# Implementation of a Turbo Decoder on a Configurable Computing Platform

Jason R. Hess

Thesis submitted to the faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Master of Science
in
Electrical Engineering

Peter M. Athanas, Chair
Mark T. Jones
Jeffrey H. Reed

September 17th, 1999
Blacksburg, VA

Keywords: Turbo Codes, FPGA, Configurable Computing
Copyright 1999, Jason R. Hess

# Implementation of a Turbo Decoder on a Configurable Computing Platform

Jason R. Hess

(ABSTRACT)

Turbo codes are a new class of codes that can achieve exceptional error performance and energy efficiency at low signal-to-noise ratios. Decoding turbo codes is a complicated procedure that often requires custom hardware if it is to be performed at acceptable speeds. Configurable computing machines are able to provide the performance advantages of custom hardware while maintaining the flexibility of general-purpose microprocessors and DSPs.

This thesis presents an implementation of a turbo decoder on an FPGA-based configurable computing platform. Portability and flexibility are emphasized in the implementation so that the decoder can be used as part of a configurable software radio. The system presented performs turbo decoding for a variable block size with a variable number of decoding iterations while using only a single FPGA. When six iterations are performed, the decoder operates at an information bit rate greater than 32 kbps.

# Acknowledgements

There are several people who deserve recognition for the assistance they provided in the successful completion of my graduate work. First and foremost, I would like to thank my advisor, Dr. Peter Athanas, for his help and guidance in not only this effort, but in all of the projects we have attempted in the three-and-a-half years of our association. His friendly demeanor, keen insight, and amazingly upbeat attitude in the face of adversity continue to inspire his students and colleagues alike.

I would also like to thank the other members of my committee, Dr. Mark Jones and Dr. Jeff Reed, for their suggestions and support throughout this effort and for serving on my committee. Special thanks goes out to Dr. Brian Woerner, my committee member-by-proxy, who was more than willing to help out on short notice and also provided many useful suggestions.

I am also grateful to my fellow students Yufei Wu and Matt Valenti for their support in helping me understand the theory of turbo codes. I am doubly grateful to Yufei for the invaluable debugging support she provided during the latter days of this project.

Wendy Akers and Martha McCollum provided priceless administrative and emotional support throughout my time with the Configurable Computing team, and without their help, this document would still have colons used improperly.

My wonderful parents and friends have given me the support I needed to complete what at times seemed to be an insurmountable task. Their simultaneous desires for my success and my sanity gave me much-needed support when the going was tough. I would especially like to recognize Dave and Paul for providing me with a place to stay in Blacksburg after the expiration of my lease left me effectively homeless.

This page would not be complete without thanking the Via family for their support of the Bradley Fellowship, which funded this work. The Fellowship provides students with the ultimate in research freedom and makes possible research into areas that would never be attempted for lack of funding. This is a laudable goal that cannot be praised enough.

# Contents

**List of Figures** ........ vi

**List of Tables** ........ viii

**List of Variables** ........ ix

**Chapter 1. Introduction** ........ 1

1.1 Motivation ........ 1
1.2 Thesis Organization ........ 2

**Chapter 2. Turbo Coding** ........ 3

2.1 Traditional Coding Schemes ........ 3
2.1.1 Block Codes ........ 6
2.1.2 Convolutional Codes ........ 8
2.2 Turbo Codes ........ 11
2.3 Decoding Turbo Codes ........ 14
2.4 Decoding Algorithms for Turbo Codes ........ 17
2.4.1 MAP Algorithm ........ 18
2.4.2 Max-Log-MAP and Log-MAP Algorithms ........ 20
2.4.3 Soft Output Viterbi Algorithm ........ 22
2.5 Designing Turbo Codes ........ 22
2.6 Summary ........ 24

**Chapter 3. Configurable Computing** ........ 25

3.1 Field Programmable Gate Arrays (FPGAs) ........ 25
3.1.1 Xilinx 4000 series FPGAs ........ 26
3.1.2 Other FPGAs ........ 28
3.2 WILDFORCE Platform ........ 29
3.3 Application Design for the WILDFORCE ........ 32
3.4 Applications of CCMs ........ 35
3.5 Summary ........ 36

**Chapter 4. Implementation of a Turbo Decoder in Configurable Hardware** ........ 37

4.1 Implementation Goals ........ 37
4.2 System Design Parameters ........ 38
4.3 Implementation Strategy ........ 40
4.4 Interleaver Module ........ 45
4.4.1 Interleaving Strategies ........ 45
4.4.2 Pattern Storage Strategies ........ 48
4.4.3 Module Implementation ........ 49

4.5 Adder Module ........ 51
4.6 Decoder Module ........ 53
4.6.1 Differences Between the First and Second Decoders ........ 53
4.6.2 Module Implementation ........ 54
4.7 Host Code ........ 56
4.8 Summary ........ 57

**Chapter 5. Results ........ 58**

5.1 Test Setup ........ 58
5.2 Implementation Statistics ........ 58
5.3 Functional Results ........ 61
5.4 Summary ........ 62

**Chapter 6. Conclusion ........ 67**

6.1 Summary ........ 67
6.2 Future Work ........ 68
6.2.1 Performance Enhancements ........ 69
6.2.1 Other Enhancements ........ 70

**References ........ 71**

**Vita ........ 74**

# List of Figures


2.1: Diagram of a communications system with forward error correction (FEC). ............ 5

2.2: Lookup table representation of a (7,4) block encoder with $d_{min} = 3$. ......................... 7

2.3: Block diagram of an encoder for a systematic cyclic code. ..................................... 8

2.4: Block diagram for a convolutional encoder............................................................ 9

2.5: Difference between conventional convolutional and RSC encoders. ...................... 11

2.6: Block diagram of a general turbo encoder............................................................ 12

2.7: Parallel vs. serial concatenated codes. ................................................................. 13

2.8: Row interleaving used in turbo codes. .................................................................. 14

2.9: Conventional turbo encoder for code rates *R*=1/3 (a) and *R*=1/2 (b)....................... 15

2.10: Turbo decoder schematic. ................................................................................. 16

2.11: Turbo decoder in iterative feedback mode. ......................................................... 17

3.1: Diagram of a Xilinx 4000 series configurable logic block (CLB)........................... 27

3.2: Picture of the WILDFORCE CCM. ...................................................................... 29

3.3: Memory interface and timing for a write directly after a read on the WILDFORCE 31

3.4: Traditional design process for CCMs.................................................................. 32

4.1: Diagram of a {7,5} RSC encoder. ........................................................................ 39

4.2: Turbo decoder schematic. .................................................................................. 40

4.3: Using the memory as an interface between modules. ............................................ 41

4.4: Top-level diagram of turbo decoder implementation............................................ 43

4.5: Illustration of why two patterns are required for a read/read interleaving scheme.... 47

4.6: Illustration of the use of a single pattern for a read/write interleaving scheme......... 48

4.7: State diagram for the interleaver module ............................................................. 51

4.8: State diagram for the adder module ................................................................... 52

5.1: Decoder layout on XC4062XL chip...................................................................... 59

5.2: Comparison of simulation and hardware results showing bit error rate (a) and frame error rate (b) for *N*=1024, 6 iterations, no puncturing and varying $E_b/N_0$. ............... 63

5.3: Comparison of simulation and hardware results showing bit error rate (a) and frame error rate (b) for *N*=1024, $E_b/N_0$=1.5 dB, no puncturing, and varying iterations....... 64

5.4: Hardware results showing bit error rate (a) and frame error rate (b) for $E_b/N_0$=1.5 dB, 4 iterations, no puncturing, and varying block size. ........................................ 65
5.5: The effect of puncturing on the hardware results for bit error rate (a) and frame error rate (b) for $N$=1024, $E_b/N_0$=1.5 dB, and 6 iterations. ........................................ 66

# List of Tables

3.1: Statistics for some members of the 4000XL series of FPGAs ................................ 28

4.1: Parameters used in turbo decoder design ................................................................ 38

4.2: Steps in decoding operation from perspective of control unit ................................. 44

4.3: Memory map for turbo decoder ................................................................................ 45

5.1: Resource utilization for turbo decoder implementation ........................................... 59

5.2: Timing information for the turbo decoder ................................................................ 60

# List of Variables

| | |
|---|---|
| $a$ | Fading channel amplitude |
| $B$ | Bandwidth |
| $C$ | Channel capacity |
| $\mathbf{d}$ | Information bits vector |
| $\hat{d}$ | Estimate of decoded message bit |
| $d_h$ | Hamming distance |
| $d_{min}$ | Minimum Hamming distance |
| $E_s$ | Signal energy |
| $f_c$ | Correction function |
| $\mathbf{G}$ | Generator matrix |
| $g(D)$ | Generator polynomial |
| $\mathbf{I}$ | Identity matrix |
| $I$ | Number of decoding iterations |
| $K$ | Constraint length |
| $k$ | Message length |
| $L$ | Number of stages in shift register |
| $L_{a1}$ | A priori information from decoder 1 |
| $L_{a2}$ | A priori information from decoder 2 |
| $L_c$ | Channel reliability factor |
| $L_{e1}$ | Extrinsic information from decoder 1 |
| $L_{e2}$ | Extrinsic information from decoder 2 |
| $M$ | Encoder memory |
| $\mathbf{m}$ | Message vector |
| $N$ | Block length |
| $N_0$ | Noise power spectral density |
| $N$ | Noise power |
| $N_e$ | Number of RSC encoders |
| n | Codeword length |

| | |
|---|---|
| $\mathbf{p}$ | Parity vector |
| $R$ | Code rate |
| $S$ | Encoder state (also signal power) |
| $t$ | Correctable bit errors |
| $w_h$ | Hamming weight |
| $\mathbf{x}$ | Codeword vector |
| $x$ | Systematic information |
| $\tilde{x}$ | Interleaved systematic information |
| $\mathbf{y}$ | Received signals vector |
| $y_1$ | Parity information for encoder 1 |
| $y_2$ | Parity information for encoder 2 |
| $y^{(s)}$ | Received systematic information |
| $y^{(p)}$ | Received parity information |
| $\mathbf{Z}$ | Parity matrix |
| $\alpha$ | Forward branch metric (also Interleaver) |
| $\alpha^{-1}$ | Deinterleaver |
| $\beta$ | Reverse branch metric |
| $\gamma$ | Branch transition probability |
| $\Lambda_1$ | Log-likelihood ratio from decoder 1 |
| $\Lambda_2$ | Log-likelihood ratio from decoder 2 |
| $\lambda^i(S)$ | Joint probability of message bit being *i* and state being S, given **y** |
| $\sigma$ | Numerator of $\lambda^i(S)$ expression |

# Chapter 1. Introduction

The availability of wireless technology has revolutionized the way communications is done in our world today. Cellular and satellite technology make it possible for people to be connected to the rest of the world from anywhere. With this increased availability comes increased dependence on the underlying systems to transmit information both quickly and accurately. Because the communications channels in wireless systems can be much more hostile than in "wired" systems, voice and data must use forward error correction coding to reduce the probability of channel effects corrupting the information being transmitted. A new type of coding, called turbo coding, can achieve a level of performance that comes closer to theoretical bounds than more conventional coding systems.

## *1.1 Motivation*

Turbo codes, which use parallel concatenated convolutional codes with iterative feedback decoding, can achieve phenomenally low error rates in environments with low signal-to-noise ratios. However, the complexity of the decoding algorithms for turbo codes has detracted from their commercial feasibility.

This thesis presents the implementation of a turbo decoder on a configurable computing machine (CCM). Use of a CCM allows complex computational structures to be implemented with much higher performance than with a general-purpose microprocessor and at much less cost than with a custom designed chip.

This decoder implementation is targeted for use in a configurable software radio system, which is slated to be integrated onto a single platform at a future date. Although this has no impact on the details of the decoding algorithm itself, the existence of other modules that the decoder must eventually interface with sets performance goals for the decoder so that it will not be a bottleneck in the overall system. Also, since the decoder will be retargeted at some point, portability must be considered when making implementation decisions.

## *1.2 Thesis Organization*

This thesis presents the details and results of the implementation of a turbo decoder in configurable hardware, as well as background information needed to understand the design choices that were made and the results that were obtained. It is organized as follows.

Chapter 2 discusses the theory of turbo codes. The theory is prefaced by a review of some of the basic coding theory concepts that are crucial to understanding turbo codes. Encoder and decoder structures, as well as several different decoding algorithms are also discussed.

Chapter 3 presents an introduction to configurable computing. It begins by presenting the general concepts of configurable computing, highlighting them with examples of the specific hardware that is used for this implementation. The traditional design cycle for CCM applications is discussed, as are the types of applications that are well-suited for implementation in configurable hardware.

Chapter 4 provides details of the actual implementation. The goals and target parameters for the design are described before delving into a discussion of the major design decisions. Finally, each of the major modules of the design is described in detail.

Chapter 5 presents experimental results obtained from the of the implementation. The experimental setup is described, followed by utilization and timing statistics for the implementation. Finally, the results from the decoder are compared to expected results from simulation for the purpose of functional verification.

Chapter 6 summarizes the work presented here and offers several ideas for improving the current version of the system in performance, modularity, and ease of use.

# Chapter 2. Turbo Coding

While turbo coding is a relatively new development in the field of coding theory, it is built on a foundation that has been developed for several decades. Indeed, turbo codes are fundamentally composed of elements that have been long associated with more traditional forms of coding, namely convolutional and block codes. For this reason, it is useful to review these fundamentals of coding theory before discussing turbo codes in further detail.

## *2.1 Traditional Coding Schemes*

In any real communication system, transmission over a channel will cause the information received at the other end to differ from what was originally transmitted. This is because the channel injects noise into the original signal. In digital communication systems, if the power in the noise is large enough relative to the power in the original signal, transmitted bits can be corrupted to the point that the receiver makes incorrect decisions about the data that was transmitted. If too many of these errors occur (in data communications "too many" can often mean one), the communication system will not be usable. Therefore, techniques must be used to reduce the probability of an error occurring for a given signal-to-noise ratio (SNR).

In 1948, Claude Shannon [1] proved that the probability of such an error occurring could be theoretically reduced to zero, given that the transmission rate does not exceed the capacity of the channel, $C$. In the case of transmission of a signal of average power $S$ over a channel of bandwidth $B$ that injects additive white Gaussian noise (AWGN) with power $N$, the capacity is given by

$$C = B\log_2\left(1+\frac{S}{N}\right) \tag{2.1}$$

For a communication system to begin to approach this limit, it cannot transmit the data "as is" with no additional processing; a single instance of high noise could cause a bit error.

In computer networks, this problem is typically solved by an automatic repeat request (ARQ) scheme [2]. In such a scheme, a checksum or other redundancy is added to a data frame to allow error detection at the receiver. If no error is detected, the receiver sends an acknowledgement (ACK) back to the sender. If there is a frame error, then the receiver either sends a negative acknowledgement (NACK) to the sender, or it does nothing and lets the sender retransmit the data after it times out of the period in which it waits for an ACK.

The ARQ technique is not typically used in wireless communications for a couple of reasons. One reason is that ARQ systems require duplex communication capabilities. While this is relatively cheap and easy to do on a wired network, it requires an increase in receiver complexity for wireless links that is often unacceptable. Also, in computer networks the probability of bit errors caused by noise is very low in many cases, so a frame retransmission is rarely necessary[1]. This is in great contrast to wireless networks, where bit errors are usually much more frequent. The number of frame retransmissions required to implement an ARQ scheme on a system with low SNR would slow throughput to a near-halt.

Since the ARQ scheme is impractical for many communication systems, forward error correction (FEC) schemes are often used instead. FEC codes are designed to improve the decisions that the receiver makes by giving it enough information to correct some of the errors that the channel has introduced into the signal. This performance improvement is largely brought about by two techniques, *redundancy* and *noise averaging* [3]. By adding redundant bits to the digital message, the encoder accentuates the uniqueness of the transmitted message. This eases the decision burden of the receiver because limiting allowable sequences to a fraction of those possible means that multiple bits will have to be corrupted by noise for the message to be decoded incorrectly. In noise averaging, the code is designed so the bits of the message affect many bits of the encoder output, allowing the receiver to average out effects of the noise over a large number of received bits.

---

[1] This is especially true for ARQ schemes used at the data-link layer for point-to-point links. While frame loss is much more frequent at higher levels in the network architecture (TCP/IP), this is often due to the entire frame being dropped at a switch or router. In this case, the entire frame would need to be retransmitted anyway, so it is not an example of the entire frame being retransmitted to fix an error in a single bit [2].

While FEC schemes can provide exceptional improvements in performance for noisy channels, their benefits do not come without costs. The use of coding requires additional processing at the transmitter and especially at the receiver, as can be seen in Figure 2.1. While encoders do not typically require large amounts of processing, the complexity of decoders can be significant. For more complicated coding techniques, the complexity of optimal decoders prohibits their implementation because they require large amounts of hardware and large amounts of processing time. In these cases, sub-optimal approximations that require less processing and provide inferior performance can be used. Also, coding reduces the data rate by a constant factor for a fixed bandwidth due to the transmission of redundant bits. The general challenge in coding system design is to provide enough randomness in the code to provide good performance in noise while providing enough structure in the code to make its decoding feasible.

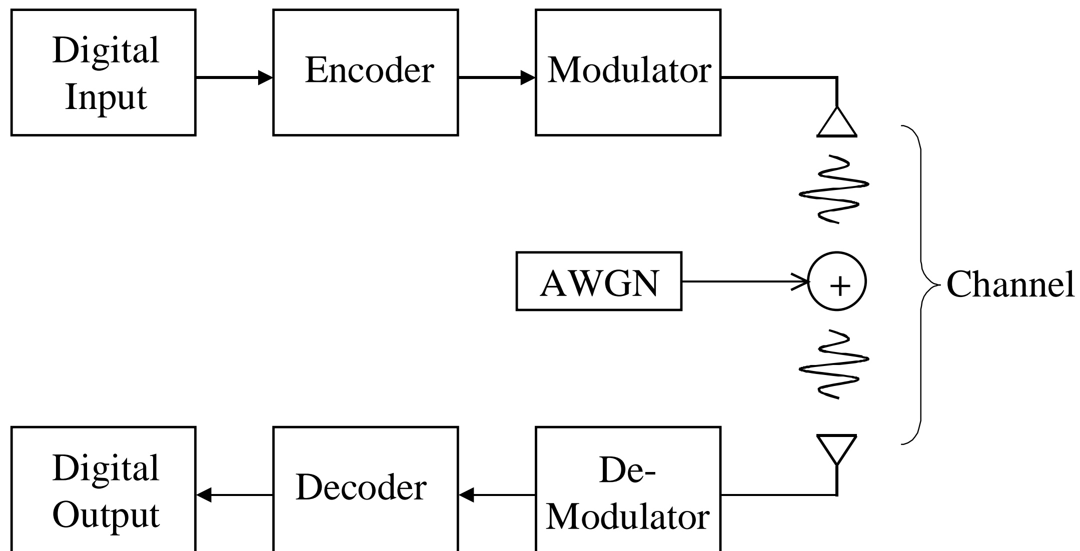



Figure 2.1: Diagram of a communications system with forward error correction (FEC).

Traditionally, coding schemes have been divided into two distinct categories: block coding and convolutional coding. Although the lines of this division have begun to blur as advanced coding schemes, like turbo coding, use elements of both types of codes, it is still useful to discuss each type of code separately to accentuate the differences between the schemes.

### 2.1.1 Block Codes

As the name implies, block coding is performed by partitioning the data stream into fixed-size messages for encoding. The messages of $k$ bits are mapped to codewords of length $n$ bits. The *code rate*, $R$, for an $(n, k)$ block code is then given by

$$R = \frac{k}{n}. \tag{2.2}$$

For some block codes, the codeword consists of the $k$ original message bits with $n - k$ parity bits appended to it. The codeword $\mathbf{x}$ is thus represented in matrix notation as

$$\mathbf{x} = [\mathbf{m} \quad \mathbf{p}] \tag{2.3}$$

where $\mathbf{m} = [m_1, m_2, \ldots m_k]$ and $\mathbf{p} = [p_1, p_2, \ldots p_{n-k}]$ are the message and parity bits, respectively. Because the message itself is part of the codeword, this code is referred to as *systematic*.

The encoder calculates $\mathbf{x}$ by a matrix multiplication of $\mathbf{m}$ with a generator matrix, $\mathbf{G}$.

$$\mathbf{x} = \mathbf{mG} \tag{2.4}$$

For the systematic code described above, $\mathbf{G}$ is a concatenation of a $k \times k$ identity matrix $\mathbf{I}$, which generates $\mathbf{m}$, and a $k \times (n-k)$ parity matrix $\mathbf{Z}$, which generates $\mathbf{p}$.

$$\mathbf{G} = [\mathbf{I} \quad \mathbf{Z}] \tag{2.5}$$

A code generated in this manner is called *linear*.

The Hamming distance between any two codewords $\mathbf{x_1}$ and $\mathbf{x_2}$ is the number of bit positions in which their binary representations differ. This will be expressed as $d_h(\mathbf{x_1}, \mathbf{x_2})$. The Hamming weight, $w_h$, of a codeword $\mathbf{x}$ is the number of 1's in its binary representation or, alternatively,

$$w_h(\mathbf{x}) = d_h(\mathbf{x}, \mathbf{0}). \tag{2.6}$$

If the Hamming distance is computed for all possible codeword pairs, then the minimum Hamming distance, $d_{min}$, for the code is found by choosing the smallest calculated distance. For a linear code, $d_{min}$ is equivalent to the minimum nonzero $w_h$. This is significant because the number of bit errors that can be corrected by the code, $t$ is given by

$$t = \left\lfloor \frac{d_{\min} - 1}{2} \right\rfloor. \tag{2.7}$$

Thus, by designing the code for a large $d_{min}$, the error-correcting power of the code will be greater. However, increasing $d_{min}$ requires[2] an increase in $n$ relative to $k$, which decreases the efficiency of the code, as defined by the code rate, $R$. By using large codewords, $R$ can be brought up to acceptable levels for a given $d_{min}$, but encoder and decoder complexity scale up with $n$, so there is a limit to the amount that the code performance can be improved by increasing codeword length.

