

TURBO CODING IMPLEMENTED IN A FINE GRAINED
PROGRAMABLE GATE ARRAY ARCHITECTURE

A Dissertation

Submitted to

the Temple University Graduate Board

in Partial Fulfillment

of the Requirements for the Degree

DOCTOR OF PHILOSOPHY

by

Robert A. Esposito

January 2009

ABSTRACT

TURBO CODING IMPLEMENTED IN A FINE GRAINED PROGRAMMABLE GATE ARRAY ARCHITECTURE

Robert A. Esposito
Temple University, 2009
Doctor of Philosophy
Advisor: Dr. Dennis Silage

One recent method to approach the capacity of a channel is Turbo Coding. However, a major concern with the implementation of a Turbo Code is the overall complexity and real-time throughput of the digital hardware system.

The salient design problem of Turbo Coding is the iterative decoder, which must perform calculations over all possible states of the trellis. Complex computations such as exponentiations, logarithms and division are explored as part of this research to compare the complexity of the traditionally avoided maximum a-posteriori probability (MAP) decoder to that of the more widely accepted and simplified Logarithm based MAP decoder (LOG-MAP).

This research considers the fine grained implementation and processing of MAP, LOG-MAP and a hybrid LOG-MAP-Log Likelihood Ratio (LLR) based Turbo Codes on a Xilinx Virtex 4 PGA. Verification of the Turbo Coding system performance is demonstrated on a Xilinx Virtex 4 ML402SX evaluation board with the EDA of the Xilinx System Generator utilizing hardware co-simulation. System throughput and bit error rate (BER) are the performance metrics that are evaluated as part of this research. An efficient system throughput is predicated by the parallel design of the decoder and BER is determined by data frame size, data word length and the number of decoding

iterations. Furthermore, traditional and innovative stopping rules are evaluated as part of this research to facilitate the number of iterations required during decoding.

ACKNOWLEDGEMENTS

I would like to thank the following people who have aided and supported me through the years. First I would like to thank Dr. Dennis Silage for his guidance through graduate school and this dissertation. He has been a great teacher and friend to me over the past four years. I sincerely thank him for everything he has taught me inside and outside of the classroom. I would also like to thank Dr. Saroj Biswas, Dr. Musoke Sendaula, Dr. Li Bai, Dr. Iyad Obeid and Dr. Bijan Mobasseri for their encouragement and guidance throughout my tenure at Temple University ECE. Last but not least I would like to thank my wife Elizabeth for her support and understanding. She was always there to help motivate me to do my absolute best.

TABLE OF CONTENTS

ABSTRACT.....	ii
ACKNOWLEDGMENTS.....	iii
LIST OF TABLES.....	vi
LIST OF FIGURES.....	vii
CHAPTER	
1. INTRODUCTION	
1.1 Turbo Codes.....	1
1.2 Programable Gate Array.....	3
1.3 Traditional Hardware Based Turbo Codes.....	4
1.4 Problem Statement and Solution.....	5
1.5 Outline of Research.....	7
2. TURBO CODES	
2.1 Shannon's Limit.....	9
2.2 Block Codes.....	10
2.3 Convolutional Codes.....	12
2.4 Viterbi Decoding.....	14
2.5 Interleaving.....	14
2.6 Code Rate and Puncturing.....	17
2.7 Phase Shift Keying.....	18
2.8 Channel Modeling.....	20
2.9 Turbo Coding.....	21
2.9.1 Parallel Concatenated Convolutional Codes.....	22
2.9.2 Serial Concatenated Convolutional Codes.....	23
2.10 Decoding Algorithms.....	24
2.10.1 Maximum a Posteriori Probability.....	24
2.10.2 Max-Log-MAP/Log-MAP.....	26
2.10.3 Soft Output Viterbi Algorithm.....	28
2.10.4 Comparison of Process.....	29
2.11 PCCC MAP Decoding.....	30
2.12 BER for PCCC.....	32
3. PROGRAMMABLE GATE ARRAY ARCHITECTURE	
3.1 Programmable Gate Array.....	34
3.1.1 Xilinx Virtex I FPGA.....	35
3.1.2 Xilinx Virtex II FPGA.....	38
3.1.3 Xilinx Virtex 4 FPGA.....	40
3.1.4 Xilinx Virtex 5 FPGA.....	42

4. SOFTWARE AND HARDWARE PLATFORM	
4.1 Software Platform.....	44
4.1.1 Hardware Description Language.....	44
4.1.2 Matlab/Simulink.....	45
4.1.3 Xilinx Integrated Synthesis Environment.....	47
4.1.4 Xilinx ChipScope.....	53
4.1.5 Xilinx System Generator.....	54
4.2 Hardware Module.....	57
4.2.1 Xilinx ML402SX Board.....	57
5. HARDWARE IMPLEMENTATION	
5.1 Parallel Concatenated Convolutional Coder.....	60
5.1.1 Recursive Convolutional Coder.....	63
5.1.2 Random Interleaver.....	64
5.2 Gaussian Channel.....	68
5.3 MAP Decoder.....	69
5.3.1 Transcendental Functions.....	70
5.3.1.1 Multiplicative/Additive Normalization.....	71
5.3.1.2 CORDIC Algorithm.....	73
5.3.1.3 Polynomial Approximation.....	73
5.3.1.4 Division.....	74
5.3.2 Distance Metric.....	75
5.3.2.1 Exponentiation by Indirect Look-Up Table.....	77
5.3.3 Forward Recursion.....	79
5.3.4 Backward Recursion.....	81
5.3.5 Recursive Normalization.....	83
5.3.6 Log-Likelihood Ratio.....	84
5.3.6.1 Division by Reciprocation.....	87
5.3.6.2 Logarithm by Taylor Series.....	91
5.3.7 MAP Performance Evaluation.....	95
5.4 PCCC Decoder.....	98
5.4.1 Extrinsic Information.....	98
5.4.2 Stopping Rules.....	100
5.4.3 PCCC Performance Evaluation.....	109
5.4.3.1 PCCC Hardware versus Matlab.....	110
5.4.3.2 PCCC Hardware versus C-Language.....	115
5.5 FPGA Synthesis Results.....	121
6. CONCLUSION AND FUTURE WORK	
6.1 Conclusion.....	127
6.2 Future Work.....	130
REFERENCES.....	131
APPENDIX: CONFERENCE PUBLICATIONS.....	135

LIST OF TABLES

Table 2.1	Repetition Code Example.....	9
Table 2.2	Decoder Complexity Comparisons.....	29
Table 5.1	Hardware vs. C-Language Program Throughput.....	120
Table 5.2	PCCC Encoder Synthesis Results.....	122
Table 5.3	PCCC Interleaver Synthesis Results.....	122
Table 5.4	MAP Decoder Synthesis Results.....	123
Table 5.5	LOG-MAP-LLR Decoder Synthesis Results.....	124
Table 5.6	Stopping Rules Synthesis Results.....	124
Table 5.7	PCCC 1 st Configuration Synthesis Results.....	125
Table 5.8	PCCC 2 nd Configuration Synthesis Results.....	125
Table 5.9	PCCC 1 st Configuration Synthesis Results Varying States.....	126

LIST OF FIGURES

Figure 2.1	(n, k) Block Code Word Structure.....	10
Figure 2.2	Convolutional Encoder.....	12
Figure 2.3	Trellis for Convolutional Encoder.....	13
Figure 2.4	Block Interleaver.....	15
Figure 2.5	Convolutional Interleaver and Deinterleaver Structure.....	16
Figure 2.6	Random Interleaver.....	16
Figure 2.7	Constellation Plot of Bandpass BPSK.....	18
Figure 2.8	Correlation Receiver.....	19
Figure 2.9	Spectrum and Autocorrelation of White Noise.....	20
Figure 2.10	Overview of Turbo Code System.....	21
Figure 2.11	PCCC Encoder.....	22
Figure 2.12	RSC Encoder.....	22
Figure 2.13	SCCC Encoder.....	23
Figure 2.14	PCCC MAP Decoding.....	30
Figure 3.1	Virtex I FPGA Architecture.....	36
Figure 3.2	Virtex I FPGA 2-Slice CLB.....	37
Figure 3.3	Virtex I FPGA Local Routing	37
Figure 3.4	Virtex II FPGA Architecture.....	38
Figure 3.5	Virtex II FGPA CLB and Slice Configuration.....	39
Figure 3.6	Active Interconnect Technology.....	40
Figure 3.7	DSP48 Slice in the Xilinx Virtex 4 FPGA.....	41
Figure 3.8	BRAM System Model in the Xilinx Virtex 4 FPGA.....	42

Figure 3.9	CLB and Slice Configuration in the Virtex 5 FPGA.....	43
Figure 4.1	Matlab versus Simulink Simulation of a (BPSK) System.....	46
Figure 4.2	Xilinx ISE New Project Wizard.....	47
Figure 4.3	Xilinx ISE New Project.....	48
Figure 4.4	Xilinx ISE Clock Wizard.....	49
Figure 4.5	Xilinx ISE Simulation of a 4-Bit Counter.....	49
Figure 4.6	Xilinx ISE Synthesis Report.....	50
Figure 4.7	Xilinx ISE FPGA Pin Mapping.....	51
Figure 4.8	Post Synthesis Simulation Output.....	51
Figure 4.9	Xilinx ISE Boundary Scan.....	52
Figure 4.10	Xilinx ChipScope Logic Analyzer Waveform Output.....	53
Figure 4.11	Xilinx System Generator Library.....	55
Figure 4.12	Xilinx System Generator FIR Filter Example.....	56
Figure 4.13	Xilinx ML402SX Board.....	58
Figure 5.1	PCCC Rate 1/3 Encoder Functional Description.....	61
Figure 5.2	PCCC Rate 1/3 Encoder Xilinx SG Implementation.....	62
Figure 5.3	RSC 2-State Functional Description.....	63
Figure 5.4	RSC 2-State Xilinx SG Implementation	63
Figure 5.5	LFSR Functional Description.....	64
Figure 5.6	LFSR Xilinx SG Implementation	65
Figure 5.7	Random Interleaver Xilinx SG Implementation	66
Figure 5.8	Control Logic Functional Description.....	67
Figure 5.9	Control Logic Xilinx SG Implementation	68

Figure 5.10	MAP Decoder Functional Description (rate 1/3 encoder).....	70
Figure 5.11	Traditional Iteration versus Cascaded Iteration	71
Figure 5.12	Euclidian Distance Functional Description.....	76
Figure 5.13	Euclidian Distance for (-1, -1) Xilinx SG Implementation.....	76
Figure 5.14	Exponentiation Functional Description.....	77
Figure 5.15	Exponentiation for (-1, -1) Xilinx SG Implementation.....	78
Figure 5.16	Forward Recursion Functional Description.....	80
Figure 5.17	Forward Recursion Xilinx SG Implementation.....	80
Figure 5.18	Recursive Normalization Functional Description.....	83
Figure 5.19	Log-Likelihood Functional Description.....	85
Figure 5.20	Log-Likelihood Xilinx SG Implementation.....	86
Figure 5.21	Division by Reciprocation Functional Description.....	88
Figure 5.22	Division by Reciprocation Xilinx SG Implementation.....	89
Figure 5.23	Division by Reciprocation Xilinx SG (Enlarged).....	90
Figure 5.24	Logarithm by Taylor Series Functional Description.....	91
Figure 5.25	Logarithm by Taylor Series Xilinx SG Implementation.....	93
Figure 5.26	MAP BER 18 bit words.....	96
Figure 5.27	MAP BER 32 bit words.....	97
Figure 5.28	Extrinsic Information Functional Description.....	99
Figure 5.29	Extrinsic Information Xilinx SG Implementation.....	99
Figure 5.30	H1 Stopping Rule Functional Description.....	102
Figure 5.31	H1 Stopping Rule Xilinx SG Implementation.....	102
Figure 5.32	LCT Stopping Rule Functional Description.....	104

Figure 5.33	LCT Stopping Rule Xilinx SG Implementation.....	104
Figure 5.34	Average Number of Iterations Magic Genie.....	105
Figure 5.35	Average Number of Iterations H1.....	106
Figure 5.36	Average Number of Iterations LCT 128 Frame.....	107
Figure 5.37	Average Number of Iterations LCT 1024 Frame.....	107
Figure 5.38	Average Number of Iterations LCT 4096 Frame.....	108
Figure 5.39	BER 32 bit Quantization 128 Frame.....	111
Figure 5.40	BER 32 bit Quantization 512 Frame.....	112
Figure 5.41	BER 32 bit Quantization 8192 Frame.....	112
Figure 5.42	BER 18 bit Quantization 128 Frame.....	113
Figure 5.43	BER 18 bit Quantization 512 Frame.....	114
Figure 5.44	BER 18 bit Quantization 8192 Frame.....	114
Figure 5.45	PCCC Decoder Single MAP Functional Description	117
Figure 5.46	PCCC Decoder Second Functional Description.....	117
Figure 5.47	PCCC Decoder Latency.....	118
Figure 5.48	PCCC Decoder Throughput Single MAP vs Dual MAP.....	119

CHAPTER 1

INTRODUCTION

1.1 Turbo Codes

Digital communications is predicated by the transmission and reception of binary data. These digital data transmissions require a system with multiple signal processing capabilities. Digital communication systems utilize channel coding where parity bits are added to the data before transmission. The receiver utilizes this transmitted parity coding to locate and correct bit errors due to channel noise and non-linearity. Bit error rate (BER) is the defined metric for digital data transmission and is measured as the ratio of the number of bits received in error to the total number of bits received. Many channel coding techniques have been proven to dramatically decrease the apparent BER in digital data transmissions. However, channel coding decreases digital communication system throughput due to the added transmission packet and the added complexity in the transmitter and receiver. In many instances digital communication channels have low signal to noise ratio (SNR) and reliable data transmission can only be attained with the addition coding.

Bandwidth utilization efficiency is another important concern in digital communication systems. In 1948 Shannon determined the theoretical maximum data rate that a digital communication system can obtain without receiving any bits in error [31]. Channel coding techniques are the practical implementation which facilitates the approach toward this theoretical limit. Before the inception of Turbo Codes in 1993 [6], channel coding techniques utilized so-called hard symbol metrics to correct errors. The advanced error correcting capabilities of Turbo Codes rely on the use of probabilistic or

soft symbol metrics to provide even greater error correction capabilities. Turbo Codes are discussed further in Chapter 1.2.

In 1993 Berrou proposed Turbo Coding which is an iterative decoding process that utilizes probabilistic or soft estimates of the binary received data [6]. It has been shown that Turbo Codes can provide an apparently low BER in digital communication channels with low SNR. The Turbo Code encoder utilizes two convolutional encoders which introduce redundancy to the transmission. The real efficacy, however, of Turbo Codes is apparent in the decoder. The Turbo Code decoder utilizes two identical soft decision modules that work in concert to produce and communicate soft estimates of the transmitted data. Each of the iterations of the Turbo Decoder produces a better estimate than that of the previous. After a nominal number of iterations the decoders are interrupted and produce a hard estimate of the transmitted data.

There are three parameters that ultimately determine the performance of Turbo Codes in the correction of data errors. The first parameter is the data frame size. Turbo Codes transmit data in a frame of a specified length. Larger length frames have been proven to decrease the BER [36]. The second parameter is the number of iterations performed in the decoder, where additional iterations to a point, can provide a decrease in BER. However, an increased number of iterations significantly decreases digital communication system throughput. This occurs because the Turbo Code decoder can only process one data frame at a time and its iterative nature is the ultimate constraint of the performance. The third parameter is the inherent structure of the Turbo Code convolutional encoder. It has been shown that a larger encoder structure (more states) provides significantly more error correction capabilities [16].

1.2 Programmable Gate Array

The programmable gate array (PGA) is a configurable parallel processing integrated circuit. Recent PGAs are configured with thousands of configurable logic blocks (CLB). The CLB is an intrinsic module that can be used to implement combinational and sequential logic. Prior implementations of the PGA inefficiently utilized the complex CLB as multipliers and memory devices. Since many digital signal processing (DSP) designs required multipliers and memory, recent PGAs have included embedded multipliers and block random access memory (BRAM) as their fine grained architecture.

The reconfigurable nature of the PGA is exploited in the hardware design process. Even if a hardware design will ultimately be implemented on an applications specific integrated circuit (ASIC), design and verification of the architecture is often performed on the PGA. Complex DSP designs are typically described in a behavioral hardware description language (HDL). This behavioral description is then translated into hardware architecture by a synthesizer. Hardware synthesizers translate behavioral HDL code into synchronous architecture of the PGA with logic flip-flops, gates, multipliers, adders and BRAM. Furthermore, the hardware synthesizer places and routes signals to the external package pins of the PGA if input and output communication is required.

Hardware platforms which include a PGA, input/output pins and user interaction devices such as push buttons, slide switches and a liquid crystal display are typically used to facilitate the design process. Furthermore, so-called on-chip logic analyzers are also implemented to verify the internal digital data without routing signals to output pins which can result in timing distortions.

1.3 Traditional Hardware Based Turbo Codes

Due to the mathematical complexity of Turbo Codes and the lack of fine grained PGA architectures, simpler logarithm transformations and estimations have been largely relied upon. However, these logarithm based architectures do not utilize the full error correcting capabilities of Turbo Codes.

In particular, Turbo Codes or the parallel concatenated convolutional code (PCCC) is a digital communication channel coding technique which provides excellent BER performance at the cost of increased hardware architecture complexity. PCCC decoding is an iterative process that utilizes various complex soft decision metrics to properly correct data transmission errors [38].

PCCC decoding is based on the maximum a-posteriori probability (MAP) of each bit in the data frame. The MAP calculation comprises complex computations such as exponentiations and logarithms which are expensive to implement in hardware [21, 22, 29]. Traditionally, when implementing the MAP algorithm in hardware, these computations are converted to the LOG domain (LOG-MAP), wherein expensive hardware multiplications are converted to less expensive additions [29, 30, 34, 37]. Also, by converting to the LOG domain, under flowing data that is an inherent problem in the MAP decoder is remedied. Typically, however, the logarithms are approximated by utilizing a maximum operation and a Jacobian correction function read from a small look-up-table (LUT) [38]. This LUT estimation, however, results in an unwanted decrease in error correction performance.

PCCC decoders are also traditionally implemented in hardware by two identical MAP hardware modules [34]. Traditional Turbo Decoders utilizes dual identical

hardware MAP decoders because they operate on different bit symbols. This traditional dual MAP implementation is, however, wasteful in hardware. Since the MAP hardware modules are identical, roughly twice the required amount of hardware resources for Turbo Decoding is traditionally utilized. This redundancy, however, is not exploited in traditional Turbo Decoder hardware implementations.

1.4 Problem Statement and Solution

This research contributes to four distinct areas of Turbo Codes: 1) An innovative hardware implementation of a MAP decoder based Turbo Decoder that is not based in the LOG domain; 2) An innovative hardware implementation of a hybrid Turbo Decoder that is based partially in the LOG domain; 3) An innovative hardware implementation of a single MAP Turbo Decoder that decreases hardware utilization; and 4) An innovative soft stopping rule based on LOG convergence and a bits per frame threshold. A summary of these four innovative contributions are detailed below.

1) MAP Based Turbo Decoder

With the increase in PGA technology and the abundance of embedded multipliers in these fine grained architectures, hardware utilization becomes less of a concern [13]. Thus, one aspect of this research is the synthesized hardware design, implementation and verification of a MAP based Turbo decoder on a fine grained architecture. Since the Turbo decoder is based on the MAP algorithm, the hardware implementation of more complex computations such as exponentiations, logarithms and underflow are explored and compared structurally to the simplified LOG-MAP decoder [14].

Design of the PCCC decoder includes exploiting the parallel processing capabilities of the Xilinx Virtex 4 field programmable gate array (FPGA) to perform calculations simultaneously whenever possible [13, 14]. Transcendental functions, such as exponentiation and the logarithm that are required in the MAP decoder, are typically performed as iterative calculations. These functions are unwrapped and implemented as cascaded or pipelined architectures in order to facilitate system throughput. The overall goal is to maintain a free flowing pipeline by utilizing the fine grained hardware resources of the FPGA.

2) Hybrid log-domain MAP Decoder

In the MAP decoder, various computations benefit from reduced hardware complexity by converting to the log domain. However, the computations in the log domain are typically estimated which leads to decreased BER performance. Thus, it is desirable to have the reduced hardware complexity of the log domain MAP decoder and the accuracy of the traditional MAP decoder.

In the designed hybrid system of this research, the feasibility of calculating (not approximating) the LLR in the LOG-MAP decoder is explored. This implementation is considered a hybrid of the traditional MAP/LOG-MAP decoder because most of the MAP computations are performed in the log domain with the exception of the LLR, and is herein referred to as the LOG-MAP-LLR.

3) Single MAP decoder based Turbo Decoder

A single MAP Turbo decoder takes advantage of the redundant hardware of the traditional dual MAP Turbo decoder [15]. With proper buffering and multiplexing [13], the single MAP Turbo decoder is able to adequately perform Turbo decoding. By only

utilizing a single MAP decoder, the hardware utilization and power consumption of the device is therefore reduced by roughly half [15]. The decrease in hardware utilization and power consumption, however, is at the expense of decreased throughput [15].

4) Soft stopping rule based on LOG convergence

Another aspect of the Turbo decoder is various stopping rule calculators that control the decoder to stop iterating. Traditional proven (hard/soft decision) calculators, as well as a new soft stopping rule referred to as the log convergence threshold (LCT) rule are explored and benchmarked in hardware. They are compared to one another based on hardware utilization and average number of iterations for a given energy per bit to noise ratio (E_b/N_o). The LCT rule sets a log convergence magnitude threshold as well as a number of bits per frame threshold. When the set number of bits per frame surpasses the log convergence magnitude threshold, then the Turbo Decoder stops iterating. It is shown that the LCT rule requires minimal hardware resources.

1.5 Outline of Research

The remaining chapters consist of expository material and are organized as follows:

Background Chapters:

Chapter 2 provides a literature review of digital communications and Turbo codes. Chapter 3 provides a synopsis of the development of the fine grained architecture of the PGA and the Xilinx Virtex 4 FPGA device. Chapter 4 describes the Xilinx electronic design automation (EDA) software and the hardware platforms utilized in this research.

Research Analysis and Results Chapters:

Chapter 5 discusses the design, implementation, verification and benchmarking of the innovative MAP, traditional LOG-MAP and innovative LOG-MAP-LLR based Turbo decoder. Chapter 5 also discusses the hardware implementation of the innovative single MAP decoder based Turbo Decoder and innovative LCT stopping rule as well as traditional stopping rules. Specifically, throughout this research, all of the hardware designs are implemented on the Xilinx Virtex 4 FPGA. Chapter 6 discusses conclusions of this research and possible future work.

CHAPTER 2

TURBO CODES

2.1 Shannon's Limit

A digital communication system must reliably transmit data through a non-ideal, band-limited and noisy channel. The limiting factors of the system are the inherent transmitting power and available bandwidth. Reliability of the system is quantitatively measured by the BER. BER is the ratio of the number of bits received in error to the total number of bits transmitted. The goal of any digital communication system is to reduce the BER to as low a value as possible. In 1948 Shannon hypothesized the maximum rate at which data could be transmitted through a noisy communication channel without error and this became known as the information capacity theorem [31]. If B denotes the channel bandwidth in Hertz and SNR denotes the received signal to noise ratio, then the information capacity C expressed in bits per second (b/sec) is:

$$C = B \log_2(1 + SNR) \text{ b/sec} \quad (2.1)$$

Shannon also proposed the concept of improving the BER by including redundancy in the transmitted data bits. A simple example of redundancy is the repetition code. Table 2.1 is an example of binary data being transmitted with a length five repetition code [19]:

Table 2.1: Repetition Code Example

Binary Data	0	1	0	1
Repetition Code	00000	11111	00000	11111

By utilizing this redundant information the receiver has five instances to correctly estimate the transmitted data. Simple redundancy improves the BER and spurred even more advanced coding techniques. These techniques must employ a code generation that

is known to both the transmitter and receiver and some are explored in the following Chapters.

2.2 Block Codes

Linear block codes are the basis for even more advanced coding techniques [19, 20]. A code is linear if the modulo-2 sum of any two code words can produce a third code word. A (n, k) block code has a structure in which k information bits, otherwise known as systematic bits, of the n code bits are identical to the message sequence. The remaining $(n-k)$ bits are called the parity check bits. The overall structure of an (n, k) block code is shown in Figure 2.1:

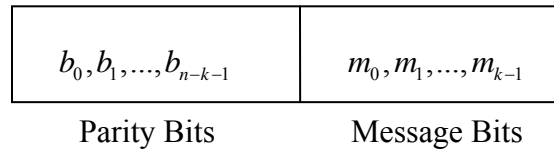


Figure 2.1: (n, k) Block Code Word Structure

For the linear block code in Figure 2.1 the following vectors can be defined:

$$\begin{aligned}
 \mathbf{m} &= [m_0, m_1, \dots, m_{k-1}] && \text{Message Vector} \\
 \mathbf{b} &= [b_0, b_1, \dots, b_{n-k-1}] && \text{Parity Vector} \\
 \mathbf{c} &= [c_0, c_1, \dots, c_{n-1}] && \text{Code Vector}
 \end{aligned} \tag{2.2}$$

The code vector is defined by the k by n generator matrix:

$$\mathbf{G} = [\mathbf{P} : \mathbf{I}_k] \tag{2.3}$$

\mathbf{P} is a user defined k by $(n-k)$ coefficient matrix that consists of linearly independent rows, and \mathbf{I}_k is a k by k identity matrix. The complete set of code vectors corresponding to all possible message vectors is generated by:

$$\mathbf{c} = \mathbf{mG} \quad (2.4)$$

At the receiver the relationship between the message vector and parity vector is established by the parity check matrix defined as:

$$\mathbf{H} = [\mathbf{I}_{n-k} : \mathbf{P}^T] \quad (2.5)$$

The importance of the parity check matrix \mathbf{H} is its ability to decode and detect bit errors. If a valid code vector is multiplied by the parity check matrix a null vector should be produced [19].

