall amplify the noise and thus, make detection of input data difficult. But since the data was channel coded at the transmitter, we can still retrieve the symbols that experienced a bad channel with the help of those that were received successfully. Following the equalizer block is the channel decoder which is the Viterbi decoder for the case of convolution channel codes. The output of the Viterbi decoder is the original bit stream sent for transmission.

As we have seen above, channel coding and consequently, the Viterbi decoder is a basic part of the receivers in both CDMA and WLAN systems. We shall now describe the details of a convolution encoder and the corresponding Viterbi decoder.


2.3 Convolution Encoding

Convolution codes were introduced in 1955 by Elias [P55]. He showed that redundancy can be introduced into a data stream through the use of a linear shift register. Figure 2.3 shows a typical rate 1/3 convolution encoder. The rate of this encoder is established by the fact that the encoder outputs three bits for every input bit. In general, an encoder with $k$ inputs and $n$ outputs is said to have rate $k/n$. The binary data stream is fed into a shift register circuit consisting of a series of memory elements. With each successive input to the shift register, the values of the memory elements are tapped and added according to a fixed pattern, creating the output coded data streams $y^0 = (y_0^0, y_1^0, y_2^0, \ldots)$, $y^1 = (y_0^1, y_1^1, y_2^1, \ldots)$, and $y^2 = (y_0^2, y_1^2, y_2^2, \ldots)$. These output streams are multiplexed to create a single coded data stream $y$ which is the convolution code word. The fixed pattern of taps of the shift register that is used to create $y$ is the generator polynomial. The generator polynomial has $n$ rows of tap values, one for each output. Another important parameter of convolution codes is the constraint length $K$, which is the maximum number of bits in a single output stream that can be affected by any input bit. This is equal to the number of memory

![Figure 2.3 A K = 3, Rate 1/3 Encoder](image-url)
elements in the shift register plus one, or the number of taps on the shift registers in the encoder. For the encoder shown in Figure 2.3, the constraint length is $K = 3$ and the generator polynomials are \{111, 111, 110\}. For an input sequence of \{1, 0, 1, 0, 0\}, the output codeword shall be \{111, 111, 001, 111, 110\}. So, 15 bits are generated for an input of 5 bits. Each block of 3 bits is generated in the following manner as illustrated in the equations 2.4 to 2.6 below.

\[
Y_t^0 = X_t \oplus Mem[1] \oplus Mem[0]
\]

(2.4)

\[
Y_t^1 = X_t \oplus Mem[1] \oplus Mem[0]
\]

(2.5)

\[
Y_t^2 = X_t \oplus Mem[1]
\]

(2.6)

where $X_t$ is the input bit at time $t$; $Y_t^i$ is the output bit $i$ at time $t$; and $Mem[0]$ and $Mem[1]$ are the memory cells of the shift register. The contents of the memory cells of the shift register define the state of the encoder. Typically, for a constraint length of $K$, there are $2^{K-1}$ states possible. The behaviour of the encoder can also be understood from Figure 2.4, which shows the state transition graph for the above encoder. A constraint length of 3 translates to $2^3-1 = 4$ states which have been denoted as $S_0$ to $S_3$. The transitions across states are labeled by the input/output bits. For instance, an input of 1 to the encoder in state $S_0$ shall produce an output of \{111\} and the encoder shall transition to state $S_2$. A convolution code is thus uniquely specified in terms of its rate, constraint length, and generator polynomials. For example, the convolution code for the WCDMA standard is rate $1/2$, $K = 9$, generator polynomials \{561, 753\} for the dedicated data channel, rate $1/3$, $K = 9$, polynomials \{557, 663, 711\} for the control channels, and rate $1/2$, $K = 7$, polynomials \{133, 171\} for the WLAN system.
2.4 The Viterbi Decoding Algorithm

The Viterbi decoding algorithm is the Maximum Likelihood (ML) decoding algorithm for convolution codes. It was proposed in 1967 as an “asymptotically optimum” approach to the decoding of convolution codes in memoryless noise [V67]. In the Viterbi algorithm, a trellis diagram is used to represent stages of the decoding operation. Each $n$-bit block generated by $k$ input bits corresponds to a trellis stage. The trellis diagram corresponding to the example of the encoder in the previous section is drawn in Figure 2.5. A trellis can be viewed as an alternate representation of a convolution code’s state transitions that explicitly shows the passage of time. The encoder states are shown vertically and valid transitions between states are labeled with the output bits corresponding to those transitions. The horizontal axis denotes time steps which are equal to the length of the input and additional zeros to flush the encoder memory cells to state $S_0$. For the rate 1/3 code discussed above, there are 2 possible
transitions from each state and thus, there are two branches leaving and entering each node in the trellis. In general, there are $2^k$ branches leaving each node, one branch for each combination of input values. After time $t = K - 1$, there are $2^k$ branches entering each node. It is assumed that after the input sequence is entered into the encoder, $K - 1$ state transitions are necessary to return the encoder to state $S_0$. This is usually done by forcing the last $K - 1$ bits of the input sequence to be zeroes. Thus, given an input sequence of $kL$ bits, the trellis diagram shall have $L + K - 1$ stages, the first and last stages starting and stopping, respectively in state $S_0$. There are $2^kL$ distinct paths through the general trellis, each corresponding to a convolution code word of length $n(L + m)$. The task of the Viterbi algorithm is to select the most likely path among these paths.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{trellis_diagram}
\caption{The Trellis Diagram}
\end{figure}
Consider the problem as an information sequence $x$ composed of $L$ $k$-bit blocks, convolutionally encoded to form the code word $y$, which is then transmitted across a noisy channel. The Viterbi decoder takes the received vector $r$ and generates an estimate $z$ of the transmitted code word. The ML decoder selects, by definition, the estimate $z$ that maximizes the probability $p(r/z)$ while the maximum a posteriori (MAP) decoder selects the estimate that maximizes $p(z/r)$. If the distribution of the source words $x$ is uniform, then the two decoders are identical. In general they can be related by Baye’s rule $p(r/z)p(z) = p(z/r)p(r)$. The input sequence $x$ can be written as $x = (x_0^0, x_0^1, \ldots, x_0^{k-1}, x_1^0, x_1^1, \ldots, x_1^{k-1}, \ldots, x_L^0, x_L^1, \ldots, x_L^{k-1})$. The output sequence $y$ will consist of $L$ $n$-bit blocks (one for each input block) as well as $m$ additional blocks, where $m$ is the length of the longest shift register in the encoder $y = (y_0^0, y_0^1, \ldots, y_0^{n-1}, y_1^0, y_1^1, \ldots, y_1^{n-1}, \ldots, y_L^{0}, y_L^{1}, \ldots, y_L^{n-1}, y_{L+m-1}^{0}, y_{L+m-1}^{1}, \ldots, y_{L+m-1}^{n-1})$. A noise corrupted version $r$ of the transmitted code word arrives at the receiver, where the decoder generates a maximum likelihood estimate $z$ of the transmitted sequence. $z$ and $r$ have the following form $r = (r_0^0, r_0^1, \ldots, r_0^{n-1}, r_1^0, r_1^1, \ldots, r_1^{n-1}, \ldots, r_L^{0}, r_L^{1}, \ldots, r_L^{n-1}, r_{L+m-1}^{0}, r_{L+m-1}^{1}, \ldots, r_{L+m-1}^{n-1})$. We can assume the channel to be memoryless i.e. the noise affecting one bit in the received word $r$ is independent of the noise process affecting all other bits. Since the probability of joint, independent events is simply the product of the probabilities of the individual events, it follows that

$$p(r/z) = \prod_{i=0}^{L+m-1} \prod_{j=0}^{n-1} (r_i^j / z_i^j)$$  \hspace{1cm} (2.7)

There are two sets of product indices, one corresponding to the block numbers (subscripts) and the other corresponding to bits within the blocks (superscripts). Equation 2.7 is called the likelihood function for $z$. Since logarithms are monotonically increasing, the estimate that maximizes $p(r/z)$ is also the estimate that maximizes $\log p(r/z)$. By taking the logarithm of each side of the equation, we obtain the log
likelihood equation,

\[ \log p(r/z) = \sum_{i=0}^{L} \sum_{j=0}^{m-1} \log p(r_i^j/z_i^j). \]  