Since block encoding is a *memoryless* one-to-one mapping, it can be viewed most simply as a look-up operation. The encoder uses the $k$-bit message as an index to a table that contains the $n$-bit codeword that it outputs, as shown in Figure 2.2. Such a look-up table would have $2^k$ entries of $n$ bits. For very large codewords, this can be prohibitively complex to implement for both encoding and (especially) decoding.

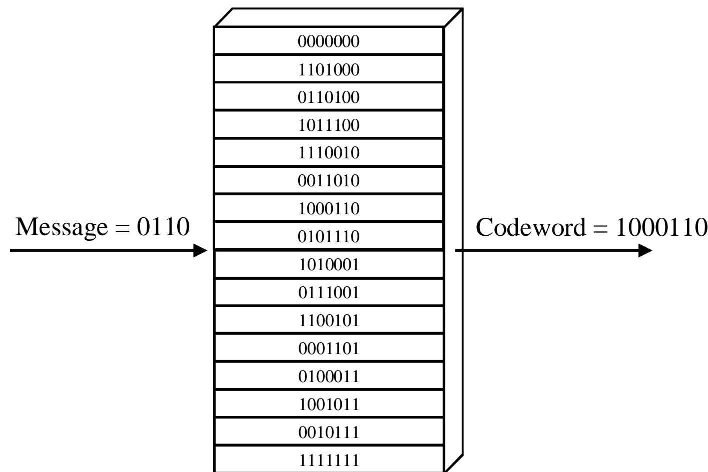



Figure 2.2: Lookup table representation of a (7,4) block encoder with $d_{min} = 3$.

A subset of block codes called *cyclic codes* can provide a solution to the implementation woes of long block codes. A sample systematic cyclic encoder is shown in Figure 2.3. By using a linear feedback shift register (LFSR), the parity bits are calculated as the message bits are being sent out of the encoder. Then, the switches are flipped, and the parity bits stream out. This implementation is much less complex than

[2] In order to maximize $d_{min}$ for a given codeword length, care must be taken when choosing **Z**. Although techniques exist for doing this, they are beyond the scope of this discussion. For more information, see [4].

the general case for block codes, because it only requires $n$ - $k$ memory elements in the LFSR to implement the encoder. Cyclic codes also have the beneficial property that their encoder structure can be represented by a single generator polynomial

$$g(D) = g_0 + g_1 D + g_2 D^2 + \ldots + g_{n-k-1} D^{n-k-1} + D^{n-k}. \tag{2.8}$$

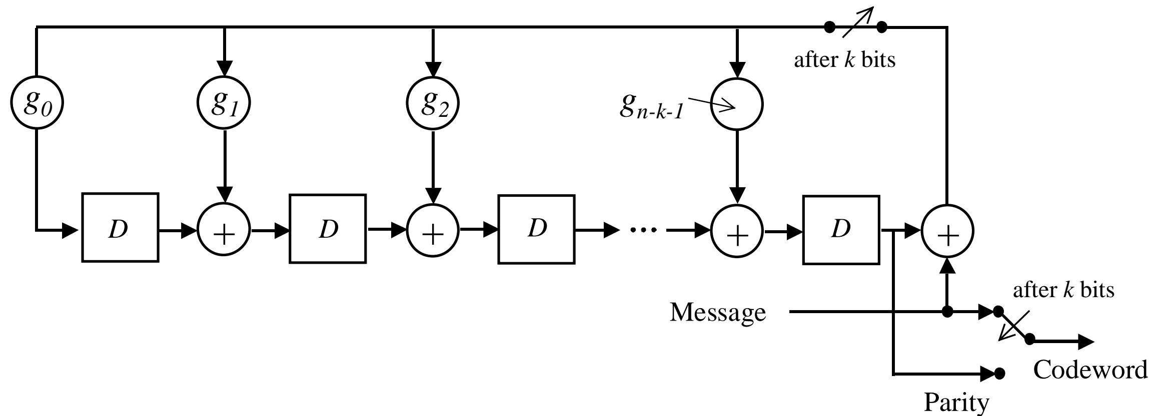



Figure 2.3: Block diagram of an encoder for a systematic cyclic code.

Decoding block codes is a much more complicated operation than encoding due to the bit errors caused by the noisy channel. If the look-up table scheme discussed for the encoder is extended to decoding, there are two implementation options. The $n$-bit block could be used as an index to a massive $2^n \times k$ bit look-up table, or the $n$-bit block could be compared with all of the entries of a $2^k \times n$ table. Neither of these options, while providing optimal results, is computationally feasible for all but the shortest block codes. One tractable, but sub-optimal alternative is to use an algebraic decoding strategy like the Berlekamp algorithm [4], which allows decoding of all combinations of errors up to the error correction capability, $t$. In the case of cyclic codes, their increased structure allows for much simpler decoding algorithms like the Meggitt decoder [5] or the Berlekamp-Massey algorithm [6].

### 2.1.2 Convolutional Codes

Like block codes, convolutional codes are characterized by encoders that output $n$ bits of encoded data for every $k$ bits of message data they receive, for a code rate $R = \frac{k}{n}$. However, unlike block codes, there is no need for the data to be partitioned into fixed-size messages in convolutional codes; the data streams can be encoded in a semi-infinite

stream. This is advantageous because the decoder does not have to wait for an entire block to be received before decoding can commence. For long block codes, this can cause intolerable latencies for sensitive applications like two-way voice communications.

Convolutional encoders, unlike block encoders, have *memory*. This means that every $k$-bit input affects the value of $K$ $n$-bit outputs of the encoder, where $K$ is a code parameter called the *constraint length*. A diagram of a convolutional encoder is shown in Figure 2.4. The memory of the encoder is implemented by a $k \times L$ shift register, where $L$ is the number of stages in the shift register. The memory of the encoder, $M$, can be found by

$$M = kL. \tag{2.9}$$

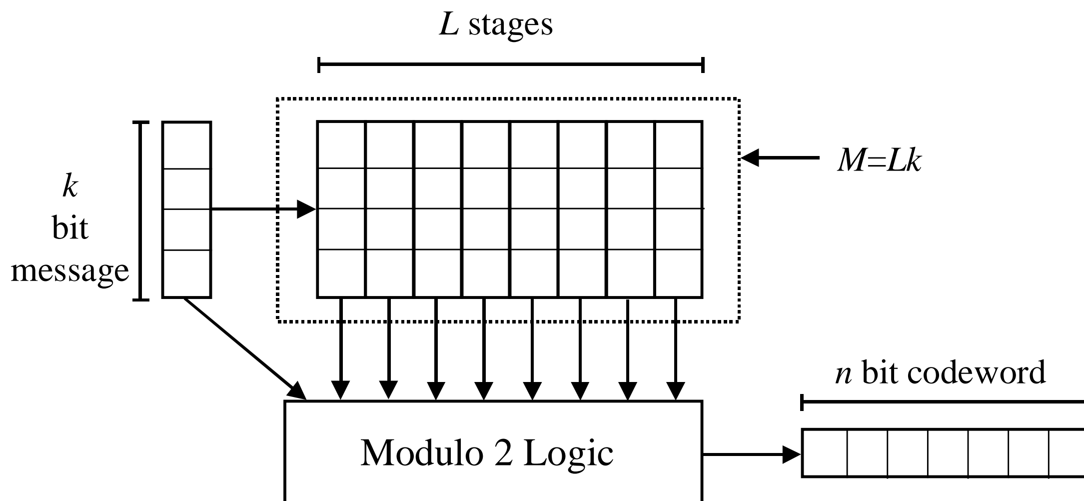



Figure 2.4: Block diagram for a convolutional encoder.

On each clock cycle, the $n$ output bits are computed by linear combination of the $k$ bits at the encoder input and the $M$ bits in the encoder memory. The linear combinations used to calculate the output bits can be expressed as a set of $n \times k$ generator polynomials, $g_{(a,b)}(D)$.

Turbo codes make use of a special case of the convolutional encoder with $k = 1$. By using only one input stream, the decoder complexity can be kept at a tractable level. This scheme has a code rate of $R = \frac{1}{n}$, which can result in an inefficient (low information throughput) code for all but the smallest values of $n$. One solution to this problem is to systematically delete some of the bits from the encoder output stream, a process known

as *puncturing*. For instance, a code with rate 1/3 can be increased to a rate of 2/5 if every sixth output bit is not transmitted over the channel. While puncturing reduces the effectiveness of the code, it is useful for increasing the code rate without a corresponding increase in decoder complexity.

The convolutional encoders used in turbo coding are also systematic and *recursive*. As with block codes, systematic implies that the encoder inputs are part of the outputs. Thus, one of the $n$ output bits for a single iteration of the encoder is the incoming message bit.

Traditional convolutional encoders do not employ feedback, and thus can be thought of as finite impulse response (FIR) filters. Recursive convolutional encoders have a feedback component that makes the encoder behave like an infinite impulse response (IIR) filter. Figure 2.5 shows the differences between conventional convolutional encoders and recursive, systematic convolutional (RSC) encoders with $k$ = 1 and similar code generators $G$.

Another advantage of convolutional codes relative to block codes lies in the decoding algorithms. Typically, block decoders rely on the demodulator to make bit decisions on the data received from the channel[3]. This practice of using *hard decisions* in decoding helps to simplify the operation of the decoder. The drawback of this method is that the decoder does not have any information about how close the received bit was to the decision threshold. Consider a system that uses polar NRZ line signaling with amplitudes of ±1 V. If hard decisions are used, received values of 0.8 V and 0.03 V will both be sent to the decoder as binary 1's even though one of the samples has a much higher likelihood of being correct than the other. The alternative is to pass the samples from the demodulator to the decoder without making a decision on them. Then, the decoder can quantify the confidence associated with each sample, which reduces the overall probability of error and provides coding gains of up to 2.5 dB [9] over systems using hard decisions. Widely available decoders for convolutional codes can use such *soft decisions* to obtain superior code performance for low SNR environments.

---

[3] Block encoders have traditionally used hard decisions because using soft decisions made the decoding operation too complex. However, recent research has been done conducted to explore the use of soft decisions for various block codes [7,8].

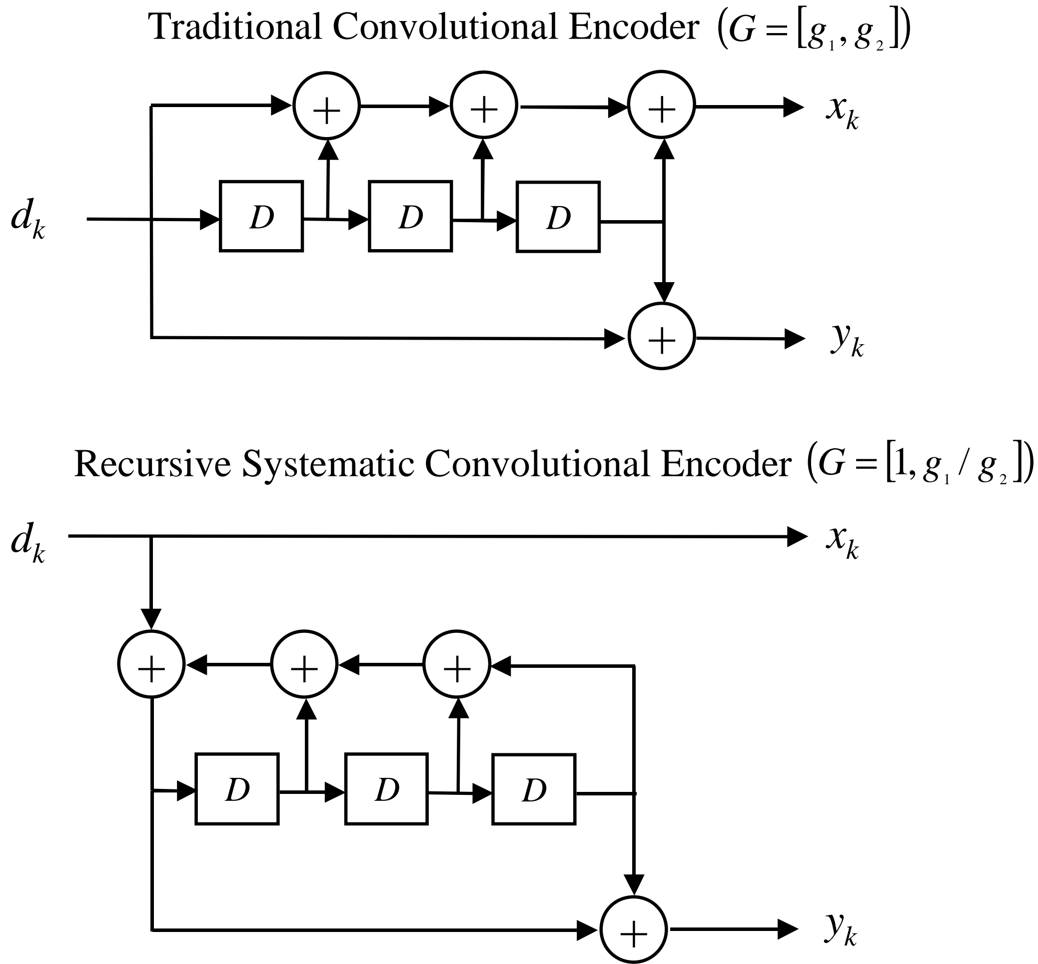



Figure 2.5: Difference between conventional convolutional and RSC encoders.

Although there exist many algorithms for decoding convolutional codes, the one that is by far the most widely used is the Viterbi algorithm (VA). Proposed in 1967 [10] for convolutional decoding, it is a maximum likelihood sequence detection algorithm that also has application to such varied areas as overcoming intersymbol interference and text recognition [11]. Although a complete exposition of the operation of the VA is outside the scope of this discussion, a detailed description can be found in the original paper [10] or in most coding theory texts [12,13].

## 2.2 Turbo Codes

The RSC codes introduced in the previous section are generally considered most suitable for application in low SNR environments, where they outperform a non-systematic

convolutional (NSC) code that has comparable memory. At higher SNRs, the NSC code will generally provide a lower bit error rate (BER) than the RSC code[4].

In 1993, Berrou and his colleagues described a new class of error-correcting code that used RSC encoders, but could achieve better performance than NSC codes at any SNR [15]. In fact, one example code in this class could come within 0.7 dB of the Shannon limit for a BER of $10^{-5}$. This innovation was dubbed *turbo coding*. A turbo encoder (Figure 2.6) uses parallel concatenation of multiple RSC encoders with pseudo-random *interleavers* (symbolized by $\alpha_k$ in Fig. 2.6) preceding all but one of the constituent encoders.

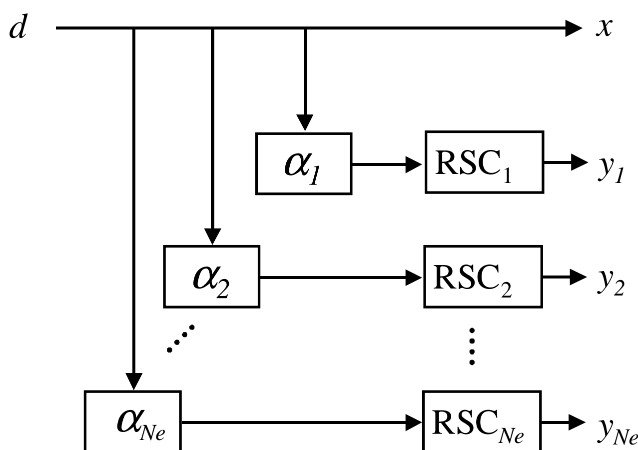


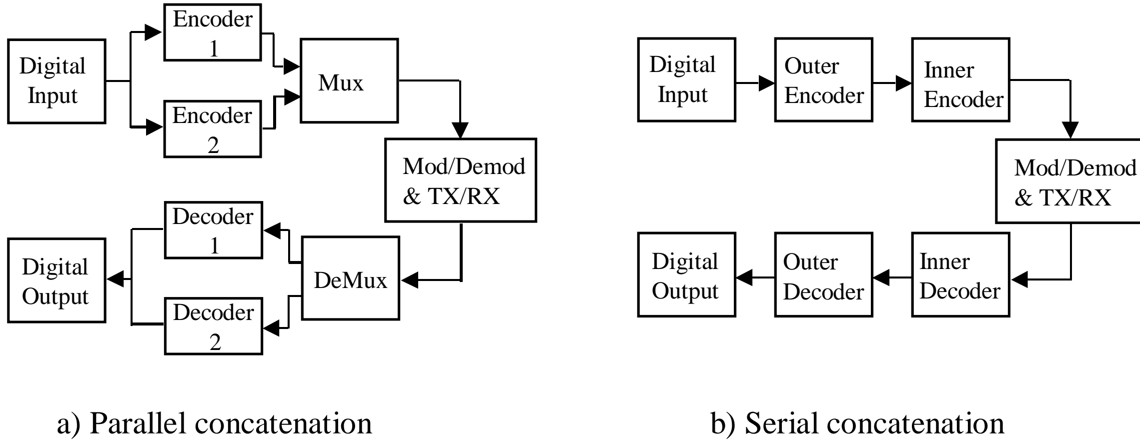

Figure 2.6: Block diagram of a general turbo encoder.

Although [15] touts turbo codes as a new class of convolutional codes, they can also be considered to be block codes. The data to be transmitted is partitioned into blocks so that the interleavers can perform their operation.[5]

Parallel concatenation (Figure 2.7a) means that multiple encoders are acting on the same data stream, and the outputs of the encoders are concatenated to form the overall encoder output. The code is then decoded by several corresponding decoders in the receiver. This is in contrast to serial concatenation, which is a technique whereby data is encoded in multiple encoders in a serial fashion in order to take advantages of properties of each code. This is shown in Figure 2.7 (b), where the data is first encoded by an outer

[4] Although this is true in the general case, it was proven in [14] that for high code rates ($R \geq 2/3$), RSC codes can be found that outperform comparable NSC codes at all SNRs.
[5] While traditional turbo codes use a block structure, stream-based versions of turbo codes have also been investigated [16].

encoder whose output is then encoded by an inner encoder. After demodulation, the inner decoding is performed first, followed by the outer decoding to produce the overall estimate of the transmitted message. One such arrangement [17], has a Reed-Solomon (RS) block code as the outer code and a convolutional code as the inner code to take advantage of the performance at low SNR of the convolutional code and the tolerance of burst errors of the RS code.




a) Parallel concatenation b) Serial concatenation

Figure 2.7: Parallel vs. serial concatenated codes.

In many cases, although it is not shown in Figure 2.7, the multiple decoders share information in order to obtain the best estimates of the transmitted data.

Interleavers can be used to help the code combat burst errors by spreading out bit information over a larger block of data. With no interleaving, convolutional codes are quite susceptible to burst errors since the $K$ output bits that contain information from a single data bit are adjacent in the encoder output stream. By interleaving the data stream before transmission and deinterleaving it after it is received, the burst errors appear to the decoder to be random. The most common type of interleaver is a *block interleaver*, which is a rectangular array where data is read into the columns and out of the rows (or vice versa). This type of interleaving is referred to as *channel interleaving*. The interleavers used in turbo coding are not channel interleavers. Referred to as *code interleavers*, they arrange the data block in a row and then rearrange the rows according to a pseudo-random pattern. This is illustrated in Figure 2.8. While this technique also has the effect of spreading out the data bits to protect against burst errors, the more

important effect is the reduction of the probability of having all of the encoders produce low Hamming weight codewords at the same time.

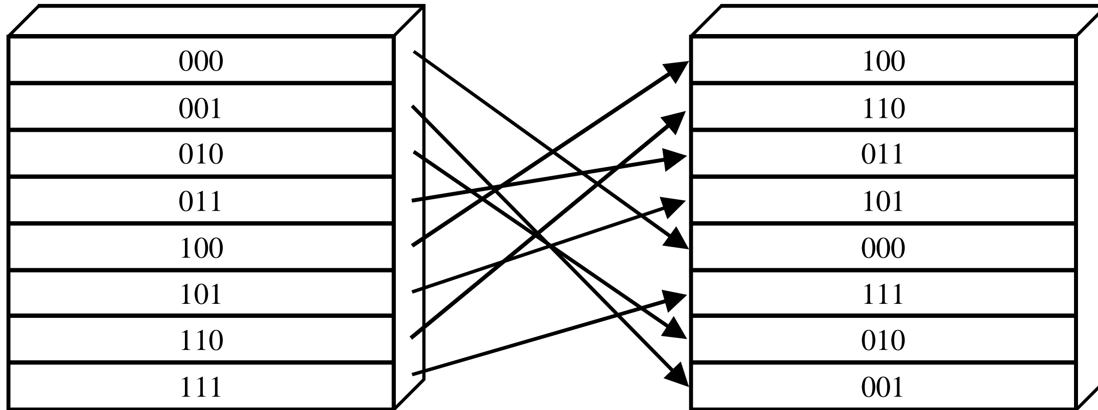



Figure 2.8: Row interleaving used in turbo codes.

The conventional turbo encoder uses two identical constituent RSC encoders (Figure 2.9). The rate of this code is 1/3 (2.9 a), unless puncturing is used, which brings the rate up to 1/2 (2.9 b). In this case, puncturing is performed by always sending the systematic information and alternating the encoder from which the parity bit comes. Thus, the sequence

$$(x_0, y_{10}, y_{20}, x_1, y_{11}, y_{21}, x_2, y_{12}, y_{22}, \ldots) \tag{2.10}$$

becomes

$$(x_0, y_{10}, x_1, y_{21}, x_2, y_{12}, \ldots). \tag{2.11}$$

## *2.3 Decoding Turbo Codes*

In a turbo encoder with $N_e$ constituent encoders, the encoder output contains a single systematic output and $N_e$ parity outputs from the RSC encoders (assuming no puncturing), $N_e - 1$ of which operate on an interleaved version of original data block. Thus, the output of the turbo encoder can be viewed as the output of $N_e$ independent RSC encoders, except the systematic information only need be transmitted for one of the encoders. The decoder can reconstruct the systematic bits for the other encoders because it knows the interleaving patterns that were used. Thus, the decoder can be decomposed into $N_e$ convolutional decoders with each one operating on the output of a single constituent encoder. In order to get the best possible estimate of the original message,

these separate decoders must be able to share the results of their calculations. To accomplish this, turbo decoders use *iterative feedback decoding*.