In any digital data transmission over a noisy channel errors can occur. The received vector \mathbf{r} is the sum of the transmitted code vector and an error vector given by:

$$\mathbf{r} = \mathbf{c} + \mathbf{e} \quad (2.6)$$

The task of the receiver is to decode the code vector from the received vector. To do this the receiver first pre-computes a syndrome vector utilizing all possible error patterns:

$$\mathbf{s} = \mathbf{eH}^T \quad (2.7)$$

The syndrome \mathbf{s} allows the receiver to decode a specific bit error pattern in the received vector. When the vector \mathbf{r} is received the decoder computes the syndrome which indicates a distinct error pattern:

$$\mathbf{s} = \mathbf{rH}^T \quad (2.8)$$

Linear block codes are able to detect and correct either single or multiple bit errors in the received code vectors.

Block codes include metrics that are important when comparing the construction of the code vectors. The first metric is the Hamming distance $d(c_1, c_2)$ which is defined as the number of bit locations in which the respective elements of the code vectors

c_1 and c_2 differ. The Hamming weight $w(c)$ is the number of nonzero elements in a code vector. The Minimum distance d_{\min} is the smallest Hamming distance between any two code vectors. A linear block code can correct up to t errors if and only if [19]:

$$t \leq \frac{1}{2}(d_{\min} - 1) \quad (2.9)$$

2.3 Convolutional Codes

In the previous Chapter linear block codes are presented. However, there are circumstances in which data bits are processed to code vectors serially rather than in blocks. Convolutional codes are used here because they do not require buffers for data storage [3, 19]. Convolutional encoders are finite state machines that consist of shift registers and modulo-2 adders. The coding rate, which is the amount of information bits transmitted in a binary symbol is expressed as:

$$r = \frac{L}{n(L+M)} \quad (2.10)$$

where L is the number of bits in the message and M is the number of stages of the shift register. To illustrate the convolutional coding process, the coder structure in Figure 2.2 is used.

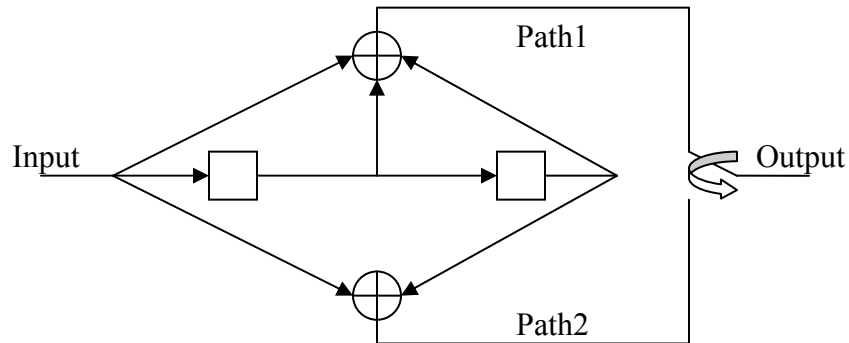


Figure 2.2: Convolutional Encoder

The generator polynomials $g^{(1)}$ and $g^{(2)}$ for the first and second paths are given by

Equation 2.11 where D is a single delay in the shift register:

$$\begin{aligned} g^{(1)}(D) &= 1 + D + D^2 \\ g^{(2)}(D) &= 1 + D^2 \end{aligned} \quad (2.11)$$

The operation of this encoder takes the message sequence (10011), which in polynomial form is $m(D) = 1 + D^3 + D^4$. The output of each of the two coded paths and the overall interleaved convolutional encoder output are given by:

$$\begin{aligned} c^{(1)}(D) &= g^{(1)}(D)m(D) = 1 + D + D^2 + D^3 + D^6 \\ c^{(2)}(D) &= g^{(2)}(D)m(D) = 1 + D^2 + D^3 + D^4 + D^5 + D^6 \\ c &= (11, 10, 11, 11, 01, 01, 11) \end{aligned} \quad (2.12)$$

The decoding procedure consists of first generating a structure specific trellis. This trellis is generated by shifting binary 0s and 1s through the encoder structure and recording the convolutional code output. The trellis is fully configured when all possible states and inputs have been determined [20]. The convolutional encoder in Equation 2.12 generated the trellis in Figure 2.3 was generated where the response to a binary 0 input is represented by a solid line, the response to a binary 1 input is represented by a dotted line and the current state is either a, b, c, or d.

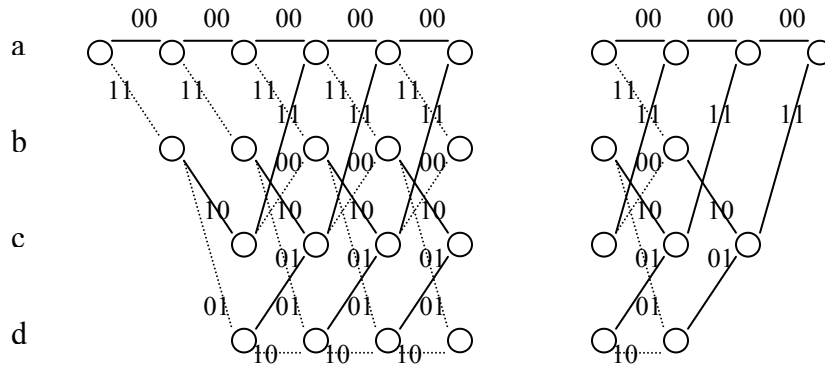


Figure 2.3: Trellis for Convolutional Encoder

2.4 Viterbi Decoding

Once the convolutional encoder trellis is completed a decoding algorithm must be devised. The basis for this decoding algorithm is the Maximum Likelihood Decoding (MLD) [20]. The MLD implies that an estimate of the code vector that minimizes the Hamming distance between the received vector and the transmitted code vector is used. Simply stated, the code vector that most closely matches the received vector is chosen. The Viterbi algorithm utilizes MLD to find the most probable path through the convolutional encoder trellis and therefore finds the best estimate of the transmitted code vector. This algorithm is implemented as follows:

- 1) Generate each stage of the trellis one at a time.
- 2) Each path entering a node is compared to the received sequence
 - a. The path with the smallest Hamming distance to the received code vector is retained.
 - b. The path with the larger Hamming distance to the received code vector is removed.
- 3) Once all surviving paths are set the transmitted code vector is determined by traversing the path with the minimum cumulative Hamming distance.

2.5 Interleaving

Interleaving is a method that rearranges the order of the data bits to accomplish a specific goal [20]. For example, Turbo Coding systems employ interleaving because error sequences in wireless digital communication systems tend to occur in bursts that can last as long as multiple symbol times:

- 1) Interleaving arranges the data in such a way that long burst errors do not destroy an entire transmitted code word.
- 2) Interleaving ensures that the outputs of the two convolutional encoders are uncorrelated, which is important in Turbo Coding.

The implementation of interleaving in Turbo Coding is discussed more in detail in Chapter 2.9.

Block interleaving is a technique that utilizes an indexed memory buffer. Data fills a column array and is then read out in row order. At the receiver end the inverse operation is performed to reproduce the original data. In Figure 2.4 interleaving any three of the 3-bit code words represented by A, B and C can be completely corrupted if there is a error burst that lasts three bit periods. However after block interleaving the error burst must have duration of seven bit periods to completely corrupt a given code word. This simple type of interleaving mitigates error bursts, however the structure of an interleaver produces delay.

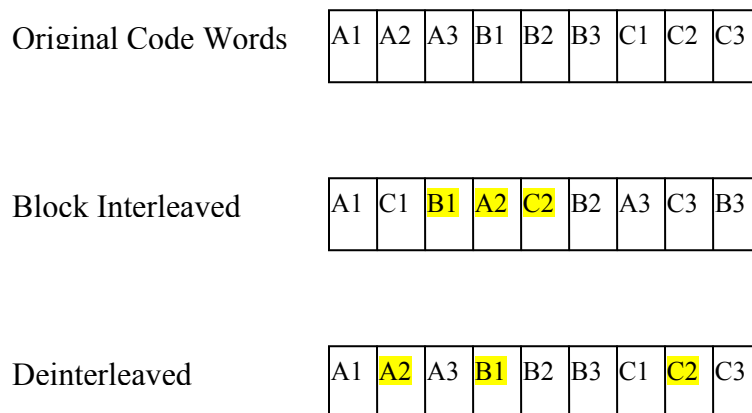


Figure 2.4: Block Interleaver

Another interleaving method utilizes a convolutional process [20]. Convolutional interleaving uses synchronized input and output digital commutators that allow data to be shifted through a bank of registers with a varying order. Both the convolutional interleaver and deinterleaver structures are shown in Figure 2.5:

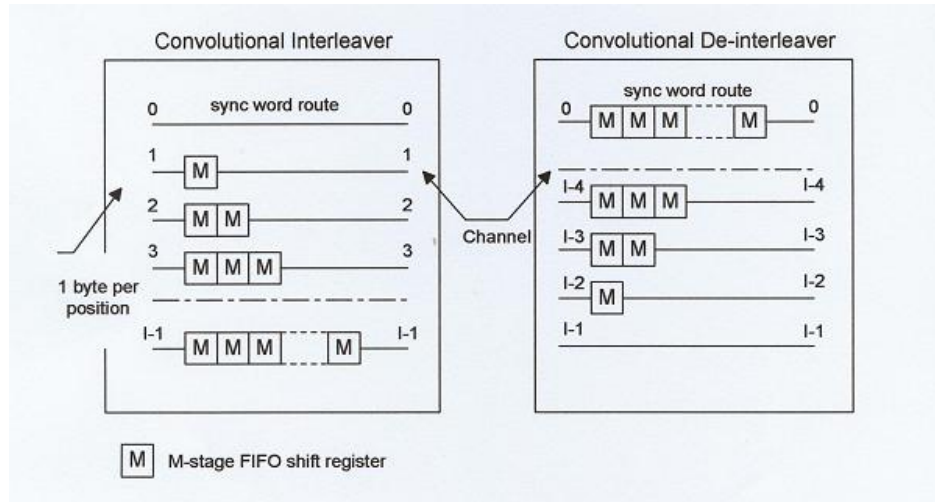


Figure 2.5: Convolutional Interleaver and Deinterleaver Structure

As for the block interleaver, more registers produce more delay in the output. In most cases a tradeoff between interleaver performance and speed has to be made.

Random interleaving is widely used in Turbo encoders and is the focus of this research. Bits are read in the order in which they are generated but they are read out of the interleaver in a pseudo random manner. The interleaver and the deinterleaver both share a digital pseudo random number generator that synchronizes the entire algorithm. Figure 2.6 shows an example of a random interleaver for a data set of length 5. The index values are shown above the data.

$$\begin{matrix} 0 & 1 & 2 & 3 & 4 \\ [0 & 1 & 1 & 0 & 0] \end{matrix} \Rightarrow \begin{matrix} 4 & 0 & 3 & 1 & 2 \\ [0 & 1 & 1 & 0 & 0] \end{matrix}$$

Figure 2.6: Random Interleaver

The pseudo random index numbers are easily generated in hardware by a linear feedback shift register which is discussed in more detail in Chapter 5.1.2. A concern with random interleaving in Turbo Coding is the size of the interleaver structure. Larger interleaver structures offer low correlation between data sets and therefore produce low bit error rates. However, this large interleaver structure also introduces large delays and thus limits the throughput of the system.

2.6 Code Rate and Puncturing

Code rate is the ratio of the number of information bits to the number of total code bits. For a convolutional code (n, k) the code rate would be $r = k / n$. In Chapter 2.2 a code vector consists of data bits concatenated with parity bits. This implies that the more parity bits that the block or convolutional encoder produces the lower is the code rate. Thus a choice must be made between more parity bits to improve the error correcting ability of the code or less parity bits to increase the data throughput.

Puncturing is a process in which certain parity bits of the code vector are omitted in a specified repeating pattern [20]. The reason for the parity bit omission is to increase the overall code rate of the system. However, puncturing increases the BER of the system. Puncturing provides an alternative in which the punctured coded data has a lower BER while gaining a larger throughput. This technique is utilized when data throughput is more important than BER performance [38].

2.7 Phase Shift Keying

Phase Shift Keying (PSK) is a form of angle modulation utilized in digital communication systems [19]. Bandpass binary PSK (BPSK) utilizes a sinusoidal carrier whose angle is modulated from 0 degrees representing data bit of 0 to 180 degrees representing a data bit of 1. In BPSK the maximal phase difference of 180 degrees separates the two symbols for detection in the presence of noise. This modulation process in what is known as a constellation plot is shown in Figure 2.7:

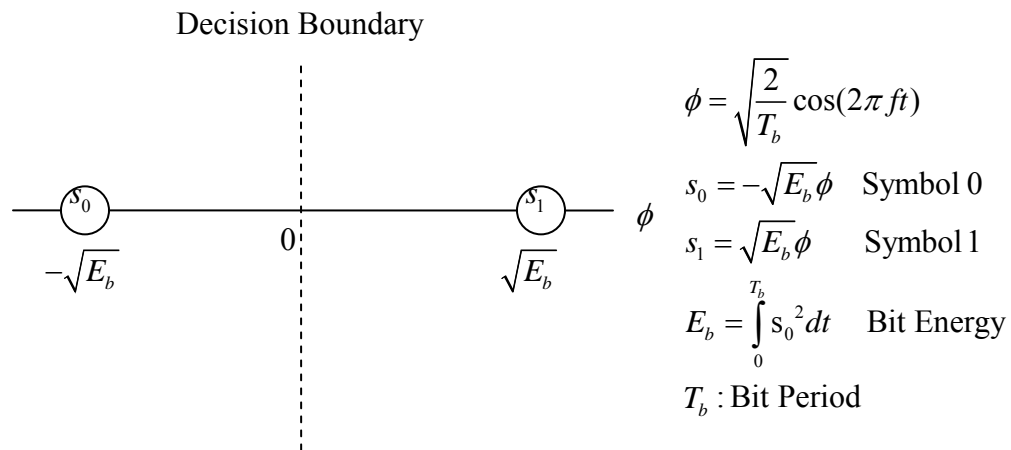


Figure 2.7: Constellation Plot of Bandpass BPSK

Notice that the symbol amplitudes are spaced according to the bit energy E_b and bit duration T_b . The binary decision threshold is located at the midpoint of zero. When the BPSK signal is transmitted through a noisy communication channel the received symbol may arbitrarily cross the decision boundary. In other words a binary 0 may be received as a binary 1 or visa versa. When this incorrect decision occurs it is called a single bit error. The receiver for the BPSK system calculates the correlation between the

received noisy signal and all possible transmitted signals. The correlation receiver for the BPSK system is shown in Figure 2.8.

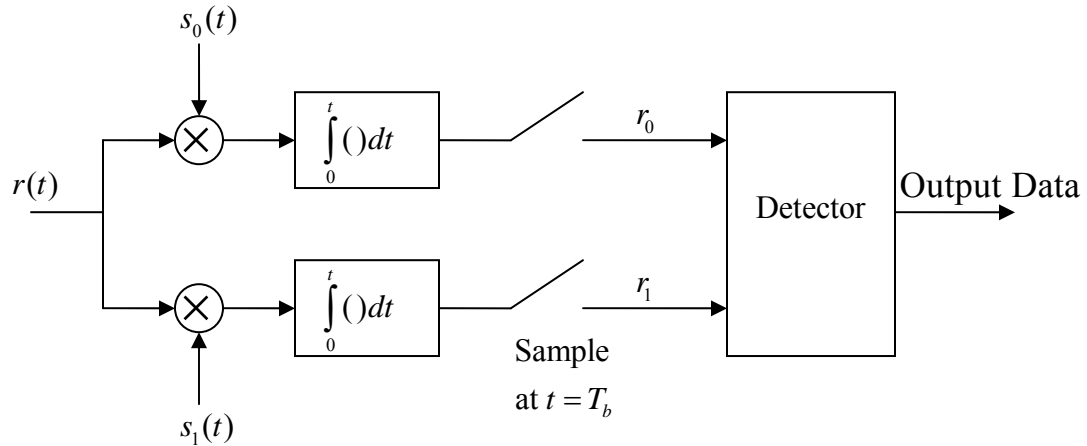


Figure 2.8: Correlation Receiver

The correlation receiver multiplies the received signal $r(t)$ with both possible transmitted symbols $s_0(t)$ and $s_1(t)$ and then integrates over a bit period for each received symbol. The detector compares both integrator outputs and selects the largest value of r_0 and r_1 .

A baseband technique that does not use carrier modulation and has an identical BER performance to that of BPSK is pulse amplitude modulation (PAM) [27]. PAM provides a simplified modulation technique where symbols are represented by a square pulse of a set amplitude and duration. The PAM correlation receiver can thus be simplified to a simple threshold detector. By normalizing the bit duration a symbol of binary 0 is represented by a negative value (-1) and symbol of binary 1 is represented by a positive value (+1) with the threshold of the receiver set at zero.

2.8 Channel Modeling

The usual channel interference that plagues communication systems is white noise [19]. This type of noise is random and has a Gaussian probability distribution with a mean value of zero and a distinct variance. Gaussian noise has a flat power spectral density (PSD) and is totally uncorrelated. The power spectrum and the autocorrelation of white noise are shown in Figure 2.9:

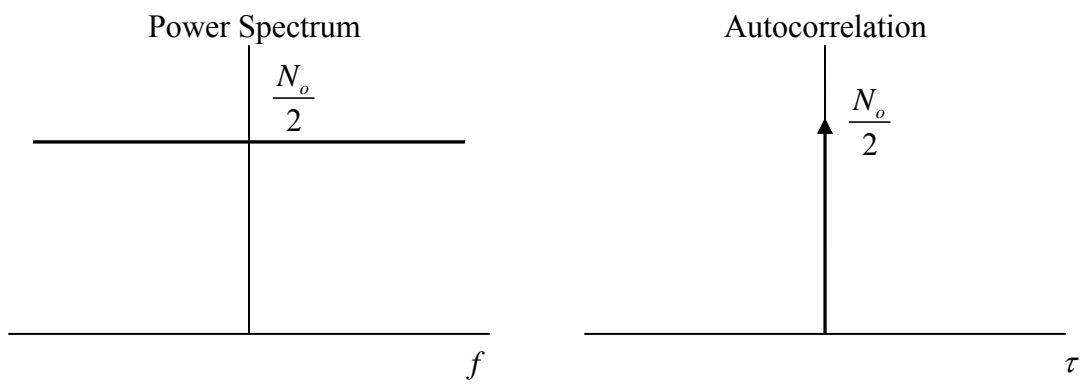


Figure 2.9: Spectrum and Autocorrelation of White Noise

This type of noise corrupts a signal in an additive fashion in the communication channel and is called Additive White Gaussian Noise (AWGN). In a mobile environment a communication system is also affected by signal fading. Fading occurs when a mobile receiver is in transit through an urban environment with no line of sight between transmitter and receiver and with multiple reflections of the transmitted signal that converge at the receiver. Fading in this environment can be modeled as a Raleigh distribution [19, 20]. The number of reflective paths and their average phase shift will determine the severity of the fading channel.

2.9 Turbo Coding

In 1993 Berrou [6] proposed the basis for Turbo Codes as implemented today. The original design concatenated two convolutional encoders with feedback known as recursive convolutional codes (RSC). To ensure that the data outputs of the two encoders are uncorrelated a random interleaver is used to rearrange the data input to a chosen encoder. The received block code is then processed by an iterative decoder that utilized deinterleavers and a probabilistic decision or a soft decision algorithm to estimate the received bits. Once either a specified number of iterations are performed or a stopping rule is evoked, the receiver makes a binary or hard decision on the data. This method of coding and decoding allows a digital communication system to approach Shannon's capacity limit [31]. The overall structure of Turbo Coding with a BPSK digital communication system is shown in Figure 2.10.

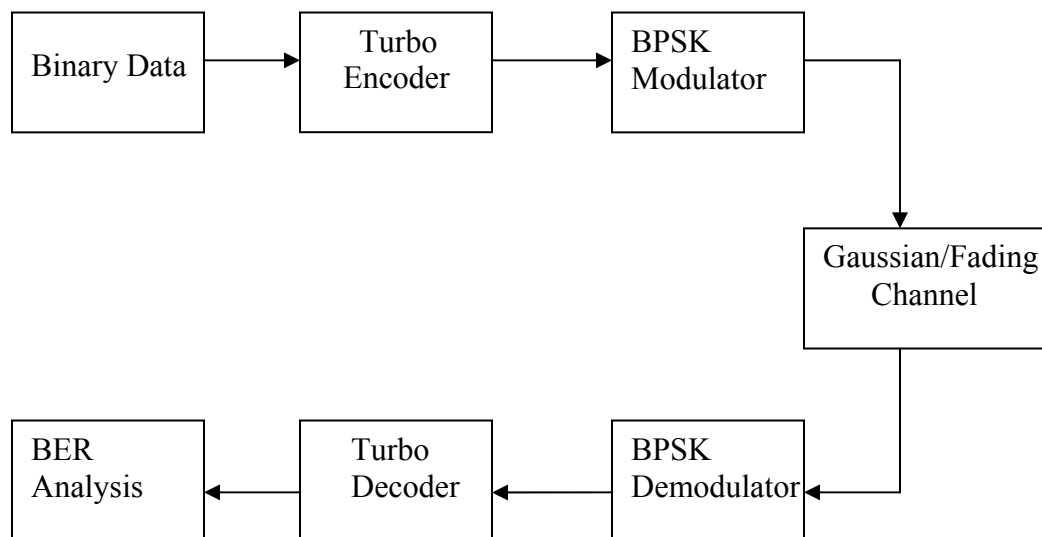


Figure 2.10: Overview of Turbo Code System

2.9.1 Parallel Concatenated Convolutional Codes

The PCCC of this research consist of two recursive convolutional (RSC) encoders [2, 11, 16, 18, 36, 45]. The data input to one of the RSC encoders is unaltered and the input to the other is randomly interleaved to ensure that the data stream is uncorrelated. The entire PCCC structure is shown in Figure 2.11:

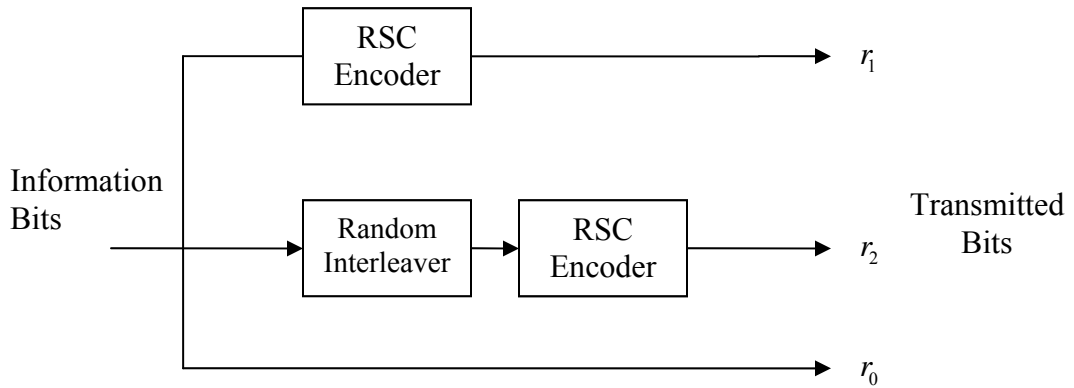


Figure 2.11: PCCC Encoder

The RSC is a convolutional encoder with feedback [38]. A simple RSC structure that could be used in the implementation of the PCCC is shown in Figure 2.12.

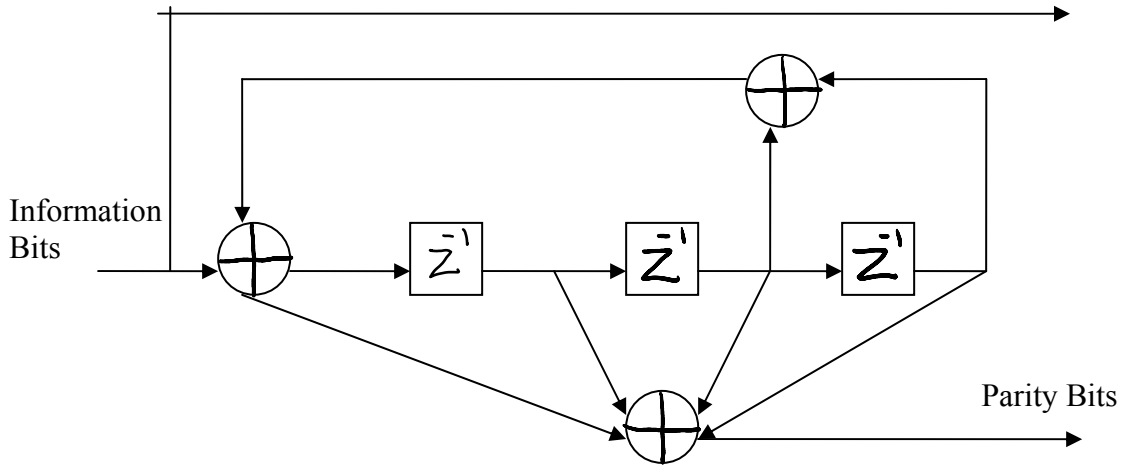


Figure 2.12: RSC Encoder

However, since the encoder has a recursive structure it has an infinite trellis. To implement this design the receiver must terminate the trellis relative to the size of the information block [16, 38]. Some of the advantages of using the PCCC in a Turbo Code structure is that the BER performance is exceptional for low values of signal to noise ratio (SNR) and its parallel nature allows for maximum throughput. Transmitting in a noisy environment does not require signal power significantly greater than the noise power.