(2.8)

In implementations of the Viterbi decoder, the summands in equation 2.8 are usually converted to a more easily manipulated form called “bit metrics” which can be denoted as a function \( M(r_i^j/z_i^j) \). The path metric for a code word \( z \) is then computed as follows

\[ M(r/z) = \sum_{i=0}^{L} \sum_{j=0}^{m-1} M(r_i^j/z_i^j). \]  

(2.9)

Also, the path metric can be expressed as the sum of branch metrics where the \( k^{th} \) branch metric is defined as the sum of the bit metrics for the \( k^{th} \) block of \( r \) given \( z \),

\[ M(r_k/z_k) = \sum_{j=0}^{n-1} M(r_k^j/z_k^j). \]  

(2.10)

The \( k^{th} \) partial path metric is thus obtained by summing the branch metrics for the first \( k \) branches that the path traverses,

\[ M^k(r/z) = \sum_{i=0}^{k-1} M(r_i/z_i). \]  

(2.11)

As shown in Figure 2.6, in the Viterbi algorithm, each node in the trellis is assigned a number. This number is the partial path metric of the path that starts at state \( S_0 \) at time \( t = 0 \) and terminates at that node. Since there are \( 2^k \) branches entering each node, we have to choose the “best” partial path metric among the metrics for all the entering paths. The path with the best metric is the survivor, while the other paths are discarded. The “best” partial path metric might be either the largest or smallest, depending on how the bit metric calculation is done. In the decoding example drawn in Figure 2.6, the branch metric computation is done on the basis of the hamming distance between the received block of 3 bits and the original codewords and the minimum branch metric is selected to be added to the partial path metrics. For
Figure 2.6 The Viterbi Decoding Algorithm

example, at $t = 1$, for the branch from state $S_0$ to $S_0$, upon receipt of $\{111\}$, the branch metric and consequently, the partial path metric for state $S_0$ is 3.

The algorithm terminates when all the nodes in the trellis have been labeled and their entering survivors determined. We then go to the last node in the trellis (state $S_0$ at time $L + m$) and trace back through the trellis. At any given node, we continue backward on a path that survived upon entry into that node. Since each node has only one entering survivor, the trace back operation always yields a unique path. This path is the ML path and the bits retrieved during the traceback are the decoded output bits.

“Hard” and “soft” decision decoding relates to the manner in which the inputs are presented to the Viterbi decoder and consequently, the way the computation of the branch metrics is done. In “hard” decision decoding, each received signal is examined
and a "hard" decision made as to whether the signal represents a transmitted zero or one. The branch metric computation in this case is the Hamming distance between the received vector \( r \) and a code word \( y \). In "soft" decision decoding, rather than assign a simple one or a zero to each received signal, a more flexible approach is taken through the use of multibit quantization. The quantization essentially partitions the decision region into "strong-one", "weak-one", and "strong-zero" type regions. The branch metric computation in soft decision decoding does a Euclidean distance calculation between the quantized input \( r \) and the codeword \( y \). It is worthwhile to note that for BPSK, the Euclidean distance calculation reduces to a dot product computation between the \( r \) and \( y \). On an additive white Gaussian noise channel, soft decision decoding can provide an increase in coding gain of 2 to 3DB over hard decision decoding.

2.5 Summary

As we saw in the above sections, the Viterbi decoder is a basic part of the receivers in WCDMA and WLAN systems. However, the decoding algorithm is highly dependent upon the code parameters, namely the generator polynomials, code rate and the constraint length. In particular, the code used in WCDMA is rate 1/3, \( K = 9 \), polynomials \{557, 663, 711\} and for the WLAN, it is rate 1/2, \( K = 7 \), and polynomials \{133, 171\}. The rate of a convolution code determines the amount of redundancy that is added to transmitted codeword. A higher rate code is thus stronger but makes inefficient use of the channel bandwidth. The constraint length determines the memory of a code. The larger the constraint length, the stronger the code is, but it is more complex at the same time. The complexity of the decoder increases exponentially with increase in the constraint length. The code for WCDMA with
$K = 9$ is 4 times more complex than that for $K = 7$. This is because $K = 9$ means
256 states per trellis stage as opposed to 64 for the $K = 7$ case. Another point of
interest here is that the data rates are higher (6-54Mbps) in WLAN as compared to
2.5Mbps for WCDMA. From an implementation perspective, this augurs well as we
have a lower decoding rate or more time to decode the more complex code. Thus, we
need an architecture which can support decoding rates in the range of 2.5-54 Mbps
and be run-time adaptive to changes in the code rate or the constraint length. Before
we describe the details of the architecture that we have proposed as part of this thesis,
we shall review some of the known architectures for the Viterbi decoder in the next
chapter.
Chapter 3

Architectures for the Viterbi Decoder

3.1 Main Functional Units of the Viterbi Decoder

This chapter discusses the various architectures for the Viterbi decoder in detail. There are three major components in a Viterbi decoder, the Branch metric unit (BMU), the Add-Compare-Select unit (ACS), and the Survivor Memory unit (SMU). Figure 3.1 shows the block diagram of a Viterbi decoder in a practical system.

Other than the three components mentioned above, it has blocks like the carrier recovery and the synchronizer so that the decoder interfaces well with the real world. The input to the receiver is a continuous stream of analog, modulated signals. The primary task of the receiver front-end is the recovery of the bit timing so that the individual received data bits can be removed from the carrier and separated from one another in an efficient manner. Both tasks are generally performed through the use of phase locked loops. The analog baseband signals after passing through the A/D converter and the block synchronizer are sent to the Viterbi decoder. The output of the multi-user detector block in Figure 2.1 and that of the parallel to serial converter in Figure 2.2 correspond to the input of the Viterbi decoder. Usually, the decoder is provided with a bit timing clock so that the decoder can distinguish between separate bits. The individual components of the decoder are treated in the subsections that follow.
Figure 3.1  A Practical Viterbi Decoder
3.1.1 Quantizer

This relates to the resolution of the A/D converter immediately following the front-end of the receiver. The number of quantization levels decides whether the decoding is hard decision or soft decision. As mentioned earlier, soft decision decoding is known to provide better coding gain than one-bit quantization or hard decision decoding. Since in practice we need to have some finite number of bits for representing the soft decision data, there has been a lot of research on this issue. It has been found that even a coarse quantization of up to 3 bits or 8 levels introduces only a slight reduction in performance with respect to the unquantized case. For higher level quantizers, the additional complexity associated with a higher resolution analog-to-digital converter is not balanced by a significant improvement in decoder performance. An 8-level quantization has been shown to result in only about .25 DB reduction in coding gain for a number of codes, as compared to .9 DB and 2.2 DB reductions for 4-level and 2-level quantization respectively.

3.1.2 The Block Synchronizer

In order to compute the branch metrics at any time, the Viterbi decoder should be able to segment the received bit stream into \( n \)-bit blocks, each block corresponding to a stage in the trellis. If the received bits are not properly divided up, the results are disastrous. Therefore, block synchronization is very important for the Viterbi decoder. One common technique of achieving synchronization relies on the fact that \( n \) is usually small (\( n \leq 4 \)). A systematic search of the bit positions can be efficiently conducted. If a guess at block synchronization is correct, one or two partial path metrics will be substantially lower than the others at each stage in the trellis after a few constraint lengths of branch metric computations. This is under the assumption
that the ML path has the smallest path metric. If a guess is wrong, then the data stream is misaligned and is essentially random. Thus, there is no dominant path and the path metrics of the survivors tend to be close together. A thresholding circuit that compares the minimum partial path metric to a fixed threshold, say after 5 constraint lengths, can be used as an indicator for synchronization.