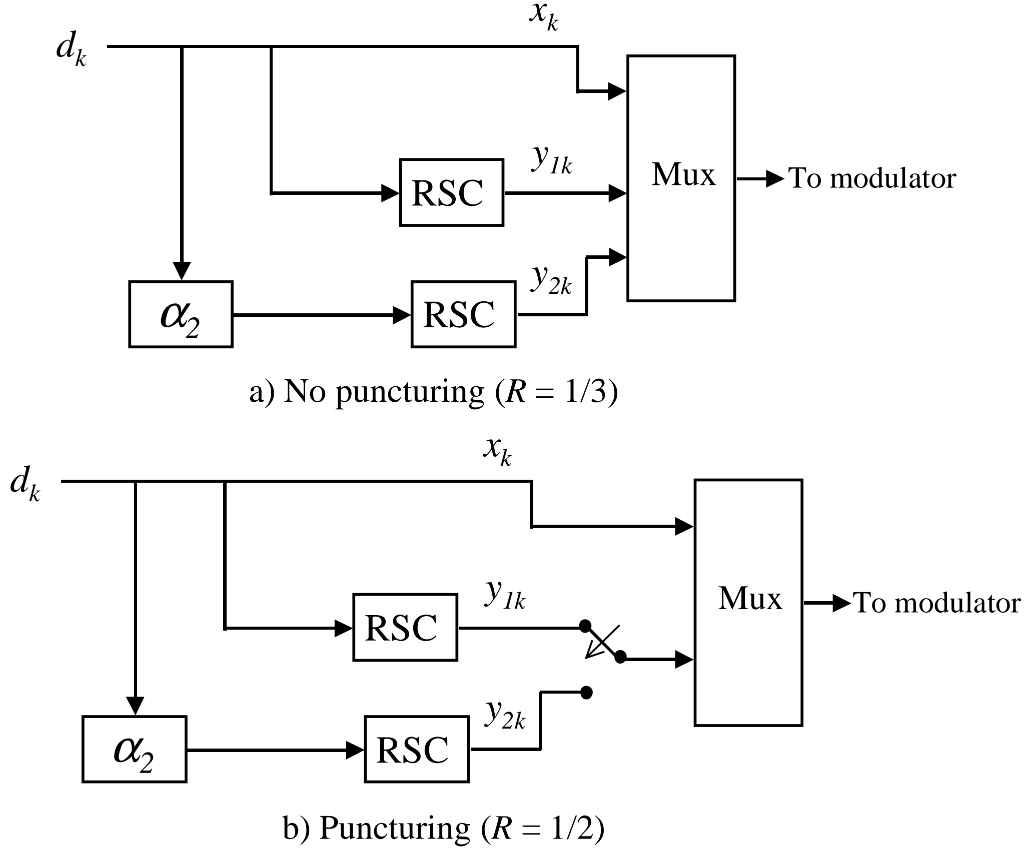



Figure 2.9: Conventional turbo encoder for code rates $R$=1/3 (a) and $R$=1/2 (b).

Figure 2.10 shows a schematic of a turbo decoder for the classical turbo code example, $N_e$=2. The first decoder uses the systematic information $x_k$, the output from the first constituent encoder $y_{1k}$, and a priori information from the second decoder, $L_{a2,k}$, to calculate soft estimates of the original data in the block known as a log-likelihood ratios (LLRs), $\Lambda_1$. The systematic and a priori information are subtracted from $\Lambda_1$ in order to prevent positive feedback. What is left over is the new information calculated by the first decoder, $L_{e1,k}$, known as the *extrinsic information*. This extrinsic information will be used as a priori information by the second decoder. The second decoder uses this a priori information along with the systematic information, and the output of the second constituent encoder, $y_{2k}$. However, $y_{2k}$ was calculated from an interleaved version of $x_k$,

so both the systematic information and extrinsic information from the first decoder must be interleaved (forming $\tilde{x}_k$ and $L_{a1,k}$, respectively) before being used in the second decoder. The second decoder produces the extrinsic information $L_{e2,k}$ that is deinterleaved and then fed back to the first encoder to be used as a priori information, $L_{a2,k}$.

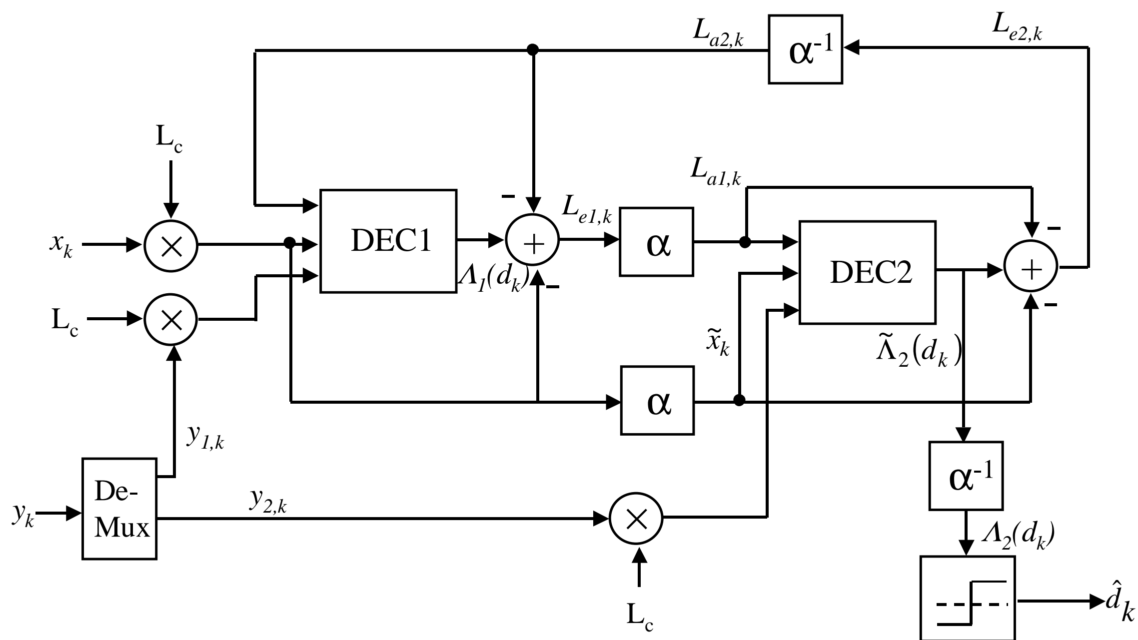



Figure 2.10: Turbo decoder schematic.

It should be noted that all of the values received from the channel are multiplied by the factor $L_c$, where

$$L_c = \frac{4aE_s}{N_0}. \tag{2.12}$$

This is an expression of the channel reliability, where $E_s$ is the signal energy, $N_0$ is the power spectral density of the noise, and $a$ is the fading amplitude of the channel. This reliability factor basically acts to amplify the values received when the channel is known to be reliable and decrease them when the channel is erratic.

After the first decoding cycle has completed, the decoder can be visualized as shown in Figure 2.11. In this form the decoder is not taking in new inputs. Instead, it is iterating toward a best estimate of the transmitted data using the received values that it now has stored in memory. After $I$ iterations, the feedback loop is broken and the LLRs

produced by the second decoder are deinterleaved one final time to put them in the same order as the original data block and a hard limiter makes the final bit decisions to produce the decoded block.

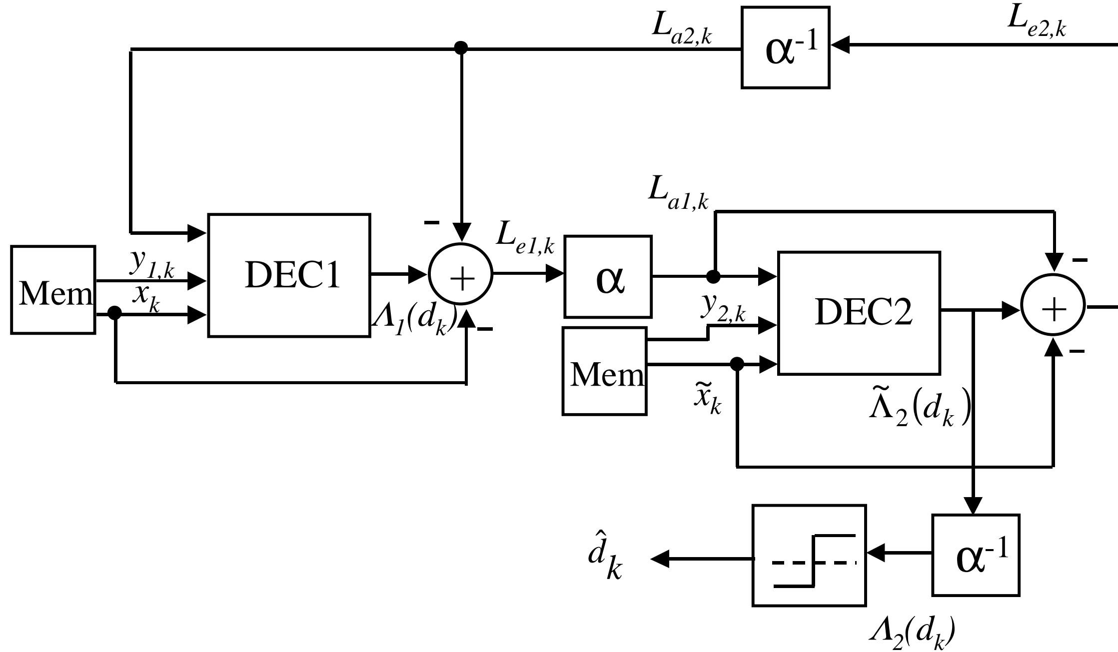



Figure 2.11: Turbo decoder in iterative feedback mode.

Up to this point, the actual operation of the convolutional decoders has been covered only in a "black box" sense; they must be able to produce LLRs from systematic and non-systematic information received from the channel and a priori information from another decoder. When this decoding structure was first proposed [15] a modified[6] version of the Bahl et al. algorithm (also known as the BCJR algorithm[7]) [18] was proposed for use in turbo decoders.

## *2.4 Decoding Algorithms for Turbo Codes*

The BCJR algorithm is optimal for producing the best estimate of the transmitted bits. It produces the maximum a posteriori (MAP) probabilities of the outputs of a convolutional encoder (or any Markov process for that matter) on a symbol-by-symbol basis. However,

[6] The BCJR algorithm had to be modified to support the recursive nature of RSC codes.
[7] BCJR are the initials of the authors of the original paper detailing this algorithm—Bahl, Cocke, Jelinek, and Raviv.

it has several attributes that make it very difficult to implement. Thus, sub-optimal approximations to the MAP algorithm, as it is often called, have been developed. These approximations attempt to ease the implementation burden of the algorithm while not losing too much of its performance. The original MAP algorithm and several approximations of it are described below.

### 2.4.1 MAP Algorithm

In 1974, Bahl et al. presented a method [18] of decoding convolutional codes that minimizes the symbol error probability. Based on methods used for the elimination of intersymbol interference [19], it works by calculating the a posteriori probabilities (APP) that the original information bit was a 0 or a 1. It was developed as an alternative to Viterbi decoding, which instead minimizes the codeword errors by optimizing its estimate of the entire transmitted sequence. Also, while the Viterbi algorithm produces hard decisions as its output, the Bahl algorithm produces soft decisions, making it ideal for use in turbo decoding.

In turbo coding literature, the Bahl algorithm is usually simply referred to as the MAP algorithm because it calculates the maximum APPs for the encoded data. For the remainder of the discussion, this convention will be followed.

Consider an RSC code of constraint length $K$ that encodes blocks of size $N$. The information bits that are encoded are $\mathbf{d} = \{d_0, d_1, \ldots, d_{N-1}\}$, and the decoder receives the noisy estimate of the encoded bits $\mathbf{y} = \{(y_0^{(s)}, y_0^{(p)}), (y_1^{(s)}, y_1^{(p)}), \ldots (y_{N-1}^{(s)}, y_{N-1}^{(p)})\}$ where $y_k^{(s)}$ and $y_k^{(p)}$ are the received estimates of the systematic and parity bits, respectively, at time $k$. The state of the encoder at time $k$ is given by $S_k$ where the state is restricted such that it begins and ends in the all-zeros state.

$$S_k = \mathbf{0}, \quad \text{for k} = 0, \text{N} \tag{2.13}$$

The MAP algorithm is designed to produce the APP for each information bit being a 1 or a 0 based on the data received from the channel. The APP can be expressed as

$$\Pr[d_k = i \mid \mathbf{y}] = \sum_{S_k} \lambda_k^i (S_k) \tag{2.14}$$

where $\lambda_k^i(S_k)$ is the joint probability

$$\lambda_k^i(S_k) = \Pr[d_k = i, S_k \mid \mathbf{y}] = \frac{\Pr[d_k = i, S_k, \mathbf{y}]}{\Pr[\mathbf{y}]}. \tag{2.15}$$

Once the APPs have been obtained, the LLR, $\Lambda(d_k)$ can be calculated from

$$\Lambda(d_k) = \log \frac{\Pr[d_k = 1 \mid \mathbf{y}]}{\Pr[d_k = 0 \mid \mathbf{y}]}. \tag{2.16}$$

This allows for hard decisions to be made by simply testing the sign of $\Lambda(d_k)$. If $\Lambda(d_k)>0$, then $d_k = 1$; otherwise, $d_k = 0$. Because the denominator of the expression in (2.15) is a constant, the numerator of the expression, $\sigma_k(S_k, S_{k-1}) = \Pr[d_k = i, S_k, \mathbf{y}]$, is often used instead of $\lambda_k^i(S_k)$. By using Bayes' rule and exploiting some properties of the encoding process, the probability $\sigma_k(S_k, S_{k-1})$ can be rewritten in the form

$$\sigma_k(S_k, S_{k-1}) = \alpha_{k-1}(S_{k-1})\gamma_i\left(\left(y_k^{(s)}, y_k^{(p)}\right), S_{k-1,} S_k\right)\beta_k(S_k), \tag{2.17}$$

where

$$\alpha_k(S_k) = \frac{\sum_{S_{k-1}} \sum_{i=0}^{1} \gamma_i\left(\left(y_k^{(s)}, y_k^{(p)}\right), S_{k-1,} S_k\right)\alpha_{k-1}(S_{k-1})}{\sum_{S_k} \sum_{S_{k-1}} \sum_{i=0}^{1} \gamma_i\left(\left(y_k^{(s)}, y_k^{(p)}\right), S_{k-1,} S_k\right)\alpha_{k-1}(S_{k-1})}, \tag{2.18}$$

and

$$\beta_k(S_k) = \frac{\sum_{S_{k+1}} \sum_{i=0}^{1} \gamma_i\left(\left(y_{k+1}^{(s)}, y_{k+1}^{(p)}\right), S_{k,} S_{k+1}\right)\beta_{k+1}(S_{k+1})}{\sum_{S_k} \sum_{S_{k+1}} \sum_{i=0}^{1} \gamma_i\left(\left(y_{k+1}^{(s)}, y_{k+1}^{(p)}\right), S_{k,} S_{k+1}\right)\alpha_k(S_k)}, \tag{2.19}$$

and $\gamma_i\left(\left(y_k^{(s)}, y_k^{(p)}\right), S_{k-1,} S_k\right)$ is the set of branch transition probabilities, which are determined by properties of the channel and the encoder. It is expressed as

$$\begin{aligned} \gamma_i\left(\left(y_k^{(s)}, y_k^{(p)}\right), S_{k-1,} S_k\right) = \Pr\left[y_k^{(s)} \mid d_k = i\right] \cdot \Pr\left[y_k^{(p)} \mid d_k = i, S_k, S_{k-1}\right] \cdot \\ \Pr[d_k = i \mid S_k, S_{k-1}] \cdot \Pr[S_k \mid S_{k-1}] \end{aligned}. \tag{2.20}$$

Thus, for each set of $2^L$ possible states, $S_k$, there will be a set of $2 \times 2^L = 2^{L+1}$ branch transition probabilities, with a single $\gamma$ indicating the probability of the encoder going

from a specific state $S_{k-1}$ to a specific state $S_k$ when a particular bit $d_k$ is encoded. It is by the calculation of the branch transition probabilities, $\gamma$, and the recursive solving for the forward and reverse branch metrics, $\alpha$ and $\beta$, that the decoder calculates the APP for each bit, which in turn is used to calculate the LLR. The decoding procedure can be broken down into several steps:

a) Initialize $\alpha$ according to

$$\alpha(S_0) = \begin{cases} 1 \text{ for } S_0 = 0 \\ 0 \text{ for } S_0 \neq 0 \end{cases} \text{ and } \alpha(S_k) = 0 \; \forall \; k \neq 0 \tag{2.21}$$

b) Initialize index variable $k = 1$

c) Calculate $\gamma_i$ and $\alpha_k$ for all states $S_k$ according to (2.18)

d) Increment $k$

e) Repeat steps c) and d) until $k = N$

f) Initialize $\beta$ according to

$$\beta(S_N) = \begin{cases} 1 \text{ for } S_N = 0 \\ 0 \text{ for } S_N \neq 0 \end{cases} \text{ and } \beta(S_k) = 0 \; \forall \; k \neq N \tag{2.22}$$

g) Initialize index variable $k = N - 1$

h) Calculate $\gamma_i$ and $\beta_k$ for all states $S_k$ according to (2.19)

i) Decrement k

j) Repeat steps h) and i) until k = 0

k) Calculate the LLR from

$$\Lambda(d_k) = \log \frac{\sum_{S_k} \sum_{S_{k-1}} \gamma_1\left(\left(y_k^{(s)}, y_k^{(p)}\right), S_{k-1,} S_k\right) \cdot \alpha_{k-1}(S_{k-1}) \cdot \beta_k(S_k)}{\sum_{S_k} \sum_{S_{k-1}} \gamma_0\left(\left(y_k^{(s)}, y_k^{(p)}\right), S_{k-1,} S_k\right) \cdot \alpha_{k-1}(S_{k-1}) \cdot \beta_k(S_k)} \tag{2.23}$$

### 2.4.2 Max-Log-MAP and Log-MAP Algorithms

Although the MAP algorithm provides the best estimates of the APP of the information bits, it is very difficult to implement in practice. The large number of multiplications are computationally expensive. One solution to this problem is to perform the entire decoding operation in the logarithmic domain. This especially makes sense because the LLR computation puts the results in the log domain anyway.

The main appeal of the log domain from a computational sense is that multiplication operations are transformed into addition operations.

$$\log(a \cdot b) = \log a + \log b \tag{2.24}$$

The drawback of the log domain is that it makes what was originally an addition operation more complex, as can be seen for the Jacobian logarithm

$$\log\left(e^{x_1} + e^{x_2}\right) = \max\left(x_1, x_2\right) + \log\left(1 + e^{-|x_2 - x_1|}\right). \tag{2.25}$$

However, for cases where $x_1$ and $x_2$ are not close in value, the second term in (2.25) is near zero, so a good approximation for this logarithm is

$$\log\left(e^{x_1} + e^{x_2}\right) \approx \max\left(x_1, x_2\right) \tag{2.26}$$

This approximation is the basis for the Max-Log-MAP algorithm, one of two MAP algorithms that operate solely in the log domain [20]. Based on this approximation, new state and branch metrics can be defined in the log domain. The new forward and reverse state metrics are

$$\begin{aligned}\overline{\alpha}_k(S_k) = \log(\alpha_k(S_k)) \approx \max_{(S_k, i)}\left[\overline{\gamma}_i\left(\left(y_k^{(s)}, y_k^{(p)}\right), S_{k-1}, S_k\right) + \overline{\alpha}_{k-1}(S_{k-1})\right] - \\ \max_{(S_k, S_{k-1}, i)}\left[\overline{\gamma}_i\left(\left(y_k^{(s)}, y_k^{(p)}\right), S_{k-1}, S_k\right) + \overline{\alpha}_{k-1}(S_{k-1})\right]\end{aligned} \tag{2.27}$$

and

$$\begin{aligned}\overline{\beta}_k(S_k) = \log(\beta_k(S_k)) \approx \max_{(S_k, i)}\left[\overline{\gamma}_i\left(\left(y_{k+1}^{(s)}, y_{k+1}^{(p)}\right), S_k, S_{k+1}\right) + \overline{\beta}_{k+1}(S_{k+1})\right] - \\ \max_{(S_k, S_{k+1}, i)}\left[\overline{\gamma}_i\left(\left(y_{k+1}^{(s)}, y_{k+1}^{(p)}\right), S_k, S_{k+1}\right) + \overline{\alpha}_k(S_k)\right].\end{aligned} \tag{2.28}$$

The new branch transition probabilities are expressed

$$\begin{aligned}&\overline{\gamma}_i\left(\left(y_k^{(s)}, y_k^{(p)}\right), S_{k-1}, S_k\right) = \\ &\log \gamma_i\left(\left(y_k^{(s)}, y_k^{(p)}\right), S_{k-1}, S_k\right) = \\ &\log \Pr\left[y_k^{(s)} \mid d_k = i\right] + \log \Pr\left[y_k^{(p)} \mid d_k = i, S_k, S_{k-1}\right] + \log \Pr\left[S_k \mid S_{k-1}\right].\end{aligned} \tag{2.29}$$

Once these metrics have been calculated, the LLR can be calculated from

$$\begin{aligned}\Lambda(d_k) \approx \max_{(S_k, S_{k-1})}\left[\overline{\gamma}_1\left(\left(y_k^{(s)}, y_k^{(p)}\right), S_{k-1}, S_k\right) + \overline{\alpha}_{k-1}(S_{k-1}) + \overline{\beta}_k(S_k)\right] - \\ \max_{(S_k, S_{k-1})}\left[\overline{\gamma}_0\left(\left(y_k^{(s)}, y_k^{(p)}\right), S_{k-1}, S_k\right) + \overline{\alpha}_{k-1}(S_{k-1}) + \overline{\beta}_k(S_k)\right].\end{aligned} \tag{2.30}$$

Although the approximation in (2.26) greatly reduces the complexity of the decoder, it also provides soft outputs that are inferior to those produced by the MAP algorithm because the second term in (2.25) is ignored. It is possible to restate (2.25) in the form

$$\log\left(e^{x_1} + e^{x_2}\right) = \max\left(x_1, x_2\right) + f_c\left(\left|x_2 - x_1\right|\right), \tag{2.31}$$

where $f_c\left(\left|x_2 - x_1\right|\right)$ can be viewed as a correction function that quantifies the error of the approximation stated in (2.26). Thus, if this correction function were factored into all of the computations of $\log\left(e^{x_1} + e^{x_2}\right)$, then the accuracy of the original MAP algorithm would be preserved, with all of the decoding occurring in the log domain. This algorithm has been dubbed the Log-MAP algorithm. Because the correction function only operates in one dimension, it can be implemented by a lookup table that is indexed by $\left|x_2 - x_1\right|$. In [20], it was shown that excellent results could be obtained by using a table with as few as 8 entries. Since this lookup table is small, the complexity of Log-MAP is smaller than that of MAP while providing essentially the same performance.