2.9.2 Serial Concatenated Convolutional Codes

In the serial concatenated convolutional codes (SCCC) the two encoders are connected in series with an interleaver in between them [4, 33]. The information bits are passed through an outer convolutional encoder, interleaved and then passed through an inner convolutional encoder. The basic SCCC structure is shown in Figure 2.13.

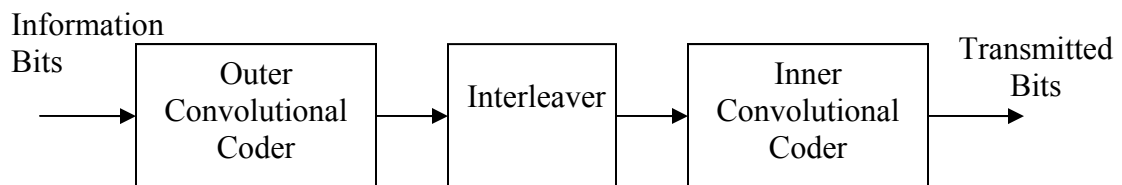


Figure 2.13: SCCC Encoder

Note that all of the transmitted bits have passed through both encoders unlike that in a PCCC. There is no direct transmission of the systematic bits or information bits as in PCCC. Here the outer convolutional encoder code words are interleaved rather than the information bits being interleaved as in PCCC. Both outer and inner convolutional encoders can be RSCs but due to the structure of the SCCC only the inner encoder is

required to be recursive [16, 38]. SCCC have different performance characteristics than the PCCC in which they obtain low BER at high SNR values but do not perform as well as the PCCC at lower SNR values [16, 38]. One crucial deficiency is the serial nature of the SCCC structure which limits the overall system throughput.

2.10 Decoding Algorithms

Turbo Codes are generated by recursive convolutional encoders (RSC) that can be represented by a trellis [5]. The decoders implemented here are also trellis based processes that are specifically designed to decode the RSC. MAP, Max-Log-MAP, Log-MAP and the Soft Output Viterbi Algorithm (SOVA) are presented here and their performance and complexity tradeoffs are discussed.

2.10.1 Maximum a-Posteriori Probability

The MAP process minimizes the bit error probability [1, 38]. For the received bit the MAP process generates a soft output in the form of an a-posteriori probability. The MAP process utilizes the log-likelihood ratio:

$$\Lambda(c_t) = \log \frac{P_r\{c_t = 1 | r\}}{P_r\{c_t = 0 | r\}}$$

(2.13)

r : received bit
 c_t : hard bit estimate

The MAP process uses this soft output ratio and compares it to a threshold value of nominally zero to compute the hard bit estimate:

$$c_t = \begin{cases} 1 & \text{if } \Lambda(c_t) \geq 0 \\ 0 & \text{otherwise} \end{cases} \quad (2.14)$$

Although the intent here is to compute the log-likelihood ratio and compare it to a threshold, it must be in a form that is solvable. It has been shown that the log-likelihood ratio can be written as [16, 38]:

$$\Lambda(c_t) = \log \frac{\sum_{(l',l) \in B_t^1} \alpha_{t-1}(l') \gamma_t^1(l',l) \beta_t(l)}{\sum_{(l',l) \in B_t^0} \alpha_{t-1}(l') \gamma_t^0(l',l) \beta_t(l)} \quad (2.15)$$

α is the joint probability of being in the present state l given the received bits from 1 to time t and is known as the forward estimator:

$$\alpha_t(l) = P_r \{S_t = l, r_t^t\} \quad (2.16)$$

β is the conditional probability of receiving the bits from time $t+1$ to τ given that the present state is l and is known as the backward estimator:

$$\beta_t(l) = P_r \{r_{t+1}^\tau | S_t = l\} \quad (2.17)$$

γ is the state transition probability and is denoted as:

$$\gamma_t^i(l',l) = P_r \{c_t = i, S_t = l, r_t^t | S_{t-1} = l'\} \quad (2.18)$$

The entire algorithm can be simplified and summarized as follows:

1) Forward Recursion:

- Initialize alpha $\alpha_0(0) = 1$ and $\alpha_0(l) = 0$ for $l \neq 0$
- For all branches in the trellis calculate γ as

$$\gamma_t^i(l',l) = p_t(i) \exp\left(\frac{-d^2(r_t, x_t)}{2\sigma^2}\right) \text{ for } i = 0,1 \quad (2.19)$$

where $p_t(i)$ is the apriori probability of each bit
and $d^2(r_t, x_t)$ is the Euclidean distance between r_t and x_t

- Initialize $\alpha_0(0) = 1$ and $\alpha_0(l) = 0$ for $l \neq 0$ and calculate $\alpha_t(l)$

for $t = 1, 2, \dots, \tau$ and $\ell = 0, 1, \dots, \text{LastState}$

$$\alpha_t(\ell) = \sum_{\ell'}^{\text{LastState}-1} \sum_{i=(0,1)} \alpha_{t-1}(\ell') \gamma_t^i(\ell', \ell) \quad (2.20)$$

2) Backward Recursion:

- Initialize $\beta_\tau(0) = 1$ and $\beta_\tau(l) = 0$ for $l \neq 0$ and calculate $\beta_t(l)$

for $t = \tau - 1, \dots, 1, 0$ and $\ell = 0, 1, \dots, \text{LastState}$

$$\beta_t(\ell) = \sum_{\ell'}^{\text{LastState}-1} \sum_{i=(0,1)} \beta_{t+1}(\ell') \gamma_{t+1}^i(\ell, \ell') \quad (2.21)$$

- Calculate $\Lambda(c_t)$

for $t = 1, 2, \dots, \tau - 1$

$$\Lambda(c_t) = \log \frac{\sum_{\ell=0}^{\text{LastState}-1} \alpha_{t-1}(\ell') \gamma_t^1(\ell', \ell) \beta_t(\ell)}{\sum_{\ell=0}^{\text{LastState}-1} \alpha_{t-1}(\ell') \gamma_t^0(\ell', \ell) \beta_t(\ell)} \quad (2.22)$$

2.10.2 Max-Log-MAP/Log-MAP

The traditional MAP process requires complex computations such as logarithms and exponentiations, and furthermore requires a large number of multiplications [16, 38].

By taking the logarithm of α , β , and γ , however, the MAP algorithm simplifies to:

$$\Lambda(c_t) = \log \frac{\sum_{l=0}^{Ms-1} e^{\alpha_{t-1}(l') \gamma_t^1(l', l) \beta_t(l)}}{\sum_{l=0}^{Ms-1} e^{\alpha_{t-1}(l') \gamma_t^0(l', l) \beta_t(l)}}$$

but.....

$$\log(e^{\delta_1} + e^{\delta_2} + \dots + e^{\delta_n}) \approx \max_{i \in \{1, 2, \dots, n\}} \delta_i$$

so.....

$$\Lambda(c_t) \approx \max_l \left[\bar{\gamma}_t^1(l', l) + \bar{\alpha}_{t-1}(l') + \bar{\beta}_t(l) \right] - \max_l \left[\bar{\gamma}_t^0(l', l) + \bar{\alpha}_{t-1}(l') + \bar{\beta}_t(l) \right]$$

where.....

$$\begin{aligned} \bar{\gamma}_t^i(l', l) &= \log \gamma_t^i(l', l) \\ \bar{\alpha}_t(l') &= \log \alpha_t(l') \\ \bar{\beta}_t(l) &= \log \beta_t(l) \end{aligned} \tag{2.23}$$

Approximating the log of the sum of exponentials as a maximum function in Equation 2.23 forces the performance of the Max-Log-MAP algorithm to be less than desirable [28, 32]. By utilizing the Jacobian the performance can be improved and is identical to the traditional MAP algorithm. The following equality can be applied when evaluating the recursive procedure from Equation [38]:

$$\begin{aligned} \log(e^{\delta_1} + e^{\delta_2}) &= \max(\delta_1, \delta_2) + \log(1 + e^{-|\delta_2 - \delta_1|}) \\ &= \max(\delta_1, \delta_2) + fc(|\delta_2 - \delta_1|) \\ \log(e^{\delta_1} + e^{\delta_2} + \dots + e^{\delta_n}) &= \log(\Delta + e^{\delta_n}) \\ \Delta = e^{\delta_1} + \dots + e^{\delta_{n-1}} &= e^{\delta} \\ &= \max(\delta, \delta_n) + fc(|\delta - \delta_n|) \\ \delta_n &= \bar{\gamma}_t^i(n', n) + \bar{\alpha}_{t-1}(n') + \bar{\beta}_t(n) \end{aligned} \tag{2.24}$$

However, this correction function is a complex recursive computation that converges to the proper correcting term. This recursive computation further decreases the throughput of the system and increases hardware utilization, and is therefore traditionally estimated by a small look up table (LUT). Thus, by estimating this correction function, the full BER potential of the decoder is not realized (not identical to MAP).

2.10.3 Soft Output Viterbi Algorithm

The soft output Viterbi algorithm (SOVA) was designed from the hard decision Viterbi algorithm that was presented in Chapter 2.4. Similar to the MAP algorithm the SOVA estimates transmitted symbols via the log-likelihood function:

$$\Lambda(c_t) = \log \frac{P_r\{c_t = 1 | r_1^\tau\}}{P_r\{c_t = 0 | r_1^\tau\}}$$

(2.25)

r : recieved bits from time 1 to τ
 c_t : hard estimate

Utilizing the Hamming distance as a path metric the decoder is obtained as the difference of the minimum cumulative Hamming distance along the various paths with partial symbol as binary zero by time τ and the minimum cumulative Hamming distance along paths with partial symbol as binary one by time τ [16, 38]. The SOVA process can be summarized as follows:

1) Forward Recursion:

- Set initial values
- Increase time to $t+1$
- Compute the branch and path metrics for branches entering the node
- Compare the path metrics and choose the survivor path
- The survivor at the final node is the maximum likelihood path

2) Backward Recursion:

- Set initial values
- Decrease time to $t-1$
- Compute the branch and path metrics for branches entering the node
- Compare the path metrics and choose the survivor path

- The survivor at the final node is the maximum likelihood path

2.10.4 Comparison of Processes

The MAP process computes the log-likelihood for all paths in the trellis. The MAP estimates the metric for both a received binary 0 and a received binary 1 and compares them to determine the best overall estimate. The Max-Log-MAP process only considers two paths of the trellis per step: the best path with a data bit of 0 and the best path with a data bit of 1. The Max-Log-MAP utilizes the difference of the log-likelihood function for each of these paths. The SOVA process also uses the two paths that are identical to the paths in the Max-Log-MAP algorithm. However, the SOVA is the least complex of the algorithms in terms of number of calculations [16, 38]. The complexity comparisons of the various decoding algorithms are shown in Table 2.2 where k is the number of systematic bits and v is the memory order of the encoder.

Table 2.2: Decoder Complexity Comparisons

	MAP	Log-MAP	Max-Log-MAP	SOVA
additions	$2 * 2^k * 2^v + 6$	$6 * 2^k * 2^v + 6$	$4 * 2^k * 2^v + 8$	$2 * 2^k * 2^v + 9$
multiplications	$5 * 2^k * 2^v + 8$	$2^k * 2^v$	$2 * 2^k * 2^v$	$2^k * 2^v$
maximum ops	0	$4 * 2^v - 2$	$4 * 2^v - 2$	$2 * 2^v - 1$
look-ups	0	$4 * 2^v - 2$	0	0
exponentiation	$2 * 2^k * 2^v$	0	0	0

Max-Log-MAP is seemingly three times as complex as the SOVA. Log-MAP is approximately twice as complex as the SOVA [33]. For the BER performance SOVA

performs better than the MAP for low SNR values but for high values of SNR it approaches the performance of the MAP [17, 18].

2.11 PCCC MAP Decoding

PCCC decoding is an iterative process that is designed as a structure with two decoders sharing soft decision information for the estimated data [16, 38]. With each iteration, the decoders work in concert to produce a better estimate of the received bit. Once the intended numbers of iterations are performed the system makes a hard binary decision for the estimated data bit. The general PCCC decoding structure is shown in Figure 2.14:

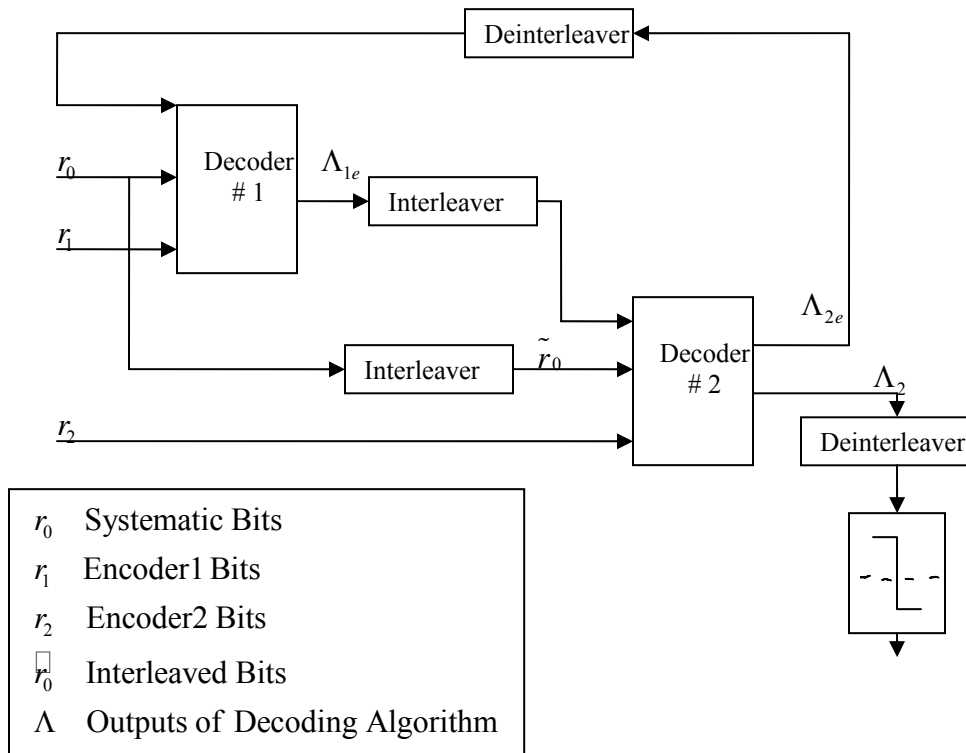


Figure 2.14: PCCC MAP Decoder

The two MAP decoders in the structure utilize a soft decision estimation algorithm. The first MAP decoder operates on the received systematic bits r_0 and parity bits r_1 that are generated by the first encoder in the PCCC structure and produces a soft estimate of the transmitted symbol. The second MAP decoder utilizes the received parity bits from the second encoder r_2 in the PCCC structure as well as an interleaved version of the received parity bits and the interleaved soft estimate produced by the first MAP decoder. Because the soft estimate from the second MAP decoder is inputted back to the first MAP decoder, the operational sequence becomes iterative. After the intended number of iterations has occurred the decoders terminate the processing of the data and a final estimate is inputted to the hard decision threshold detector. A summary of the process is as follows [38]:

- 1) Initialize the feedback output of decoder 2: $\Lambda_{2e}^{(0)}(c_t) = 0$
- 2) For all iterations
 - a. Compute $\Lambda_1^{(r)}(c_t)$ and $\Lambda_2^{(r)}(c_t)$ using:

$$\Lambda(c_t) = \log \frac{\sum_{l', l=0}^{M_s-1} \alpha_{t-1}(l') p_t(1) \exp\left(-\frac{\sum_{j=0}^{n-1} (r_{t,j} - x_{t,j}^1(l))^2}{2\sigma^2}\right) \beta_t(l)}{\sum_{l', l=0}^{M_s-1} \alpha_{t-1}(l') p_t(0) \exp\left(-\frac{\sum_{j=0}^{n-1} (r_{t,j} - x_{t,j}^0(l))^2}{2\sigma^2}\right) \beta_t(l)} \quad (2.26)$$

- b. Compute the extrinsic information of the first decoder $\Lambda_{1e}^{(r)}(c_t)$ as

$$\Lambda_{1e}^{(r)}(c_t) = \Lambda_1^{(r)}(c_t) - \frac{2}{\sigma^2} r_{t,0} - \tilde{\Lambda}_{2e}^{(r-1)}(c_t) \quad (2.27)$$

- c. Compute the extrinsic information of the second decoder $\Lambda_{2e}^{(r)}(c_t)$ as

$$\Lambda_{2e}^{(r)}(c_t) = \Lambda_2^{(r)}(c_t) - \frac{2}{\sigma^2} \tilde{r}_{t,0} - \tilde{\Lambda}_{1e}^{(r)}(c_t) \quad (2.28)$$

- d. After the iterations make a hard decision based on the output of the second decoder $\tilde{\Lambda}_2^{(l)}(c_t)$

2.12 BER for PCCC

BER is a standard metric in digital communication systems which calculate the average number of bit errors a system produces with channel noise. BER is plotted against the bit energy E_b to noise power N_0 ratio where E_b is calculated as in Figure 2.7 and N_0 is the power shown in Figure 2.8. PCCC is specifically designed to improve BER performance but at the expense of computational complexity, throughput, and resources [20]. Two important aspects of PCCC determine its BER performance in the presence of additive white Gaussian (AWGN) noise. One is the length of the data-frame/interleaver. A long data-frame/interleaver is important because the MAP decoding process then produces more information during the forward and backward recursion calculations which improves the BER performance.

The second aspect of PCCC that affects the BER is the intended number of iterations. The MAP decoders in the PCCC calculate a soft estimate of the bits in the partial symbols of the data frame. This information is communicated recursively during the decoding process. It has been demonstrated that an increased number of iterations provides better BER performance at the expense of throughput.

Another comparison is made between the MAP, Log-MAP and SOVA algorithm. The MAP algorithm has the best BER performance because it does not perform estimations. The LOG-MAP performance comes close to attaining the same BER performance due to the Jacobian correction function. However, due to the approximation of the recursive correction function in the LUT the LOG-MAP algorithm does not reach the full potential of the MAP algorithm.

CHAPTER 3

PROGRAMMABLE GATE ARRAY ARCHITECTURE

3.1 Programmable Gate Array

Integrated circuits (IC) are created by the interconnection of thousands of transistor circuits on a semiconductor substrate to implement logic and memory devices. This type of semiconductor design is called Very Large Scale Integration (VLSI). In simple terms, VLSI is the design verification and manufacturing of dedicated circuits called Application Specific Integrated Circuits (ASIC). The ASIC can typically consist of a processor, peripherals and memory devices such as random access memory (RAM) and read only memory (ROM).

The two major benefits of the ASIC is its low cost to manufacture and speed. The design of an ASIC is relatively costly, but once completed the devices can be manufactured at low cost. The ASIC is an excellent choice for consumer products that are manufactured in very large numbers, such as those for game consoles, personal computers and cellular telephones. The ASIC also executes operations expeditiously because of their compact VLSI architectures.

Two major drawbacks of the ASIC are its cost and time to market. The inherent cost of designing a single ASIC chip can be in the range of millions of dollars. Also the ASIC design process is time consuming and could be problematical if the customer needs the ASIC within a relatively short period of time.

Another drawback to the ASIC is its inability to be reconfigured. These considerations for the ASIC spurred the development of the programmable gate array (PGA) which is a robust IC architecture that is relatively low cost but inherently

reconfigurable. The PGA is an IC that contains configurable logic blocks (CLB) that can be set or programmed to replicate the functionality of combinational and sequential logic circuits. The coarse grained architecture of the early PGAs has evolved to the fine grained architecture of the current PGA which now includes embedded adders, multipliers, and memory.

The functionality of the PGA provides the ability to configure and verify a specific algorithm and then, if required, to make changes easily to the design. The PGA can be configured rapidly which provides an ideal platform for the iterative design process. If a design is going to be implemented on a relatively small scale then the PGA can be directly implemented in the final product. However, if the design is implemented on a much larger scale than the PGA is primarily used for algorithmic design and verification. After the design is verified the PGA design can be translated into an ASIC architecture, although the mapping of the design is not direct.

3.1.1 Xilinx Virtex I FPGA

The highly regular coarse grained architecture of the PGA is typified by the (1997) Xilinx Spartan II device. The inception of the fine grained Virtex I (2001) is an architectural improvement over that of the Spartan II PGA. Xilinx proprietary names their device a field programmable gate array (FPGA). The overall architecture of the Virtex I FPGA is shown in Figure 3.1 [43].

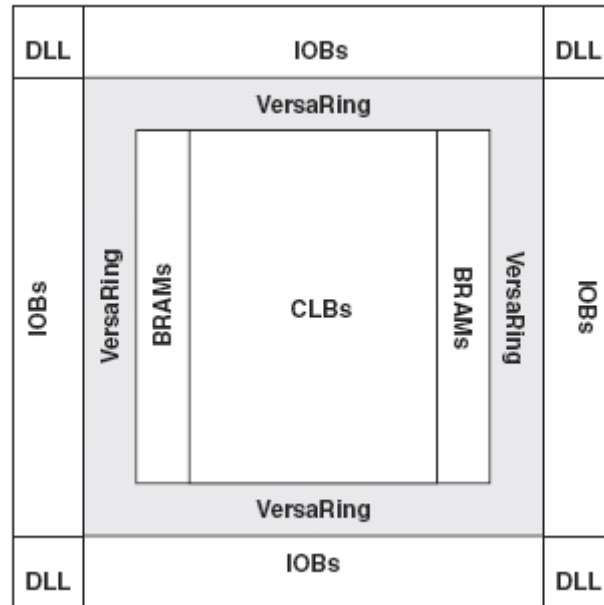


Figure 3.1: Virtex I FPGA Architecture

In this fine grained architecture the configurable logic blocks (CLB) of the Virtex I FPGA are interconnected using a complex routing matrix. This routing matrix allows the CLB to be connected in various ways to duplicate combinational and sequential logic circuits. The routing matrix interconnects the CLB, but the peripheral interconnect matrix (VersaRing) provides routing from the CLB to the input and output blocks (IOB). The Virtex I also includes block random access memory (BRAM) and delay lock loops (DLL) for synchronous clock distribution.

Each CLB consists of four logic cells (LC) distributed in two slices. These LC consist of a look-up table, carry and control logic, and type D flip flop as a storage element. A 2-Slice CLB of the Virtex I FPGA is shown in Figure 3.2.

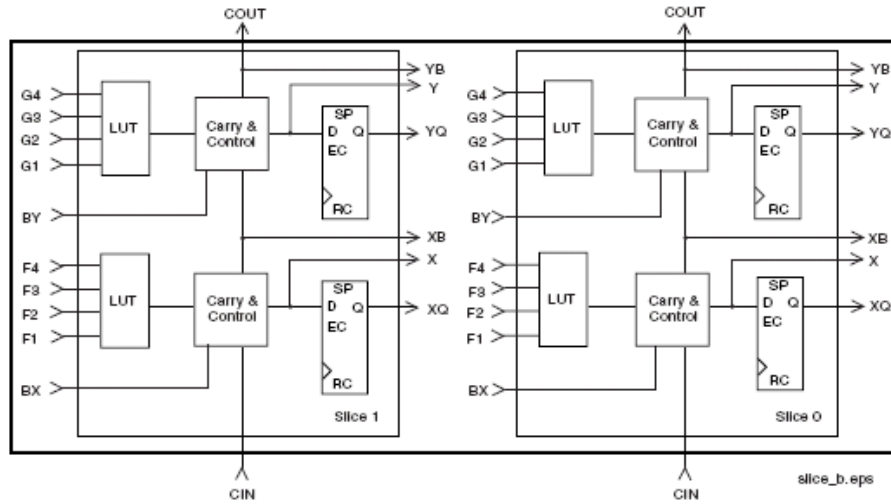


Figure 3.2: Virtex I FPGA 2-Slice CLB

The LUTs can be configured as 32 by 1-bit synchronous RAM or 16-bit shift registers that can be used to store data. Arithmetic processing is provided by dedicated exclusive or (XOR) logic gates as a 1 bit full adder and a dedicated AND logic gate for multiplication operations. Local routing is performed on three levels. The first is connection of the LUTs and general routing matrix (GRM). The second is the feedback paths between internal LUTs. The third is the direct paths between horizontal and adjacent CLBs without the inherent delay of the GRM. The local routing is shown in Figure 3.3.

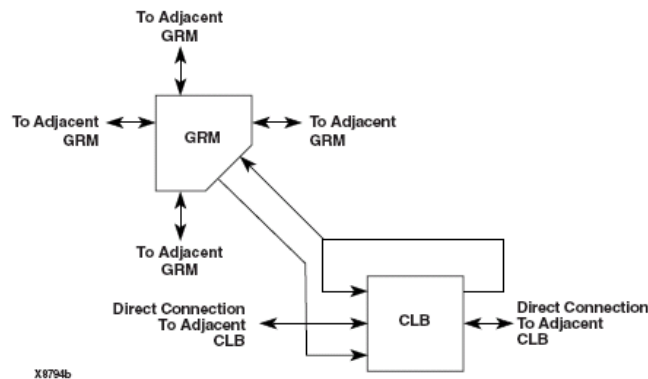


Figure 3.3: Virtex I FPGA Local Routing

3.1.2 Xilinx Virtex II FPGA

The Xilinx Virtex II FPGA (2003) architecture consists of configurable logic blocks (CLB), block random access memory (BRAM), and a digital clock manager (DCM). The DCM is an improved circuit for synchronous clock distribution. An overview of the Virtex II architecture is shown in Figure 3.4.