3.1.3 Branch Metric Unit

The Branch Metric Unit computes the branch metrics at each stage of the trellis corresponding to a $n$-bit block of input data. There are two aspects to the computation of the branch metrics. Firstly, the Branch Metric Unit needs to know the codewords against which it compares the receive input bit stream. This is usually done by having an encoder built inside the Branch Metric Unit. Secondly, the actual difference between the codewords and the received symbol needs to be calculated. Since there are $2^{K-1}$ states and $2^k$ branches going into each state at each trellis stage, $(2^{K-1})(2^k)$ branch metrics need to be calculated per stage. However, there are only $2^n$ unique codewords corresponding to an $n$-bit codeword, and so only $2^n$ unique branch metrics need to be computed at each stage. It would be helpful if there was some symmetry in the way the branch metrics repeat over the trellis states, but this is generally irregular and highly dependent upon the generator polynomials chosen for a particular code. The computation of the branch metrics is thus highly dependent upon the rate of the code i.e the factor $n$, as well as the generator polynomials.

A typical implementation of the branch metric calculation unit is a based on a look-up table containing the various bit metrics. To compute each metric, the table needs to be looked up $n$ times and the various bit metrics summed together. However, if we decide on using a metric such as the hamming distance for hard decision decoding, all that needs to be calculated for a branch metric is the number
of bit differences or the hamming distance. This can be carried out by exclusive OR operations. On the other hand, for soft decision decoding, since the Euclidean distance needs to be calculated, a subtracter and a multiplier would be required. So, from an implementation view, hard decision decoding is simpler for the branch metric unit than soft decision decoding. The results of the branch metric unit are passed to the add-compare-select Unit for path metric updation and storage.

3.1.4 Add-Compare-Select Unit

The Add Compare Select Unit takes in the branch metrics computed by the branch metric computer and computes the partial path metrics at each node in the trellis. The surviving path is identified, the partial path metrics updated and and the survivor information stored in the traceback memory. These functions are performed most efficiently by breaking the trellis into a number of identical elements. For example, the trellis diagram of a rate $1/n$ convolution code can be broken up into elements containing a pair of origin and destination states and four interconnecting branches, as shown in Figure 3.2 for the case of a $K = 3$ encoder. Each stage of the trellis comprises of 2 butterfly elements in this case. By extension, one trellis stage of a constraint length $K$ coder can be made up of $2^{K-2}$ butterfly elements. Thus, only one circuit needs to be designed for the path metric updating operation. The basic functions performed by this circuit are add, compare and select. An ACS circuit is shown in the Figure 3.3. A butterfly is made of 2 ACS units. The entire decoder can be based on one such ACS, resulting in a slow, but low cost implementation. On the other hand, a separate ACS circuit can be dedicated to every element in the trellis, resulting in a fast, massively parallel implementation. For a rate $1/n$ code with constraint length $K$, $2^{K-1}$ ACS operations need to be performed at each stage of the trellis. The ACS circuit is thus the primary functional unit of the Viterbi
Figure 3.2 One stage of a $K = 3$ encoder - two butterfly elements

Figure 3.3 An Add-Compare-Select Unit
decoder; its design and utilization are the principal factors in determining the decoder complexity and speed. However, it should be noted that for a rate \( k/n \) code, the ACS circuit needs to make a \( 2^k \) way comparison of partial path metrics to determine the surviving path. The complexity of the ACS circuit thus increases exponentially with \( k \) and mostly, only rate \( 1/n \) codes are used in practice. Puncturing [JC79] is another technique employed to derive rate \( k/n \) codes from \( 1/n \) codes. This enables the use of the same rate \( 1/n \) decoder at the receiver.

### 3.1.5 Survivor Management Unit

The survivor management unit is responsible for keeping track of the information bits associated with the surviving paths designated by the add-compare-select unit. There are two basic design approaches: register exchange and traceback.

In register exchange [GPF89], there is a shift register associated with every trellis node throughout the decoding operation. The register for a given node at a given time contains the information bits associated with the surviving partial path that terminates at that node. As the decoding operations proceed, the contents of the register in the bank are updated and exchanged as dictated by the survivor branches. At the end of the decoding operation, the sequence of input bits is contained in the register which corresponds to the final state of the decoder.

In the register exchange method, the registers must be capable of sending and receiving strings of bits to and from other registers. In a fast decoder, all of the exchanging must take place simultaneously, leading to a hardware intensive implementation. This approach is thus conceptually simple but hardware intensive because it involves an interconnection network among the registers to move bits back and forth between them.
In the traceback approach [RC90], rather than storing the information bits at each trellis stage, a history of the surviving branches at each node is stored. Information bits are obtained by tracing back through the trellis as dictated by this connection history. Consider for example the rate 1/3, \( K = 3 \) code discussed earlier in 2.3. State \( S_2 \) in such a four-state encoder corresponded to encoder shift register contents \{01\}. In general, if we denote shift register contents \{xy\} by state \( S_{xy} \), then state \( S_{xy} \) can be preceded only by states \( S_{y0} \) or \( S_{y1} \) because the input bit is the leftmost bit in the encoder shift register. A zero or a one may thus be used to uniquely designate the surviving branch entering a given state. For a rate 1/\( n \) code, just 1 bit is necessary to designate the surviving branch. Either registers or memory can be used to store the survivor history bits. But note that in the traceback approach, the contents of the register associated with each state are not exchanged with other registers, simply updated by one bit with each successive stage of decoding. Traceback implementations of the survivor management unit are significantly faster than register exchange because they do not require register contents to be moved about, although the trace back operation is somewhat more complicated.

Also worth mentioning is the "unification property" of Viterbi decoding which makes possible decoding without having to wait for the entire message sequence. It can be shown that survivor sequences of all the states in a given time stage share a common subsequence. That is, if we follow all survivor sequences back \( X \geq X_{\text{min}} \) stages, they all merge to the same state. This property is referred to as the unification property and it allows us to truncate the trellis diagram and save only some number \( S \geq X_{\text{min}} \) most recent stages. The number \( X_{\text{min}} \) is obtained empirically and is usually five times the constraint length.
3.2 Architecture Realizations for the Viterbi Decoder

The Viterbi Algorithm is computationally demanding not because its algorithm is complex in a conceptual sense. In fact, the essence of the algorithm is a relatively simple procedure of identical add, compare, select, and traceback operations. Rather, the computational burden arises because a relatively simple set of operations must be applied to a large number of basic nodes or states at each discrete time step. The number of states grows exponentially with constraint length. With the limitations of present fabrication technology there has been a great incentive to devise algorithms that assign more than one state per processor and/or constrain interprocessor communication such that the area necessary to wire processors does not dominate the area required by the processors themselves. Locally connected processor arrays are of interest here because they satisfy the latter constraint. In fact, the Viterbi Algorithm has benefited much from research in the use of processor arrays, [GT88] and [C90], for popular algorithms like sorting, polynomial multiplication, matrix transposition and Fast Fourier Transform. We shall briefly go over some of the architectures commonly used for implementing the Viterbi decoder.

3.2.1 Uniprocessor

The most straightforward implementation of the Viterbi Algorithm is a completely sequential one where every state is evaluated, in sequence, in a single arithmetic logic unit, driven by a programmed control unit (i.e., a microprocessor). This approach, though processor poor, and speed wise slow, requires a very small area. At the same time, it must co-ordinate random access to the processor’s memory which shall be used for storing and updating the path metrics computed at each stage. Commonly, two path metric memories are used for this purpose, one for the new path metrics,
and the second for the old path metrics. These are swapped after every time step in
the decoding process. However, at the expense of complicating the address decoding
circuitry, it is possible to update the path metrics in-place and save processor memory.
A slight improvement to the completely serial approach is doing two states of a
butterfly, rather than a single state, sequentially. For this, we would need two add
compare select units in hardware. Using such an approach, it would require $2^{K-2}$ time
steps to process all the states. Infact, current Digital Signal processors (TMS320C54)
use this approach by having dedicated ACS instructions for the Viterbi. We shall see
the details of the DSP implementation in a later chapter.