### 2.4.3 Soft Output Viterbi Algorithm

The soft output Viterbi algorithm (SOVA) is another logarithmic domain procedure for performing soft output decoding of convolutional codes. Proposed in 1989 [21], it provides an alternative to the Max-Log-MAP approximation. While SOVA and Max-Log-MAP provide identical performance for hard decisions, Max-Log-MAP performs somewhat better than SOVA for soft decisions. This is offset by the fact that the implementation complexity of SOVA is less than that of Max-Log-MAP.

## *2.5 Designing Turbo Codes*

There are many variables in the design and implementation of a turbo coding system that can greatly affect the overall performance and usability of the system. Generally, these factors provide tradeoffs between coding gain, code rate, implementation complexity, decoding throughput, and decoding latency. While many of these factors will be covered in greater detail in Chapter 4, a brief discussion of them is given here to supplement the

theoretical discussion. Some of the factors affecting the design of a turbo coding system are the following:

*Block Length (or Interleaver Size):* This is one of the most important tradeoffs. Longer blocks provide the best coding gain (in fact, the original code that came within 0.7 dB of the Shannon limit used blocks of length 65,536), but decoding complexity and latency scale up linearly with block size.

*Number of RSC Encoders:* While more encoders will improve the coding gain of the turbo code, the decoding complexity and latency increase immensely with additional encoders as each new encoder requires a corresponding convolutional decoder and interleaver/deinterleaver pair in the turbo decoder. Also, the code rate declines as the number of constituent encoders increases unless heavy puncturing is used. Because of this, turbo coding systems are generally implemented with two constituent encoders.

*RSC Encoder Constraint Length:* Another way to improve the coding gain, the decoding complexity is exponentially related to constraint length, so small constraint length encoders ($K$=3,4,5) are typically used. Generally, increasing the constraint length is only beneficial if the block length is also increased.

*Puncturing:* Puncturing allows the user to trade coding gain for increases in the code rate. Decoding complexity and speed are not usually significantly affected by this.

*Number of Decoding Iterations:* The more iterations that the decoders can use to share information, the better the bit estimates will be up to a saturation point. However, this increase in reliability follows a law of diminishing returns, and each iteration incurs additional latency.

*Windowing Techniques:* Proposed in [22], this method reduces the memory requirements of the MAP decoder at the cost of increased latency.

## *2.6 Summary*

This chapter introduced the relatively new class of block codes called turbo codes that provide extraordinary performance that approaches the Shannon limit. Coding theory fundamentals, including block and convolutional codes and general terminology, were introduced to provide a foundation for the discussion of turbo codes. The parallel concatenated structure of turbo encoders and the iterative feedback structure of turbo decoders were also presented. This was followed by a discussion of different methods of performing soft output decoding of convolutional codes, an essential element of the turbo decoder. Finally, several of the parameters that must be considered when designing a turbo coding system were discussed.

# Chapter 3. Configurable Computing

While Chapter 2 described the theory behind the turbo decoding algorithm being implemented, that knowledge alone does not allow one to proceed much further than the realm of simulation. In order to implement the algorithm, one must also understand the details of the platform on which it will be constructed and the design procedure used to realize the theoretical design in hardware. This chapter gives an overview of the concept of configurable computing followed by a description of hardware used in configurable computing applications with a focus on the actual hardware used for this implementation. The general design process for applications on this platform is discussed, followed by a description of applications that are typically well-suited to these architectures.

Configurable computing uses special hardware that has the ability to change its functionality very rapidly by downloading a new configuration. This is a compromise between the two extremes of using general-purpose microprocessors and custom-built application-specific integrated circuits (ASICs) to perform a computational task. General purpose processors place a premium on flexibility. They can perform a wide variety of tasks by simply varying the instructions that they have stored in memory. In contrast, custom ASICs are designed to perform a particular task and nothing else. Although they are highly inflexible, they offer optimum performance in terms of speed, power consumption, and chip area. Configurable computers attempt to offer the performance benefits of ASICs while still maintaining the flexibility of a general-purpose processor by allowing the hardware to vary to suit a given task. This kind of hardware flexibility requires a special kind of device on which any digital logic can be implemented and then changed completely with the simplicity of a memory read. The main hardware devices that are used to power configurable computing are *field-programmable gate arrays* (FPGAs).

## 3.1 Field Programmable Gate Arrays (FPGAs)

An FPGA consists of a grid-like array of logic cells that are connected by a programmable metal interconnect to each other and to an array of I/O cells that reside on

the edges of the chip. The cells and programmable interconnect switches are built on top of a structure resembling a static RAM (SRAM), so that the current configuration of the hardware on the chip is defined by the values stored in these RAM elements. In order to reconfigure the hardware, the memory elements must simply be loaded with new values, meaning that the hardware can be changed entirely in a matter of milliseconds.

After being introduced in 1985 by Xilinx, FPGAs were mainly used to implement "glue logic" for board-level designs. As the densities of the devices increased, they became popular for prototyping ASIC designs. Because FPGAs can be reprogrammed frequently they offer a fast, inexpensive way to try out different design alternatives and perform functional verification before a design is committed to an ASIC implementation. As the technology progressed further, researchers began using systems based on multiple FPGAs to do high-performance computations for various applications. More recently, the introduction of FPGAs that can achieve much faster configuration times by means of partial reconfiguration or a special context switching approach has led to research in *run-time reconfigurable* systems [23,24].

In order to understand how FPGAs are programmed, a more detailed discussion of their inner structure is in order. Although there are several vendors that manufacture competing products, Xilinx FPGAs were used for this design, so much of the discussion centers on the details of Xilinx parts.

### 3.1.1 Xilinx 4000 series FPGAs

The device used for the turbo decoder implementation is a member of Xilinx's 4000 series of FPGAs. The atomic element for logic implementation in these FPGAs is the configurable logic block (CLB). A diagram of a 4000 series CLB is shown in Figure 3.1. The CLB consists of two four-input lookup tables (LUTs), a three-input LUT, two flip-flops (FFs), and several multiplexers that are used for internal routing of signals in the CLB. The CLB is reconfigured by changing the data in the LUTs and the select bits for the multiplexers.

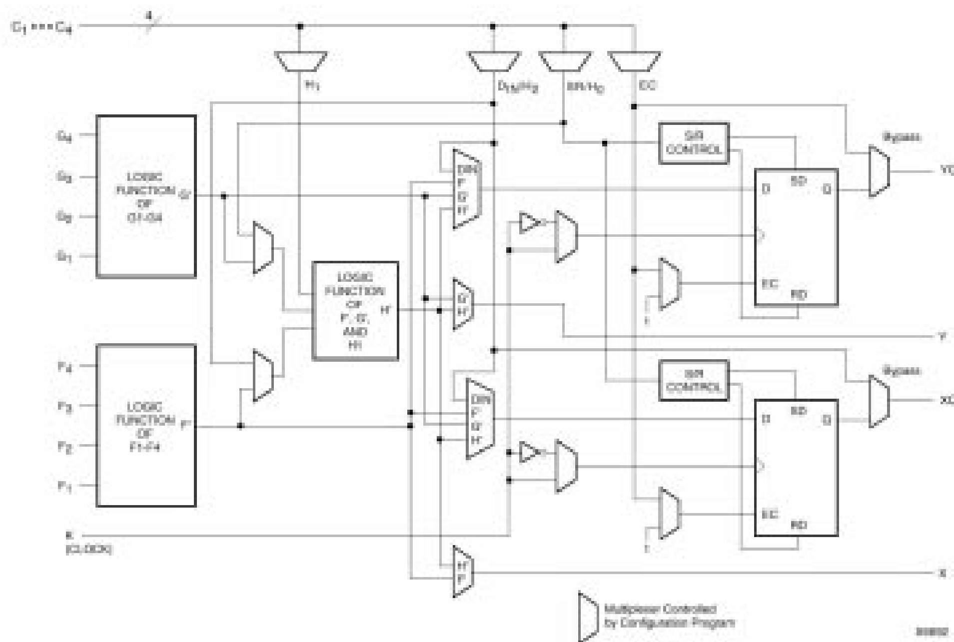



Figure 3.1: Diagram of a Xilinx 4000 series configurable logic block (CLB) [25].

Because of the structure of the CLB, with the outputs of the two four-input LUTs (F and G) being used as possible inputs to the three-input LUT (H), a single CLB can be used to implement two functions of four unrelated variables and a third function of three unrelated variables, a single function of five variables, any function of four variables plus some functions of six variables, or some functions of nine variables [25].

Another useful feature of the CLB is that it can be configured as an on-chip RAM. This RAM is significantly faster than any off-chip memory. A single CLB may be configured to be a single-ported $16 \times 1$, $16 \times 2$, or $32 \times 1$ RAM with either edge-triggered or level-sensitive timing, or a dual-ported $16 \times 1$ RAM with edge-triggered timing. The dual-ported RAM allows simultaneous reads and writes to be performed. Of course, larger RAM structures can be constructed by combining several of these basic RAMs with some decoding logic.

The specific device used for the turbo decoder implementation is the Xilinx 4062XL-3 FPGA. This device contains the equivalent of approximately 62,000 logic gates implemented in 2,304 CLBs. While this is near the highest capacity of the 4000XL family as shown in Table 3.1, recent advancements in process technology have provided 4000 series devices with gate counts as high as 250,000 [26].

| Device | Max. Logic Gates | Max. RAM Bits | Total CLBs | Max. User I/O |
|---|---|---|---|---|
| 4005XL | 5,000 | 6,272 | 196 | 112 |
| 4010XL | 10,000 | 12,800 | 400 | 160 |
| 4013XL | 13,000 | 18,432 | 576 | 192 |
| 4028XL | 28,000 | 32,768 | 1,024 | 256 |
| 4036XL | 36,000 | 41,472 | 1,296 | 288 |
| 4052XL | 52,000 | 61,952 | 1,936 | 352 |
| 4062XL | 62,000 | 73,728 | 2,304 | 384 |
| 4085XL | 85,000 | 100,352 | 3,136 | 448 |

Table 3.1: Statistics for some members of the 4000XL series of FPGAs [25].

The Max. Logic Gates and Max. RAM Bits numbers from Table 3.1 reflect the cases where every CLB is used for the specified purpose. A more realistic design would likely contain CLBs used for logic and RAM, so neither maximum would be achieved. It is also interesting to note that all of the CLB counts are perfect squares. This is due to the fact that the CLB arrays are always square. For instance, the 2,304 CLBs on the 4062XL are laid out in a $48 \times 48$ array.

### 3.1.2 Other FPGAs

While the 4000 series devices are capable for many computational tasks, there have been many innovations in FPGA technology since their introduction. The Xilinx 6200 series of FPGAs were the first to support partial reconfiguration. While the 4000 series devices must be completely reconfigured no matter what changes are made, the 6200 supported reconfiguration of only the columns of CLBs that needed to be changed. With careful application design, the reconfiguration time could be reduced immensely, allowing the first experimentation with run-time reconfiguration.

Lockheed Sanders has made inroads toward truly run-time reconfigurable computing with the introduction of a context-switching FPGA [27]. This device uses logic cells that can store up to four configurations that can be switched between one another in a single clock cycle.

The COLT processor [23] is an experimental device that is reconfigured by a self-directing data stream. The data stream contains programming headers that configure the necessary resources just before the data is processed in a technique called *wormhole run-time reconfiguration*.

Xilinx reached a milestone in FPGA technology with the release of the Virtex series of devices which, in addition to supporting partial reconfiguration and dedicated on-chip memory, included the first FPGA to hold the equivalent of one million logic gates [26]. While this brief summary only begins to cover the different advances in FPGA designs, it gives an indication of the various approaches that have been taken to improve the performance of the technology.

## 3.2 WILDFORCE Platform

Although the main hardware for the design resides in the FPGA, it is useless without a structure that provides access to external memory, a way to program the chip, and user control of the chip's operation. These capabilities and more are provided by the WILDFORCE board from Annapolis Micro Systems. Pictured in Figure 3.2, WILDFORCE is a configurable computing machine (CCM) that uses five Xilinx FPGAs as processing elements (PEs). One of the FPGAs is designated as the control PE (CPE0), and the other four are referred to as PEs 1 – 4. The latter are connected in a systolic array

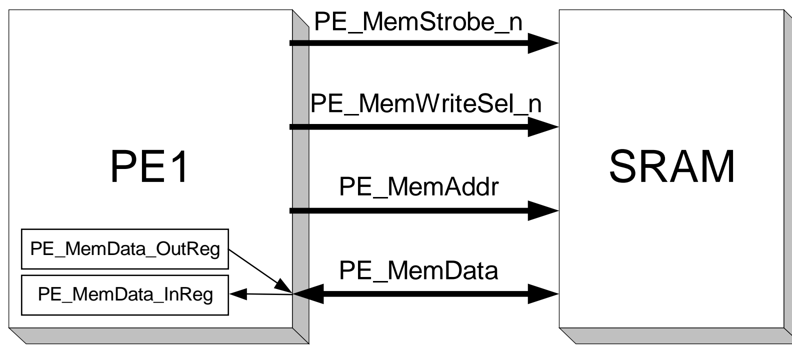
Figure 3.2: Picture of the WILDFORCE CCM [28].

by a 36-bit bus that is used for inter-PE communication. CPE0 has connections of varying bit widths to each of the other PEs on the board. Additionally, another channel of inter-PE communications is provided by a configurable crossbar switch that can be programmed in nibbles of four bits.

Each PE is also connected to a daughter card connection that can be used to house an external SRAM module. The memory modules are addressable at the word (32 bit) level, and available in capacities up to one megaword (4 MBytes). The WILDFORCE used for this implementation uses five 4062XL FPGAs, each with a one megaword SRAM.

The memory interface, and a timing diagram for a consecutive read and write are shown in Figure 3.3. The "_n" notation used for the memory strobe and write select signals denotes that the signals are active low. In other words, when PE_MemStrobe_n = '0' then the memory is listening to the signals on the buses, and when PE_MemWriteSel_n = '0' the memory is in write mode (assuming that the strobe is also active). In the illustrated case, a read of address A1 is performed, followed by a write of data D2 to address A2. On the WILDFORCE, data from a memory read is not available until two cycles after the request is made, which is why the data, D1, that is stored in A1 is not received by the PE until after the write has already been performed. This detail can often make designing efficient state machines difficult for applications that require that the address or data used in a memory operation be dependent on the data from a previous memory read.

Because the WILDFORCE is a PCI card, it must be installed in a PC to be used. User-developed software running on the host PC's processor actually controls the WILDFORCE through the PCI bus by using an application programming interface (API) that comes with the board. By using the API, the user can program the PEs, control the board clock and reset signals, and communicate with the board by a number of means. Data can be exchanged between the host and CCM through the FIFOs that CPE0, PE1, and PE4 are attached to. Additionally, the processor can use DMA transfers to write to and read from the PE memories on the board. Finally, the WILDFORCE can signal the host processor of an event by using its interrupt line.



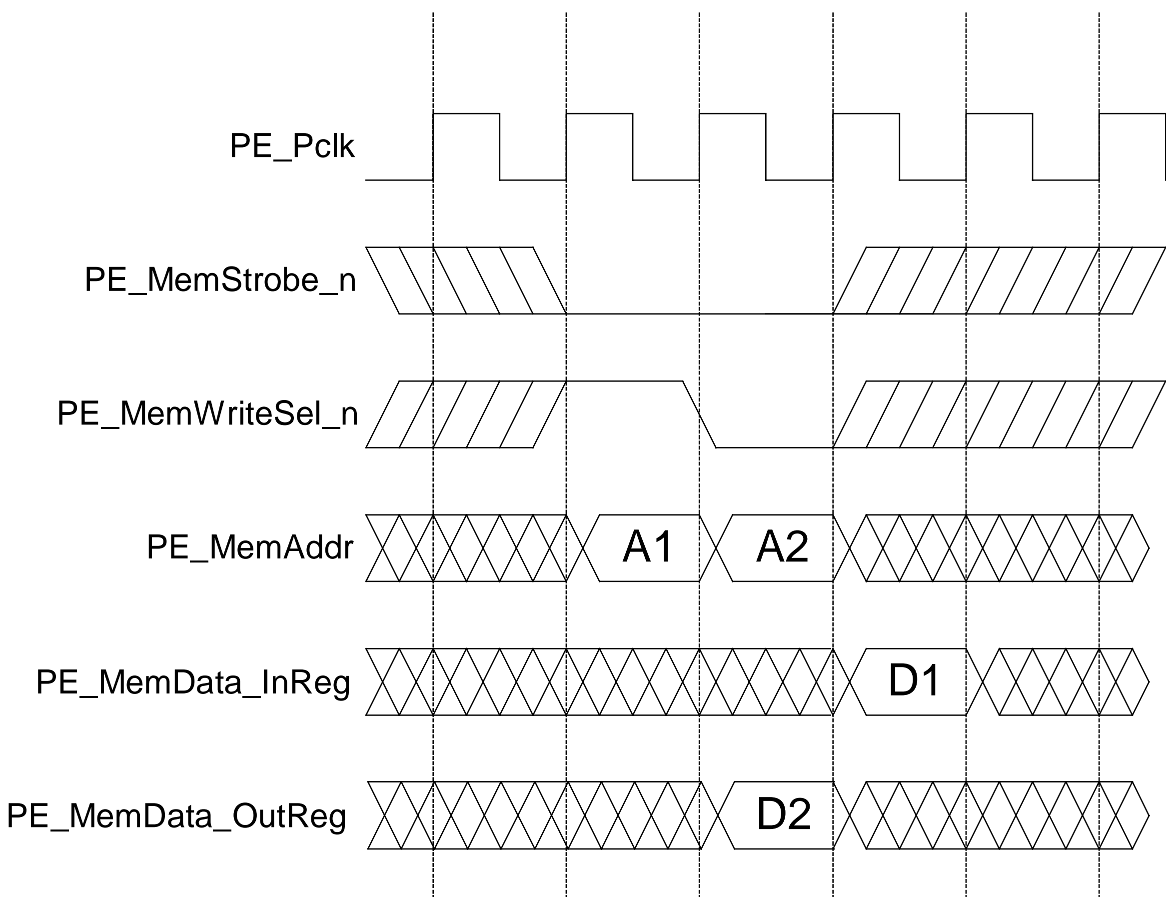




Figure 3.3: Memory interface and timing for a write directly after a read on the WILDFORCE.

## 3.3 Application Design for the WILDFORCE

Designing applications for CCMs can be a complicated process because the designer must create an FPGA-based implementation of the hardware as well as software to control it. Getting the hardware and software to interoperate properly is often difficult because neither of them can be truly fully tested until they run with the other.

The traditional approach to application design for the WILDFORCE, which was the approach used for the turbo decoder implementation, is shown in Figure 3.4. Because most of the later stages in the process require little creative input from the designer, any errors or undesirable performance require the designer to make the modifications in the partitioning and description phases and redo the later phases of the process. A more detailed description of the stages follows Figure 3.4.

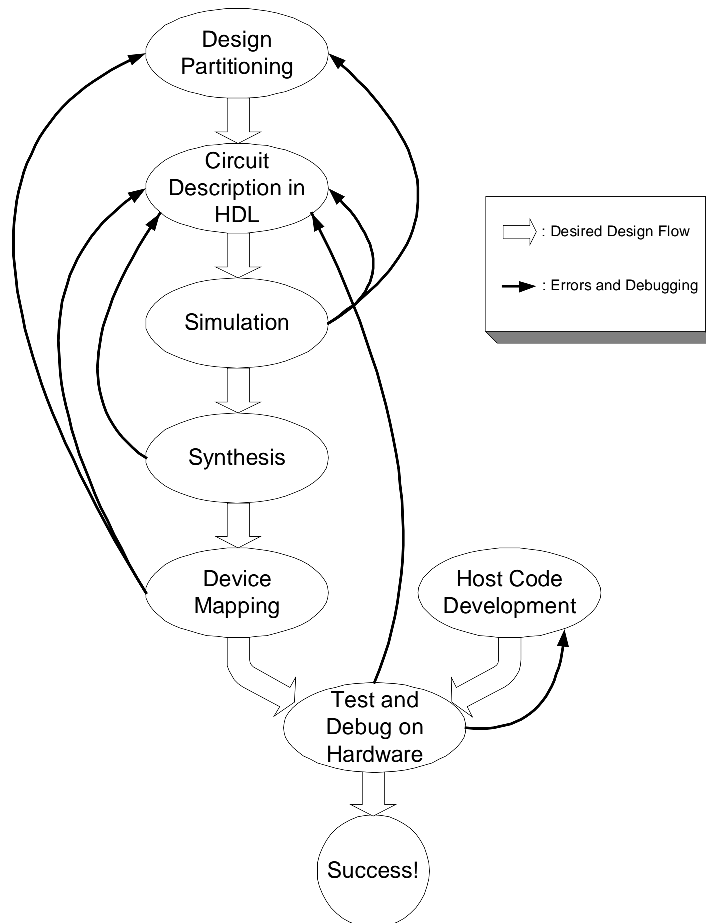



Figure 3.4: Traditional design process for CCMs.