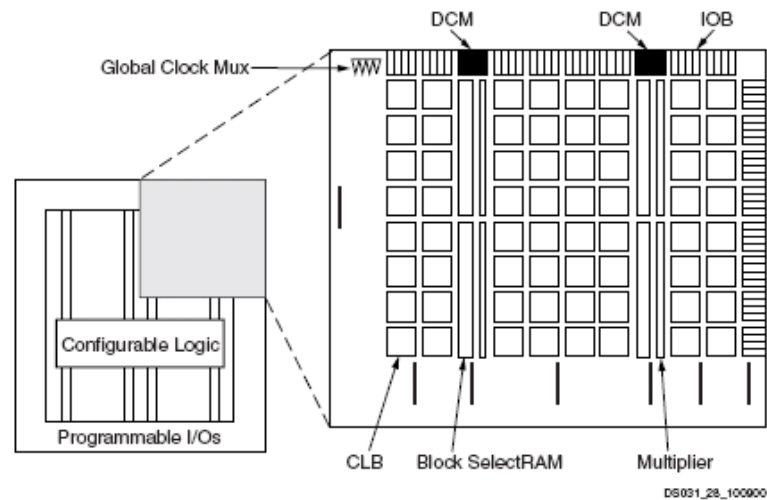


Figure 3.4: Virtex II FPGA Architecture

In the Xilinx Virtex II FPGA each CLB consists of four slices that are interconnected through a switch matrix. Each slice consists of two 4-input look up tables, wide multiplexers, 16-bits of RAM or a 16-bit variable tap shift register. The CLB and the configuration of a single slice are shown in Figure 3.5.

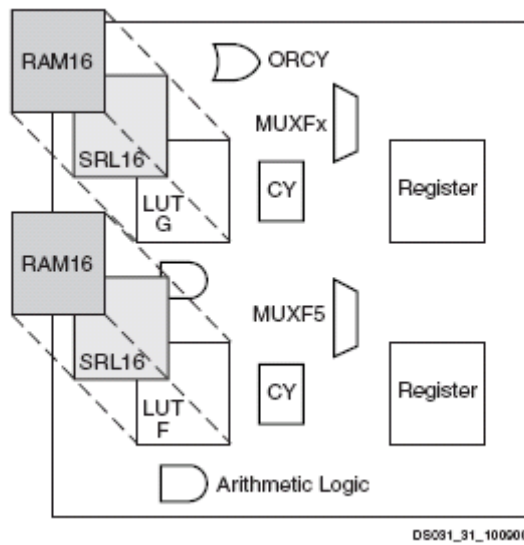
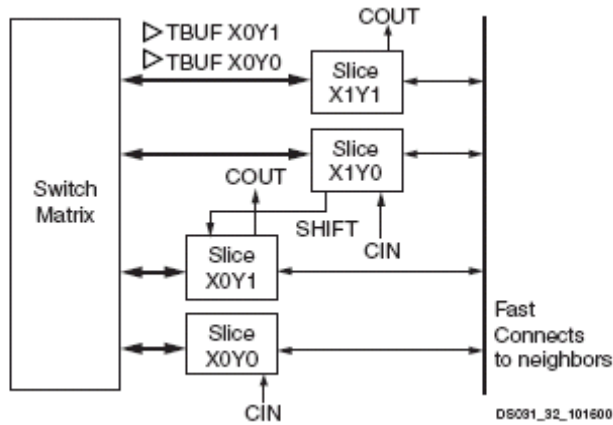


Figure 3.5: Virtex II FPGA CLB and Slice Configuration

The DCM eliminates clock distribution delays and utilizes a frequency synthesizer to generate clocks with a wide range of frequencies and duty cycles with dynamic phase shift control.

Active interconnect technology (AIT) is a programmable routing matrix introduced in the Xilinx Virtex II FPGA. A benefit of the AIT is that all logic elements, RAM and embedded multipliers are connected to an identical switching matrix for global routing access. The active interconnect technology is shown in Figure 3.6.

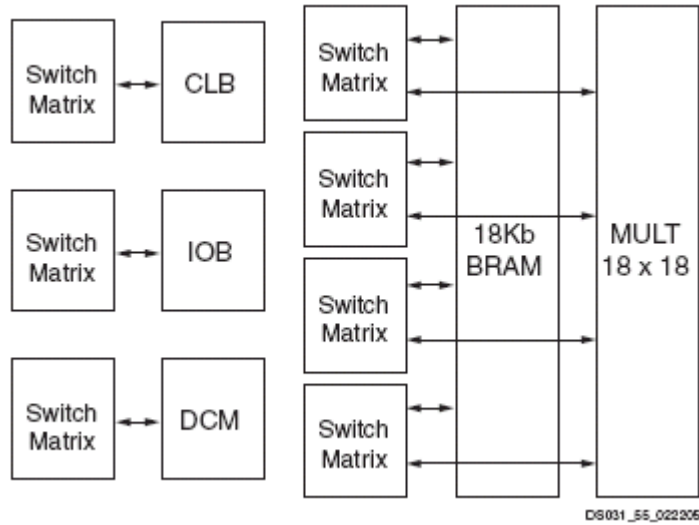


Figure 3.6: Active Interconnect Technology

3.1.3 Xilinx Virtex 4 FPGA

The Xilinx Virtex 4 FPGA (2005) is utilized in this research [43]. The Xilinx Virtex 4 FPGA also introduced the concept of a device family with various architectures. The Virtex 4 has three complimentary fine grained architectures that specifically target three different applications: LX for logic applications, FX for embedded platform applications and SX digital signal processing applications. Some of the features are a digital clock manager (DCM) that utilizes a phase matched clock divider, the DSP48 slice that includes an 18-bit by 18-bit 2's compliment multiplier and a 48-bit accumulator, RAM that can be configured into 18 Kbit or 36 Kbit cascaded blocks, 6.5 Gb/s transceivers for high speed input/output (I/O) applications, and a Power PC hard core which provides a reduced instruction set computer (RISC) processor.

The DSP48 slices consists of embedded 18-bit by 18-bit multipliers and corresponding logic that facilitates operations such as multiply, accumulate, multiply and

accumulate, division, square root, multiplexer, barrel shifter and counter. These DSP48 slices are specifically targeted in this research for efficiency of the Turbo Code process.

The architecture of the DSP48 slice is shown in Figure 3.7.

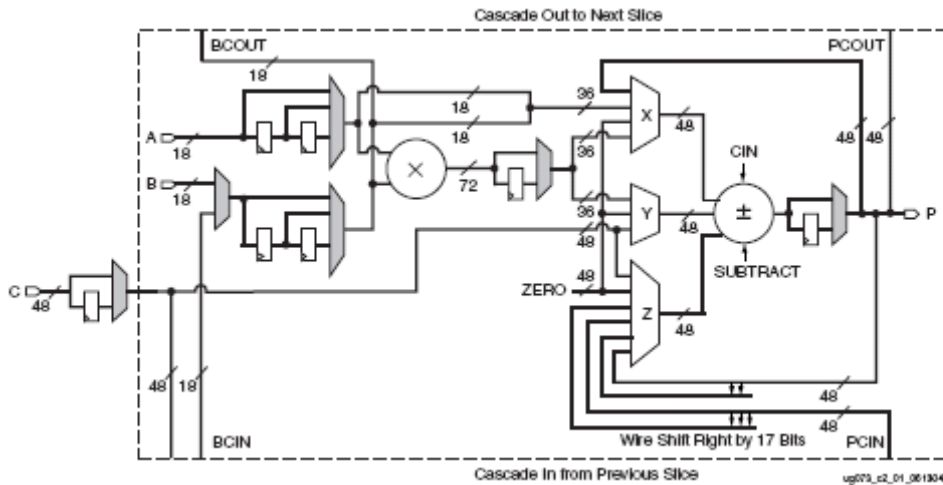


Figure 3.7: DSP48 Slice in the Xilinx Virtex 4 FPGA

Block RAM (BRAM) provides buffers and look-up tables. Buffers are important to the Turbo Code decoding process due to its iterative nature and because the process computes data on a frame by frame basis. The Xilinx Virtex 4 FPGA provides dual port BRAM modules that can store 18 Kbits of data. The bit width and depth for the BRAM can be configured to store 16 Kbit by 1-bit, 8 Kbit by 2-bit, and 512-bit by 36-bit data. BRAM modules also have the ability to be cascaded to allow up to 32 Kbits of storage. The bit width and depth and dual port configuration of the BRAM modules are essential to maintain data flow synchronization in the Turbo Code decoding procedure. The system model of the BRAM is shown in Figure 3.8.

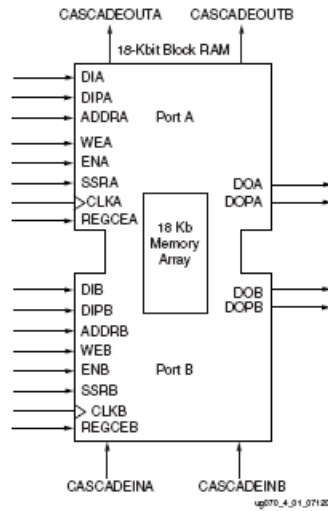
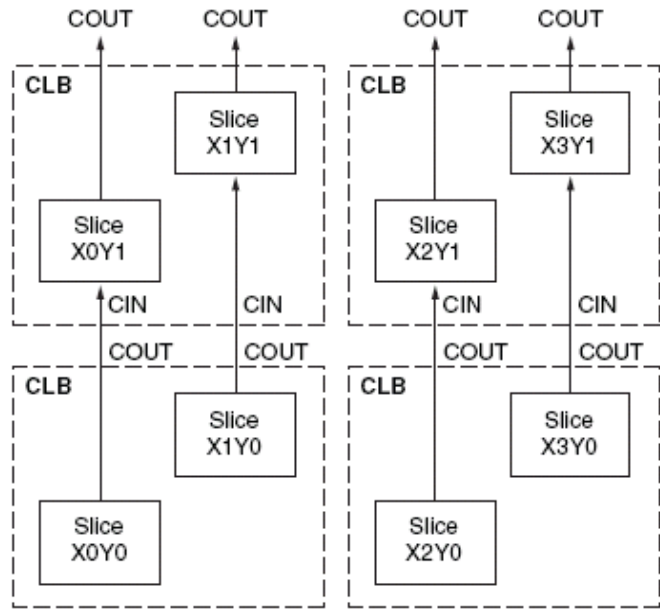


Figure 3.8: BRAM System Model in the Xilinx Virtex 4 FPGA

3.1.4 Xilinx Virtex 5 FPGA

The Xilinx Virtex 5 FPGA (2006) is the newest device that would significantly improve this research since it is the most advanced fine grained FPGA now available from Xilinx. Block RAM is a major feature in the Xilinx Virtex 5 FPGA that can be utilized in many applications. Each CLB in the Xilinx Virtex 5 FPGA consists of two slices connected to a switching matrix and arranged as a column. Each slice consists of four LUTs, four storage elements, multiplexers, carry logic and data storage via distributed RAM. Furthermore the Xilinx Virtex 5 FPGA improved the DSP48 architecture to include a 25-bit by 18-bit embedded multiplier for large word multiplications. The relationship between the CLBs and the slices is shown in Figure 3.9.



UG190_5_02_122605

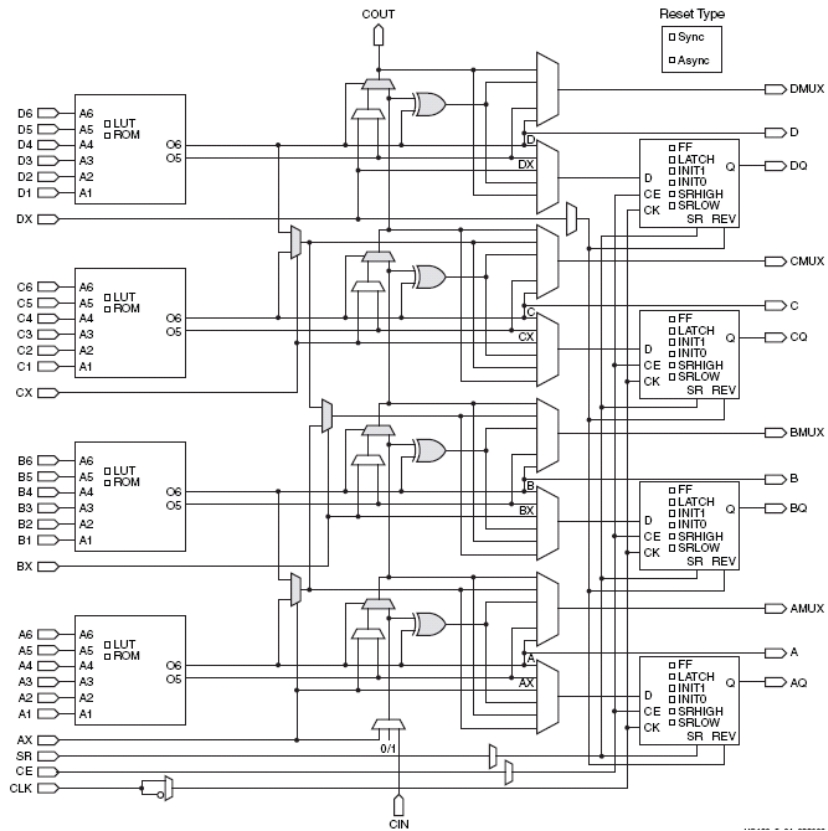


Figure 3.9: CLB and Slice Configuration in the Virtex 5 FPGA

CHAPTER 4

SOFTWARE AND HARDWARE PLATFORM

4.1 Software Platform

This research employs Xilinx electronic design automation (EDA) software called the System Generator (SG) which operates in concert with the Mathworks Matlab/Simulink environment [7, 42]. The Xilinx SG facilitates the design of FPGA hardware systems in the graphical Simulink environment [12]. Hardware modules that are provided by the Xilinx SG can be used in design or new modules can be created utilizing the Verilog hardware description language. These EDA tools and others are discussed in detail in the following Chapters.

4.1.1 Hardware Description Language

A hardware description language (HDL) is an EDA tool that provides a software platform for the design and verification of an FPGA architecture. Processes can be expressed as conversational behavioral code. Hardware design can be summarized by the following steps. A behavioral HDL description of the process is created first. The HDL code is then simulated to verify functionality. The design is then synthesized or translated to hardware. Finally a post synthesis simulation is performed and the hardware design is placed and routed to the FPGA.

The synthesis tool takes the behavioral description and creates an optimal Boolean description. It accomplishes this by removing redundancy in hopes of minimizing the required logic and to satisfy time constraints. Post synthesis verification is accomplished by stimulating the pre and post synthesized code with the same inputs. It

then compares the outputs of both systems to see if they match. If the pre and post synthesis outputs match then the functionality was correctly translated into hardware. Also post synthesis timing verification is performed.

If timing verifications are not met the system will re-synthesize and make the proper modifications in order to meet the timing specifications. The final step is placement and routing. This procedure optimally arranges the cells and connects the signal paths on the PGA. It should be noted that it is also possible to manually affect the placement and routing.

A behavioral HDL description of a specific process is accomplished by using one of several conversational languages. This research essentially utilizes Verilog for all hardware designs although the Xilinx SG is a graphical interface rather than a source text file for such a behavioral description.

4.1.2 Matlab/Simulink

Matlab is a vector based high level language similar to C [7]. The computer aided design (CAD) environment includes numerous toolboxes with built in functions for diverse applications. Some of the toolboxes include but are not limited to: digital communications, digital signal processing and statistical data processing. All non-hardware related Turbo coding simulations in this environment were performed using Matlab.

Simulink is essentially Matlab code embedded in a graphical user interface. Matlab toolbox functions can also be expressed as blocks in Simulink that can be connected to form a system [12]. Each of the Simulink blocks have parameters that can

be easily altered which facilitates a systematic design approach. To illustrate the similarities and differences between the Matlab and Simulink CAD environment a binary phase shift keying (BPSK) system is presented. The discrete signal outputs of both Matlab and Simulink are equivalent which verifies that they are performing the same operation even though they represent entirely different approaches. This software comparison is shown in Figure 4.1.

MATLAB:

```

%GENERATE BINARY DATA
data=round(rand(1,5))           1  0  0  1  1

%MODULATE
mod=pskmod(data,2)             1 -1 -1  1  1

%DEMODULATE
demod=pskdemod(mod,2)          1  0  0  1  1

```

SIMULINK:

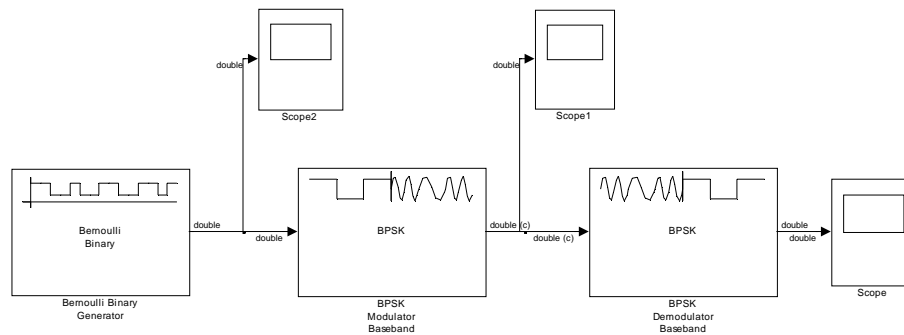


Figure 4.1: Matlab versus Simulink Simulation of a (BPSK) System

Furthermore, the Simulink model consists of built-in tokens such as a random data generator, BPSK modulator and demodulator. Each of the tokens has unique

parameters that can be altered to obtain different functionality. The scope tokens are utilized to view and verify the waveforms at particular points in the Simulink model.

4.1.3 Xilinx Integrated Synthesis Environment

The Xilinx Integrated Synthesis Environment (ISE) is a computer aided design (CAD) environment that enables hardware description language design, synthesis and verification of digital designs [40]. This Chapter illustrates the procedure for designing and verifying a digital system in the Xilinx ISE environment. The first step is to set the device properties for the field programmable gate array (FPGA) by specifying the family, package, and preferred HDL simulator. The new project window that is displayed in the Xilinx ISE is shown in Figure 4.2. This particular example targets a Spartan-3 FPGA in a 256 pin package.

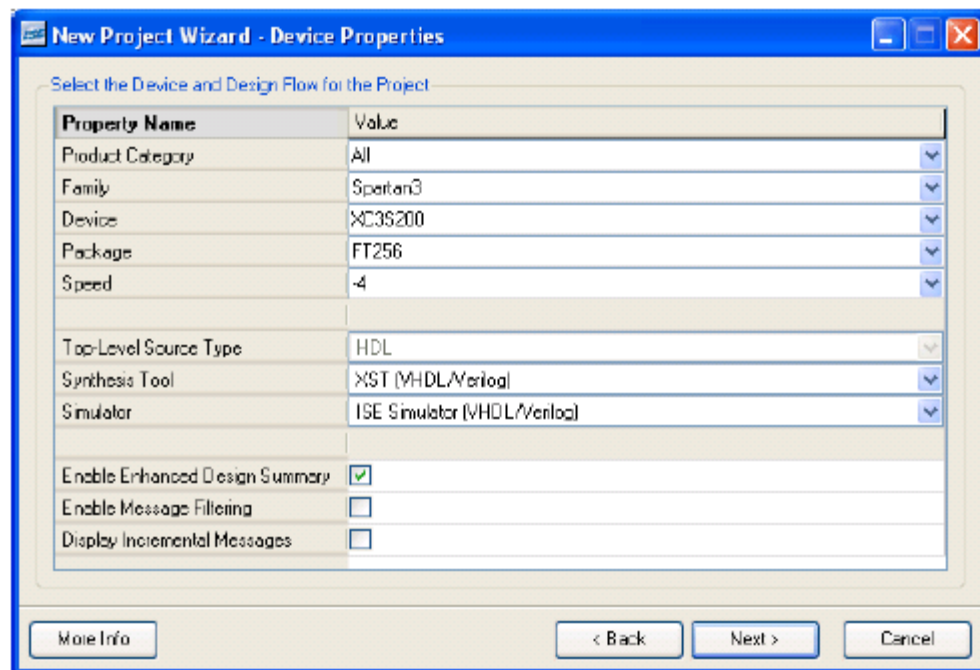


Figure 4.2: Xilinx ISE New Project Wizard

The next step is the creation of a new HDL project. The project must have a unique name and include new or existing HDL files. The Xilinx ISE example in the upper of the design window as shown in Figure 4.3 shows a project called counter with an included Verilog file which contains the description of a 4-bit counter. The Verilog code is shown in the right hand window of Figure 4.3.

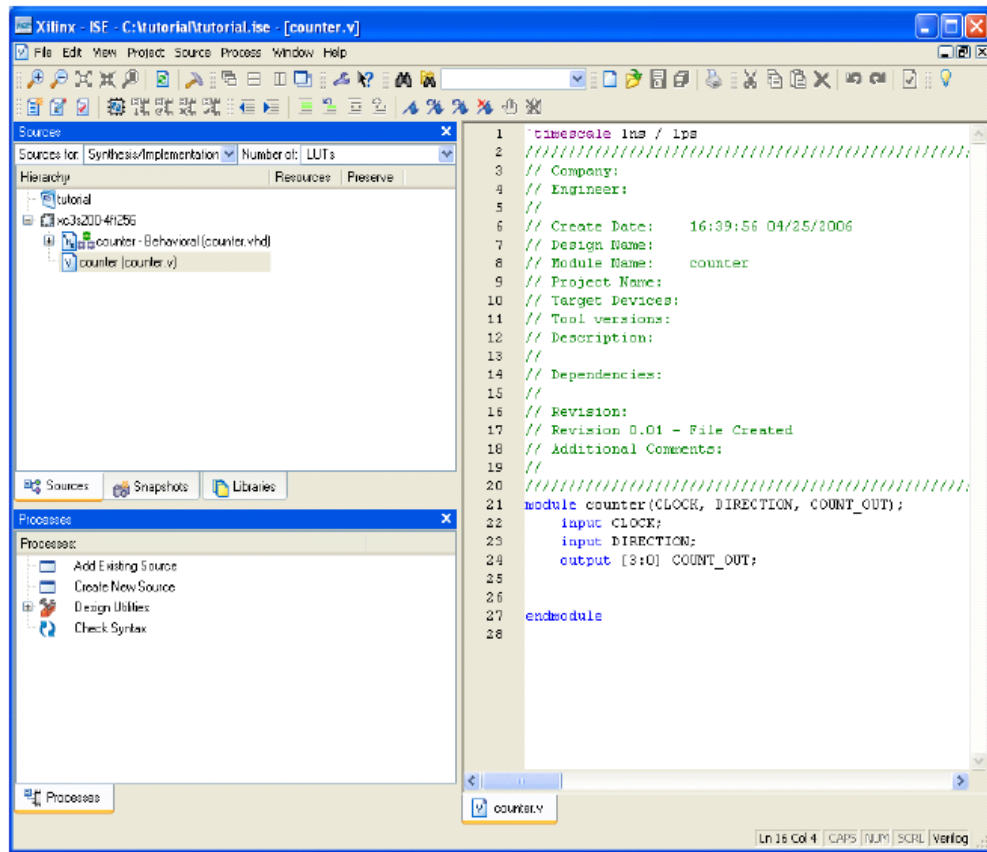


Figure 4.3: Xilinx ISE New Project

Next the clock information is set in the clock wizard. Figure 4.4 shows the clock wizard that provides a means of specifying the frequency and duty cycle of the onboard oscillator used in the design. In this example the clock has a period set at 40 ns.

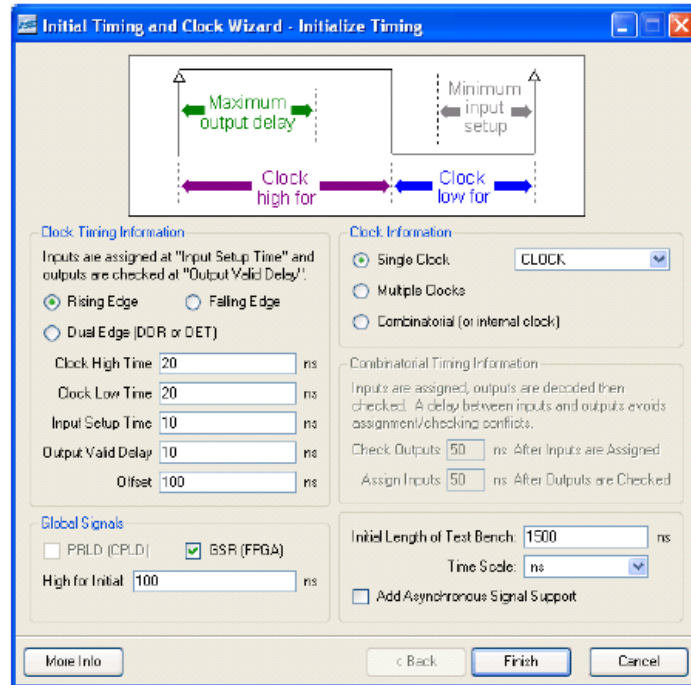


Figure 4.4: Xilinx ISE Clock Wizard

Simulation of the source code in the project is the next step. The design is a 4-bit counter that is driven by the positive edge of the clock. The simulation of the 4-bit counter for a period of 1500 ns is shown in Figure 4.5. All 4 bits of the counter can be seen changing binary state and thus produce a 4-bit counter output.