3.2.2 Parallel

The other extreme is a fully parallel implementation of the Viterbi algorithm where
each state is assigned one processor and the interprocessor connection network is a
shuffle exchange graph. In the context of a VLSI realization, this type of fully par-
allel layout, though dominated by large interprocessor wire area, is the architectural
organization with the greatest possible throughput for a given fabrication technology.
Most of the interconnection wiring is due to the routing between ACS units, and this
does not scale very well for large constraint lengths. Consider for example the Big
Viterbi decoder(BVD-1988), one of the first implementations for constraint length
14, [DGP96]. The first prototype of the BVD filled two large cabinets and operated
at 1 Kbit/s. A more recent effort in this direction is a bit serial implementation of
a Viterbi decoder for 3rd generation W-CDMA systems [HYK99]. This is for a rate
1/3 and $K = 9$ Viterbi decoder chip. Bit serial arithmetic for the ACS units is used
for savings in area. Also, they use a floorplan of the ACS modules such that the
256 ACS units are clustered together in groups which reduce interconnection wiring.
This bit serial implementation of the Viterbi decoder chip can operate at 2-20 Mbps. Thus, a fully parallel approach is suitable for high speed Viterbi decoding.

### 3.2.3 General Cascade

An alternative to the above extremes is the General cascade architecture of the Viterbi decoder [GT88]. In this architecture, a ring of processors work in parallel with each processor computing an entire trellis stage. This enables full path metric information to be passed locally from one processor to its neighboring processor, removing the necessity of bit serial arithmetic, as in the fully parallel approach, and thus allowing pipelining. Furthermore, because an entire stage need not be processed before moving onto the next stage, there is an improvement in throughput. The number of processors required to implement the cascade architecture for a decoder of constraint length $K$ is $K - 1$. FIFOs and switches are used to co-ordinate the transfer of data between processors. This intermediate approach, though slightly complex because it has to manage the transfer of data between processors, is an efficient alternative to the completely sequential/parallel approaches. It takes $O(2^K / K)$ time to compute all states while taking up $O(K)$ area in terms of the number of ACS processors.

### 3.2.4 Systolic

This approach gives a linear scale solution to the Viterbi algorithm. Typically, algorithms that have a feed-forward structure lend themselves suitably to a linear scale solution i.e. an architecture which leads to at maximum a linear increase in hardware complexity for a linear speedup in the throughput rate if the limit of the computational speed of the hardware is reached. However, the Viterbi contains a data dependent feedback loop in the ACS unit. This feedback component presents a bottleneck to the throughput rate of the decoder unlike the BMU and the SMU which can
be easily pipelined for high speed implementation. In [GH89] and [GH90], Fettweis and Meyr have proposed a hierarchical decoding approach which breaks the ACS bottleneck by formulating the Viterbi Algorithm as a matrix-vector problem using semi-ring algebra. A systolic array solution for the ACS units emerges from this approach. The area consumed by the systolic solution is comparable to that of the fully parallel solution i.e. $O(2^K)$ while the time it takes is of $O(2^K)$, which is similar to that taken by a uniprocessor implementation.

The various implementations of the Viterbi decoder discussed above are suitable for use in different systems depending upon the requirements of the system. For example, if the system is such that the decoding throughput is low priority and the area needs to be minimized, a uniprocessor architecture would be ideal. On the other hand, for high speed decoding, a parallel or cascade structure would be preferred. However, none of these architectures address the issue of reconfigurability of the decoder. Each of these is designed for a particular configuration of the decoder. We have tried to overcome this problem in our design of a reconfigurable decoder. In Chapter 4, we will review the current work on reconfigurable Viterbi decoders and develop our own architecture for such a decoder.
Chapter 4

A Reconfigurable Decoder Architecture

In this thesis, a flexible and reconfigurable architecture for the Viterbi decoder has been developed, simulated and synthesized. The synthesis of the design has been done on Field Programmable Gate Array Device. The proposed architecture of the Viterbi decoder provides dynamic reconfigurability for constraint lengths between 3 and 9 and for rates 1/2 and 1/3. This flexibility is achieved through an architecture that employs a fully parallel approach, as discussed in the last chapter, for the highest constraint length and the highest rate. The same physical hardware is used for smaller constraint lengths and lower rates by mapping the decoder structure for the smaller constraint length onto that for the larger constraint length. The reconfiguration can be done on-the-fly by toggling the reset and select inputs to the Viterbi decoder. Since this design uses the fully parallel implementation approach for all constraint lengths, we get a high decoding throughput and moreover, the throughput does not degrade for different constraint lengths.

A very important challenge to overcome in reconfiguring the Viterbi decoder is to identify suitable realizations of each block in the decoder, namely the branch metric unit, the add-compare-select unit and the survivor management unit, which can easily scale to changes in the constraint length and the rates. As we have seen in the previous chapter, the constraint length $K$ is a very important parameter of the decoder. It decides the complexity of the ACS units in terms of the the number of units required for a fully parallel implementation and also more importantly, the interconnection
network between the ACS units. On the other hand, the rate affects only the branch metric calculation and so has relatively small impact on the design of the decoder.

4.1 Previous work

Previous implementations of “reconfigurable” Viterbi decoders have focused on providing “programmed” flexibility rather than “dynamic” reconfigurability. In [PP93], the authors have proposed a foldable Viterbi decoder that uses a foldable state scheduling scheme to achieve varying constraint length decoders. Foldability as they use it means mapping of larger problem sizes to a smaller set of fixed hardware. They have fixed hardware in the sense of a fixed number of butterfly processor elements which are scheduled to process all the states over time. This way they achieve 100 percent hardware utilization but the throughput suffers for larger constraint lengths. They have used the Omega network, which is essentially a shuffle exchange network, to map the problem for a large K onto a small number of butterfly units. The Omega network allows for a fixed interconnect between the ACS units and is thus well suited to provide this”programmed” reconfigurability. “Programmed” flexibility as opposed to “dynamic” reconfigurability means that for each configuration i.e. a set of rate and constraint length, the computational kernel remains unchanged but a new set of configuration data are loaded into the hardware. Such flexibility, which is attained by loading a different bit stream into the FPGA for each configuration is slow and relies on the device being re-programmable. For instance, it could not be used for an ASIC architecture. Also, the reconfiguration time for such a re-programmable device, say an FPGA, is of the order of a few milli seconds, which is much more than what is desired when we need a device to switch at run time from one standard to another.
Another hardware implementation of a flexible Viterbi decoder which provides dynamic reconfigurability is that of a DSP coprocessor [DhAg00]. This decoder is geared towards a solution for Third Generation base station architectures i.e. a data rate of 2.5 Mbps and a constraint length of 9. It is mainly concerned with low power, small area, relatively low data rates, and a flexible system interface for co-ordination with the DSP. Under the constraints of small area and low power, the architecture used for the ACS units is the general cascade approach discussed in the previous chapter. Typically, cascade structures are built for the size of the trellis that must be solved and the outputs fed back to the inputs through another reordering network. However, one could also replace this feedback network with a memory and use the cascade device as a higher radix unit. In their approach, they use a cascade structure of four ACS units capable of operating on a radix-16 trellis. Coupled with state metric memory, it can be used to allow operation on trellises up to 256 states. Operation of the cascade on larger trellises requires pulling out the radix-16 trellises one by one from the larger trellis. The disadvantage of this approach is because of the inherently sequential nature of the cascade algorithm, there is a loss of throughput for higher constraint lengths. The manner in which achieves reconfiguration is through a set of two internal parameter banks. Each of the banks can hold descriptions of the decoding problem and one can be loaded as the decoder is executing from the other. The parameters of the decoding problem include generator polynomials, code rate, constraint length, and frame structure details.