*Design Partitioning:* The first stage of the process requires the designer to synthesize knowledge of the application requirements, platform architecture, and host API into a feasible high-level system design. If multiple PEs will be used, the application must be divided in a way that provides the best performance within the constraints of the architecture. Even within a PE, the design can be partitioned into modules to ease design and testability. Also, the designer must plan anticipated interaction with the host, so that a data flow for the system can be established, giving an indication of the communications responsibilities of each of the PEs used in the design.

*Circuit Description:* In this stage, the entire functionality of the application is described in a hardware description language (HDL). The two most prevalent HDLs are Verilog and VHDL. For WILDFORCE designs, VHDL is preferred because all of the simulation modules and chip interfaces are written in VHDL. The logic core of each PE is coded according to the design from the previous stage and its signals are connected to a core interface that allows the PE to be modeled for board-level simulation and chip-level synthesis.

There are generally two schools of thought for circuit description, behavioral and structural description. Behavioral description simply expresses the operation of the circuit in terms of the outputs that are expected from a set of inputs, while structural description expresses the circuit as an instantiation and interconnection of simpler components. Consider the example of two numbers (A, B) being added to produce a sum (C). A behavioral description of this operation would simply be A + B = C, whereas a structural description would describe A and B as inputs to an adder with C as the output, and the adder itself would be defined by a similar interconnection of simpler logic modules. Behavioral circuit descriptions are generally easier to code than their structural counterparts, but the designer does not have as much control over the implementation of the circuit with behavioral description.

*Simulation:* Once the circuit descriptions have been coded, a simulator is used to replicate the operation of the entire board using user-designed PE cores and models provided with the WILDFORCE for the PE interfaces, board, memories, and other system components.

The purpose of the simulation is to provide functional verification of the design. Memory and FIFO contents can be specified by the user to provide test vectors for the simulation. For this design, Workview Office from Viewlogic was used as the simulator [29].

*Synthesis:* Each PE description, consisting of the logic core and the interface logic, is synthesized to produce a netlist of smaller modules that are specific to the FPGA being targeted. FPGA Express from Synopsys was the synthesis tool used for this design [30].

*Device Mapping:* The netlist produced by the Synthesis process specifies logic elements and their interconnections, but does not specify their exact placement on the FPGA. The M1 tools from Xilinx are used to place the elements specified in the netlist and allocate the routing resources needed for the connections, while trying to conform to any timing or placement constraints specified by the designer. If this can be done successfully, the output of this stage is a binary file that specifies the configuration of the FPGA to which the design is mapped. The M1 software also produces an estimate of the clock speed at which the design can be reliably run.

*Host Code Development:* The code running on the host processor must program the FPGAs on the board using the configurations produced in the mapping stage, read and write from the memories on the board, handle interrupts from the board, control the board clock, and perform any necessary pre- or post-processing of the data on which the board operates.

*Hardware Test/Debug:* Although some parts of the host code (such as the data processing) can be tested ahead of time, a large portion of the code is only testable when it is run with actual hardware configurations to interact with the API calls. Similarly, the hardware design cannot be tested for timing correctness until it is implemented on the board. Thus, incorrect results often produce a debugging quandary because the error could exist in the software or the hardware.

Although the traditional method is still the prevalent way that CCMs are programmed, there are research efforts that try to improve the process by making it simpler or more portable. JHDL is a set of FPGA CAD tools implemented in Java that allows the user to design the structure and layout of a circuit by creating circuit modules with standard object-oriented techniques [31]. This allows the user to perform simulation and execution in a unified environment, which makes debugging the final hardware implementation much easier. Janus is a set of architecturally independent tools that provide the ability to port applications between different architectures and allow application-based intellectual property to be easily migrated between platforms [32]. JBits is an API to the Xilinx configuration bitstream [33]. This allows dynamic modification of the bitstream configurations, allowing compilation of new configurations to be done very quickly.

## *3.4 Applications of CCMs*

Although they can theoretically support any application that a general-purpose processor with equal memory resources can support, the special features of CCMs make them especially well-suited for use with several classes of applications. Because FPGAs can be configured into highly parallel, pipelined structures, the greatest speedups over general-purpose processors are often seen in dataflow applications with a large degree of parallelism and a small number of data dependencies, a description which describes many image and signal processing operations. Many of the early CCM applications were centered around image processing [34]. Number factoring for cryptography [35] and template matching for automatic target recognition [36] are other algorithms where the customizable hardware of the FPGAs allows for great speedups over conventional processing. Many of these systems are limited not by chip area, but by memory bandwidth.

As RTR becomes more prevalent, the advantages of fast reconfiguration also make CCMs optimal for some applications. Great cost savings can be achieved over custom ASIC systems by using a single FPGA with RTR to perform all of the computations. This approach works best for applications that can be broken into distinct

stages where the memory (on- or off-chip) can be used to buffer the partially processed data between configurations, as was done for image interpolation in [24].

CCMs are also very useful for the development of systems where high performance is required and the requirements for the application are expected to change periodically because of changing standards and protocols or the desire to support several protocols with the same piece of hardware. The reconfigurability of the system allows for great cost-savings over the frequent replacement of ASICs with performance that is superior to a general-purpose system. This model applies to the ever-changing world of the Internet. A prototype for a run-time reconfigurable router that could be programmed by inserting a programming header into the packet stream is presented in [37]. This model also applies to the software radio system for which the turbo decoder is targeted.

## 3.5 Summary

This chapter presented the basic concepts of configurable computing and some specifics about the platform that was used for the turbo decoder implementation. FPGAs, the configurable elements that make up CCMs, were defined, followed by a detailed description of the Xilinx XC4000 series FPGAs that were used in this design. The WILDFORCE CCM was also described, with emphasis given to details that will be most relevant to the design, namely the memory interface and host interaction. The application design procedure for CCMs in general, and the WILDFORCE in specific, was explained. Finally, a discussion of the applications that are well-suited to configurable computing was presented.

# Chapter 4. Implementation of a Turbo Decoder in Configurable Hardware

With the background established for both the algorithm being implemented and the platform for which it has been targeted, it is time to look more closely at the details of the implementation that was performed. This chapter presents an overview of the goals and strategy of the implementation followed by descriptions of the major hardware modules developed for the design.

## *4.1 Implementation Goals*

Because the turbo decoder is designed to work as part of a larger communications system, there are certain implementation requirements that must be met so that the decoder will operate well with the other modules in the larger system. The first of these goals is decoding rate. Other modules in the system are planned for operation at up to 32 kbps information rate, so the decoder should be able to support this speed so that it is not a bottleneck in system performance. This means that for a rate $R = 1/3$ code the decoder should receive bits at an average rate of 96 kbps and produce decoded information bits at 32 kbps.

While doing this, the decoder is limited by the constraints of the WILDFORCE board – namely FPGA area, interconnection bandwidth, memory bandwidth, and clock rate. While 5 FPGAs provide a considerable amount of chip area, the realities of limited chip interconnect, low memory bandwidth, and non-shared memories often limit the efficient use of all of this hardware, especially in memory-intensive applications that do not have prodigious amounts of explicit parallelism, like turbo decoding. It must also be noted that the WILDFORCE platform is not necessarily the final target for this application, which makes conservation of chip area more important to allow for integration with other parts of the communications system on future platforms that may have fewer FPGAs.

## *4.2 System Design Parameters*

At the end of Chapter 2, several tradeoffs that are of paramount importance to turbo coding systems are described. While an ideal system would be extremely flexible along all of these degrees of freedom, this is not always a realistic or practical expectation. More often than not, the limitations of the hardware on which the algorithm is being implemented affect the parameters that are used in the design. Simulation studies can be used to evaluate the performance of the code under different decoder configurations, and these results can be used in combination with knowledge of the hardware to produce a set of tradeoffs that will produce a code that maximizes performance within the constraints of the hardware.

| **Parameter** | **Target Value** |
|---|---|
| Block Length | 1024 bits (flexible) |
| Number of RSC Encoders | 2 |
| Constraint Length of RSC Encoders | 3 |
| Generator Function of RSC Encoders | {7,5} (octal notation) |
| Puncturing | Flexible ($R$ = 1/2 or 1/3) |
| SISO Decoding Algorithm | Max-Log-MAP |
| Number of Decoding Iterations | 6 (flexible) |

Table 4.1: Parameters used in turbo decoder design.

The parameters that were used for the decoder design are summarized in Table 4.1. Although high block lengths generally provide better code performance, they also incur greater frame latency, since an entire frame must be received before decoding can begin and the decoding time is proportional to frame length. Since the decoder was aimed at least for partial use in voice communications, the target block length was restricted to a reasonably small 1024 bits to avoid packetization delays. Although this was the target for the design, the method of implementation is such that the decoder hardware complexity does not increase with block length, so larger block lengths can be supported. This is favorable for data transmissions where high latency and packetization can be tolerated if the result is reduced bit error rates.

The RSC encoder parameters were fixed because flexibility would have been difficult to implement and because a larger constraint length or more encoders would have significantly increased the complexity of the decoding hardware. The generator function {7,5} is an octal representation of the taps that exist on the RSC encoder. The first term expresses the function for producing the feedback variable, $a_k$, and the second term produces the encoder output where $a_k$ is the input to the lower shift register. If it is expressed in binary as {111,101}, is it easy to see this relationship in Figure 4.1.

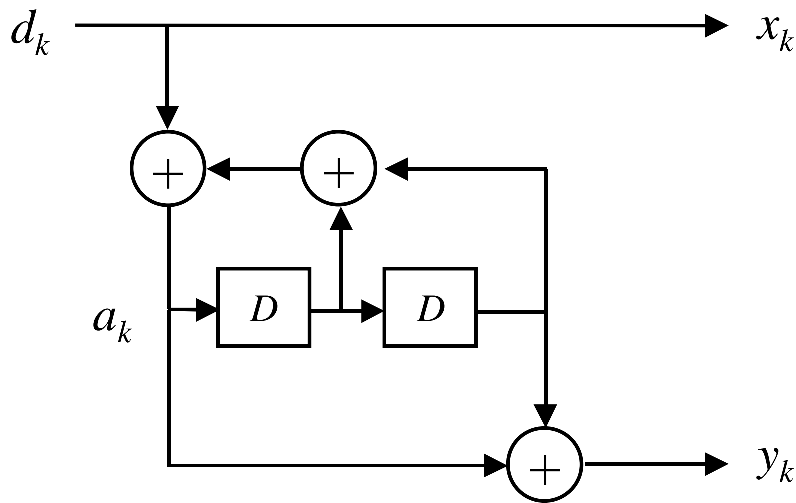



Figure 4.1: Diagram of a {7,5} RSC encoder.

Puncturing was easy to implement flexibly because it is just a matter of the decoder clearing the values that have been punctured out. Max-Log-MAP was chosen as the decoding algorithm for the constituent decoders in order to simplify the implementation due to concerns that the lookup operation for the correction factor would degrade the speed of the decoder, which already had a large computational delay. The number of iterations was chosen to be six to achieve acceptable performance while keeping necessary bounds on latency. Like the block length, the implementation allows this parameter to be changed easily at run-time so this parameter could be modified depending on the quality of service (QoS) required by the data.

## 4.3 Implementation Strategy

To describe the strategies used for the implementation of the turbo decoder, it is useful to review what operations it must perform. Figure 4.2 shows the diagram of a turbo decoder from Chapter 2.

It is obvious from inspection of Figure 4.2 that the decoding operation can easily be split up into modules. Modularity is useful because it allows the system to be developed and tested in smaller, more manageable parts. By defining an interface between the modules, each of the modules can be designed to work with inputs from and provide outputs to other modules. The three major modules from Figure 4.2 are the adder, interleaver, and constituent decoder modules. The other modules (hard decision, demultiplexing, and scaling) can be implemented by processing data on the host, so they will not be implemented on the FPGAs for this application.

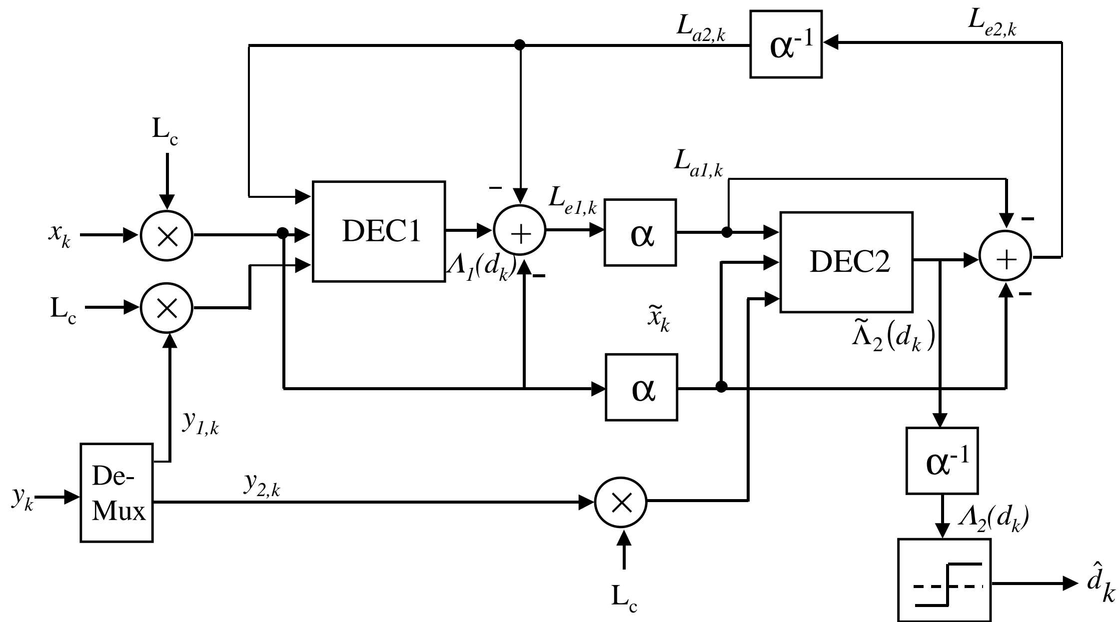



Figure 4.2: Turbo decoder schematic.

Another thing that stands out about a system such as this is that a large number of operations require the entire block of data to be available before the operation can produce any output. This can be true for the interleavers and deinterleavers, depending on the implementation, and it is certainly true for the constituent decoders, which must perform both forward and backward recursion over the data to produce their soft

estimates. Only the adders, where no reordering is required and output bits are dependent only on the current set of inputs, could possibly be implemented to stream outputs as its inputs are received. However, since none of the other modules can process the data in a stream-oriented fashion, this is not very useful.

Even if streaming could be done in all of the modules, there would still be problems with stream synchronization. This problem is best illustrated by looking at one of the adders. Two of the three inputs to the adder module, the systematic and a priori information bits, are also required to generate the third input, the LLR of the message bits. Thus, the systematic and a priori information would have to be delayed to account for the delay of the decoder. Since, this delay is proportional to the total block length, the storage necessary to buffer this information on the FPGA would be immense. This problem only gets worse when one considers that the decoder must perform multiple iterations while using the same systematic and parity bits, so the entire blocks of input data must be buffered.

The massive memory storage requirements require the use the off-chip storage provided by the SRAMs on the WILDFORCE board. Because the modules will need to use off-chip memory to retrieve their inputs and store their outputs, the memory can be used to provide a uniform interface between all of the modules.

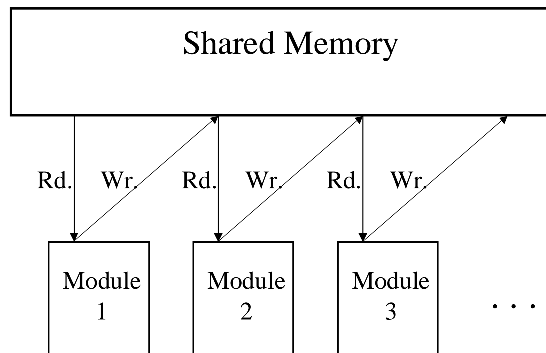



Figure 4.3: Using the memory as an interface between modules.

Illustrated in Figure 4.3, the scheme of using memory as a module interface works by handing each module a memory address for each one of its inputs and outputs. The address is a pointer to an array that contains a block length of data. Thus, for the output

of one module to be used as input by a second module, the second module need only be given the pointer to the output array of the first module.

This scheme of using the memory as a module interface is very useful in testing because it eliminates many possible timing issues that could occur if the modules were directly interfaced on chip. When a module is tested individually, if its outputs to memory are correct, then the designer can be sure that it will work with other modules in the system. Although using the memory as an interface is much slower than using a direct connection on the chip, memory use cannot be avoided for this design, so it an added benefit that it eases the testing process.

As shown in Figure 4.3, this strategy of using the memory as a module interface requires that all of the modules have access to the same shared memory. On the WILDFORCE board, this is nearly equivalent to stating that all of the modules must reside on the same PE, since the PE memories are not shared. While this may seem to be an egregious under-utilization of system resources, the alternative is not very appealing either. Since PE memories are not shared, applications that work well when spanned across multiple PEs are those that require little or no feedback. Since the turbo decoder is essentially based on iterative feedback, this would not work very well. If the decoder were to span multiple PEs, then a module would have to be created to transfer several blocks data from the SRAM across the systolic bus where another module would write this data to memory. While this transfer is occurring, the SRAMs associated with each PE would be occupied for several block lengths of transfers, preventing any other processing during that time and detracting substantially from any performance gains that could be gained.

A top-level diagram of the decoder implementation is shown in Figure 4.4.

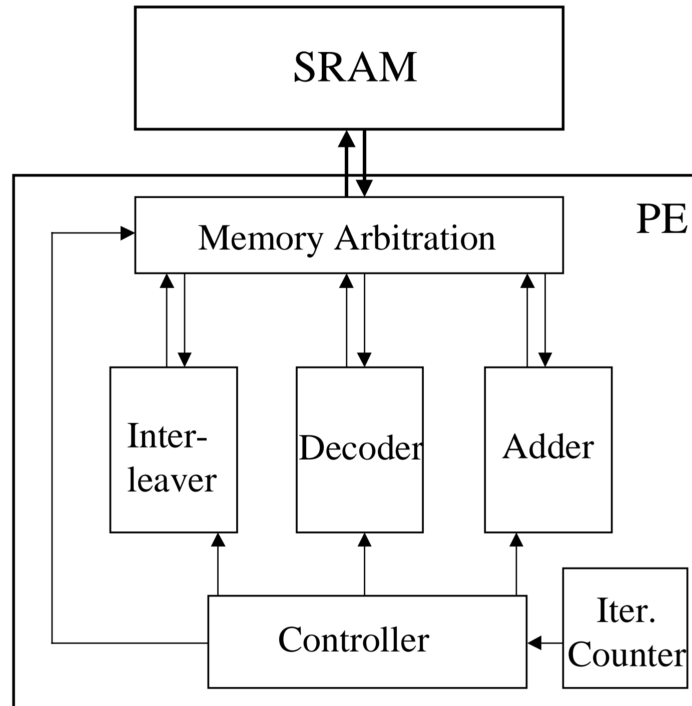



Figure 4.4: Top-level diagram of turbo decoder implementation.

The main function of the controller is to execute the decoding algorithm by successively turning the modules “on” and “off.” When a module is activated, the controller sends a “start” signal and a set of memory addresses for inputs and outputs. It also instructs the memory arbitration unit to connect the “active” module to the PE’s port to the SRAM. The controller performs memory access for the “active” module until it receives a “done” signal from that module. Successive modules are activated to complete the iteration of the decoder, and the iteration counter is incremented. If the iteration limit has not been reached, then the cycle is repeated again. The succession of steps that make up a decoding cycle is shown in Table 4.2.

Since data dependencies and memory bus contention prevent any of these steps from occurring in parallel, it is possible to use the modules for more than one purpose. For example, the interleaver module can perform either interleaving or deinterleaving and the decoder module can act as either the first or second constituent decoder, depending on control signals supplied by the controller. By making the modules slightly more flexible, needless duplication of hardware that would consume large amounts of chip area is avoided.

| Operation | Memory Pointers Supplied |
|---|---|
| 1. First decoder | $x, y_1, L_{a2}, \Lambda_1$ |
| 2. First adder | $x, L_{a2}, \Lambda_1, L_{e1}$ |
| 3. First interleaver | $L_{e1}, L_{a1}$ |
| 4. Second interleaver | $x, \tilde{x}$ |
| 5. Second decoder | $\tilde{x}, L_{a1}, y_2, \tilde{\Lambda}_2$ |
| 6. First deinterleave (only on last iteration) | $\tilde{\Lambda}_2, \Lambda_2$ |
| 7. Second adder | $\tilde{x}, L_{a1}, \tilde{\Lambda}_2, L_{e2}$ |
| 8. Second deinterleave (not on last iteration) | $L_{e2}, L_{a2}$ |
| 9. Increment iteration counter | -- |
| 10. If iteration limit not exceeded, return to 1. | -- |

Table 4.2: Steps in decoding operation from perspective of control unit.