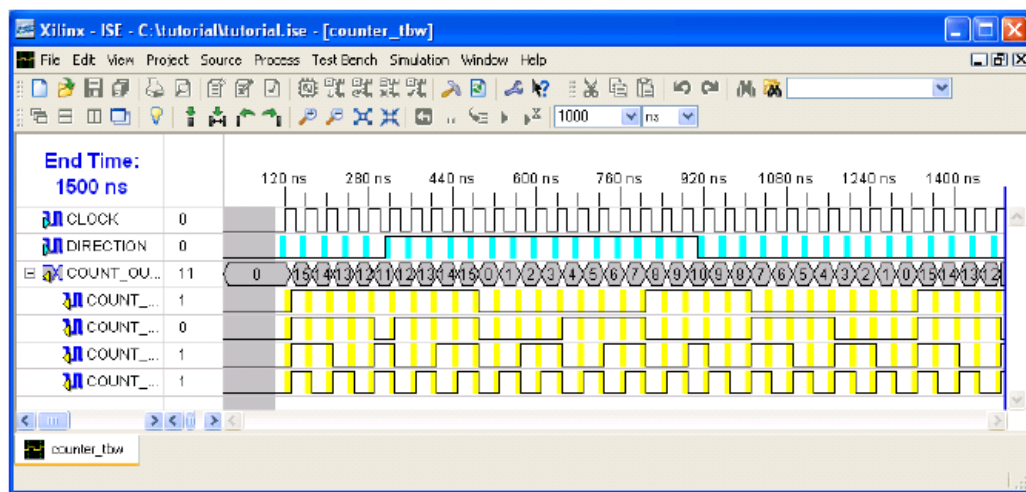


Figure 4.5: Xilinx ISE Simulation of a 4-Bit Counter

Once the system is verified in simulation the next step is the synthesis hardware mapping of the Verilog code. The post synthesis report generated by the Xilinx ISE is shown in Figure 4.6. In the right side of the design window the targeted device and its corresponding utilized resources are shown. In this design the 4-bit counter utilized approximately 1 percent of the available FPGA resources such as look-up tables, flip-flops, and slices. Furthermore, after the design is correctly synthesized the FPGA pins that the signals are mapped to are shown. The pins for the clock and counter bits are shown in Figure 4.7. These locations can be specified before synthesis using a user constraint file (UCF).

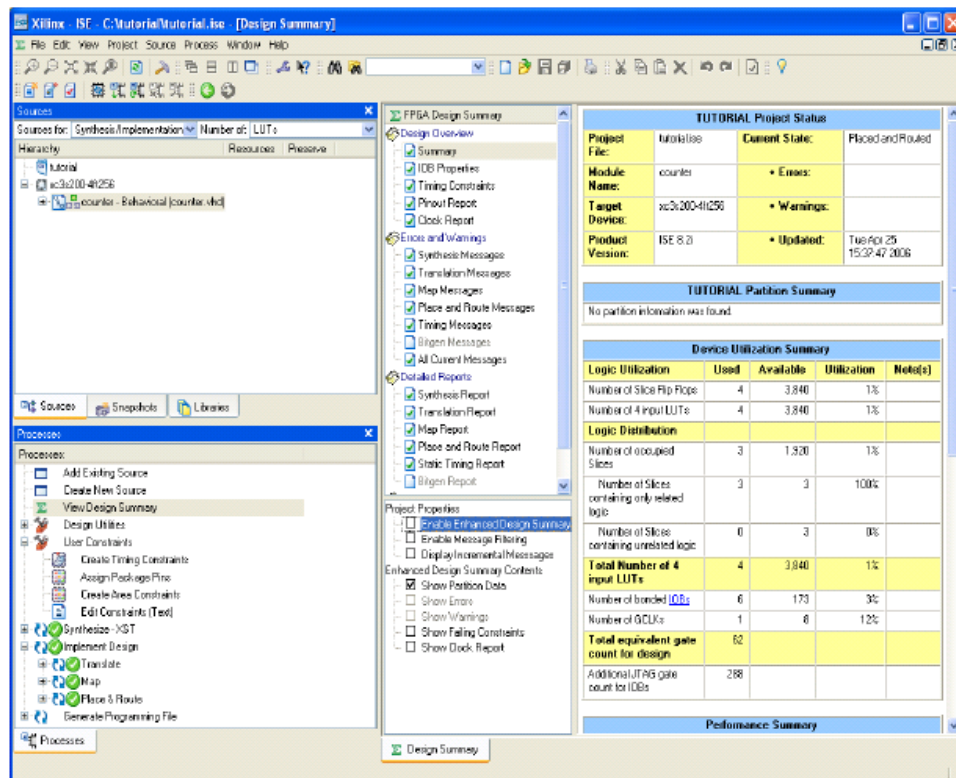


Figure 4.6: Xilinx ISE Synthesis Report

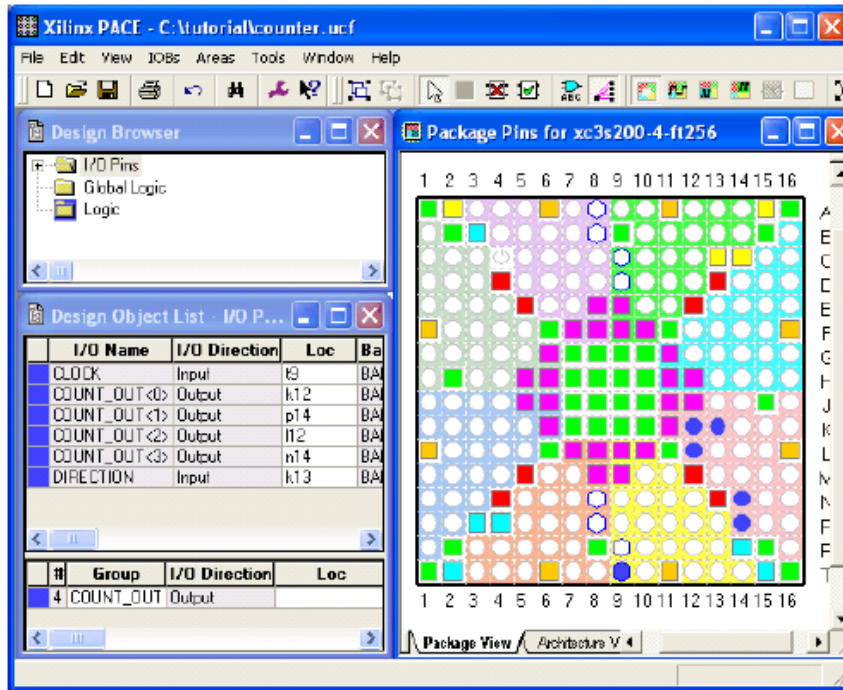


Figure 4.7: Xilinx ISE FPGA Pin Mapping

Once the design is synthesized and mapped to the FPGA a post-synthesis verification simulation can be performed. Post synthesis simulation is important because it verifies that the synthesis tool correctly translated the behavioral Verilog HDL code to hardware.

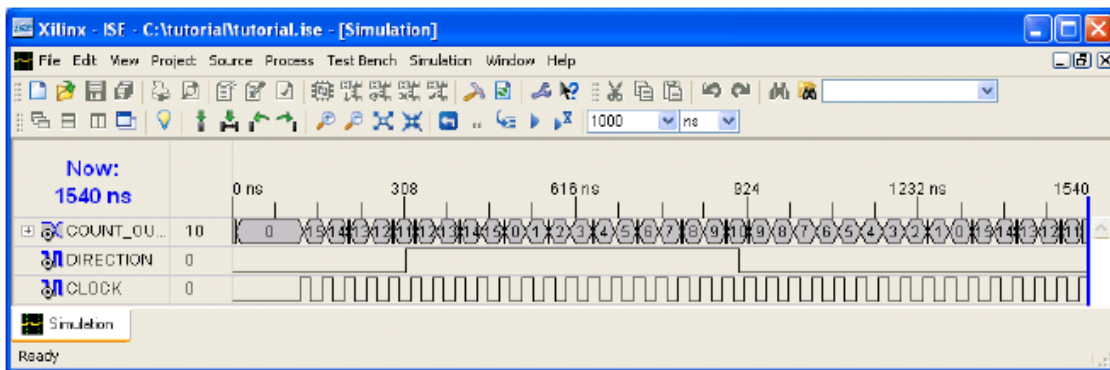


Figure 4.8: Post Synthesis Simulation Output

The output of the post- synthesis simulation is shown in Figure 4.8. The waveforms prove that the 4-bit counter is operating correctly.

The final step in the design process is the downloading of the synthesized design to the FPGA. The FPGA hardware module communicates with the Xilinx ISE via a cable using Joint Test Action Group (JTAG) protocol. The cable can be either a serial, parallel or Universal Serial Bus (USB). ISE communicates with the hardware module using a boundary scan protocol that recognizes the FPGA and other onboard devices. Xilinx ISE recognized the Spartan-3 FPGA along with a memory chip as shown in Figure 4.9. Once the FPGA is identified the synthesized design bit file is downloaded to the board.

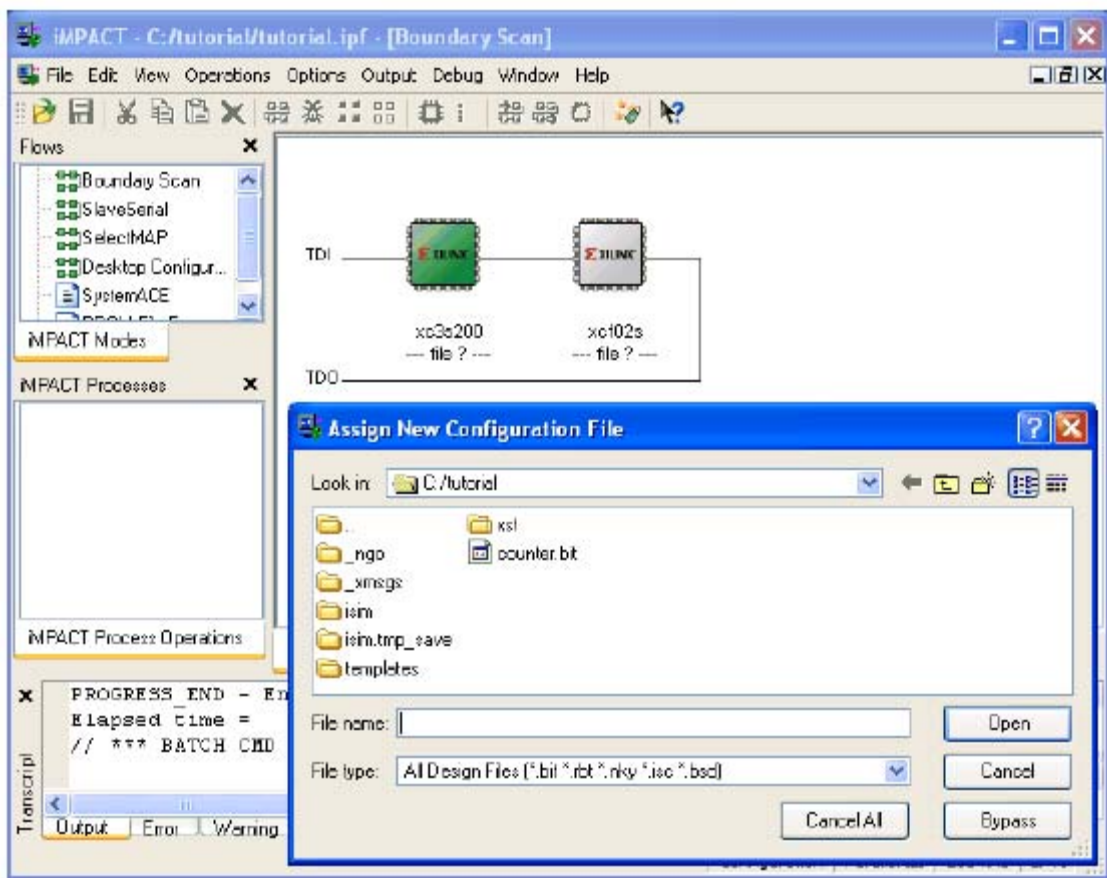


Figure 4.9: Xilinx ISE Boundary Scan

4.1.4 Xilinx ChipScope

The Xilinx ISE has a unique digital signal capturing feature called ChipScope [39]. This is an integral hardware logic analyzer that is downloaded to the FPGA along with the synthesized design. In order to assess the performance of a complex design with a conventional external logic analyzer the digital signals would have to be routed to output pins which of itself creates timing problems in the architecture. Digital signals are not routed in using ChipScope since the logic analysis is completely integral to the FPGA and exists in addition to the hardware design. ChipScope utilizes the same cable and JTAG protocol used in the downloading of the hardware synthesis bit file. The JTAG port transfers logic information to the host PC where it is displayed. Figure 4.10 shows what ChipScope captures and displays on the host PC as a design is running on the FPGA.

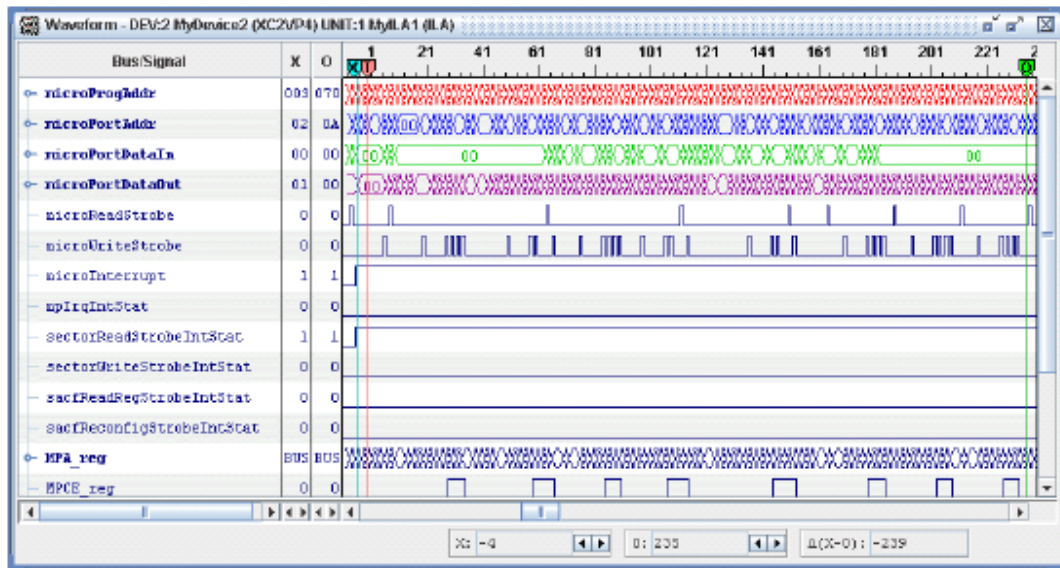


Figure 4.10: Xilinx ChipScope Logic Analyzer Waveform Output

4.1.5 Xilinx System Generator

The Xilinx System Generator (SG) is a FPGA hardware based toolbox that is integrated with the Simulink libraries [12, 42]. Normally Simulink is not a FPGA hardware synthesis computer aided design (CAD) environment, but rather only modeling software. System Generator provides Simulink with exclusive hardware blocks that can interact with other processing blocks. A Simulink simulation of an FIR filter that is using these Xilinx SG hardware blocks is shown in Figure 4.12. The Xilinx SG provides blocks that are specific to applications such as digital signal processing (DSP), digital communications and digital image processing. A list of some of the blocks that the SG provides are shown in Figure 4.11 including arithmetic functions such as an accumulator, adders/subtractors. Also shown are digital data blocks including a so-called bit basher that has the ability to strip and concatenate bits to the inputs. Furthermore a black box token can be used to import unique user defined Verilog HDL code into the SG environment where it can interact with other hardware blocks.

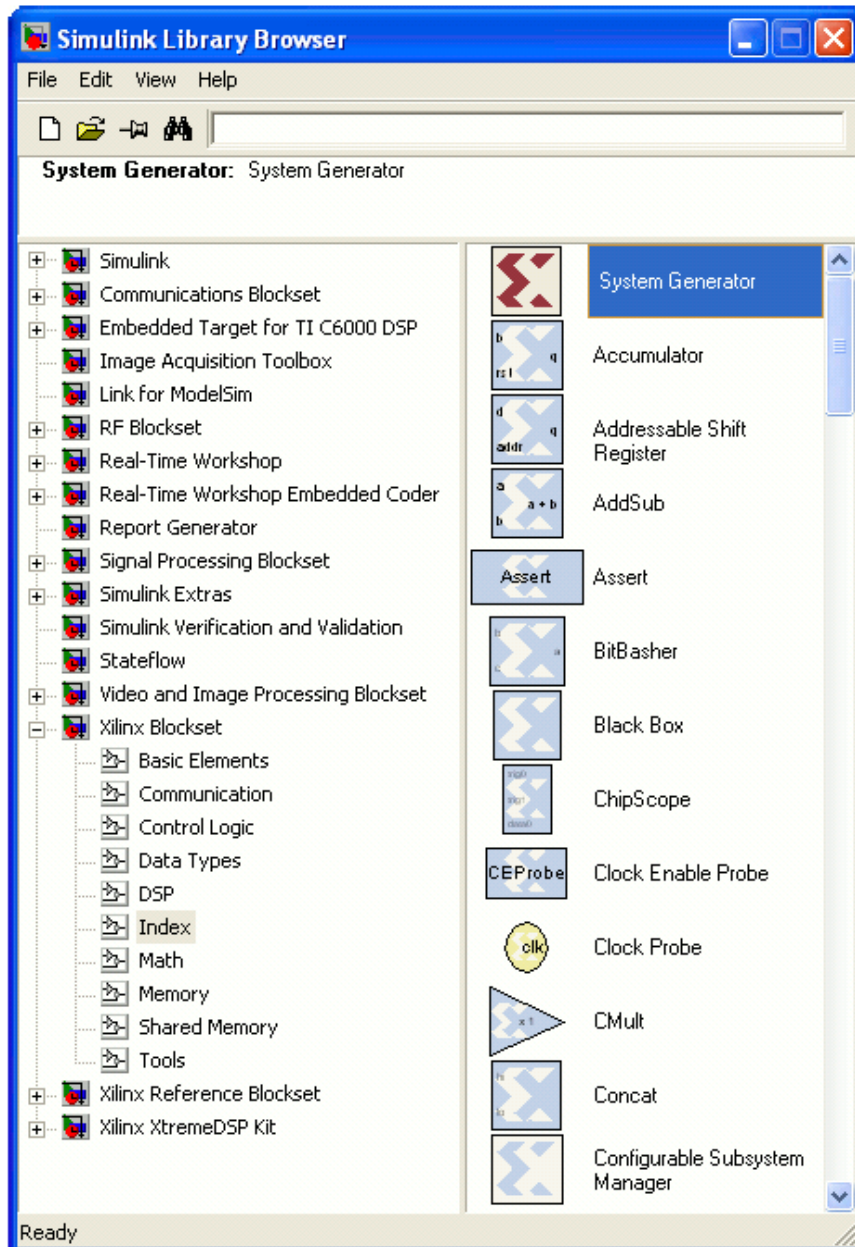


Figure 4.11: Xilinx System Generator Library

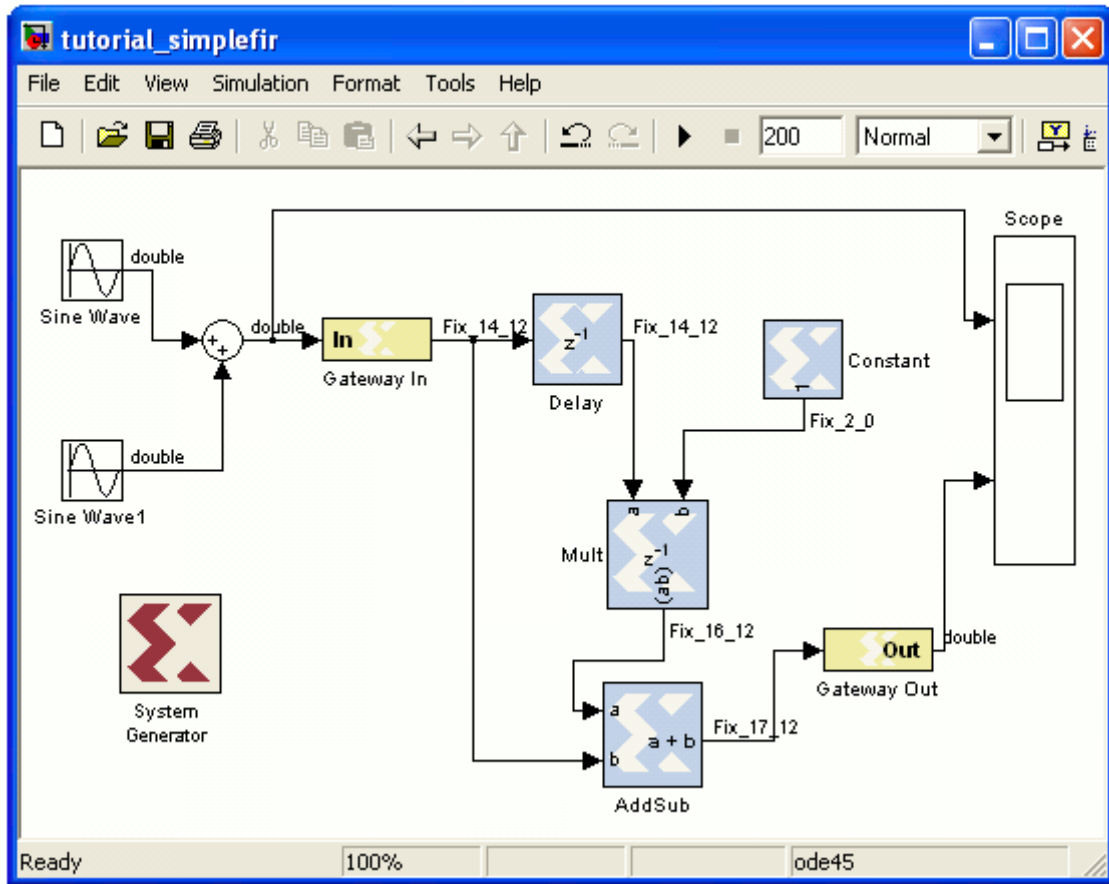


Figure 4.12: Xilinx System Generator FIR Filter Example

In the FIR filter example, Simulink simulated analog sinusoids are inputted to a hardware gateway which quantizes the signal to a specified number of bits. The hardware blocks inside the gateways implement a simple FIR filter in this example. The output then exists by the hardware gateway out and is viewed in the Simulink environment. An important feature of SG is its ability to provide co-hardware simulation where the Xilinx hardware blocks within the gateways are synthesized using a link to the ISE environment [12]. The synthesized code executes on the FPGA and can be compared with a Simulink simulation to verify that the outputs agree. All outputs are provided to the PC via the JTAG connection in a similar manner to that used for the ChipScope analyzer.

The Xilinx SG provides an environment that is visually conducive to the design procedure and a real-time method of inputting and outputting digital data for verification purposes. If a specific functionality cannot be obtained from the integral libraries, a SG token containing either Verilog HDL or Matlab code can be included.

4.2 Hardware Module

The hardware module for this research includes the advanced Xilinx Virtex 4 XC4VSX35 fine grained FPGA architecture. This particular Virtex 4 device contains 192 embedded DSP48 multipliers, 192 18 Kbit blocks of RAM, Input/Output ports for interaction with the host PC, input slide switches and push buttons for user interaction, and a liquid crystal display (LCD).

4.2.1 Xilinx ML402SX Board

The Xilinx ML402SX is the development board used in this research and contains the following [12, 41]: (1) Virtex 4 SX, 64-MB of DDR RAM, (4) 100MHz Oscillator, (11) AC97 Audio Codec, (8) General Purpose Switches and Buttons, (13) LCD, (14) 4Kb of EEPROM, (15) VGA Output Connector, (17) Compact Flash Storage Device, (12) RS232 Serial Port, (16) PS/2 Mouse Port, (21) Ethernet Port, (24) JTAG Port. The front and back of the ML402SX board with tag numbers for easy component reference to the list are shown in Figure 4.13.

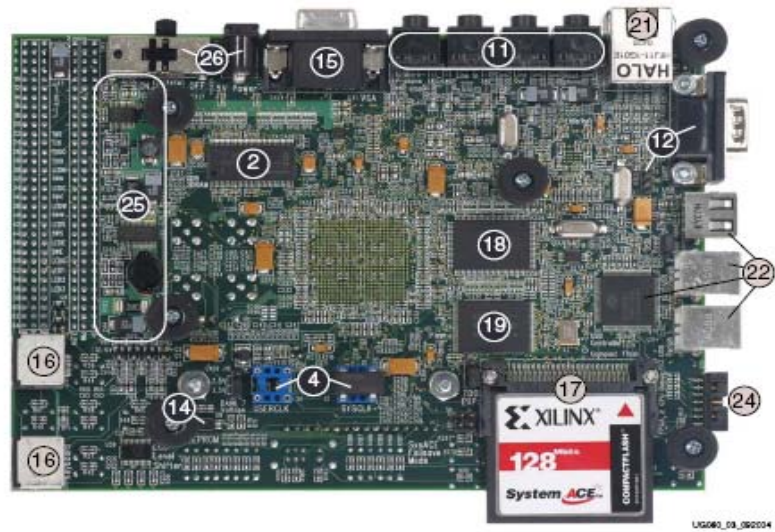
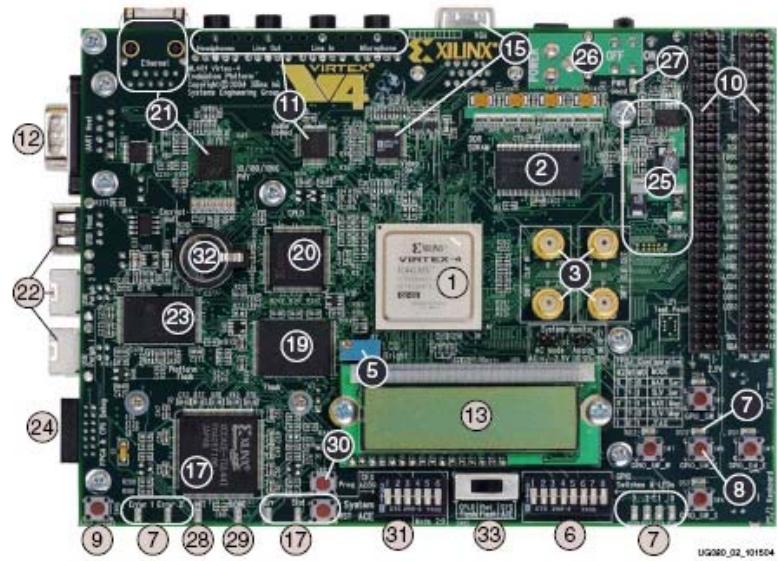


Figure 4.13: Xilinx ML402SX Board

The key external resources of the ML402SX board that are used in this research are the JTAG port, LCD Screen, Buttons/Switches, and LED's. The JTAG port facilitates the high speed data communication between the ML402SX board and the host PC. The Xilinx ISE software downloads the synthesized hardware design to the Virtex 4 SX device through the same JTAG port. After the process is executing on the ML402SX

board, the Xilinx System Generator and Simulink send and receive data signals over the JTAG connection.