4.2 The Proposed Architecture

As mentioned earlier, the key challenge in designing a reconfigurable decoder architecture is the realization of suitable structures for the three major blocks in the decoder
i.e. the BMU, ACS, and the SMU. Since we were aiming for high decoding throughput, in the range of 2Mbps for W-CDMA to 54 Mbps for WLAN, we chose a fully parallel implementation of the decoder for all the constraint lengths we want to support. This means that we have hardware for the largest constraint length decoder and the smaller constraint length decoders use only part of the total resources available. Also, this obviates the need for any state metric memory and address generation circuitry which makes the design simpler. However, now since we shall be using registers for holding the state metrics, the interconnection network between the outputs of the ACS units and the state metric registers becomes even more complex as it also needs to have the ability to configure itself into a different trellis structure for different constraint lengths. One of the major contributions of this thesis is to define a basic component that forms an essential building block of the Viterbi decoder, given any constraint length, code rate and generator polynomial. This approach of defining a basic computational core which can be used for different functional realizations is similar to the approach discussed in [EB98]. Our basic building block, hereafter referred to as the Viterbi Core is shown in Figure 4.1. The Viterbi Core is designed to operate on a butterfly i.e. two states at a time. It is composed of a branch metric unit, two registers for storage of the path metrics, an add-compare-select unit which does the path metric updating for a butterfly, and optional multiplexors at the inputs of the path metric registers. The Viterbi core can be replicated as many times as dictated by the constraint length of the code to yield different configurations of the decoder. For example, for a $K = 5$ decoder, which has 16 states or 8 butterflies, the Viterbi core needs to replicated 8 times. The different generator polynomials corresponding to each code are stored internally in registers. Note that the input to the multiplexors is the $selectK$ input line, the toggling of which changes the constraint length of the decoder and an input to the BMU is the $selectR$ input, which controls the code rate
Figure 4.1 The Viterbi Core
of the decoder. Select\(K\) and select\(R\) are the selection inputs to the decoder which determine the mode of operation of the decoder.

The exact operation of the decoder i.e. how the reconfiguration is achieved, and the behaviour of the different components of the Viterbi core shall be explained in later sections. Also, note that we have considered only rate 1/\(n\) codes in this thesis. This is because for this class of codes, the design of the ACS units is simple for which reason, these are the most commonly used convolution codes in the standards. Now we will describe the detailed architecture of the individual units in the reconfigurable decoder core.

### 4.2.1 BMU Core

As discussed in the previous chapter, the branch metric unit takes in the received codeword and the select\(R\) line as inputs and produces the branch metrics as the outputs. For a rate 1/\(n\) code, there are \(2^n\) unique codewords corresponding to a codeword of length \(n\), and there are \(2^K\) branch metrics that need to be computed at each stage of the trellis. The number of unique codewords is typically smaller than the number of branch metrics. For example, for a rate 1/2, \(K = 5\) code, the number of unique codewords is 4 whereas the number of branches per trellis stage is 32. These unique codewords repeat irregularly over the branch metrics, depending on the generator polynomials. Also, the codewords associated with a particular branch or state transition change when going from one constraint length to the other. Note that changing constraint lengths implies a change in the generator polynomials. Thus, there lies a problem in the computation of branch metrics in a flexible manner which permits adaptation to various constraint lengths and polynomials.

This problem is solved by utilizing the symmetry in the codewords associated with a butterfly, and by having a number of BMUs compute the branch metrics in parallel
for the whole trellis. Out of the four codewords involved in one butterfly, as drawn in Figure 4.2 only one of them is unique. The others are either the same or flipped versions of this one unique codeword. This is reflected in terms of branch metrics in the following equations, 4.1 and 4.2.

\[
BM(j, 2j) = BM(j + 2^{K-2}, 2j + 1) \quad \text{(4.1)}
\]

\[
BM(j, 2j + 1) = BM(j + 2^{K-2}, 2j) = -BM(j, 2j) \quad \text{(4.2)}
\]

This property comes about as a consequence of the generator polynomials being such that there is a one at both ends of the polynomial, and the fact that codewords are produced by exclusive OR operations of the shift register contents. Most commonly used generator polynomials, among these being the ones for WCDMA and WLAN, adhere to this requirement. Another important observation that we make from figure 4.3 which depicts the trellis interconnections for \( K = 3 \) and \( K = 4 \) is that independent of the constraint length, there are state transitions from state \( 2i \) to state \( i \), where \( i \) is the butterfly index. In other words, there are transitions from states 0 to 0, 2 to 1, 4 to 2, 6 to 3, and so on, for all constraint lengths. Thus, we can dedicate one BMU unit per butterfly which shall compute the one unique codeword needed.
Figure 4.3  The Trellis for $K = 3$ and $K = 4$

per transition per butterfly. Further, since each BMU needs to calculate the branch metrics for different state transitions, each BMU needs to be initialized to a different initial state. For example, for a $K = 3$ decoder, the first BMU needs to compute the metrics for the transition from state 0 to state 0, while the second BMU needs to compute the branch metric value corresponding to the transition from state 2 to state 1. As discussed above, once the metric from state 2 to state 1 is known, the metrics from state 2 to state 3, state 3 to state 1, and state 3 to state 3 are also known. Thus, each BMU needs to be initialized to the appropriate state value, which means that the shift register for the encoder in the BMU core is loaded with the correct state. In the example discussed above, the first BMU’s encoder shift register is loaded with 0, and the second BMU shall have a 2 in its shift register.
The BMU is internally composed of an encoder which computes the codeword and some additional circuitry to calculate the hamming distance between the computed codeword and the received codeword. Since we have hardware for the maximum constraint length, this type of architecture is readily usable for the smaller constraint lengths, the only difference being that for the smaller constraint lengths, some of the BMUs shall not be required. For the example of \( K = 5 \) and \( K = 7 \), there shall be 32 butterflies for \( K = 7 \), but for \( K = 5 \), only the computations of the first 8 are really needed. The BMU that computes one butterfly is a part of the Viterbi core described earlier. Since we want the architecture to be adaptable to changes in the code rate in addition to the constraint length, the circuitry inside the BMU is designed for the highest rate code. This means that the encoder shift register needs to be of length equal to the highest constraint length. The taps of the shift register corresponding to different code rates are hardwired and can be selected at run-time by changing the \( selectR \) input of the decoder.

4.2.2 ACS Core

This is the computationally most demanding part of the Viterbi decoder. Also, since it involves a feedback in the way in updates the path metrics, it involves extensive interconnection wiring between the ACS units. As seen in the Viterbi core in Figure 4.1, there are two path metrics updated for every butterfly. Updating a path metric means adding the branch metrics from two states to get two partial path metrics and then selecting the minimum out of these two. This means that an ACS butterfly has inputs as two branch metrics, and two path metrics and two path metrics and two decision bits as outputs. The results of the path metric updates are stored back to registers while the decision bits are written to survivor memory. Thus, there is no memory usage for the storage of path metrics in the case of this fully parallel
architecture which makes our design simple. However, this simplicity comes at the cost of making the connections between the ACS units and the path metric registers complicated.

The interconnection network between the ACS units and the path metric storage registers is very complex and highly dependent upon the constraint length. In general, for a typical butterfly calculation, as seen previously in Figure 4.2, the input states are $2i$ and $2i+1$, and the output states are $i$ and $i+2^{K-2}$. This shows the dependence of the metric updates on the constraint length. For example, if $K = 3$, the ACS unit for the first butterfly would update the path metrics for states 0 and 2, while if $K = 4$, the first butterfly needs to write the updated path metrics to path metric registers 0 and 4. So, for the fully parallel architecture we are considering, we need to find a way to route the outputs of the ACS units to the appropriate path metric registers. Let us suppose that we want to support a range of constraint lengths from $[K_1, K_2]$, $K_1 < K_2$. So, we have $2^{K_2-2}$ ACS units and $2^{K_2-1}$ registers for path metric storage. Let us index the outputs of the ACS units as $[\text{ACS}(0), \text{ACS}(1), \ldots, \text{ACS}(2^{K_2-1})]$, ACS($2i$) and ACS($2i+1$) being the outputs of ACS unit $i$; and the inputs to the path metric registers as $[\text{PM}(0), \text{PM}(1), \ldots, \text{PM}(2^{K_2-1})]$. From the trellis diagrams for the various constraint lengths, we make the following important observation: $\text{PM}(i)$ can be fed from either ACS($2(i - 2^{K-2}) + 1$) or ACS($2i$) depending upon the value of $K$. Equations 4.3 to 4.5 describe the interconnections between the outputs of the ACS units and the inputs to the path metric registers as a function of $K$.