All of the internal variables in the decoder ($L_a$, $\alpha$, $\beta$, $\Lambda$) are represented as 16-bit signed fixed-point numbers. However, the data coming in from the channel ($x$, $y_1$, $y_2$) are represented as 8-bit signed fixed-point numbers. Since the memory words on the WILDFORCE are 32-bits long, there is some opportunity to save memory space and decrease the number of reads required to perform some of the calculations by concatenating multiple variables and storing them in the same memory word. This is performed twice in this design, with $x$ and $y_1$ being concatenated and $\beta_1$ and $\beta_2$ being concatenated. The resulting memory map is shown in Table 4.3 (where $N$ represents the block length).

| Memory Range | Variable(s) |
|---|---|
| $0 \rightarrow N - 1$ | Interleaver Pattern |
| $N \rightarrow 2*N - 1$ | $x$ & $y_1$ |
| $2*N \rightarrow 3*N - 1$ | $y_2$ |
| $3*N \rightarrow 4*N - 1$ | $L_{a2}$ |
| $4*N \rightarrow 5*N - 1$ | $\Lambda_1$ |
| $5*N \rightarrow 6*N - 1$ | $L_{e1}$ |
| $6*N \rightarrow 7*N - 1$ | $L_{a1}$ |
| $7*N \rightarrow 8*N - 1$ | $\tilde{x}$ & $\tilde{y}$ |
| $8*N \rightarrow 9*N - 1$ | $\tilde{\Lambda}_2$ |
| $9*N \rightarrow 10*N - 1$ | $L_{e2}$ |
| $10*N \rightarrow 11*N - 1$ | $\beta_1$ & $\beta_2$ |
| $11*N \rightarrow 12*N - 1$ | $\beta_3$ & $\beta_4$ |
| $12*N \rightarrow 13*N - 1$ | $\hat{d}$ |

Table 4.3: Memory map for turbo decoder.

With the overall system implementation described, the individual modules will now be described in greater detail.

## 4.4 Interleaver Module

The interleaver module, while conceptually simple, provides the greatest implementation flexibility of any of the modules. Multiple schemes exist for how the actual interleaving and deinterleaving are performed as well as for how the interleaving patterns are stored and retrieved. Some of these design alternatives are discussed below, followed by a more detailed description of the design that was selected.

### 4.4.1 Interleaving Strategies

If interleaving and deinterleaving are to be performed by the same hardware module, how the data is manipulated during interleaving and deinterleaving determines the memory and hardware requirements of the modules. The four alternatives for doing this are described below.

*Read (on interleave)/Read (on deinterleave):* In this method, interleaving is performed by reading the data to be interleaved according to an interleaving pattern and then writing it to memory in a linear fashion. The pattern is an array with length equal to $N$, where $N$ is the block length, with entries that hold every number between 0 and $N - 1$ in some order. The pattern values represent the addresses from where the values to be interleaved will be read from the module input array. Thus, the $i^{th}$ entry in the output array is found by reading from the address the $i^{th}$ location in the pattern array. For deinterleaving in this scheme, the data are read according to an interleaving pattern and then written linearly. However, this pattern must be different from the one used in the interleaving stage. Thus, this method has the advantage that the hardware for interleaving is identical to that for deinterleaving, but it requires storage for two pattern arrays. This effect is illustrated in Figure 4.5 for the simple case of a block with only eight entries.

*Write/Write:* This method is very similar to the read/read method, except that the data is read linearly and the written to output in an interleaved fashion. As is the case for the read/read scheme, the write/write scheme uses the same hardware for interleaving and deinterleaving, but uses two different patterns.

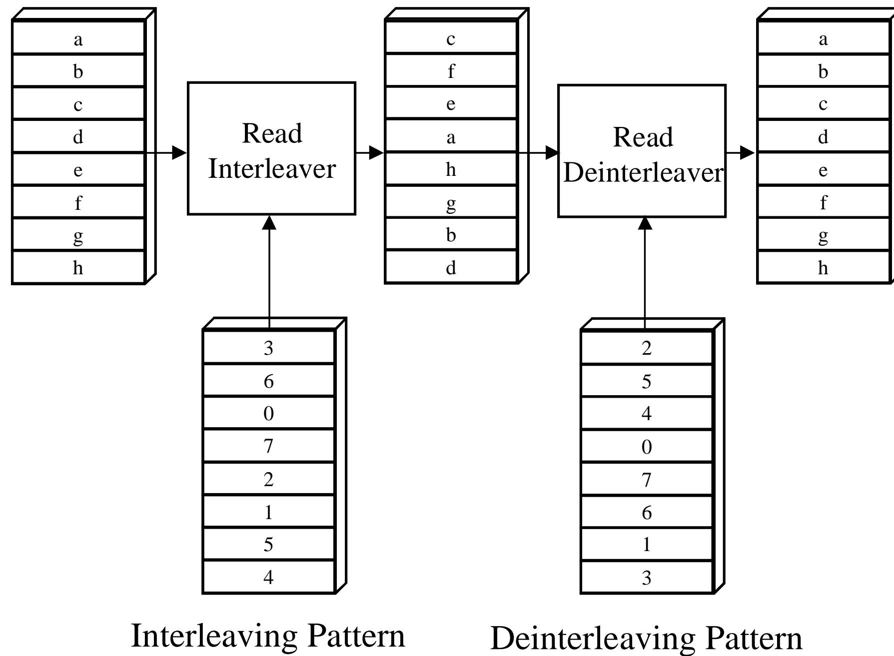



Figure 4.5: Illustration of why two patterns are required for a read/read interleaving scheme

*Read/Write:* When performing interleaving in the read/write scheme, the data is read according to an interleaving pattern and written linearly. However, when deinterleaving in this scheme, the data is read linearly and written according to an interleaving pattern. By doing this, the interleaver and deinterleaver can use the same interleaving pattern. This is shown in Figure 4.6. Thus, the hardware is more complex since it must operate differently when performing the two operations, but only one pattern array must be stored.

*Write/Read:* This method is analogous to the read/write method except that the interleaver reads linearly and writes according to the interleaving pattern and the deinterleaver reads in an interleaved fashion and writes linearly. This approach also requires the hardware to perform the two operations differently but with the advantage of using a single array for pattern storage.

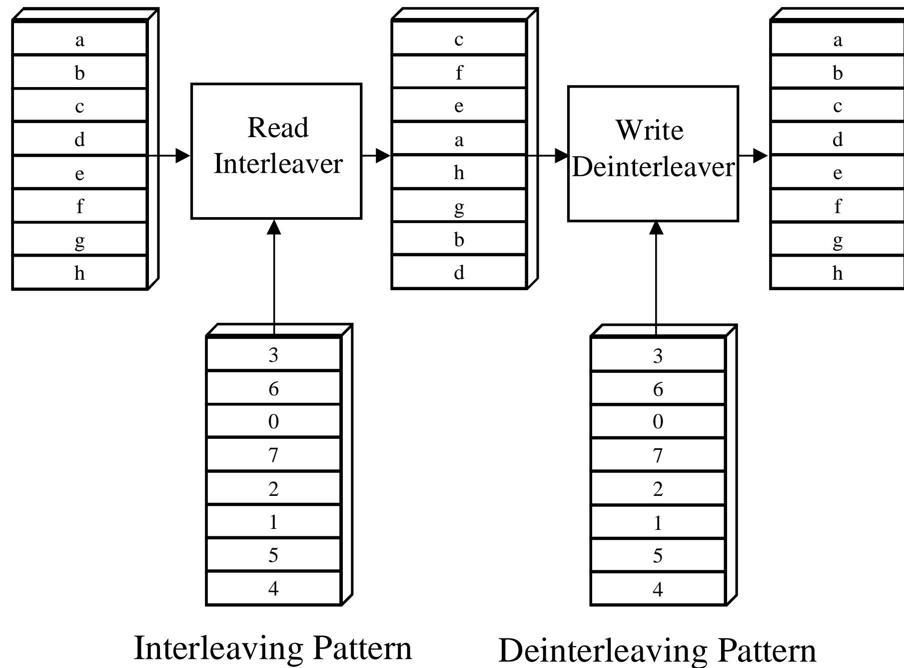



Figure 4.6: Illustration of the use of a single pattern for a read/write interleaving scheme

### 4.4.2 Pattern Storage Strategies

In addition to the different approaches to performing interleaving discussed in Section 4.4.1, there are also several different ways of implementing storage for the interleaver patterns on the WILDFORCE platform. These alternatives embody implementation tradeoffs between chip area, memory space, decoder throughput, scalability, and system portability. The options for storing the interleaver pattern are:

*Single SRAM*: This scheme has the interleaving pattern (along with the input and output arrays for the interleaver) stored on the SRAM associated with the PE the decoder is implemented on. Thus, for each member of the input array, two reads and a write from the SRAM are required. This means that performing one block (de)interleave requires approximately $3*N$ clock cycles. Additionally, the pattern data requires either $N$ or $2*N$ memory words, depending on whether a read/write class scheme or a read/read class scheme is used. Although slow, this implementation has the advantage of being very scalable with regard to block size and very portable to other platforms since it uses very little chip area and a single memory port.

*Two SRAMs*: This method has the interleaving data stored in the SRAM associated with a neighboring PE. During interleaving, a part of the chip would retrieve patterns from the neighboring PE through the systolic bus and supply them to the interleaver. The input and output array would, of course, still be stored on the SRAM associated with the decoder PE. Thus, only a single read and write from the decoder's memory would be required to interleave a single datum, and the entire operation would require approximately 2*$N$ clock cycles. The main disadvantage of this method is that it wastes an entire PE to perform a memory fetch. Although this is not a large problem on the WILDFORCE (unless multiple PEs are being used for the decoder), using multiple memories could provide portability problems when moving to other platforms.

*On-Chip RAM*: Since it is possible to configure CLBs on the FPGA as RAM elements, the interleaving pattern can be stored on the FPGA. By doing this, it is able to provide the improved performance of the two SRAM scheme without using another PE. The major disadvantage of this method is that it can consume large portions of the chip area. The 1024x10 RAM required for pattern storage uses almost 400 of the 2304 CLBs on an XL4062 FPGA. If two patterns are required, then more than 1/3 of the chip will be used for pattern storage. This also limits the flexibility of the decoder to use large block sizes since the area required for pattern storage for large blocks would exceed that of the entire FPGA.

*Sequence generation*: Another possible implementation would use a linear feedback shift register (LFSR) to produce the interleaving pattern. Although this would reduce the number of memory accesses needed to perform the interleaving and would require relatively little chip area, this implementation choice has problems with flexibility. Because the taps on the LFSR must be carefully chosen to produce a sequence that is full-cycle, changing the block length requires redesigning the LFSR structure.

### 4.4.3 Module Implementation

In order to preserve flexibility and portability, the method of storing the interleaving patterns in the SRAM associated with the decoder PE is used for this design. The

read/write scheme is used for interleaving and deinterleaving. Thus, the interleaver module must be able to function by interleaving on a read or a write, depending on the mode it is in. When it is in interleave mode (apply pattern to reads), the pattern must be fetched from memory before the data can be read, but it can be written to the next address in the output array as soon as it arrives. When it is deinterleaving (apply pattern to writes), the data can be fetched without needing to wait for the pattern value, although both the data and the pattern must be fetched before the write can occur. The constraints of the memory architecture of the WILDFORCE, namely that data values are not valid until two clock cycles after they are requested for a read, create complications in the design that can be handled by inserting idle states into the module's controlling state machine. The state diagram for the interleaver module is shown in Figure 4.7.

When the module is functioning as an interleaver, it applies the interleaving pattern to the reading of the data. For this reason, the pattern read must be completed before the data read can be started. The first pattern is fetched before entering the main loop in the process so that the data can be fetched from an address based on the pattern value. After a request for the first data word is issued, the counter is incremented and another pattern read is issued. Since the write cannot be performed until the results of the data read have become available, this time is used to prevent delays later by pre-fetching the next pattern value. The counter must be incremented because it is used as the index into the pattern array, and the first value has already been fetched. Once the data arrives, it is written the following cycle. Because the counter has already been incremented for the pattern read, the write address must be based on the value of the counter less one. After the last data word is read, then the interleaver goes to a special state to write the last word since the counter cannot exceed its maximum value.

When the module is deinterleaving, it applies the interleaving pattern to the writing of the data words. Since the address of the data being read is independent of the pattern value, they can be fetched on consecutive cycles, and the write can proceed once both have arrived. In this case, the write address is based on the value of the pattern data, so the counter can be incremented at the beginning of each loop iteration without having to store a lagged version to perform write addressing.

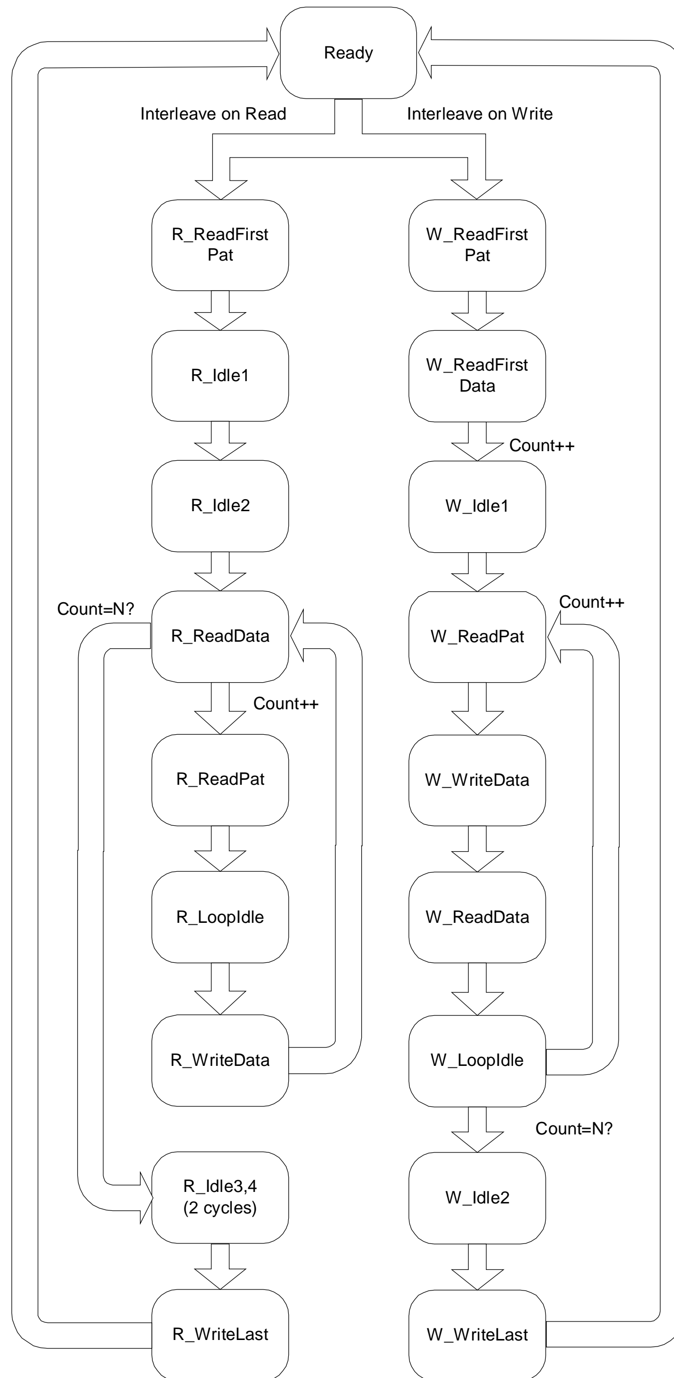



Figure 4.7: State diagram for the interleaver module

## *4.5 Adder Module*

The adder module is the simplest of the modules. Simply put, the adder must read three data words, perform an arithmetical operation on them, and write the result back to

memory for each element in the associated arrays. Like the interleaver module, however, some clock cycles can be saved if the module's state machine is made more complex to cope with the delay in reading the data. In the module state diagram shown in Figure 4.8, the first addend is fetched before the main processing loop. This is done because the next addend request is issued in the processing loop before the current result is written to memory. This is done to make efficient use of the delay between the request for the two subtrahends and their arrival. As is the case with the interleaver module when it is interleaving on reads, the writing of the result for this module must be based on a decremented version of the counter since it is incremented to fetch the next addend before the write is performed.

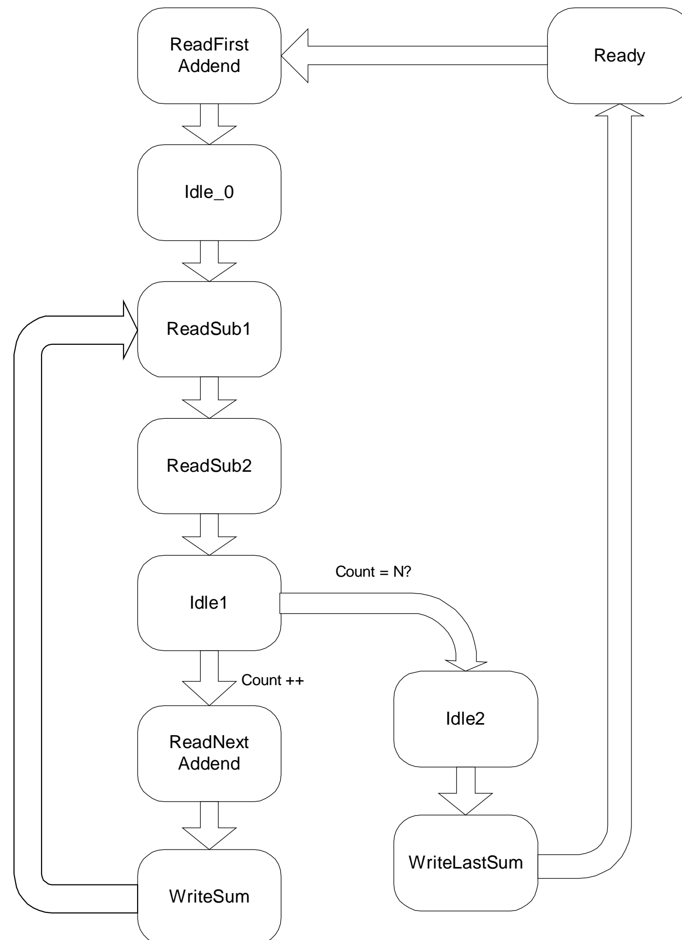



Figure 4.8: State diagram for the adder module

## 4.6 Decoder Module

Like the interleaver module, the decoder module is designed to be able to perform multiple functions in the decoder schematic. The module must be able to act as either the first or second constituent decoder. Although these two modules are computationally very similar, their implementations differ in a couple of ways.

### 4.6.1 Differences Between the First and Second Decoders

The first difference between the functionality of the two decoders is actually an artifact of the memory organization. As discussed in section 4.3, there are several instances where variables are concatenated and stored in the same memory location. One of these concatenations is performed with the systematic data, $x$, and the parity information from the first encoder, $y_1$. This is done to decrease the number of memory reads required for decoding, since both pieces of information will be needed at the same time for decoding. This technique only enhances the performance of the first decoder because it is the only one that uses $x$ and $y_1$ for decoding. The second decoder uses the interleaved version of the systematic information, $\tilde{x}$, and the parity information from the second decoder, $y_2$. Since only $x$ (and not $\tilde{x}$) is transmitted in the encoded bit stream, $\tilde{x}$ is not known until $x$ is interleaved. However, $y_2$ is encoded from the interleaved systematic bits, $\tilde{x}$. Thus, $y_2$ cannot be stored with its corresponding systematic information since it is not included in the encoder output. The end result of this is that the second decoder module must perform separate read operations to fetch $\tilde{x}$ and $y_2$ while the first decoder can get both the systematic and parity information with a single read. Thus, the module state machine must perform differently depending the decoding operation that is being performed.

The second difference is related to *trellis termination.* Since the sequence of message bits that are encoded is not deterministic, it is not possible to predict what the final state of the encoder will be at the end of the block. This degrades the performance of the decoder since the calculation of the backward state metrics, $\beta$, is based on the assumption that the encoder ends in the all-zeros state. Trellis termination attempts to rectify this situation by using the final $M$ bits in a block to return the encoder to the all-zeros state, where $M$ is the memory of the encoder. Instead of using all $N$ message bits for meaningful information, if only $N - M$ are used, then the values for the remaining $M$

bits can be calculated based on the state and generator matrix of the encoder in order to force the state to the all-zeros at the end of the block. Although this pattern will terminate the first encoder correctly, unless the interleaver is designed very carefully it will not terminate the second encoder, since the termination bits will be spread throughout the block on which the second encoder operates. Thus, the second encoder is usually not terminated, which means that the second decoder has no prior knowledge of the ending state for its backward metric calculations. Therefore, in the second decoder all of the initial values for $\beta$ should have the same weight.

The implementation consequences of this are related to the fact that each decoder uses the same memory space for storing the $\beta$ array. This can be done because $\beta$ is only used internally within each decoding function. Each decoder is responsible for initializing its $\beta$ array. Because we have just shown that the two decoders must initialize $\beta$ differently, this means that the decoder module state machine must perform the initialization differently depending on which constituent decoding operation it is performing.

### 4.6.2 Module Implementation

The state diagram for the decoder module is shown in Figure 4.9. When the decoding operation is started, the $\beta$ array is initialized according to the decoder setting specified by the top-level system controller. Because the calculation of $\beta_k$ requires knowledge of $\beta_{k+1}$, the initialization must define the last set of metrics, $\beta_N$. For this implementation with encoders with memory $M = 2$, four metrics must be initialized, which can be accomplished in two clock cycles due to the concatenation of the $\beta$ values. After this, the decoder calculates the $\beta$ array starting from the final state and ending with the earliest state and stores is to memory. Depending on which decoder it is currently implementing, a single processing loop for calculating a set of four $\beta$ values requires either six or seven clock cycles. After the reverse state metrics are calculated, the decoder calculates the forward state metrics, $\alpha$, and combines them with the previously calculated $\beta$ results to form the final decoding results, the bit LLRs represented by $\Lambda$. Because the LLRs are calculated as soon as the $\alpha$ values are available, only the current set of $\alpha$ values need to be stored, so they can be used to calculate the next set of $\alpha$ metrics. After the entire

block has been traversed in the forward direction, all of the LLRs have been calculated, and the decoder module can return to its ready state.