A concern inherent to the JTAG port is its relatively slow speed. The JTAG port could become bottlenecked because of the parallel cable or USB cable. In order to avoid this data communication bottleneck high speed Ethernet 1000Mb/s could be used in place of the JTAG USB cable for passing data to and from the board. Finally the LEDs and switches are used to trigger events such as the start and stop of digital data flow.

CHAPTER 5

HARDWARE IMPLEMENTATION

5.1 Parallel Concatenated Convolutional Coder

The encoder of the PCCC shown in Chapter 2.9.1 consists of two identical recursive convolutional coders (RSC) and a random interleaver [3, 30, 37]. A stream of data bits are inputted through one encoder while the same stream of data is randomly interleaved and inputted through a second identical encoder. The salient reason why the encoders are functionally identical is to ensure that the parity bits that they generate are uncorrelated. Uncorrelated streams of data are affected by channel noise differently and therefore provide the decoder with two different possibilities to decode a single symbol [38].

For each data input bit of a rate $1/3$ code, there are three total bits transmitted: the first bit is a parity bit from the output of the first encoder, the second is a parity bit from the output of the second encoder and the third bit is the information bit that produced the two parity bits. The functional description and the Xilinx System Generator (SG) hardware implementation of the PCCC encoder structure as part of this research are shown in Figure 5.1 and Figure 5.2 respectively. A more detailed description of the Xilinx SG hardware implementation PCCC encoder is described in Chapter 5.1.1 and 5.1.2.

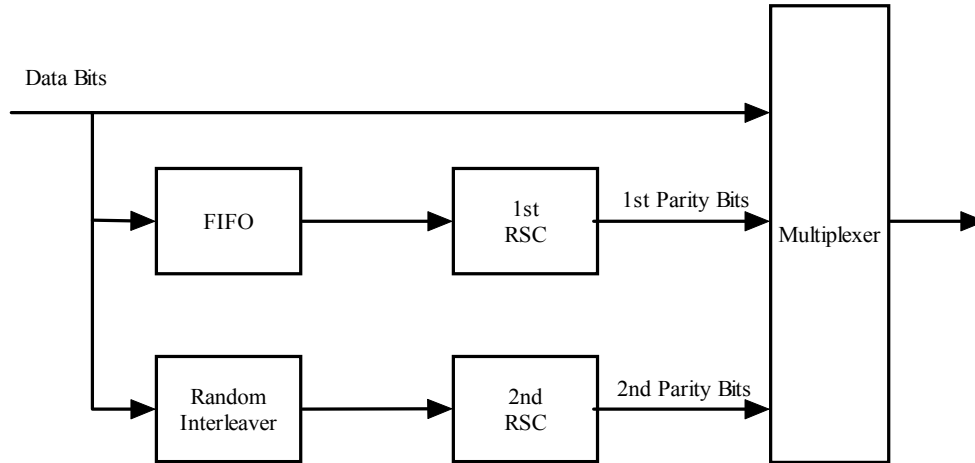


Figure 5.1: PCCC Rate 1/3 Encoder Functional Description

The first RSC has a delayed input which synchronizes the data flow with the second RSC. The second RSC input is buffered to the Block Ram (BRAM) in sequential order and then outputted in a pseudo-random order. The dual port nature of the BRAM blocks allow bits to be written and read simultaneously, which implies that data flows to the RSC without interruption. This particular hardware implementation produces an information bit, a parity bit from the first encoder and a parity bit from the second encoder on every clock cycle once the data pipeline is full.

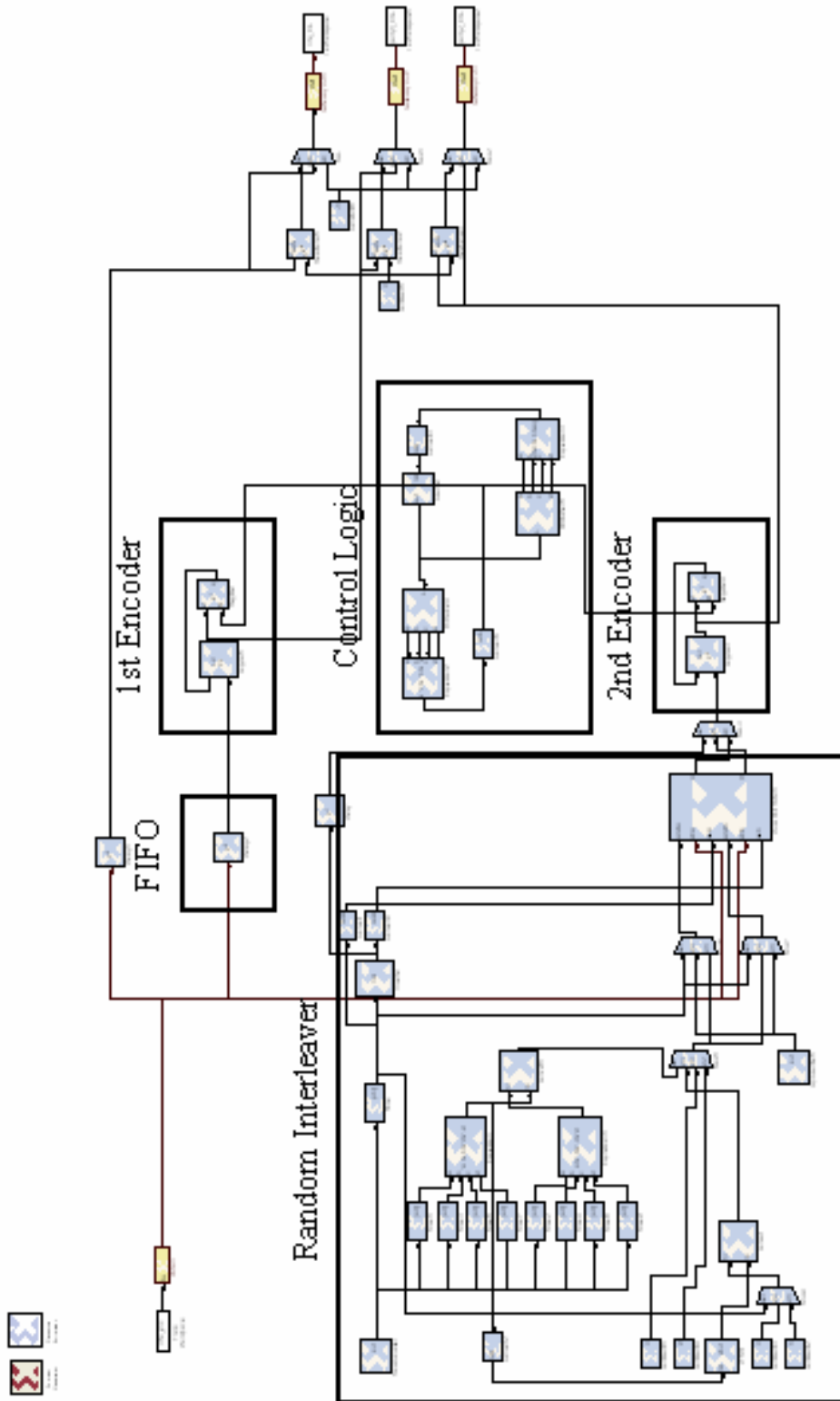


Figure 5.2: PCCC Rate 1/3 Encoder Xilinx SG Implementation

5.1.1 Recursive Convolutional Coder

The hardware design of both the first and second recursive convolutional coders (RSC) was implemented utilizing registers and modulo-2 adders. The functional description and Xilinx SG hardware implementation of a 2-state RSC are shown in Figure 5.3 and Figure 5.4 respectively.

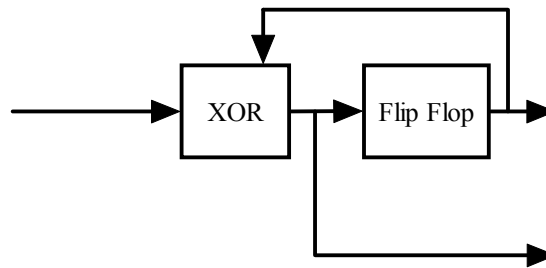


Figure 5.3: RSC 2-State Functional Description

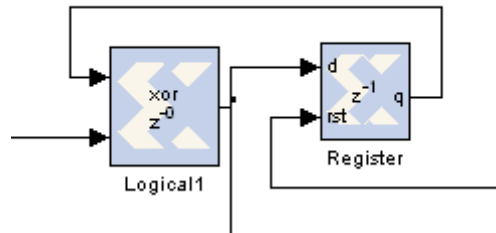


Figure 5.4: RSC 2-State Xilinx SG Implementation

Data is inputted to the XOR block and is sequentially shifted through the registers on every clock cycle. This allows the encoder to produce a parity bit on every clock cycle.

The RSC structure is directly dictated by the number of states in the encoder. For example, for the two state encoder implemented as part of this research, a single register is needed. In order to implement a four state encoder, however, two registers would be

needed. In general, the number of states required is related to the number of registers to the power of two or $\#States = 2^{\#Registers}$.

5.1.2 Random Interleaver

A random interleaver is required for the second encoder of the PCCC encoding process. A frame of data is input to the interleaver where it pseudo randomly changes the position of the bits in the frame. In order to implement this interleaver, a data buffer and random number generator is required [19, 20]. This implementation of the pseudo random number generator utilizes a linear feedback shift register (LFSR) which is a sequence of registers that are interconnected and initialized to a particular state. The LFSR then pseudo-randomly cycles through all possible states except '0' on each clock cycle. The LFSR functional description and Xilinx SG hardware implementation as part of this research is shown in Figures 5.5 and Figure 5.6 respectively.

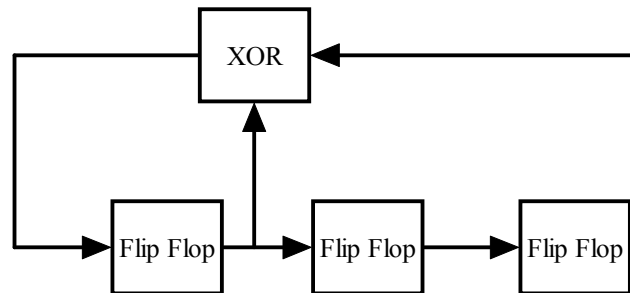


Figure 5.5: LFSR Functional Description

In one example, for a frame size of eight bits, a three state LFSR is chosen and arbitrarily initialized to a binary state. In the FPGA hardware, the pseudo random number generator is implemented with a Xilinx SG LFSR token that has parameters for the number of states and the initial seed value. Furthermore, the external hardware

includes a multiplexer, a constant and concatenation token which provide the all zero state which the LFSR cannot produce. This structure is shown in Figure 5.6.

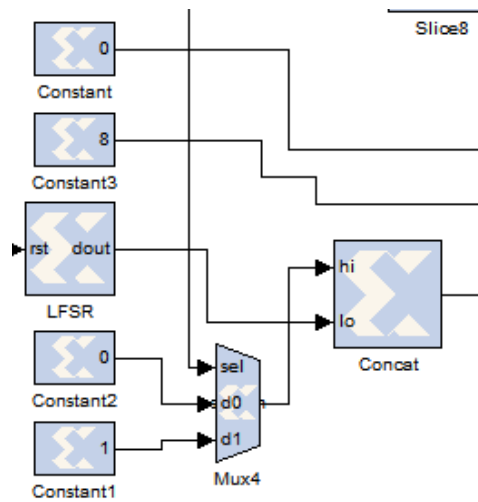


Figure 5.6: LFSR Xilinx SG Implementation

The fixed binary constants are concatenated with the 3-bit LFSR state output in order to produce a 16 state address. Output of the concatenation block is the direct address of a block RAM module that is used for data buffering. The random number generator LFSR connected to a dual port RAM block, which is the full structure of the PCCC random interleaver, is shown in Figure 5.7.

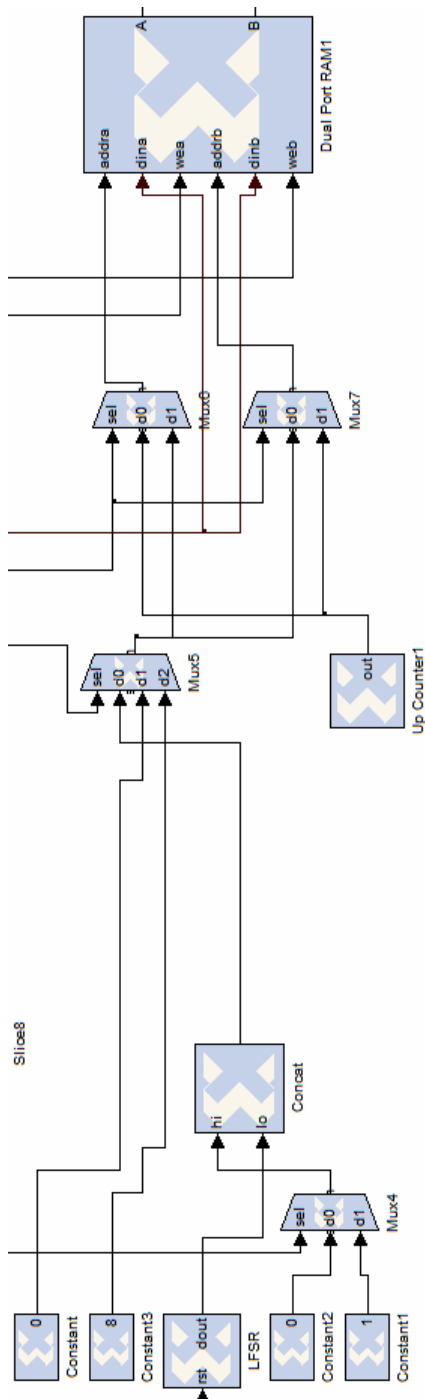


Figure 5.7: Random Interleaver Xilinx SG Implementation

The random interleaver works as follows: first the data frame is written to RAM in sequential order by a 4-bit up counter, then the next data frame is written to RAM in sequential order while the previous data frame is simultaneously read from RAM in a

pseudo-random order. A single dual port RAM on the Xilinx Virtex 4 FPGA is able to implement a random interleaver of up to 18 Kb. If the frame exceeds 18 Kb, then more than one block of RAM is required. This implementation provides a freely flowing pipeline.

In order to synchronize the encoders to an all-zero state after each frame, control logic is required. A counter is utilized to reset the state of the first and second encoders after each frame is complete. In this example, the frame is eight bits, and thus the eight bits require eight clock cycles to pass through the encoders. Therefore, the encoders are reset every eight clock cycles to provide an all-zero state for the next frame in the sequence. The control logic functional description and Xilinx SG hardware implementation as part of this research is shown in Figure 5.8 and Figure 5.9 respectively.

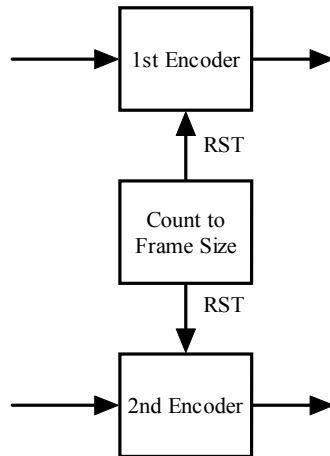


Figure 5.8: Control Logic Functional Description

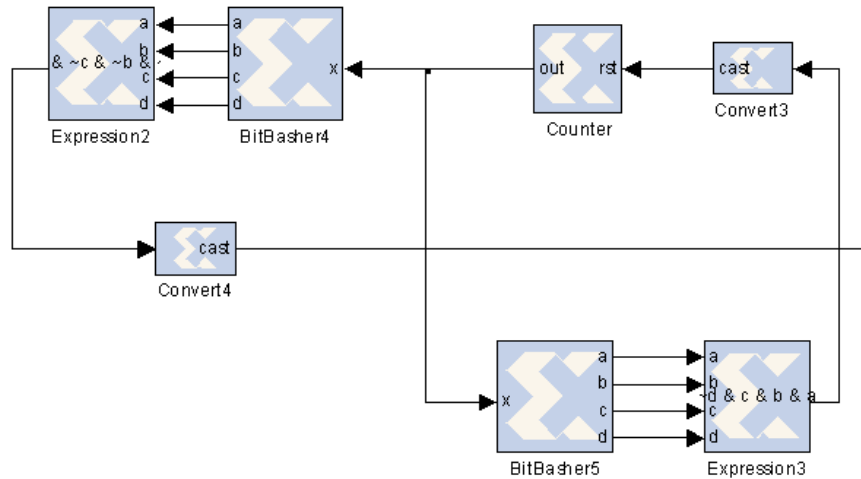


Figure 5.9: Control Logic Xilinx SG Implementation

5.2 Gaussian Channel

The Xilinx SG has a digital communications library that has hardware tokens which implement various channel models. The Gaussian token generates pseudo random white noise by utilizing a combination of the Box-Muller algorithm and the Central Limit Theorem [42]. The Xilinx token accepts inputs of binary 1s and 0s, and then transforms them to the simple two's complement numbers +1s and -1s to simulate baseband BPSK modulation. The token furthermore accepts a signal to noise ratio (SNR) constant which is used to scale the variance of the noise that is added to the modulated symbols. The precision of the output is a two's complement number consisting of 17 total bits with 11 fractional bits. With the implementation of this Gaussian token, the PCCC system has a suitable channel which can provide a noise corrupted bit to the receiver on every clock cycle.

5.3 MAP Decoder

As previously discussed in Chapter 2.10, the PCCC decoder is traditionally defined by two MAP decoders which calculate and communicate soft bit decisions in an iterative process [38]. After a set number of iterations, the PCCC decoder makes a hard decision on each soft data bit. The following Chapters discuss the hardware implementation of the MAP process. The overall PCCC decoder is discussed in Chapter 5.4.

The MAP decoder consists of the following steps [17]: calculate the distance metric γ ; calculate the forward recursion α ; calculate the backward recursion β ; calculate the log-likelihood function Λ ; and perform a hard decision based on the log-likelihood function. These MAP calculations, however, are performed over all possible trellis states which are independent with respect to one another [38]. Due to this independence, each state can be computed in parallel. Furthermore α and β are independent of one another and can also be computed in parallel, thus increasing the throughput of the system. The functional implementation of the MAP decoder as part of this research is shown in Figure 5.10 where N is the number of encoder states and four Gamma states are computed for a rate 1/3 encoder [14].

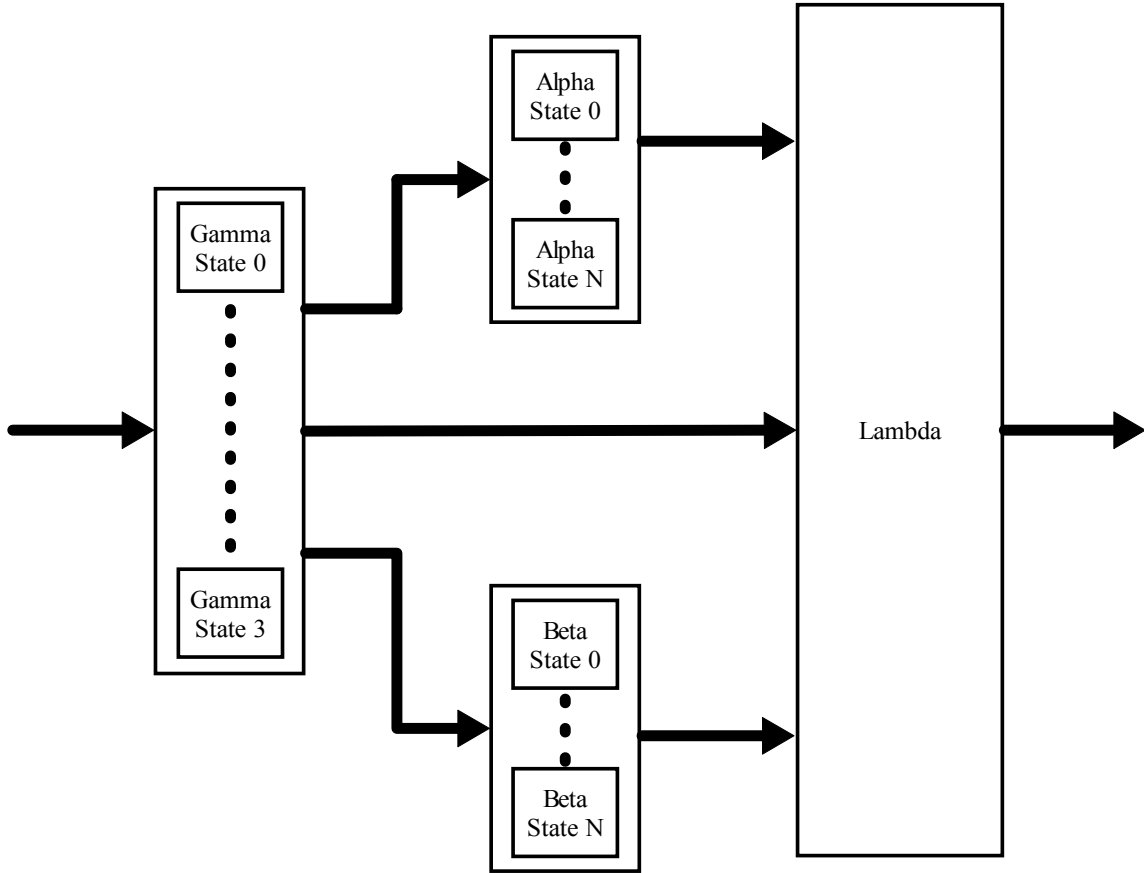


Figure 5.10: MAP Decoder Functional Description (rate 1/3 encoder)

Hardware implementations of the parameters γ, α, β and Λ are discussed in detail in Chapters 5.3.2, 5.3.3, 5.3.4 and 5.3.6 respectively. In this research, the MAP decoder was designed to assume a two-state RSC encoder.

5.3.1 Transcendental Functions

The major concerns due to their complexity in MAP decoding are the implementation of the exponential, the natural logarithm and the division function. It is important that all possible computational algorithms be explored to determine the most efficient in FPGA resources and execution. Three basic algorithms are evaluated as part of this research for exponentiation and natural logarithms: multiplicative/additive (MN,

AN) normalization, the CORDIC algorithm and polynomial representations [26]. Furthermore, fundamental division algorithms and division by convergence is assessed [26].

Due to their iterative nature the MN, AN, and CORDIC algorithms are implemented in a cascaded manner so that an apparent bottleneck in the pipeline does not occur. For example, the cascaded translation of a three step iterative process is shown in Figure 5.11. In a normal iterative calculation, the input data must be interrupted or buffered while the iterations are being completed. With a cascaded structure, the data flow does not have to be interrupted and thus the system throughput is increased.

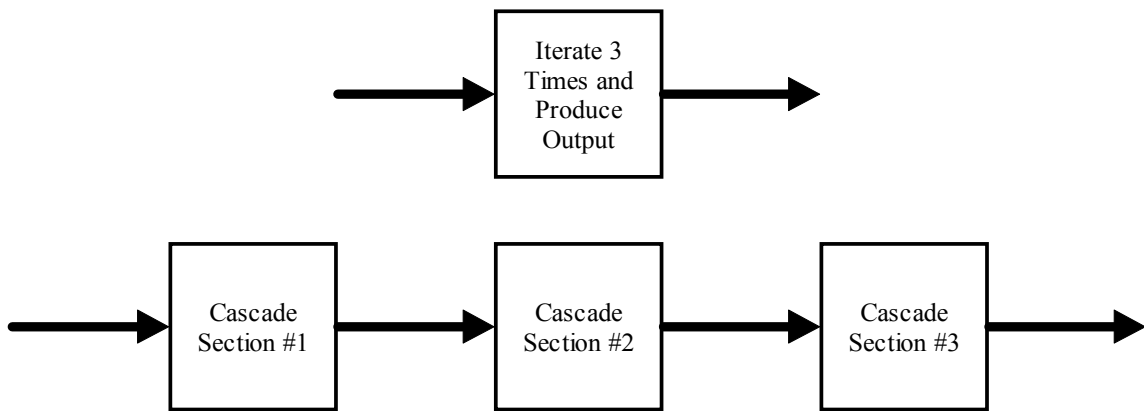


Figure 5.11: Traditional Iteration versus Cascaded Iteration

5.3.1.1 Multiplicative/Additive Normalization

Multiplicative normalization (MN) is an iterative process where one of two coupled equations converges to a desired value [26]. Additive normalization (AN) is a similar process that utilizes addition rather than multiplication. Natural logarithms can be computed using (MN) by the following process in Equation 5.1 [26].

$$\begin{aligned}
x^{(i+1)} &= x^i(1 + d_i 2^{-i}) & d_i \in \{-1, 0, 1\} \\
y^{(i+1)} &= y^i - \ln(1 + d_i 2^{-i}) \\
x^{(0)} &= x \\
y^{(0)} &= y \\
\ln(1 + d_i 2^{-i}) & \text{ is read from a table}
\end{aligned} \tag{5.1}$$

The value of d_i is chosen so that $x^{(m)}$ converges to 1 and in response $y^{(m)}$ converges to $y + \ln(x)$. If the initial value of $y^{(0)}$ is chosen as 0 then the process converges directly to $\ln(x)$. Exponentiation using (AN) is shown in the process in Equation 5.2 [26].

$$\begin{aligned}
x^{(i+1)} &= x^i - \ln(1 + d_i 2^{-i}) \\
y^{(i+1)} &= y^i(1 + d_i 2^{-i}) & d_i \in \{-1, 0, 1\} \\
x^{(0)} &= x \\
y^{(0)} &= y \\
\ln(1 + d_i 2^{-i}) & \text{ is read from a table}
\end{aligned} \tag{5.2}$$

The value of d_i is chosen so that $x^{(m)}$ converges to 0 and in response $y^{(m)}$ converges to ye^x . If the initial value of $y^{(0)}$ is chosen as 1 then the process converges directly to e^x .