for $0 \leq i < 2^{K_1-2}$

$$PM(i) = ACS(2i) \ \forall K$$

(4.3)
for \((2^{K_1-2} \leq i < 2^{K_2-2})\)

\[
PM(i) = \begin{cases} 
ACS(2(i - 2^{K_2-2}) + 1); & \text{K s.t. } i - 2^{K_2-2} \geq 0, \forall K \in [K_1, K_2) \\
ACS(2i); & \text{otherwise}
\end{cases}
\]  \hspace{1cm} (4.4)

for \((2^{K_2-2} \leq i \leq 2^{K_2-1})\)

\[
PM(i) = ACS(2(i - 2^{K_2-2}) + 1) \text{ for } K = K_2
\]  \hspace{1cm} (4.5)

For the simple case of a decoder which supports \(K = 3\) and \(K = 4\), Figure 4.4 depicts the interconnections. The highlighted connections are the ones that change if the constraint length changes from 3 to 4. The Following equations reflect the same.

\[
PM(0) = ACS(0);
\]
\[
PM(1) = ACS(2);
\]
\[
PM(2) = MUX(selectK, ACS(1), ACS(4));
\]
\[
PM(3) = MUX(selectK, ACS(3), ACS(6));
\]
\[
PM(4) = ACS(1);
\]
\[
PM(5) = ACS(3);
\]
\[
PM(6) = ACS(5);
\]
\[
PM(7) = ACS(7);
\]

In this way, by using a 2-input multiplexor to feed the path metric registers, and controlling the output of the multiplexor by the \(selectK\) input of the decoder, we can route the ACS outputs to the correct path metric registers. The multiplexors are the additional overhead which is needed to make the interconnection network reconfigurable. Apart from the path metrics, the ACS units also produce two decision bits for the the two states of a butterfly. A decision bit is 1 if the surviving branch comes from a state with higher index and zero otherwise. These decisions are used
Figure 4.4  The Interconnection Network for \( K = 3 \) and \( K = 4 \)
(The italicized connections change for \( K = 3 \) and \( K = 4 \))
by the Survivor Management Unit to reconstruct the sequence of states the decoder
goes through. Decision bits for all the states are produced simultaneously and must
be written to memory in parallel.

4.2.3 Survivor Management

This consists of the controller for the Survivor memory and the Survivor memory
itself. The survivor memory size is chosen according to the largest constraint length
decoder. For a decoder which needs to support a maximum constraint length of
$K = 9$, the survivor memory must have a width of 256 bits. Each of these 256
bits is written to by the decision bits produced by the ACS units at each stage of the
trellis. Each bit position represents a state in the trellis diagram, and each location in
memory represents a time step in the decoding process. Thus, the length of the input
message $(L)$ determines the length of the survivor memory. The survivor memory
should be at least equal to $(L + K - 1)$, taking into account the trailing zeros at the
end of the message. When writing the decision bits into memory, the same problem
occurs as in writing the updated path metrics to the registers. This is because the
outputs of the ACS units again need to be written to different bit positions in memory
depending upon the constraint length. This problem is solved in a similar manner by
using 2-input multiplexors at the data input to the survivor memory.

As discussed in the last chapter, there are two approaches to Survivor manage-
ment, namely register exchange and traceback. We chose the traceback scheme over
the register exchange method because the traceback is more suitable for reconfigura-
tion purposes. This is because of its more “programmable” approach as compared to
the “hardwired” approach of the register exchange method. Because of the trailing
zeros at the input, after the whole message is received, the encoder always ends in
state 0. Thus, the decoder can start decoding from state $S_0$, as soon as the complete
message has been received. The traceback controller goes through 3 states during the entire decoding process.

In the first state, the survivor memory is being filled with the decision bits produced by the ACS units for all the states. This takes a number of time steps equal the message length plus constraint length minus one. In the second state, the traceback operation is performed. It starts in state \( S_0 \) at the last memory address written to and generates the sequence of states and consequently the sequence of input bits experienced by the decoder in the reverse order. It reads the bit position corresponding to the state of the decoder at each time step and calculates the state at the previous time step. A bit of 0 in position \( i \) at memory location \( t \) indicates that the state at time \( t - 1 \) was \( (2i) \mod 2^{K-1} \) while a bit of 1 at the same location would mean the state at time \( t - 1 \) was \( (2i + 1) \mod 2^{K-1} \). This is derived from the butterfly properties discussed earlier. Also, once we know the state, the input bit is easily found as the left most bit of the state. The traceback algorithm can be summarized as follows: Consider a constraint length \( K \) decoder which needs to decode a message of length \( L \), which means we need \( K - 1 \) bits to represent a state value.

Start in state \( S_t = i = 0 \) at \( t = L + K - 1 \)

While \( t \geq 0 \)

input bit \([t]\) = left most bit or \( K - 1 \)st bit of \( i \)

Read Bit \( i \) at memory location \([t]\)

If Bit \( i = 0 \) then

\[ S_{t-1} = i = (2i) \mod 2^{K-1} \]

else

\[ S_{t-1} = i = (2i + 1) \mod 2^{K-1} \]

end if
\[ t = t - 1 \]

end while

So, if we use \( K_{\text{max}} - 1 \) bits for the state variable of the decoder which can support up to constraint length \( K_{\text{max}} \), for any other smaller value of constraint length, performing the above algorithm on just the \( K - 1 \) least significant bits of the state is sufficient. In this way, by providing a \( \text{selectK} \) selection input to the SMU, the traceback controller can switch to different constraint lengths by operating on a different number of bits in the state variable. It takes another message length plus constraint length number of time steps to traceback the entire trellis and yield the input bits in the reverse order. In the third and final state, the decoder outputs the decoded bits in correct order. The traceback unit is thus easily adaptable for different lengths mainly because of two reasons. Firstly, it uses memory width needed for the largest constraint length so that the smaller ones are easily accommodated. Secondly, during the traceback, because of the nature of the traceback algorithm, it can flexibly perform the operations required for different constraint lengths.

### 4.3 Summary

The design of the ACS, BMU and the survivor memory was done so that the Viterbi decoder can adapt to changes in the constraint length and coding rates dynamically. Since we are interested in high decoding throughput, a fully parallel structure for all constraint lengths was chosen. A basic building block, the Viterbi core, was identified, which can be replicated to generate different configurations of the decoder. The Viterbi core relies on the symmetry properties of the butterfly computations. The ACS, BMU and path metric storage registers are parts of the Viterbi core which are designed to operate on butterflies in the trellis. An interconnection network between
the ACS units and the path metric storage registers makes use of 2-input multiplexors to reconfigure itself for different constraint lengths. The reconfiguration for different constraint lengths was a bigger challenge than that for different rates because the constraint length affects the number of ACS units and their interconnections whereas the rate just affects the branch metric computations. The reconfiguration for different rates is achieved by having the BMU designed to operate for the highest rate code. This reconfigurable architecture has been synthesized on a Xilinx XCV800 FPGA. In the next chapter, we will present results of our prototype implementation of the reconfigurable architecture on the FPGA.
Chapter 5

Prototype Implementation and Results

The proposed architecture was implemented on a Xilinx XCV800 Field programmable gate array (FPGA). VHDL was used for the hardware description and Foundation software tools used for mapping the high level description into a gate level implementation for the the FPGA device. In this chapter, we shall first give a brief overview of the FPGA architecture, then present results on the implementation of the decoder in the XCV800 FPGA, and finally conclude with a comparison of the FPGA implementation with that of a DSP implementation.

5.1 FPGA Development Environment

FPGAs are powerful programmable logic devices and have recently become very popular. This is not only due to the fact that programmable logic saves development cost and time over complex ASIC designs, but also because the gate count per FPGA chip has increased considerably over the last few years. FPGAs provide the benefits of custom CMOS VLSI, while avoiding the initial cost, time delay, and inherent risk of a conventional masked gate array. FPGAs are customized by loading configuration data into internal memory cells. The configuration data can be thought of as a single instruction which the FPGA executes each time. FPGAs can be programmed any number of times and the Virtex family devices can support clock speeds upto 200 MHz. The re-programmable feature of FPGAs is provided by static RAM technology. FPGAs have three major configurable elements: configurable logic blocks (CLBs), in-
put/output blocks (IOBs), and interconnects. CLBs provide the functional elements for constructing the user’s logic. IOBs provide the interface between the package pins and the internal signal lines. The programmable interconnect resources provide routing paths to connect the inputs and outputs of the CLBs and IOBs onto the appropriate networks. Customized configuration is established by programming internal SRAM memory cells that determine the logic functions and internal connections implemented in the FPGA. The XCV800 FPGA has an array of 56 x 84 CLBs, a maximum available 512 IO pins and a total of 114, 688 RAM bits.