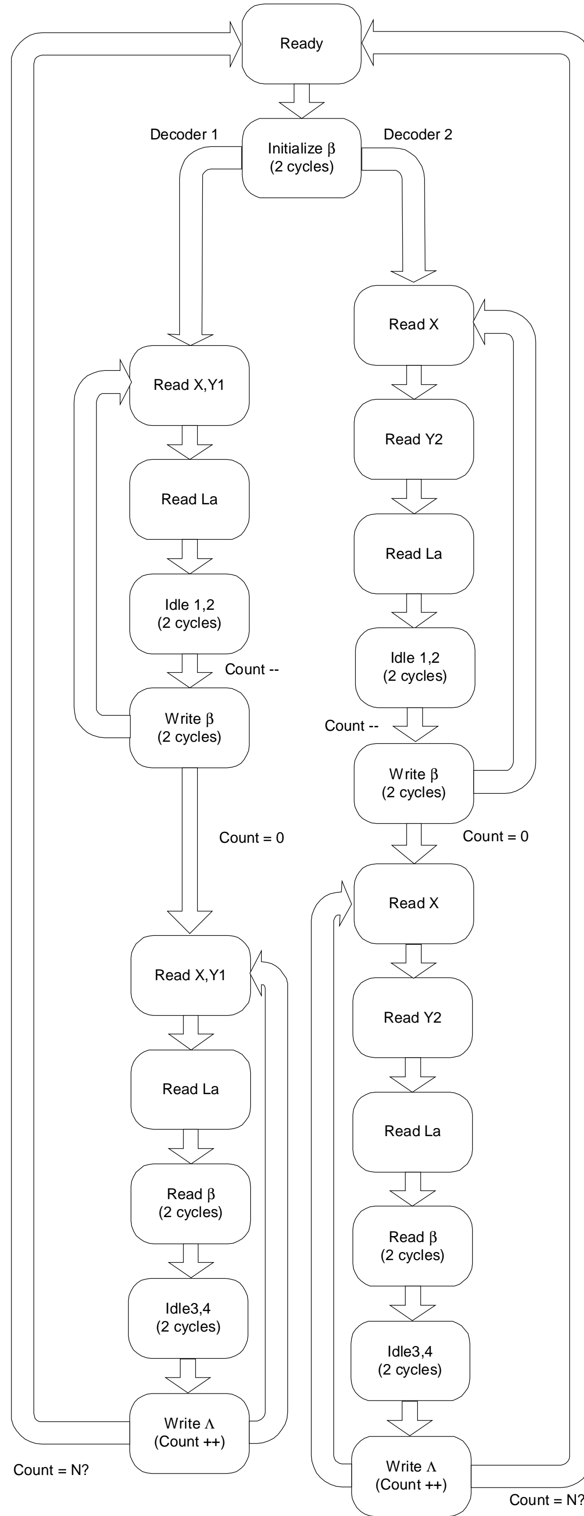



Figure 4.9: State diagram for the decoder module

## 4.7 Host Code

The WILDFORCE board cannot perform without some interaction from the host PC. Of course, the host code is responsible for providing a means to program the FPGAs on the board and for giving the user access to the board clock and reset functions. However, in this design, the host code is also used for several additional purposes that aid in verification, benchmarking, and simulation of a larger communications system.

Because the PC is the main way to get data to the board, its access to the PE memory can be used to replace some of the interface modules that would need to be in place in a system that was part of a larger communications system. For instance, in a real system that streams received blocks to the decoder, a module would have to be in place to multiplex the systematic and parity bits into separate arrays, with the details of the implementation depending on whether or not puncturing is used. In this system, the host code can write the systematic and parity bits directly into the PE memory, thus simulating the actions of such a module.

In this design, the host code generates a random message bit array, encodes it using a turbo encoder based on {7,5} RSC encoders, adds simulated additive white Gaussian noise (AWGN) to the encoder output according to the desired SNR, and writes these simulated received bits to the SRAM on the WILDFORCE. It also writes the number of decoding iterations and the block size to reserved locations in the memory. Then, the board is reset, and the decoding operation begins. The host program waits for an interrupt signal from the PE, which signals that the decoding has been completed. Then, the results are read from the memory, and a new block of encoded data is loaded to the board. While the board is processing the new data, the results from the previous calculation are put through a hard decision and compared against the encoded bits in order to generate bit error rate (BER) and frame error rate (FER) statistics.

By doing this, the host program can be used to document the error performance of the decoder. It can also be used for validation and benchmarking of the decoder. If the SNR is set to infinity, then the decoder should produce a zero BER. If this is not the case, then the decoder is not performing correctly and should be debugged. Also, the

host program can be used to measure the time that the decoder takes to process a block, which can be used to calculate an overall information throughput for the system.

## 4.8 Summary

This chapter discussed the details of the implementation of the turbo decoder on the WILDFORCE platform. The overall goals of the implementation were discussed, and the choice of system parameters targeted to achieve those goals was justified. An implementation strategy that emphasizes modularity and the use of memory as a module interface was presented, along with a discussion of the memory resources that are required for the implementation. Then, each of the three major modules—the interleaver, the adder, and the decoder—were discussed in detail along with the major design choices that were made in their development. Finally, the functionality of the program running on the PC host that controls the WILDFORCE board was discussed.

# Chapter 5. Results

The following chapter presents the results of the implementation of the turbo decoder on the WILDFORCE CCM. First, the experimental setup from which the results were derived is described. Then, details of the hardware implementation are discussed in the context of the attainment of performance goals while obeying the constraints of the system. Functional correctness of the design is shown by comparing the code performance of this implementation to software-based simulations of turbo codes. Additional results are presented that show the performance of the hardware while varying several design parameters.

## 5.1 Test Setup

The system that was used for the implementation was a WILDFORCE PCI board with five Xilinx 4062XL FPGAs, each with a local SRAM with 1MB of capacity that is addressable in 32 bit words. Because the PC must control the board operation and because the main datapath to the board is through the host PCI bus, the host has several tasks in this design. The control program running on the PC was used to generate random message data, perform turbo encoding of the data, simulate the effects of the AWGN channel and scale "incoming" signals with the reliability factor, $L_c$, before being placed in the PE's memory. Because the host can maintain tight control over the board through the WILDFORCE API, the host was also used to manage the decoding of multiple blocks for the purpose of calculating the bit error rate and frame error rate (FER) of a code running on the system. These calculations were done by comparing the hard decision outputs of the system with the original message bits. Finally, by using timing functions in the host code, the decoding process was timed for benchmarking purposes.

## 5.2 Implementation Statistics

Table 5.1 summarizes the FPGA resource utilization for the turbo decoder. With about three-fourths of the FPGAs logic resources used by this implementation, there is still room for additional hardware that would be required if the design were on a stand-alone

platform, or even for integration with other modules in the software radio for which the turbo decoder was designed.

| Resource | Utilization |
|---|---|
| CLBs | 1734 of 2304 (75%) |
| CLB flip-flops | 441 out of 4608 (9%) |
| IOBs | 75 out of 193 (38%) |
| IOB flip-flops | 52 |

Table 5.1: Resource utilization for turbo decoder implementation.

The utilization of the chip resources can also be seen in the chip layout for the decoder's FPGA configuration, shown in Figure 5.1.

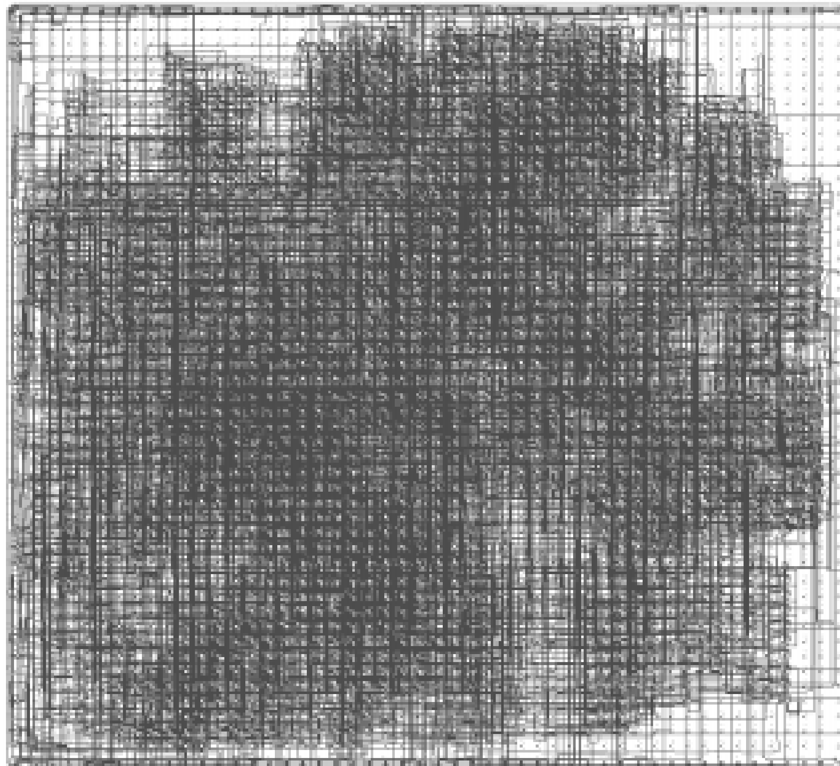


Figure 5.1: Decoder layout on XC4062XL chip.

The timing requirements that dictate the speed at which the design can be operated are summarized in Table 5.2.

| Timing metric | Value |
|---|---|
| Average Connection Delay | 8.122 ns |
| Maximum Pin Delay | 44.539 ns |
| Minimum Clock Period | 324.39 ns (3.083 MHz) |

Table 5.2: Timing information for the turbo decoder.

A single decoding iteration requires the use of the first and second constituent decoders, two adder modules, and three interleaver modules.[8] If the state machines for each module are analyzed, it is found that each contains one or more loops that traverse the entire block of data. By counting the number of states in each loop, the number of cycles required to perform the module's operation can be approximated as a function of the block length. Thus, the first decoder requires approximately 13*N* clock cycles, while the second decoder requires 15*N*. The adder and interleaver modules require on the order of 5*N* and *4N*, respectively. For our target iteration count of 6, then the total number of cycles required to decode a single *N*-bit block of information is

$$5\times(13N+15N+3(4N)+2(5N))+(13N+15N+3(4N)+5N)=295N\ \frac{\text{cycles}}{\text{block}}. \quad (5.1)$$

In order to meet the target information decoding rate of 32 kbps, the system clock must then run at

$$32\frac{\text{kb}}{\text{sec}}\times 295\frac{\text{cycles}}{\text{bit}}=9.44\text{ MHz}. \quad (5.2)$$

Although this rate is higher than the maximum clock rate given by the place and route tool, all is not lost. The clock speed estimations given by the CAD tools are generally very conservative. In fact, this design was found to run in actual hardware at speeds up to 11.5 MHz while producing correct results. By using this speed instead, it is found that information rates of up to (11.5 MHz) / (295 cycles/info.bit) = 38.9 kbps can be supported, which meets the target throughput. One final note is that the Xilinx software was run without any timing constraints. If constraints are used, the minimum clock

[8] The reason that only three interleaver modules are required is that the LLRs from the second decoder only need to be deinterleaved during the last decoding iteration. During this state, there is no need to deinterleave the extrinsic information because it will never be used as a priori information for the next stage. In fact, it is also unnecessary to calculate the extrinsic information during the last iteration, so only one adder is required for that iteration.

statistic should be increased significantly. Analysis also shows that the decoder module has by far the greatest path delays. By including additional registers in this module to pipeline the computations, it is likely that the clock speed and the overall system throughput can be increased further.

## 5.3 Functional Results

Although the performance has been found to meet the implementation goals, this is only a success if the decoder performs in a functionally correct manner. In order to verify that this is the case, the results from the implementation are compared to those from a turbo coding MATLAB simulation.

Figure 5.2 compares the performance of the hardware and the simulation for a decoder that uses six iterations with $N = 1024$ and no puncturing for values of $E_b/N_0$ from 0.5 dB to 2.0 dB. The BERs are shown in Figure 5.2 (a), and the FERs are shown in Figure 5.2 (b). As the figure shows, the hardware results match the expected results very closely, which indicates that the hardware is performing the decoding correctly. The differences that do exist between the simulation and hardware results are due to the randomness in the generation of the encoded bits and the channel noise for the two instances.

In Figure 5.3, the hardware and simulation results are compared for different numbers of decoding iterations while $N = 1024$, $E_b/N_0 = 1.5$ dB, and no puncturing is used. As in Figure 5.2, the hardware results match the simulation results for both BER (a) and FER (b). These results also demonstrate the law of diminishing returns that applies to the number of decoding iterations. For situations where few iterations are used, each additional iteration provides a sizable performance increase. However, as the iteration count increases, the relative performance improvement derived from each additional iteration decreases until the latency cost of performing the extra iterations does not justify the corresponding decoding improvement.

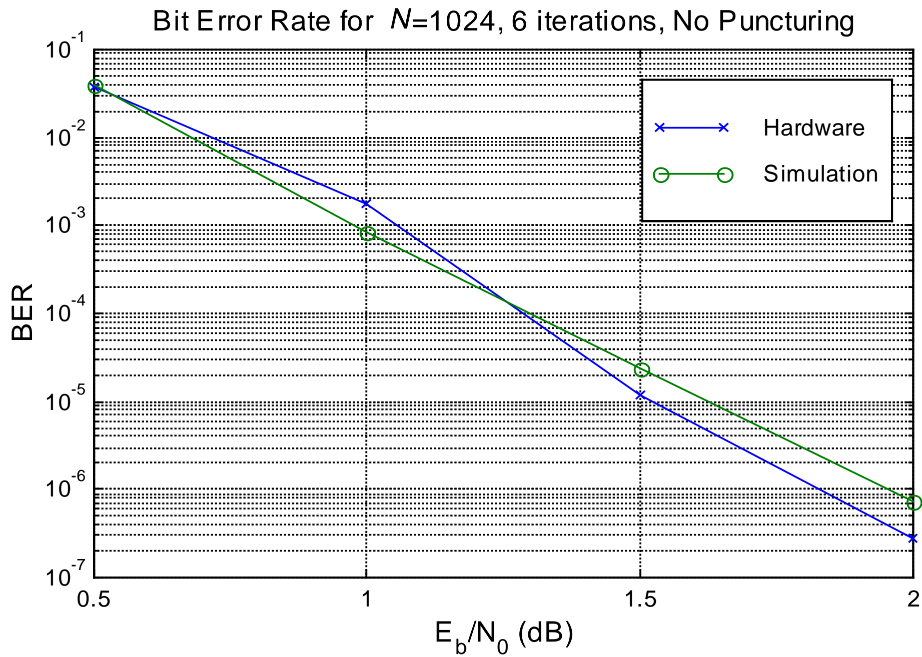
The strong correlation between the simulation and hardware results shown in Figures 5.2 and 5.3 verifies that the implementation is functioning correctly. Figures 5.4 and 5.5 further demonstrate the behavior of the decoder when varying some of the other design parameters. In Figure 5.4, the block size is varied from 16 to 4096 for the case of

$E_b/N_0$ = 1.5 dB with four iterations and no puncturing performed. The graphs show that the error rates decrease as the block size increases. This makes sense because larger blocks can provide a lower probability of low weight codewords by spreading the data bits across a larger space than the smaller blocks can. It should be noted, however, that care was taken to choose interleavers with very good spreading properties for this experiment. At high SNRs ($E_b/N_0 \approx 2$ dB), the use of a "good" interleaver can provide BERs that are more than an order of magnitude lower than those from randomly generated interleavers.

Figure 5.5 shows the effect of puncturing on the decoding results of a system that performs six iterations with $N$ = 1024. The results show that the punctured system requires an $E_b/N_0$ between 0.5 and 0.65 dB greater than the unpunctured system to achieve the same BER and FER. This is offset by the fact that the punctured system utilizes the available bandwidth 50% more efficiently than the unpunctured system.

## 5.4 Summary

This chapter presented the results of the implementation of the turbo decoder. It was found that the implementation was able to meet the stated performance goals while staying well within the constraints of the FPGA resources. Functionally, the implementation produced BERs and FERs that were very close to the expected results generated by software simulations.



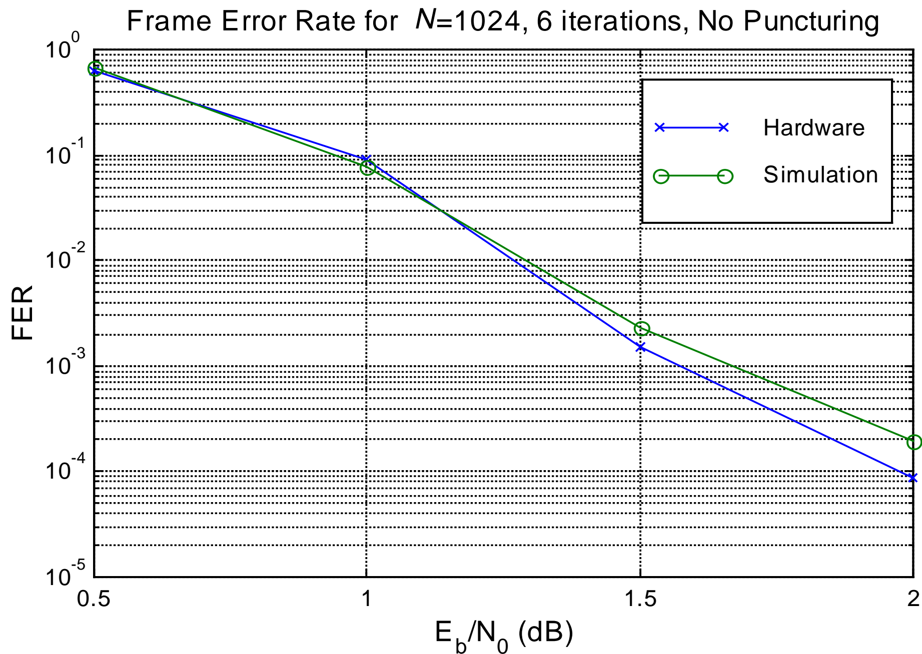
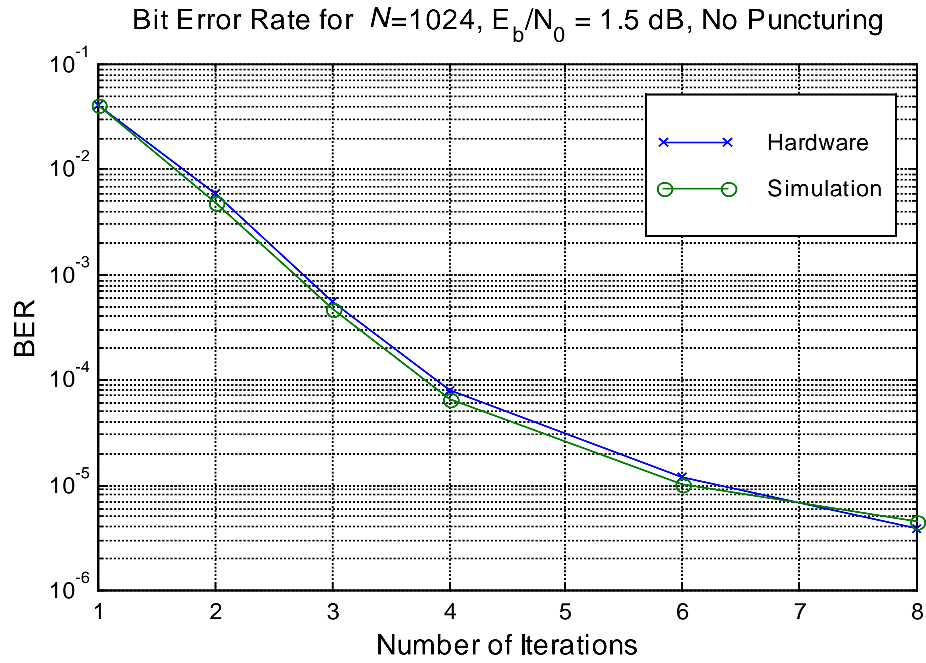




Figure 5.2: Comparison of simulation and hardware results showing bit error rate (a) and frame error rate (b) for $N$=1024, 6 iterations, no puncturing and varying $E_b/N_0$.