However, in order to obtain an accurate convergence, the AN algorithm requires numerous iterations, each of which introduces delay into the hardware pipeline. Thus, the faster converging MN is relied upon in the Turbo Decoder of this research and not AN.

5.3.1.2 CORDIC Algorithm

The CORDIC algorithm convergence is based on a unit vector with end point equal $(x, y) = (1, 0)$ being rotated by an angle z . Therefore the new endpoint is $(x, y) = (\cos(z), \sin(z))$ [26]. Thus other trigonometric functions can be found in terms of $\cos(z)$ and $\sin(z)$. Shown in Equation 5.3 are the iterative process used in the CORDIC algorithm.

$$\begin{aligned}x^{(i+1)} &= x^i - d_i y^i 2^{-i} \\y^{(i+1)} &= y^i + d_i x^i 2^{-i} \\z^{(i+1)} &= z^i - d_i \tan^{-1}(2^{-i}) \quad d_i \in \{-1, 1\} \\ \tan^{-1}(2^{-i}) & \text{ is read from a table}\end{aligned}\tag{5.3}$$

To utilize these equations, $z^{(i)}$ is initialized to the desired angle. The next step is to choose $d_i = \text{sign}(z^{(i)})$ so that it performs a rotational convergence to the desired angle. After a certain number of iterations z converges to zero and (x, y) will converge to $(\cos(z), \sin(z))$. The natural log can be calculated by the approximation in Equation 5.4.

$$\ln(w) = 2 \tanh^{-1} \left| \frac{w-1}{w+1} \right|\tag{5.4}$$

However, in order to obtain an accurate convergence, the CORDIC algorithm requires numerous rotations, each of which introduces delay into the hardware pipeline. Thus, CORDIC is not relied upon in the Turbo Decoder of this research.

5.3.1.3 Polynomial Approximation

Functions can be approximated by polynomials such as the Taylor-Maclaurin series expansions [26]. The expansions in Equation 5.5 can be used to approximate the exponential and the natural logarithm.

$$e^x \approx 1 + \frac{1}{1!}x + \frac{1}{2!}x^2 + \frac{1}{3!}x^3 + \dots \quad (5.5)$$

$$\ln(x) \approx -y - \frac{1}{2}y^2 - \frac{1}{3}y^3 - \dots \quad \text{where } y = 1 - x$$

5.3.1.4 Division

Fundamental hardware division is performed using a shift and addition process [26]. This technique results in a system bottleneck due to the number of shifts and additions necessary to compute such a quotient. Division by convergence is an alternative technique which can be accomplished utilizing several unique methods. The first method is division by repeated multiplications [26]. This is accomplished by repeatedly multiplying the divisor and dividend by a sequence of multipliers. This convergence technique is given in Equation 5.6.

$$q = \frac{z}{d} = \frac{zx^{(0)}x^{(1)}\dots x^{(m)}}{dx^{(0)}x^{(1)}\dots x^{(m)}} \quad (5.6)$$

$$d^{(i+1)} = d^{(i)}(2 - d^{(i)})$$

$$z^{(i+1)} = z^{(i)}(2 - d^{(i)})$$

Iterations are performed until the divisor converges to 1. Therefore, if the divisor converges to 1 then the dividend converges to the desired quotient.

The second scheme is division by reciprocation. In this scheme the reciprocal of the divisor is found using the Newton-Raphson iteration [26]. Once the reciprocal of the divisor is found it can then be multiplied by the dividend to obtain the quotient. If repeated division by the same divisor is required, then this algorithm is an efficient method.

5.3.2 Distance Metric

The MAP decoding process starts with the calculation of the distance metric γ . Each MAP decoder receives a noise corrupted symbol consisting of a parity bit and data bit. This metric computes the distance of the noise corrupted symbols to the possible modulated symbols in the encoder trellis for all branches [38]. This distance metric is calculated for all received symbols in the data frame which makes it possible to determine a path through the trellis that has the smallest cumulative Euclidian distance, as shown in Equation 5.7.

$$\begin{aligned} & \text{for } t = 1, 2, \dots, \tau \quad \text{and} \quad \ell = 0, 1, \dots, \text{LastState} \\ \gamma_t^i(\ell', \ell) &= p_t(i) \exp\left(\frac{-d^2(r_t, x_t)}{2\sigma^2}\right) \text{ for } i = 0, 1 \\ & \text{where } p_t(i) \text{ is the apriori probability of each bit} \\ & \text{and } d^2(r_t, x_t) \text{ is the squared Euclidean distance between} \\ & \text{the received symbol } r_t \text{ and trellis symbol } x_t \end{aligned} \quad (5.7)$$

Evaluating the argument of the exponential in Equation 5.7 requires a Euclidian distance calculator and one multiplication in order to normalize the value with respect to the channel noise variance. As part of this research it is assumed that the receiver knows the noise variance of the channel at any given time and that the initial a-priori probability of each binary bit is 0.5. The squared Euclidian distance can be calculated between symbols consisting of two bits by Equation 5.8.

$$\begin{aligned} & [x_1, x_2] \quad [r_1, r_2] \\ & ((x_1 - r_1)^2 + (x_2 - r_2)^2) \left(\frac{1}{2\sigma^2} \right) \end{aligned} \quad (5.8)$$

The functional and Xilinx SG hardware implementation of the squared Euclidian distance computation is shown in Figure 5.12 and Figure 5.13 [14].

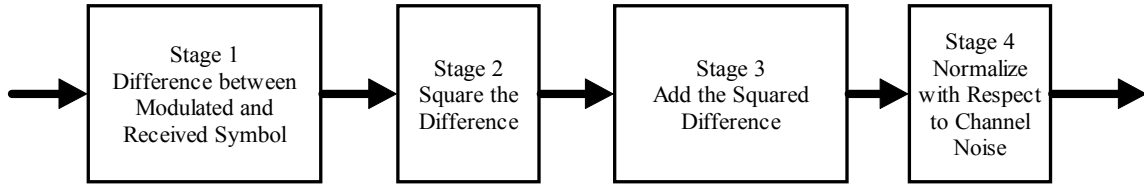


Figure 5.12: Euclidian Distance Functional Description

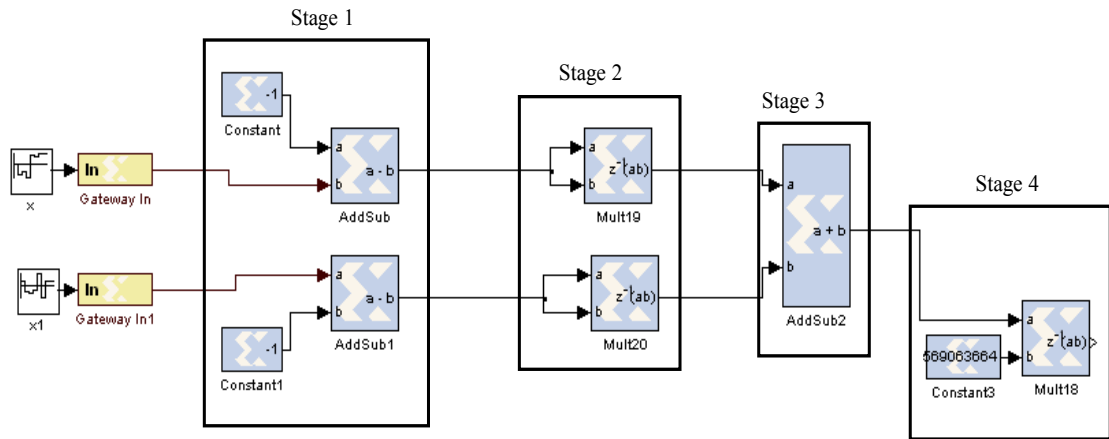


Figure 5.13: Euclidian Distance for Symbol (-1, -1) Xilinx SG Implementation

Stage 1 is the element by element subtraction of the ideal modulated symbol and received noise corrupted symbol (-1, -1). Stage 2 squares the output of the difference utilizing two multipliers. Stage 3 then provides the addition of the squared difference values. Stage 4 is a multiplication that normalizes the squared Euclidian distance to the reciprocal of twice the channel variance. Note that since the channel noise variance is constant it can be implemented as a multiplication of its reciprocal rather than actual division. This is important because division is inefficient to implement in hardware. It is also shown that the Stage 1 and Stage 2 implement parallel computations, and thus each of the four possible received symbols can be computed independently [14]. Thus, the other three possible transmitted symbols (1, 1), (-1, 1) and (1, -1) are similarly computed.

5.3.2.1 Exponentiation by Indirect Lookup Table

Exponentiation is the final and most computationally intensive step of the distance metric calculation. Evaluating such functions is not an easy task to perform in hardware. As part of this research implementation of exponentiation is performed using an indirect look-up table (LUT) [14]. An indirect LUT performs a specific computation before and after the look-up, which provides some benefit. For example, a direct look-up table for 32 bit words would require 2^{32} entries. In one example, in order to reduce the amount of entries to a reasonable number, the following identity $e^{(A+B+C+D)} = e^A e^B e^C e^D$ is used. By breaking the 32-bit data word essentially in to four 8-bit words, it is possible to use four small parallel look-up tables of size 2^8 and three multiplications.

It is noted that the data word can be broken down into any number of smaller bit words at the expense of the number of multipliers required to perform the reconstruction. In general, smaller the data words decrease the size of the LUT, but increase the number of multipliers that are required [14]. The functional and Xilinx SG hardware implementations of this technique are shown in Figure 5.14 and Figure 5.15 respectively.

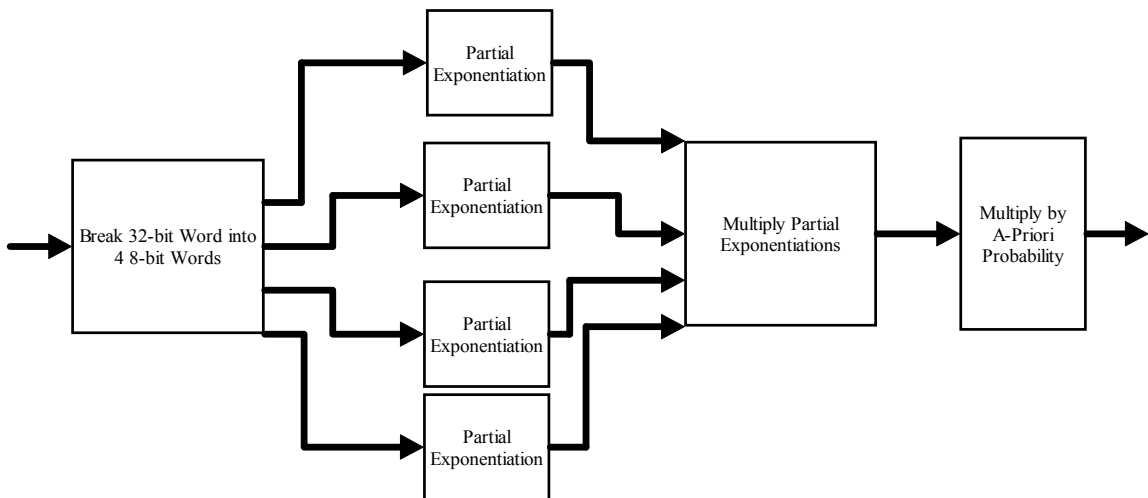


Figure 5.14: Exponentiation Functional Description

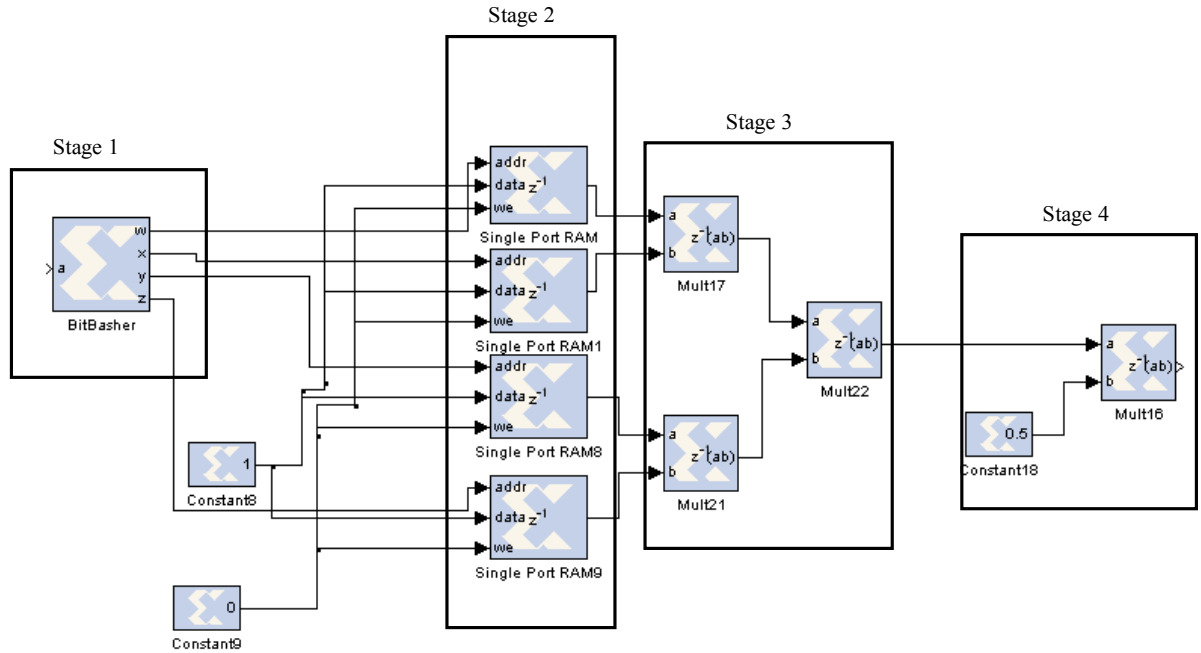


Figure 5.15: Exponentiation for Symbol (-1, -1) Xilinx SG Implementation

Stage 1 is a so-called bit-basher that splits the 32 bit word into four 8 bit words.

Stage 2 is four single port block RAMs that are utilized as small parallel LUT. Stage 3 is the multiplication of the LUT outputs by cascaded hardware multipliers. Finally, stage 4 is the multiplication of the exponentiation with the a-priori probability (initially 0.5). The four LUT's and two multipliers are designed to operate in parallel, thus further increasing the throughput of the system.

The other three symbols (1, 1), (-1, 1) and (1, -1) are independent from each other, and thus exponentiation for the other three symbols are similarly computed in parallel in an identical structure. This research design may be further configured for other bit word lengths by utilizing various configurations of parallel block RAMs, thus providing a general configurable design that can be optimized for specific system requirements [14].

In the LOG-MAP algorithm, this exponential step is avoided because by converting the processing to the log domain, the log and exponential functions essentially cancel each other out. Thus, in the LOG-MAP algorithm, the squared Euclidian distance computation of Equation 5.8 is the only computation that must be computed for the distance metric [38]. Therefore, the LOG-MAP decoder can also utilize the squared Euclidean distance hardware design as shown in Figure 5.13.

5.3.3 Forward Recursion

Forward recursion α is the second calculation in the MAP decoding process. It is essentially a calculation that starts at the beginning and then proceeds to the end of the trellis for the entire frame. As it proceeds through the trellis, the process utilizes the pre-calculated γ values at each branch node and the previous α value to calculate a weighted path [38]. This forward recursion is implemented using Equation 5.9.

$$\begin{aligned} & \text{for } t = 1, 2, \dots, \tau \quad \text{and} \quad \ell = 0, 1, \dots, \text{LastState} \\ \alpha_t(\ell) &= \sum_{\ell'}^{\text{LastState}-1} \sum_{i=(0,1)} \alpha_{t-1}(\ell') \gamma_i^i(\ell', \ell) \end{aligned} \quad (5.9)$$

The functional and Xilinx SG hardware implementation of α is shown in Figure 5.16 and Figure 5.17 respectively.

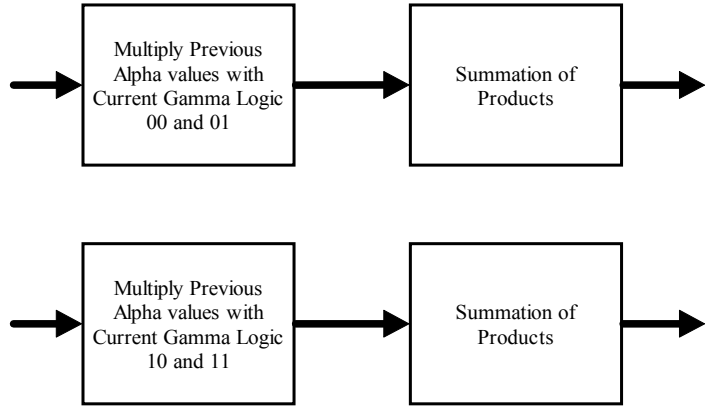


Figure 5.16: Forward Recursion Functional Description

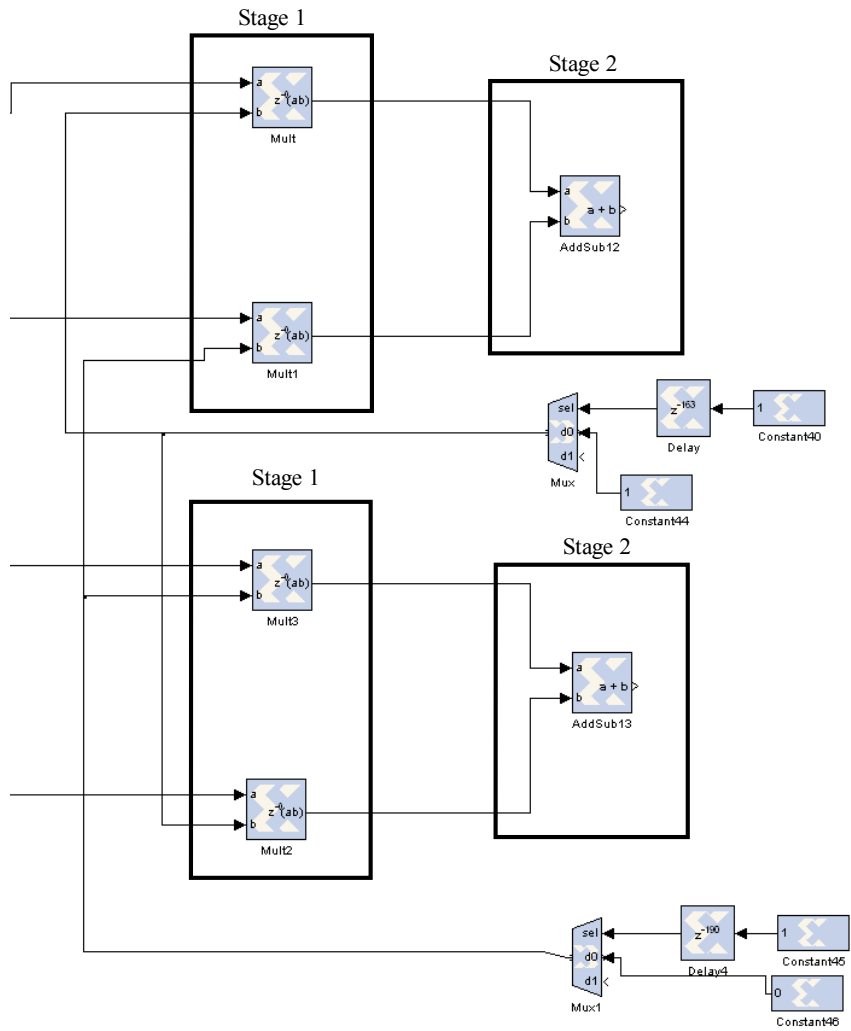


Figure 5.17: Forward Recursion Xilinx SG Implementation

Stage 1 is four parallel multipliers which multiply the previous α value with the present γ value for both a logic 0 and logic 1 state. Stage 2 is the parallel summation of the multiplier outputs and also shows the feedback path for the previous α value. It should be noted that as the number of states in the decoder grows, the throughput will not decrease due to the parallel architecture of the forward recursion (the states are computed independently). However, Stage 1 and stage 2 can both be expanded to independently compute the forward recursion for more states without increasing the computation time [14].

In the LOG-MAP algorithm, the multiplications in Figure 5.17 are replaced by additions. This replacement is possible because $\log(AB) = \log(A) + \log(B)$. Therefore, the hardware structure for the LOG-MAP forward recursion would be nearly identical to that shown in Figure 5.17 with the exception of replacing the multipliers with adders and the addition of a maximum function to estimate the logarithm of the exponentials. This replacement is beneficial because hardware adders and maximum operations are less expensive in terms of hardware than multipliers.

5.3.4 Backward Recursion

Backward recursion β is the third calculation in the MAP decoding process. It is a calculation that starts from the end and then works its way to the beginning of the trellis. As it traverses backwards through the trellis the process utilizes the future γ values at each branch and future β values to calculate a weighted path. Since β starts from the end of the trellis it must wait for the entire data frame to be received in order for

the calculation to commence. This aspect reduces the throughput of the decoder significantly. Backward recursion is implemented with Equation 5.10.

$$\begin{aligned} & \text{for } t = \tau - 1, \dots, 1, 0 \quad \text{and} \quad \ell = 0, 1, \dots, LastState \\ \beta_t(\ell) &= \sum_{\ell'}^{LastState-1} \sum_{i=(0,1)} \beta_{t+1}(\ell') \gamma_{t+1}^i(\ell, \ell') \end{aligned} \quad (5.10)$$

This calculation is equivalent in hardware to that of the forward recursion. The only difference is the addition of dual port RAM first-in-last-out (FILO) buffer which is written to using an up-counter and read from using a down-counter. This FILO ensures that the data frame is reversed for the calculation. Once the data frame is reversed by the FILO, β can be calculated using the same structure as the α process.

After the calculation, however, the data must be sent through another identical FILO buffer to reverse the frame back to its original order. The FILO buffer is implemented utilizing a dual port block RAM and up/down counters. However, similar to forward recursion, backward recursion can also be expanded to independently compute the metrics for more states if necessary, thus increasing the throughput.

The backward recursion computation impacts the latency of the decoder. This is due to the frame having to be reversed and then re-reversed in time. Thus, the backward recursion adds a delay equivalent to two frame sizes. In this example, the backward recursion would have a latency of 128K clock cycles.

In the LOG-MAP algorithm, the multiplications are replaced by additions and maximum operations, similar to the forward recursion as described in Chapter 5.3.3. The LOG-MAP decoder, however, also requires the BRAM FILO structure to flip and re-flip the data frame in the same manner as the MAP decoder. The only difference between the

MAP and LOG-MAP backward recursion are the replacement of the multipliers with the adders and maximum operations.

5.3.5 Recursive Normalization

In general, forward and backward recursions are simple calculations to implement in FPGA hardware (a simple addition and multiplication). Both of these calculations, however, are inherently under-flowing [38]. As the number of states and the number of iterations grows, the forward and backward recursion values tend toward zero. This under-flowing is a concern because this hardware implementation only has a fixed number of fractional bit accuracy [14]. Therefore, these values must be normalized in some manner.

Both recursion values are utilized in the calculation of the log-likelihood function which is inherently a ratio. Therefore, normalization of the recursion values on a symbol by symbol basis is possible. As part of this research, the recursion values are normalized to the range 0-1 after each symbol is calculated. In order to normalize the values, the maximum forward and backward recursion values are determined. The recursion values are then shifted an amount equivalent to the most significant 1-bit of the larger number needed to be in the most significant fractional position [14]. The functional implementation of this normalizing procedure is shown in Figure 5.18.

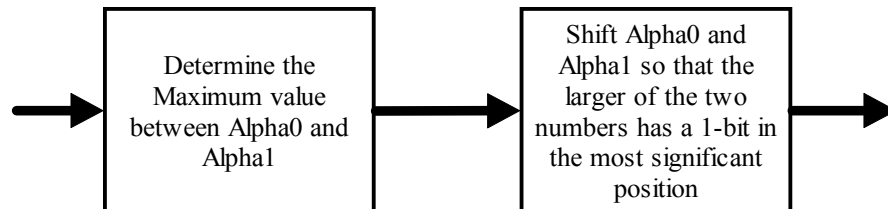


Figure 5.18: Recursive Normalization Functional Description

Both forward and backward recursion calculations require the previous output of the multiplier in order to calculate the present output. If normalizing is a time consuming procedure which introduces excess delay in the feed-back loop, well known data dependency issues will arise. Furthermore, if normalization is performed by dividing by the maximum value, the system will have to be up-sampled by the latency of the feedback loop which greatly reduces the system throughput [12, 26]. Performing normalization by detecting the maximum value and then shifting appropriately, allows recursive normalization to be realized in a single clock cycle, thereby not adversely affecting the throughput of the system.