The design cycle for FPGA programming begins with the design entry in VHDL. Once the design has been developed in VHDL, functional verification of the logic can be done by simulating the design. The next step involves synthesis of the design to generate an optimized netlist describing the gate level design in Xilinx format. This netlist, along with constraint files is fed into the Xilinx Implementation tools. The constraints can be timing constraints or forcing pin assignments on the device. The design goes through map, place and route, and timing phases to generate a bit stream which can be downloaded to the device. Finally, timing verification of the design can now be done using the same test vectors that were used for functional verification.

5.2 Prototype Implementation

In this section we describe the architecture from the perspective of implementation on the XCV800 FPGA. As mentioned above, VHDL was used to describe for hardware description. At the topmost level, the Viterbi decoder appears as shown in Figure 5.1.
5.2.1 System Partition

Shown in Figure 5.2 is the block diagram of the internals of the reconfigurable Viterbi decoder. The main components of the decoder are the controller unit, the Viterbi core, the interconnection network and the traceback memory. Also part of the design are configuration registers which hold parameters such as all the different generator polynomials, and the different values of the constraint lengths and code rates. The top level VHDL design comprises of the following entities.

Inputs/Outputs

The input(2:0) is available three bits in parallel with the bits being either 0 or 1, which implies hard decision decoding. Note that when the chip is used as a rate 1/2 decoder, only the upper 2 bits of the input shall be used. The selectK input is the selection input which determines the constraint length to be chosen for any particular realization of the decoder. It is 3 bits wide because it can select any of the 7 values from 3-9 for the constraint length. Similar to the selectK input is the selectR input which selects the rate of the decoder. It is one bit wide for selecting either a rate of 1/2 or 1/3. The other inputs of the decoder are a clock and reset input. The clock is global and is the only clock used in the design. The reset pin is important
because it is used in conjunction with the \textit{selectK} and \textit{selectR} input to reconfigure the decoder configuration. Reset needs to be made low every time the selection inputs are changed for a different configuration. Thus, the decoder requires just one clock cycle for reconfiguration purposes. There is only one output pin of the decoder at which the decoded message is available.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{viterbi_decoder.png}
\caption{The Viterbi Decoder Implementation}
\end{figure}

\textbf{Controller}

The controller takes in the control inputs and generates the appropriate signals for the ACS, BMU and the traceback memory. It configures the BMU through the \textit{selectR}
input, manages the ACS interconnections through the selectK input and controls the reading/writing of traceback memory. A behavioral description of the controller unit was written. It was implemented as a 3-state state machine. It accepts the whole incoming message in the first state, writing the decision bits to traceback memory during this phase. In the second state, it does the traceback to yield the input bits in reverse order, and finally it outputs the buffered input bits in the correct order in the third state.

**Viterbi Core**

The Viterbi Core (Figure 4.1) comprises of the ACS and BMU units. These were described behaviorally and generated multiple number of times as dictated by the maximum constraint length of the decoder. For the case of \( K = 9 \), we instantiated the core 128 times for the 256 state decoder. The BMU unit takes in \( selectR, selectK \), input(2:0), and an index for initialization purposes as inputs and generates two branch metrics as outputs. It uses the index to set the encoder to the appropriate value, as discussed in the previous chapter. It calculates only one branch metric uniquely and then simply negates it to produce the second branch metric. The width of the branch metrics was chosen to be 2 bits because for hard decision decoding of rate 1/2 and rate 1/3 codes, the Hamming distance can take on values 0-3. These are fed into the ACS unit, which takes in two additional path metric values from the path metric storage registers and generates two decision bits as well as two updated path metrics.

**Interconnection Network**

This defines the path from the ACS outputs to the path metric registers. It uses 2-input multiplexors to route the path metrics such that reconfigurability can be achieved. The operation of the multiplexors is governed by the selectK input as
described earlier by equations 4.3 to 4.5. Also part of the interconnection network are another set of multiplexors which connect the decision bits produced by the ACS units to traceback memory.

Traceback Memory

The traceback memory was implemented using the in-built RAM blocks of the Xilinx FPGA. The XCV800 FPGA has 28 4096-bit Synchronous RAMs each of which can be configured to have port widths from 1, 2, 4, 8, upto 16 bits. Since our design can support a constraint length upto 9, and we need to write in parallel 256 bits to memory, we needed to concatenate 16 of the RAMB4_S16 blocks of the XCV800 to generate the full traceback memory. Note that a port width of 16 implies a memory of size 256x256 or 8-bit address lines.

The way we have configured the traceback memory affects the maximum length of the message that can be decoded. Since the length of the memory is 256, the maximum packet length that can be supported for a constraint length 9 decoder is $256 - 9 + 1 = 248$. The last 8 inputs are the zeros for flushing the memory of the encoder.

5.3 Implementation Results

We verified the operation of the reconfigurable Viterbi decoder using the functional simulator the Xilinx tools provide. We could decode messages for different configurations of rate and constraint length by toggling the selection inputs to the decoder. The Xilinx tools generate mapping and timing reports as part of the implementation process. We have used these reports to derive information such as the percentage of
CLBs used in the device and the maximum clock frequency at which the circuit can run.

5.3.1 Area/Timing Analysis

From the mapping information, we verify that one of the four global clock buffers in the FPGA was used for routing the global clock inside the FPGA. Also, Out of the 166 available IOBs, 9 are used by the input/outputs of the Viterbi chip. To compare the CLB resources consumed by the reconfigurable decoder implementation with a standard fully parallel static implementation, we have summarized the equivalent gate count for the reconfigurable case and for the corresponding fixed one in Table 5.1. We can see that the gate count for the reconfigurable decoder is slightly higher because of the overhead associated with the multiplexors and the control logic that manages the reconfiguration. However, the overhead is just around 3% even when we compare a rate 1/2-1/3, $K = 3 – 7$ decoder with a corresponding fixed implementation. There seems to be very little degradation in performance in terms of the maximum clock speed.

<table>
<thead>
<tr>
<th>$K$</th>
<th>Area(gates)</th>
<th>Area Overhead(%)</th>
<th>Speed (MHz)</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>23,055</td>
<td>-</td>
<td>60.4</td>
</tr>
<tr>
<td>3 – 5</td>
<td>23,481</td>
<td>1.8</td>
<td>59.7</td>
</tr>
<tr>
<td>7</td>
<td>86,845</td>
<td>-</td>
<td>60.1</td>
</tr>
<tr>
<td>3 – 7</td>
<td>89,407</td>
<td>2.9</td>
<td>59.3</td>
</tr>
</tbody>
</table>

Shown in Table 5.2 are the statistics for Area usage for fixed implementations of constraint lengths 3, 5, 7 and 9. This has been done for a rate 1/2 decoder. As one would expect, increasing the constraint length by 2 approximately quadruples the area.
consumed by the Viterbi decoder. Also shown are the clock speeds in MHz at which
the FPGA can be driven for the above configurations. Since all the implementations
are parallel, there is not much difference in the clock speeds. It is important to

<table>
<thead>
<tr>
<th>$K$</th>
<th>Total Area(gates)</th>
<th>Slices(% used)</th>
<th>Speed(MHz)</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>19,404</td>
<td>1%</td>
<td>71.1</td>
</tr>
<tr>
<td>5</td>
<td>23,055</td>
<td>3%</td>
<td>60.4</td>
</tr>
<tr>
<td>7</td>
<td>86,845</td>
<td>12%</td>
<td>60.1</td>
</tr>
<tr>
<td>9</td>
<td>341,240</td>
<td>50%</td>
<td>-</td>
</tr>
</tbody>
</table>