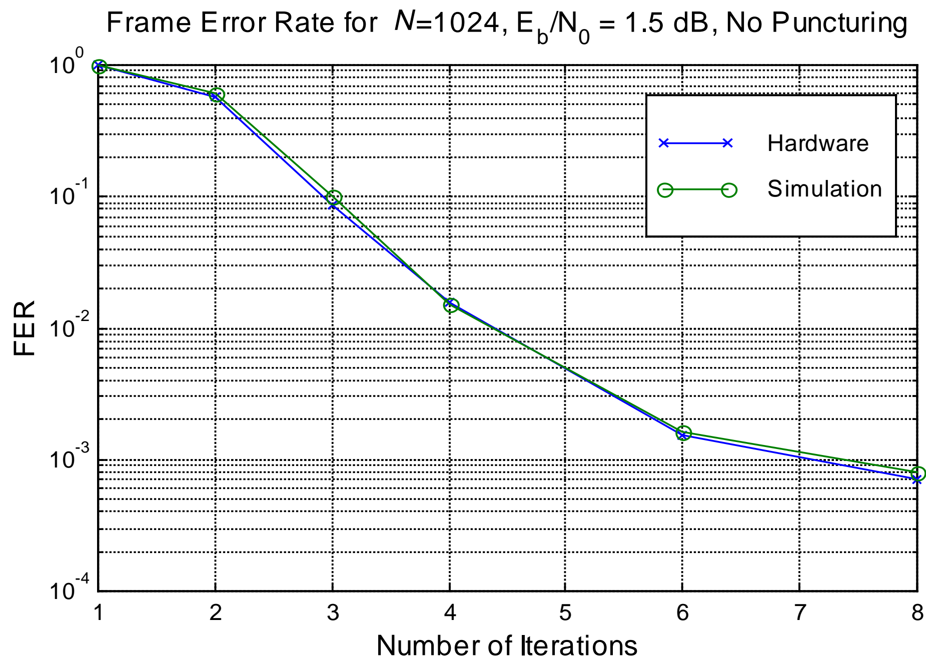
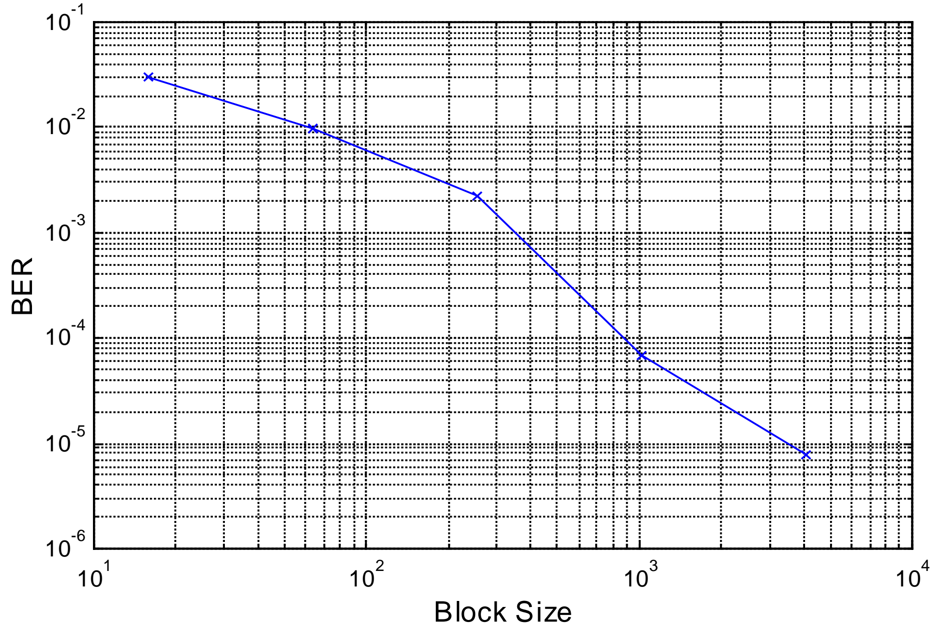




Figure 5.3: Comparison of simulation and hardware results showing bit error rate (a) and frame error rate (b) for $N$=1024, $E_b/N_0$=1.5 dB, no puncturing, and varying iterations.



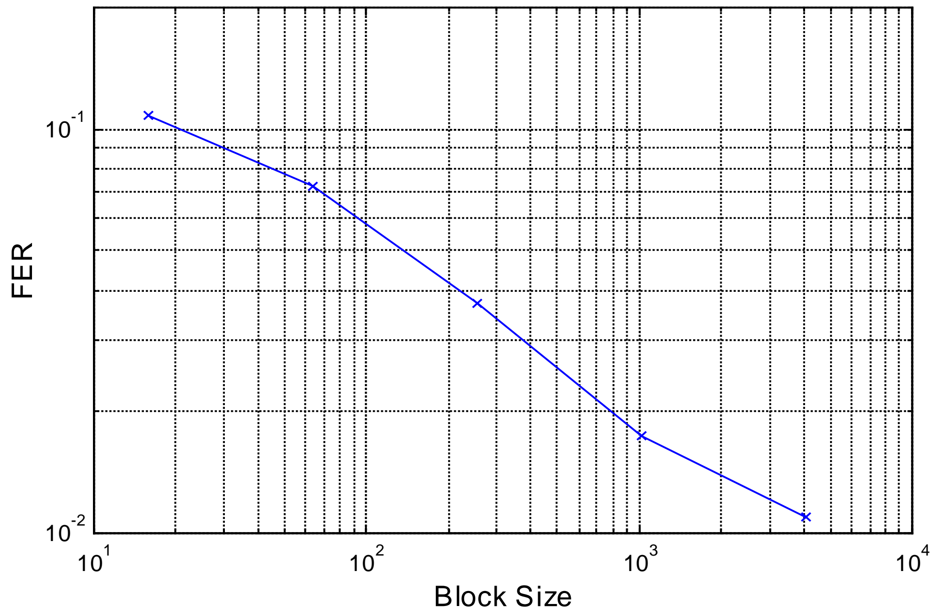
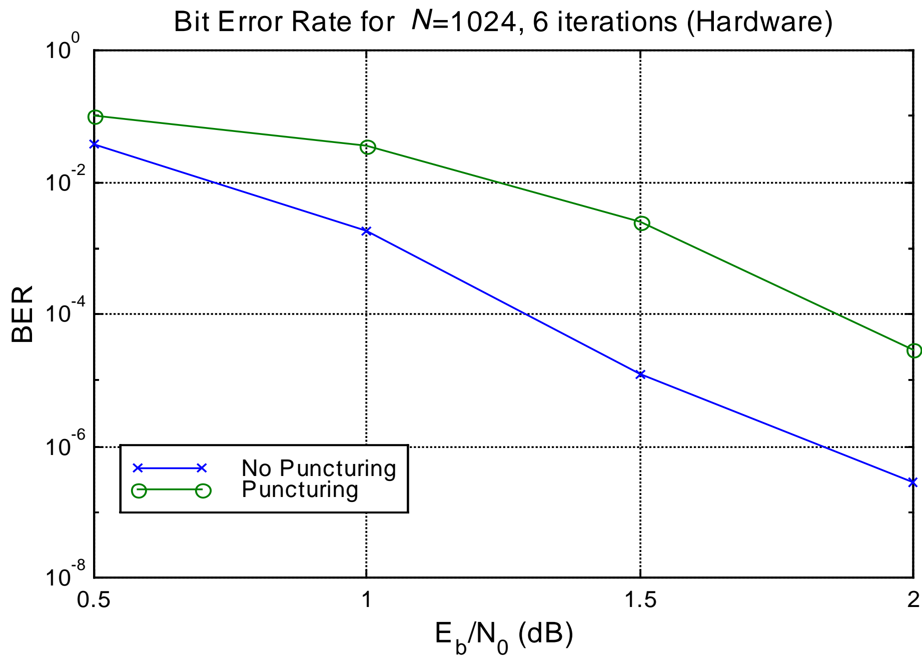




Figure 5.4: Hardware results showing bit error rate (a) and frame error rate (b) for $E_b/N_0$=1.5 dB, 4 iterations, no puncturing, and varying block size.



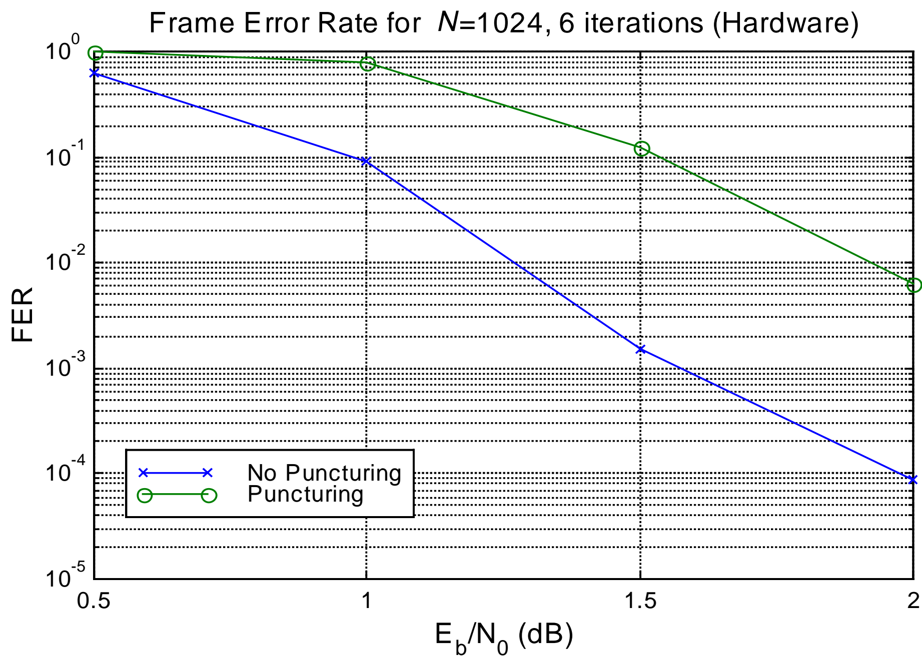




Figure 5.5: The effect of puncturing on the hardware results for bit error rate (a) and frame error rate (b) for $N$=1024, $E_b/N_0$=1.5 dB, and 6 iterations.

# Chapter 6. Conclusion

This final chapter provides an overview of what has been presented in the document, followed by a discussion of possible extensions and improvements to the work.

## *6.1 Summary*

This thesis presented an implementation of a turbo decoder on a configurable computing platform. Chapter 1 introduced the concept of turbo decoding in the context of the motivation for this thesis. The software radio system that the decoder is designed for is briefly described.

In Chapter 2, the theory of turbo coding is presented in much greater detail. In order to provide background and context, other concepts of coding theory were introduced, including convolutional and block codes. The structure of turbo encoders was described. The iterative feedback structure used for decoding turbo codes was also presented. Several soft-input, soft-output decoding algorithms that can be used for the constituent decoders were discussed, including Log-MAP, Max-Log-MAP, and SOVA. Finally, the design of turbo codes was discussed.

Chapter 3 dealt with the other major realm of this design—configurable computing. After configurable computing was defined, the devices at the heart of many configurable computing machines, FPGAs, were described, with special attention being given to the Xilinx FPGAs that were used for this implementation. An overview of the features of the WILDFORCE CCM was also presented with an emphasis on those features that are most crucial to the turbo decoder design. The design process for CCM applications was presented with an emphasis on designing for the WILDFORCE. This process requires the co-design of hardware configurations for the FPGAs and software to control the board from the host, which can create difficult debugging problems. Several alternate approaches to CCM design that were not taken in this project were also described. CCMs are best suited for designs where parallel hardware and reconfigurability can best be exploited, like image and signal processing.

Chapter 4 provided a detailed look at the design and implementation of the turbo decoder for the WILDFORCE platform. Goals and target parameters for the system were presented in the first two sections. The strategy for implementing the turbo decoder was also described. Several key elements of this strategy included the use of modularity to ease implementation and testing, the use of the external memory as an interface between modules to simplify coding and remove timing issues, and the use of a single PE to provide portability to other CCMs and room for integration with other modules. The three major modules used in the design were also described. The interleaver and adder modules used pre-fetching of data to reduce idle cycles that are caused by the two cycle delay on memory reads. The interleaver and decoder modules were designed to be flexible so that they can implement two slightly different processes without requiring hardware to be wasted on two distinct modules for these processes.

In Chapter 5, the results of the implementation were presented. The role of the host program in channel simulation and design verification for this project was described. Resource utilization and timing statistics for the implementation were given, and it was shown that the design can operate at the targeted rate of 32 kbps information throughput. Functional verification of the design was also presented by comparing the performance of the implemented decoder with the expected results from MATLAB simulations.

The contributions of this research include:

- An implementation of a turbo decoder that emphasizes flexibility and portability, making it ideal for use in a software radio.
- A technique for using off-chip memory to simplify the interface between modules that reside on a single FPGA.
- A technique for saving FPGA area by building flexible hardware modules that can be time-shared to perform the overall decoding operation.
- A technique for optimizing the state machines to overcome inefficiencies created by delays in performing memory reads.

## 6.2 Future Work

This section presents several enhancements to the design that could be made. While some of them are only necessary if the throughput requirements of the decoder increase,

others will be required when the decoder is ported to a stand-alone platform and integrated with other parts of the software radio.

### 6.2.1 Performance Enhancements

There are many ways that the performance of the decoder can be enhanced. Some of these are described in this section.

One of the more glaring details in the current implementation is the fact that it only uses a single PE on the WILDFORCE, while the rest of the FPGAs are unused. Although portability concerns were a driving factor in the decision to do this, if the implementation were going to be used on the WILDFORCE extensively, it would be good to take advantage of the additional configurable resources. Using these resources effectively can be a difficult challenge however. If the block size were fixed to a fairly small size, then it is possible that some of the modules could be made more efficient by buffering the data on the FPGA, however this reduces the design flexibility. The most effective way to use multiple PEs for the design is to pipeline the decoding process by iterations. The first PE could perform the first two iterations and then pass its data onto a neighboring PE for more iterations, while the first PE begins decoding a new block of data. The major difficulty with this approach is that because the PEs do not have access to a shared memory, the passing of data between them would have to be implemented by a module that reads the relevant data blocks from the first PE's memory and passes them to a corresponding module on the second PE that writes the data to memory. The time required to do these transfers reduces the performance gains of parallelizing the system in this manner, but the performance would still be increased.

Further tuning of the state machines, specifically for the decoder module, should be able to trim a few cycles out of the major decoding loop, which would offer some improvement. A more dramatic improvement could be achieved by integrating the adder and decoder modules to reduce the memory accesses. Although they are conceptually distinct, both modules use the same set of variables, and integration would not be difficult. This relatively simple change could offer a performance increase of about 15%.

As mentioned in Chapter 5, the available clock speed could also be increased by using more constraints with the Xilinx tools. Additionally, the insertion of registers to

employ pipelining in the slower decoder module would allow the overall clock rate to be increased, resulting in an increase in throughput.

Perhaps the easiest way to increase the performance of the design is to move to a platform that provides multiple memory ports. The memory-intensive nature of this implementation means that the ability to perform two memory operations per cycle instead of one would almost double the throughput of the design. Because some of the possible future target platforms for the decoder offer this feature, the implementation should be able to keep up with any increases in the performance of the other software radio modules to avoid becoming the system bottleneck.

Finally, a way to improve the decoding performance (in terms of accuracy, not throughput) would be to use Log-MAP decoding instead of the Max-Log-MAP that is implemented here. The additional lookup tables and extra addition operations this would require have the potential to slow the decoding process significantly, although this could be offset by some of the other performance improvements mentioned here. Careful use of registers for pipelining could also be used to control the reduction in throughput.

### 6.2.1 Other Enhancements

Beyond just increasing the throughput of the decoder, there are other enhancements that can (and in some cases, must) be made to the design if it is to be used as part of a larger software radio.

In order to make the decoder less dependent on a host processor to do data manipulation, a multiplexer unit would need to be implemented to receive incoming data and separate the information and parity bits according to whether or not puncturing is used. Additionally, some sort of handshaking protocol would need to be implemented in order to provide synchronization between the decoder and modules that are before and after it in the larger system. Also, the decoder module would have to perform the reliability scaling, preferably as the incoming data is being multiplexed, so it would need to have some access to a SNR estimate for the channel.

# References

[1] Shannon, C.E., “A mathematical theory and communication,” *Bell System Technical Journal*, vol. 27, pp. 379-423 and 623-656, 1948.

[2] Peterson, L.L. and Davies, B.S., *Computer Networks: A Systems Approach*, Morgan Kaufmann, San Francisco, 1996.

[3] Couch, L.W., *Digital and Analog Communication Systems*, Macmillan Publishing Co., New York, 1993.

[4] Berlekamp, E.R., “Nonbinary BCH decoding,” *IEEE Int. Symp. Inform. Theory*, San Remo, Italy, 1967.

[5] Meggitt, J.E., “Error correcting codes and their implementation,” *IRE Trans. Inform. Theory*, vol. 7, pp. 232-244, Oct. 1961.

[6] Massey, J.L., “Shift-register synthesis and BCH decoding,” *IEEE Trans. Inform. Theory*, vol.15, pp.122-127, January 1969.

[7] Wicker, S., *Error Control Systems for Digital Communications and Storage*, Prentice Hall, Englewood Cliffs, NJ, 1995.

[8] Lin, S., Kasami, T., Fujiwara, T., and Fossorier, M., *Trellises and trellis-based decoding algorithms for linear block codes*, Kluwer Academic Publishers, 1998.

[9] Valenti, M.C., “Iterative detection and decoding for wireless communications,” Doctoral dissertation, Va. Poly. Inst. & State University, Blacksburg, VA, July 1999.

[10] Viterbi, A.J., “Error bounds for convolutional codes and an asymptotically optimum decoding algorithm,” *IEEE Trans. Inform. Theory*, vol. 13, pp. 260-269, April 1967.

[11] Forney, G.D., “The Viterbi Algorithm”, *Proceedings of the IEEE*, vol.61, no. 3, pp. 268-278, 1973.

[12] Lin, S., Costello, D.J., *Error Control Coding: Fundamentals and Applications*, Prentice-Hall, Englewood Cliffs, NJ, 1983.

[13] Blahut, R.E., *Theory and Practice of Error Control Coding*, Addison Wesley, Englewood Cliffs, NJ, 1980.

[14] Berrou, C. and Glavieux, A. “Near optimum error correcting coding and decoding: Turbo-codes,” *IEEE Trans. Commun.*, vol. 44, pp. 1261-1271, October 1996.

[15] Berrou, C., Glavieux, A., and Thitimajshima, P., “Near Shannon limit error-correcting coding and decoding: Turbo-codes (1),”, *Proc., IEEE Int’l. Conf. on Communications*, Geneva, Switzerland, pp. 1064-1070, 1993.

[16] Hall, E.K. and Wilson, S.G., “Stream-oriented turbo codes,” *Proc. IEEE Veh. Tech. Conf.*, Ottowa, CA, 1998.

[17] Forney, G.D., *Concatenated Codes*, MIT Press, Cambridge, MA, 1966.

[18] Bahl, L.R., Cocke, J., Jelinek, F., and Raviv, J., “Optimal decoding of linear codes for minimizing symbol error rate,” *IEEE Trans. Inform. Theory*, vol.20, pp.248-287, March 1974.

[19] Chang, R.W., and Hancock, J.C., “On receiver structures for channels having memory,” *IEEE Trans. Inform. Theory*, vol. 12, pp. 463-468, Oct. 1966.

[20] Robertson, P. and Hoeher, P., “Optimal and sub-optimal maximum a posteriori algorithms suitable for turbo decoding”, *European Trans. on Telecommunications*, vol. 8, pp. 119-125, March-April 1997.

[21] Hagenaer, H. and Hoeher, P., “A Viterbi algorithm with soft-decision outputs and its applications,” *Proc. of IEEE GLOBECOM ’89*, pp. 11-17, Nov. 1989.

[22] Viterbi, A.J., “An intuitive justification and a simplified implementation of the MAP decoder for convolutional codes,” *IEEE Journal Sel. Areas in Comm.*, vol. 16, pp.260-264, February, 1998.

[23] Bittner, R., Athanas, P. and Musgrove, M., "Colt: An experiment in wormhole run-time reconfiguration," *SPIE Photonics East ‘96*, Boston, MA, November 1996.

[24] Hudson, R.D., Lehn, D.I., and Athanas, P.M., "A run-time reconfigurable engine for image interpolation," *Proc. IEEE Symp. on FPGAs for Custom Comp. Machines (FCCM ’98)*, Napa, CA, April 1998.

[25] Xilinx, *The Programmable Logic Data Book*, 1998.

[26] Xilinx website, *http://www.xilinx.com*

[27] Scalera, S.M. and Vazquez, J.R., "The design and implementation of a context switching FPGA," *Proc. IEEE Symp. on FPGAs for Custom Comp. Machines (FCCM '98)*, Napa, CA, April 1998.

[28] Annapolis Micro Systems website, *http://www.annapmicro.com*

[29] Viewlogic website, *http://www.viewlogic.com*

[30] "FPGA Express", Datasheet, Synopsys, July 1998.

[31] Bellows, P. and Hutchings, B., "JHDL – an HDL for reconfigurable systems," *Proc. IEEE Symp. on FPGAs for Custom Comp. Machines (FCCM '98)*, Napa, CA, April 1998.

[32] Hudson, R.D., Lehn, D.I., Hess, J.H., Atwell, J.W., Moye, C.D., Shiring, K.J., and Athanas, P.M., "Spatio-temporal partitioning of computational structures onto configurable computing machines," *SPIE Proceedings*, Vol. 3526, pp. 62-71, November 1998.

[33] Guccione, S.A. and Levi, D., "XBI: A Java-based interface to FPGA hardware," *SPIE Proceedings*, Vol. 3526, pp. 97-102, November 1998.

[34] Abbott, L., Athanas,P., Chen, L. and Elliott, R.L., "Finding lines and building pyramids with Splash-2," *Second FPGAs for Custom Computing Machines Workshop*, Napa, California, pp. 155-163, April 1994.

[35] Kim, H.J., "Data-specific number factoring on context switching configurable computers," UCLA Electrical Engineering Department, Technical Paper #98-2, 1998.

[36] Rencher, M. and Hutchings B. L., "Automated target recognition on SPLASH 2", *Proc. IEEE Symp. on FPGAs for Custom Comp. Machines (FCCM '97)*, Napa, CA, April 1997.

[37] Hess, J.R., Lee, D.C., Harper, S.J., Jones, M.T., and Athanas, P.M., "Implementation and evaluation of a prototype reconfigurable router," *Proc. IEEE Symp. on FPGAs for Custom Comp. Machines (FCCM '99)*, Napa, CA, April 1999.

# Vita

Jason Hess was born in Newport News, VA on June 29, 1975. After spending the whole of his formative years in that shoestring city near the coast (which, despite the inquiries of the uninitiated, is most certainly not a newspaper), he enrolled in Virginia Tech's engineering program in the fall of 1993. While there, he performed undergraduate research in graphics applications for FPGAs. After obtaining a BS in electrical engineering in 1997, he interned for a summer with Annapolis Micro Systems where he designed applications for configurable computing machines. In the fall of 1997, he returned to Virginia Tech for graduate work as a Bradley Fellow. His research focused on implementing communications hardware in configurable hardware. In October, he will join Cisco Systems in Austin, TX as a hardware engineer.