Table 5.2 Fixed Implementation: Area/Speed

note that the clock speed of the circuit is not the actual decoding throughput of the
decoder. This is because the decoder begins to output the message bits only after is
has received the entire message and performed the traceback through all the states.
So, for a message frame of length $L$, it would take $L + K - 1$ clock cycles for the
decoder to receive the message, another $L + K - 1$ to do the traceback, and finally $L$
to output the decoded bits in the correct order.

$$\text{Number of cycles/frame} = 3L + 2K - 2 \quad (5.1)$$

$$\text{Decoding throughput} = (\text{Clock Frequency}) \frac{L}{3L + 2K - 2} \quad (5.2)$$

If we do a timing analysis of the decoder implementation, we can find the critical path
in our design. Since the BMU and the ACS units are combinational units, we would
expect the path from the input to the traceback RAM through the BMU and ACS
to be the longest path. With the current design mapping using the Xilinx software
tools, we can get a maximum clock rate of 80 MHz. Since the Xilinx place and route
tools do not always yield the most efficient place and route solution, if the routing is
done to minimize routing delays, it might be possible achieve clock speeds in excess
of 100 MHz. Another method to increase the speed of the design would be to pipeline the BMU/ACS units. This would degrade the latency of the decoder but improve the clock speed.

5.3.2 Comparison with a DSP implementation

It is interesting to compare the results of this parallel implementation with a serial implementation done on a TI TMS320C54x Digital Signal Processor [H96]. The TMS320C54x incorporates a special hardware unit to accelerate the Viterbi metric update computation. This is essentially an add-compare-select-store unit with dual accumulators and a splittable ALU which is capable of performing a Viterbi butterfly in four cycles.

\[
\text{Number of cycles/frame} = L(5(2^{K-2}) + 2^{K-5} + n(2^{n-1}) + 16.25)
\]  

(5.3)

where \(L\) is the frame length, \(K\) is the constraint length, and \(n\) denotes a rate \(1/n\) coding rate. We can see the clear dependence of the number of cycles per frame on the constraint length of the code. Table 5.3 and Table 5.4 compare this DSP implementation with our fully parallel one for the WLAN and WCDMA standards. However, the comparison is not completely fair because we have not taken power and cost into account. Also, a more recent TI C64x DSP has an on-chip Viterbi coprocessor which is exclusively meant to reduce the burden of Viterbi decoding from the DSP. It can decode up to 50 channels of 7.95 Kbps each giving an aggregate throughput of about 4 Mbps. It is apparent that an FPGA implementation is much closer at achieving the required data rates. For the WCDMA case, the DSP can attain decoding rates required for indoor environments, while the FPGA exceeds the required data rate requirement. For the WLAN, the FPGA is able to achieve the rates up to the 24 Mbit/s standard easily. Rates up to 54 Mbps and even 100 Mbps
can be achieved on the FPGA as technology improves to yield not only faster gate array devices but also software tools which efficiently map the hardware design onto the devices.

**Table 5.3** *DSP Benchmarks: WLAN/WCDMA*

<table>
<thead>
<tr>
<th>Standard</th>
<th>$R$</th>
<th>$K$</th>
<th>$L$</th>
<th>$Cycles/L$</th>
<th>$F_{max}$ (MHz)</th>
<th>Rate (Mbps)</th>
<th>Desired</th>
</tr>
</thead>
<tbody>
<tr>
<td>WLAN</td>
<td>1/2</td>
<td>7</td>
<td>192</td>
<td>40752</td>
<td>200</td>
<td>0.94</td>
<td>6-54</td>
</tr>
<tr>
<td>WCDMA</td>
<td>1/2</td>
<td>9</td>
<td>192</td>
<td>129840</td>
<td>200</td>
<td>0.29</td>
<td>2.5</td>
</tr>
<tr>
<td>WCDMA</td>
<td>1/3</td>
<td>9</td>
<td>192</td>
<td>131376</td>
<td>200</td>
<td>0.29</td>
<td>2.5</td>
</tr>
</tbody>
</table>

**Table 5.4** *FPGA Benchmarks: WLAN/WCDMA*

<table>
<thead>
<tr>
<th>Standard</th>
<th>$R$</th>
<th>$K$</th>
<th>$L$</th>
<th>$Cycles/L$</th>
<th>$F_{max}$ (MHz)</th>
<th>Rate (Mbps)</th>
<th>Desired</th>
</tr>
</thead>
<tbody>
<tr>
<td>WLAN</td>
<td>1/2</td>
<td>7</td>
<td>192</td>
<td>588</td>
<td>80</td>
<td>26.1</td>
<td>6-54</td>
</tr>
<tr>
<td>WCDMA</td>
<td>1/2</td>
<td>9</td>
<td>192</td>
<td>592</td>
<td>80</td>
<td>26.1</td>
<td>2.5</td>
</tr>
<tr>
<td>WCDMA</td>
<td>1/3</td>
<td>9</td>
<td>192</td>
<td>592</td>
<td>80</td>
<td>25.9</td>
<td>2.5</td>
</tr>
</tbody>
</table>
Chapter 6

Conclusions and Future Work

6.1 Conclusions

As mobile and wireless communication becomes increasingly ubiquitous, the need for dynamic reconfigurability of hardware shall pose fundamental challenges for communication algorithm designers as well as hardware architects. In this thesis, we attempted to solve this problem for the particular case of the Viterbi decoder, which is a crucial component at the physical layer of most wireless communication systems.

We have proposed a new Viterbi decoder architecture which allows for run-time adaptation to different code configurations i.e. set of code rates, constraint lengths and generator polynomials. The new architecture is a fully parallel structure for all configurations and thus suitable for very high data rate decoding. We identified some properties of the decoder trellis structure and used them to design a basic building block which can be replicated to yield different decoders. We then coded this design in VHDL and implemented it on a Xilinx XCV800 FPGA. We have showed that this architecture is capable of supporting data rates upto 26 Mbps for constraint lengths 3 - 9 and for rates 1/2 to 1/3. The area overhead associated with a reconfigurable decoder implementation which can support constraint lengths 3 to 7, when compared to a static constraint length 7 decoder is just around 2.9%. Thus, our design does not consume much more resources than an equivalent non-reconfigurable decoder but yields the facility of run-time reconfiguration. Also, we benchmarked the performance of our FPGA implementation against a DSP implementation which uses a specialized
instruction set for the Viterbi. We found that for data rates in the range 2-54 Mbps, a DSP solution is not feasible since a highly hardware intensive solution is required to support these high data rates. The maximum data rate achievable by the most powerful current DSP is upto 4 Mbps.

6.2 Future Work

We believe that the proposed architecture has considerable application in the context of multi-tier networks, where different decoding standards shall have to be supported on one device. As part of our future work, we would like to develop an interface to the decoder, so that it can be a part of an overall communications system. For example, as part of the WLAN system, the Viterbi decoder needs to get its inputs from the output of an FFT block while as part of the WCDMA system, it is fed from the multi-user detector. One solution would be to implement this architecture as a coprocessor to a DSP which would then be free to run other receiver algorithms.

Also, the current implementation of the decoder is limited to decoding of message packets of maximum length 248. This limitation arises because we are using the in-built RAM blocks of the Xilinx FPGA. Since the maximum packet size for a WLAN data packet is 4096 bits, we need to be able to support decoding for longer messages. We could do this by either increasing the size of the traceback memory, or by adopting a traceback approach which relies on the unification property mentioned in 3.1.5. In the latter approach, we need to use memory banks of length only five times the constraint length[RC90], and using this approach, we can start decoding without having to wait for the entire message. This should not only improve our decoding latency but also increase the decoding throughput to three times it’s current value, the corresponding loss in decoding accuracy being negligible. Another step
towards making the decoder more realistic would be to provide the capability of soft
decision decoding, since soft decision decoding is usually employed in all practical
communication systems.

Finally, as a follow-up to this work, we would like to investigate the possibility
of a reconfigurable architecture which is not fully parallel for the highest constraint
length. This is interesting because we could then potentially reconfigure the decoder
for any large value of constraint length without being limited by the need to have a
correspondingly large amount of hardware resources.
Bibliography