In the LOG-MAP algorithm recursive normalization is avoided [38]. Normalization is not needed in the LOG-MAP algorithm because the small numbers that underflow in the MAP algorithm due to multiplication, would actually increase due to the additions. Thus, the exclusion of normalization hardware structures is a major benefit to utilizing the LOG-MAP decoder rather than the MAP decoder.

5.3.6 Log-Likelihood Ratio

The log-likelihood ratio (LLR) is the soft estimate MAP decoder output and is the next step in decoding is the ratio Λ . The natural logarithm of a ratio using the γ , α and β values is calculated with Equation 5.11.

$$\begin{aligned}
& \text{for } t = 1, 2, \dots, \tau - 1 \\
\Lambda(c_t) &= \log \frac{\sum_{\ell=0}^{LastState-1} \alpha_{t-1}(\ell) \gamma_t^1(\ell', \ell) \beta_t(\ell)}{\sum_{\ell=0}^{LastState-1} \alpha_{t-1}(\ell) \gamma_t^0(\ell', \ell) \beta_t(\ell)} \quad (5.11)
\end{aligned}$$

After Λ is calculated, a hard decision is made with the threshold set at zero.

Positive values are decided as logic 1 and negative values as logic 0. The log-likelihood ratio process consists of three steps. The first step in Equation 5.11 is the multiplication and summation of the distance metric, the forward recursion and the backward recursion. The only difference between the numerator and denominator terms in Equation 5.11 is that the distance metric is calculated with respect to either a logic 0 input or logic 1 input.

Implementation of the functional and Xilinx SG hardware implementation of this multiplication/summation procedure is shown in Figures 5.19 and 5.20. It should be noted that the multiplications and summations for the logic 0 and 1 are independent and thus computed in parallel [14].

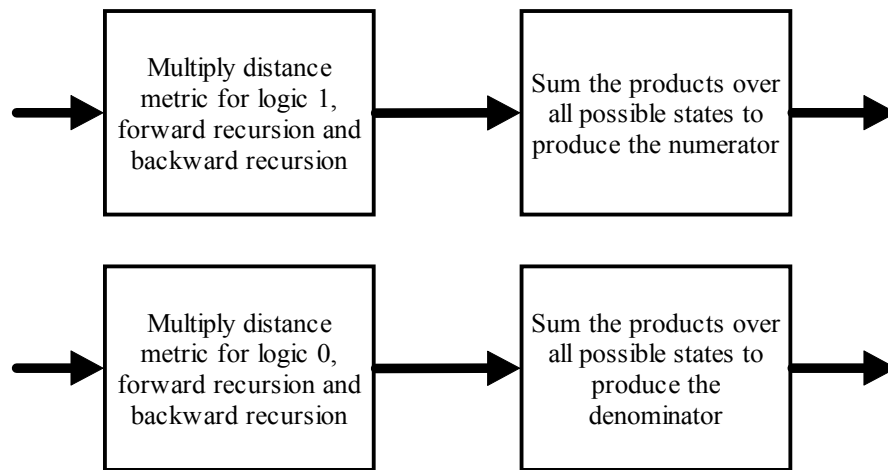


Figure 5.19: Log-Likelihood Functional Description

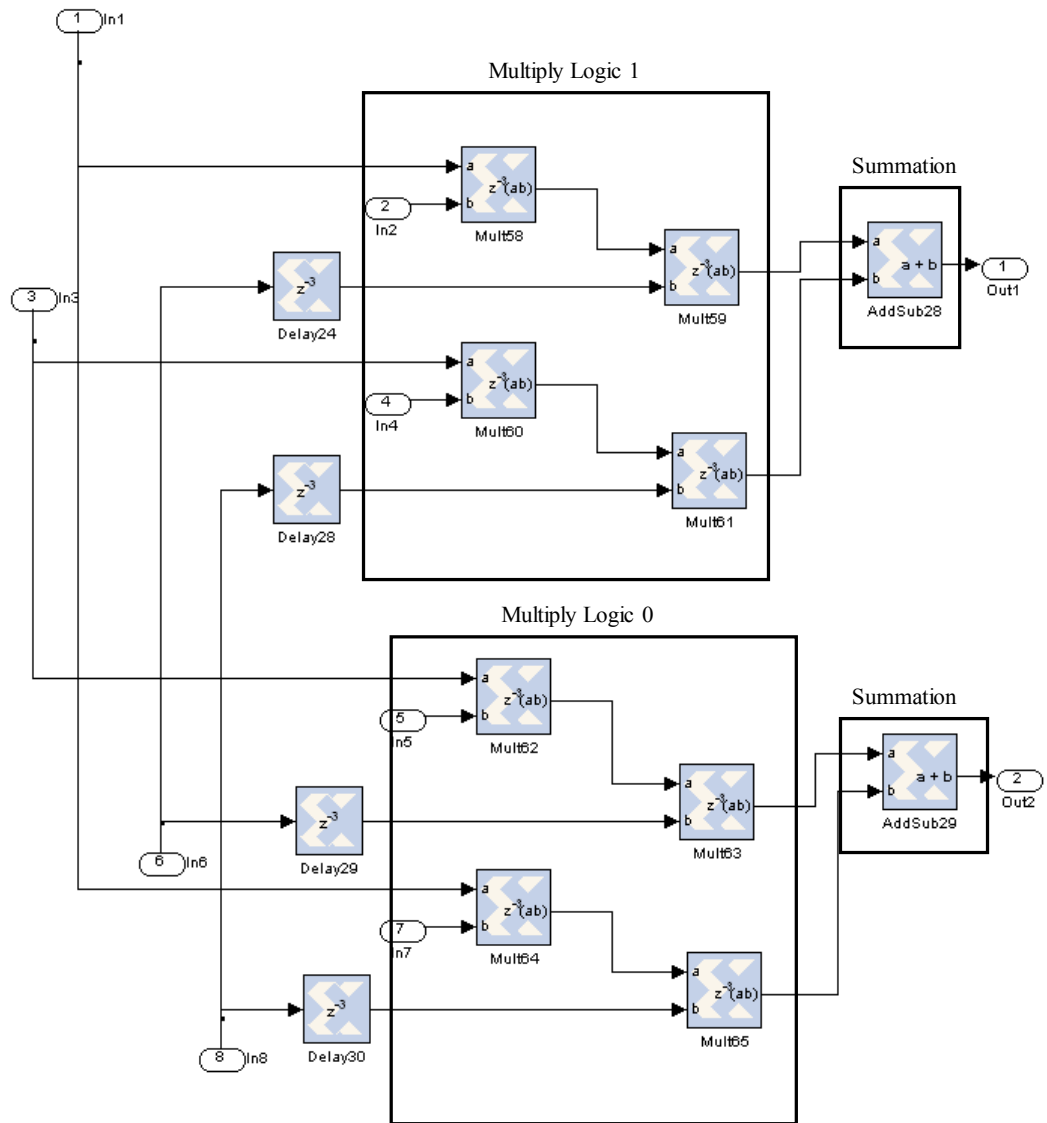


Figure 5.20: Log-Likelihood Xilinx SG Implementation

The second step of the LLR is more computationally intensive than the first step. In an initial implementation, the log-likelihood ratio may be computed by a division of the numerator and denominator obtained in the first step, followed by a logarithm of the ratio [14]. This initial implementation, however, has a possible fault in that the division

may produce either a very large or a very small number. This is due to the fact that the numerator and denominator of the ratio is the probability that the bit is a logic one and logic zero respectively.

For a 32 bit example, the ratio may be $0.999/2^{-32}$, which results in a very large number $4.29e9$ that would require 32 integer bits to represent. In another example, the ratio may be $2^{-32}/0.999$, which results in a very miniscule number $2.33e-10$ that would require 32 fractional bits to represent. Thus this extreme range requires a large number of integer as well as fractional bits which may be unwarranted in some circumstances.

An alternative to this initial implementation is a second implementation that is more efficient in systems with a fixed number of bits. The second implementation takes advantage of the identity $\log(A/B) = \log(A) - \log(B)$. Therefore, in the second implementation, the sequential division and then logarithm of the first implementation is replaced by two logarithms and a subtraction. Furthermore, the two logarithmic calculations may be computed in parallel since they are independent.

A major advantage to the second implementation is that the extreme range required by the division in the initial implementation is avoided because the logarithm is computed separately for two numbers in the range 0-1. Shown in the following Chapters are the division and logarithm hardware designs that may be utilized in either the initial LLR implementation or the second LLR implementation.

5.3.6.1 Division by Reciprocation

Division by reciprocation is an iterative calculation that computes the reciprocal of a specified number d where $q=z/d$. The method is based on the Newton-Raphson

iteration to determine the root of a given function. For example, the function as described as $f(x) = 1/x - d$ leads to the recurrence as shown in Equation 5.12.

$$q = \frac{z}{d}$$

$$x^{(i+1)} = x^{(i)}(2 - x^{(i)}d^{(i)}) \quad (5.12)$$

for $d = [.5, 1)$ $x^{(0)} = 1.5$

In order to implement the reciprocation technique as shown in Equation 5.12, pre and post-processing must be performed. First, the denominator of the ratio must be scaled to the range [0.5, 1) in order for the process to converge in a minimum number of iterations [26]. Second, the process must go through at least 3 iterations to obtain a reasonable result. Third, the reciprocal must be re-scaled. Finally, the reciprocal must be multiplied by the numerator of the desired ratio. Implementation of the functional and Xilinx SG hardware implementation of this four step reciprocation procedure is shown in Figure 5.21 and Figure 5.22.

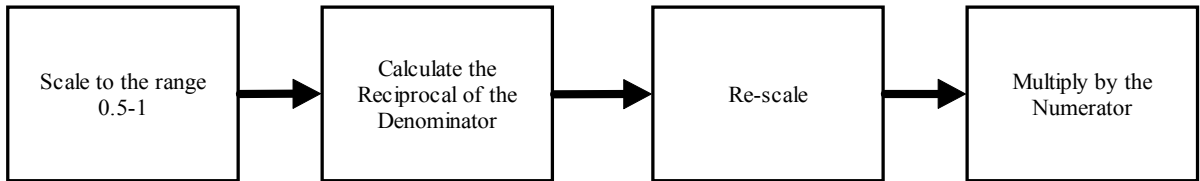


Figure 5.21: Division by Reciprocation Functional Description

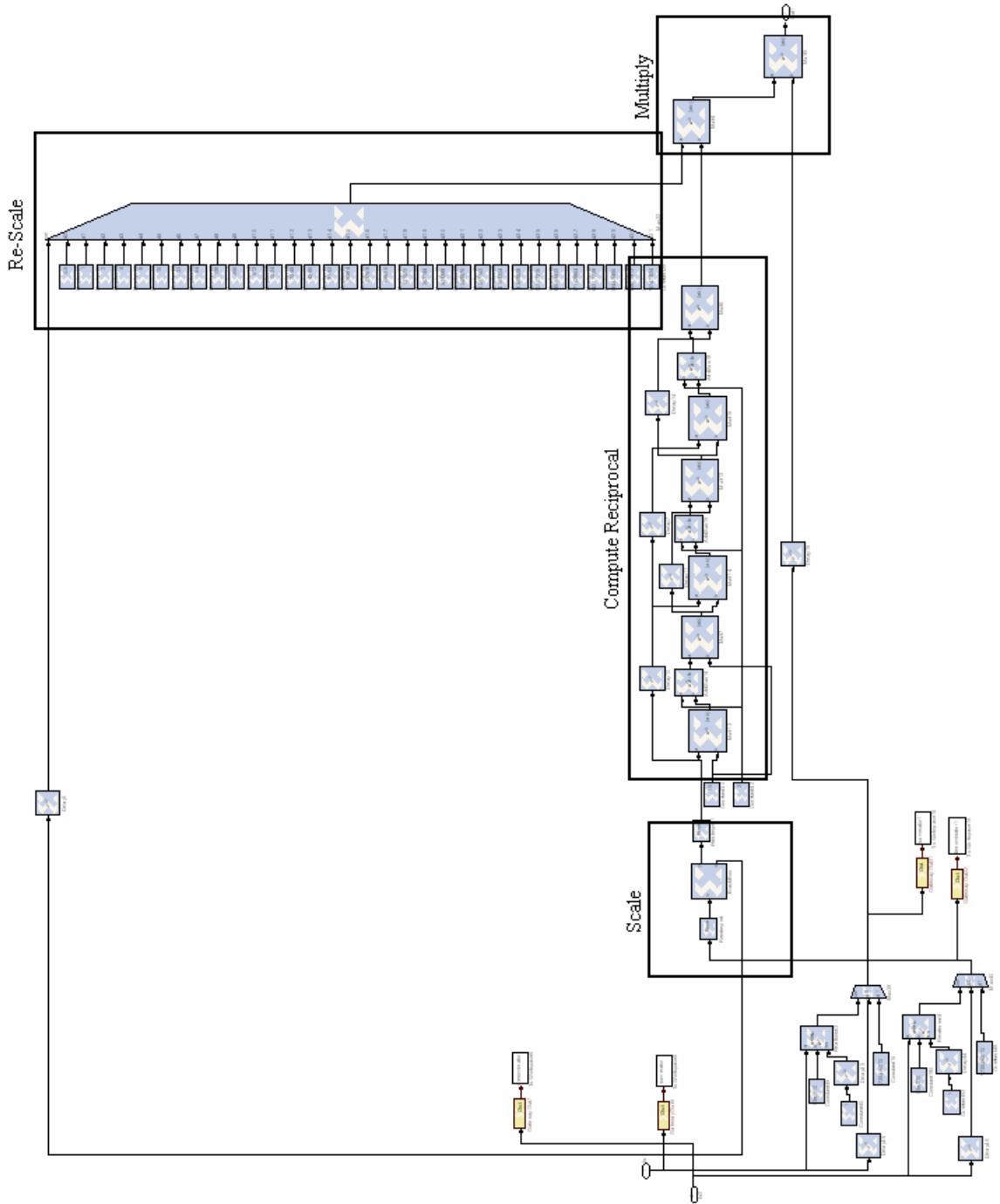


Figure 5.22: Division by Reciprocation Xilinx SG Implementation

Scaling the denominator in hardware is not an easy task. In order to scale the number in the range $[0.5, 1)$, the denominator bits are shifted until the most significant bit

is in the first fractional position. Since the first fractional position has a weight of 0.5 and there are no integer bits, it is possible to ensure that the denominator is at least 0.5 and less than 1. During the shifting process, the process must keep track of how many bit positions the denominator is shifted which is critical for rescaling after the iterations are complete. Once the denominator is scaled to the proper range, division by reciprocation can commence.

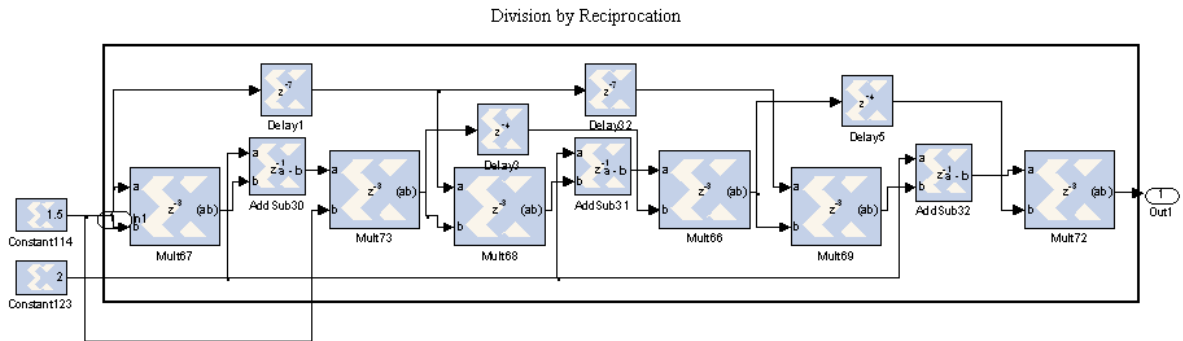


Figure 5.23: Division by Reciprocation Xilinx SG (Enlarged)

As shown in an enlarged section of Figure 5.23, a cascaded version of the reciprocation procedure is implemented. Cascading three such sections is equivalent to iterating three times while maintaining consistent pipeline flow. After the reciprocation is complete, the result must be rescaled.

For example, a denominator of 0.0056 is scaled by 100 in order to produce 0.56 which is in the range [0.5, 1). The reciprocal is then calculated as 1.78. In order to rescale properly, 1.78 is multiplied by 100 to produce the correct result of 178. In hardware re-scaling is a single multiplication of the reciprocation result and the bit value equivalent to the number of shifts recorded during scaling. After the reciprocation is computed, the result is multiplied by the numerator to complete the division.

Division by reciprocation for 18 bit word lengths utilizes eight embedded multipliers and produces an output in eight clock cycles, and continues to produce an output every clock cycle thereafter. The traditional shift and subtract division technique would take 18 clock cycles to produce an output [26]. Thus, division by convergence produces outputs faster than the traditional technique, at the expense of multiple hardware multipliers.

5.3.6.2 Logarithm by Taylor Series

Polynomial representation of a function is useful in hardware, because complex functions can be calculated with multiplications and additions. The Taylor series representation, as shown in Chapter 5.3.1.3, is chosen for the computation of the natural logarithm needed to compute the LLR. In order to compute the Taylor series, the following hardware pipeline must be implemented. First, the value must be scaled in the range $[0.5, 1)$ in order to obtain proper convergence of the series. Second, at least the first fourteen terms of the series is calculated in order to obtain an accurate result [14, 26]. Third, the result must be re-scaled. Finally, the terms of the series must be accumulated. The functional and Xilinx SG implementation of the Taylor series is shown in Figure 5.24 and 5.25 respectively.

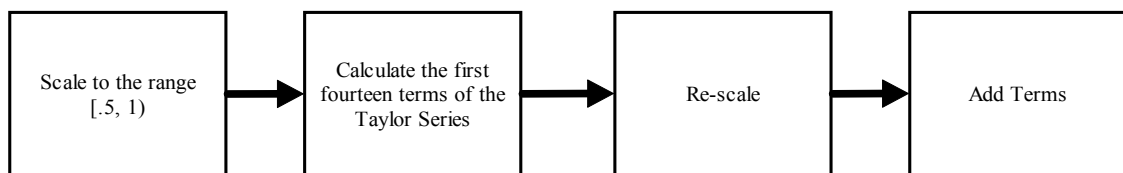


Figure 5.24: Logarithm by Taylor Series Functional Description

Scaling the value is identical to the scaling discussed in the division by reciprocation procedure in Chapter 5.3.6.1. The value is shifted so that the most significant bit is in the first fractional position. Calculation of the first eight terms of the series can take advantage of parallel processing. As shown in Figure 5.25, the third and fourth terms can be computed simultaneously as well as the fifth through eighth terms.

By computing the terms in parallel, the overall computation time is reduced. After the logarithm is approximated by the series, the result are rescaled. In order to rescale properly, the following identity $\log(AB) = \log(A) + \log(B)$ was utilized. For example, before computing the logarithm, the value A was scaled by scaling factor B. Therefore, the resulting logarithm is computed as $\log(A) = \log(AB) + \log(1/B)$. In hardware, $\log(1/B)$ is stored in a small 32 entry LUT for 32 bit words. The number of shifts during scaling is used to determine the correct entry in the LUT to utilize in rescaling and is accomplished by a single addition.

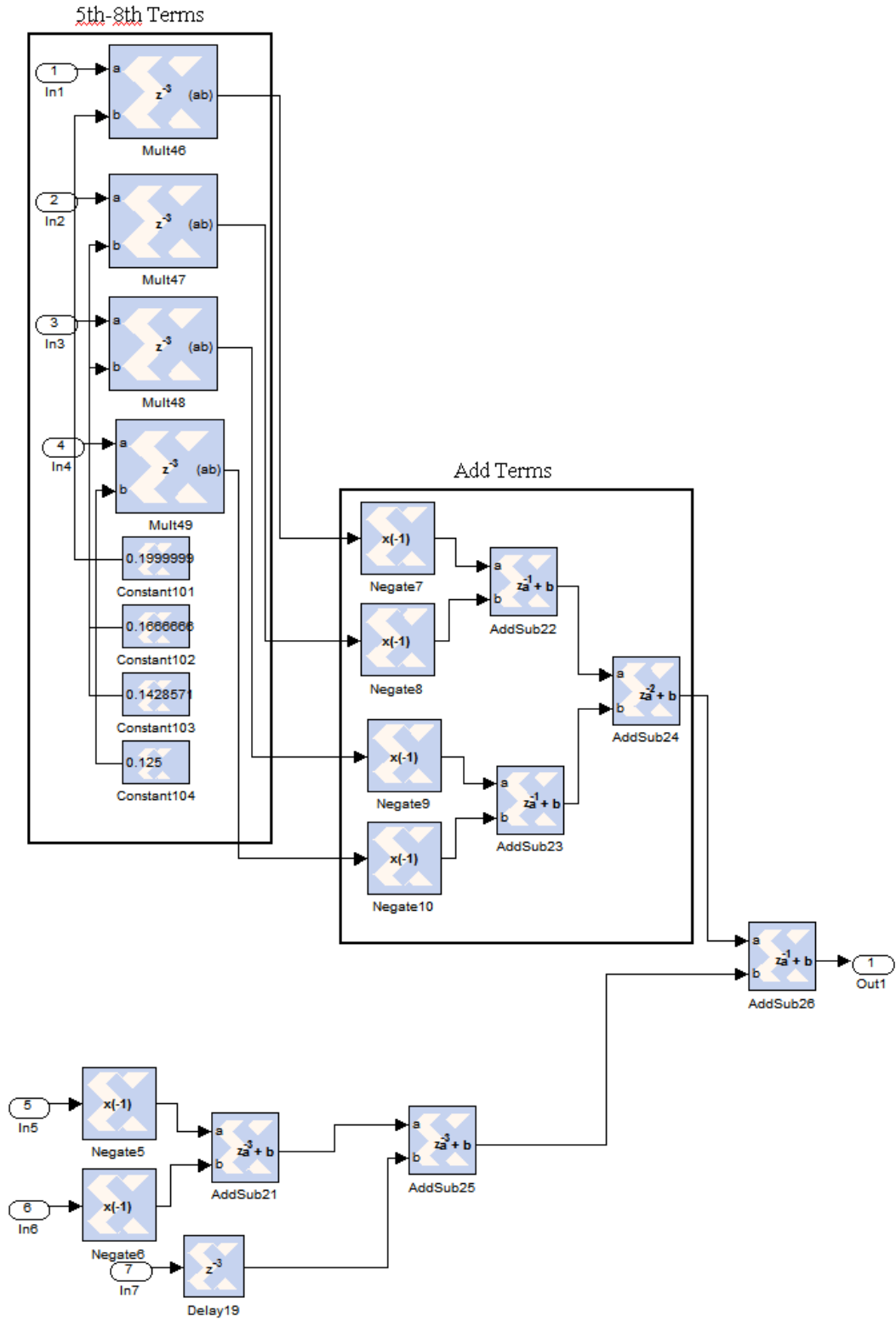


Figure 5.25: Logarithm by Taylor Series Xilinx SG Implementation

After computing the LLR, a hard decision is made in order to distinguish between a binary 0 and binary 1. This is accomplished by a single comparator with a threshold set at zero.

In the LOG-MAP algorithm, the logarithm itself is typically not computed, but rather a maximum operation corrected by a Jacobian function is utilized [4, 10]. The correction function is a recursive function that is typically estimated by a small LUT. This estimation is traditionally what separates the performance of the MAP algorithm and LOG-MAP algorithm [16, 38].

One solution would be to compute the Jacobian function with each iteration in order to improve the BER performance of the LOG-MAP decoder. However, the Jacobian is a complex function that contains multiple exponentiations and logarithms that may not yield an efficient hardware solution. Another solution is to compute the log-likelihood ratio LOG-MAP-LLR, rather than estimating it. By computing the LLR, the LOG-MAP-LLR decoder increases the BER performance without an extreme increase in hardware utilization. This solution can be performed as $\log(A/B)$ utilizing the logarithm hardware in Figure 5.25. Two identical logarithm calculators can perform $\log(A) - \log(B)$ between the numerator and denominator which as discussed earlier will maintain a small dynamic range easily representable by 18 bit words. Furthermore, the exponentiations required in the LOG-MAP-LLR can be computed by multiple exponentiation tables identical to those utilized in the distance metric of the MAP decoder.

Thus, a hybrid LOG-MAP-LLR design is asserted to be a good compromise terms of BER performance and hardware utilization. As discussed earlier, by converting to the log domain, the benefits are that: multiplications are replaced by additions, exponentiation in the distance metric, and normalization is not required because there is no underflow. Then, by computing the LLR rather than estimating it, the BER performance of the LOG-MAP decoder is improved. Computing the forward and backward recursion in the LOG-MAP decoder is not feasible due to large hardware utilization. The backward and forward recursion computations in the LOG-MAP decoder would require a logarithm and multiple exponentiations for each state of the encoder. For example, a 16 state encoder would require 32 logarithms to support the forward and backward recursion computations. This would be too costly in terms of utilizing hardware multipliers since each log calculator requires eight multipliers. This solution would only be feasible with a log LUT, at the expense of bit accuracy.

5.3.7 MAP Performance Evaluation

In order to evaluate the hardware performance of the MAP decoder, a Matlab simulation is utilized. The MAP decoder hardware BER is compared to the Matlab simulation BER in order to evaluate the inherent accuracy of the FPGA hardware implementation. A simulation is executed in Matlab for uncoded pulse amplitude modulation (PAM) and RSC PAM with a 2-state MAP decoder for both 18 and 32 bit word lengths. Different word lengths were chosen to provide a balance between hardware utilization and computational accuracy since improved accuracy requires more hardware resources.

In the first BER simulation, 18 bit word lengths were chosen to minimize hardware utilization. By utilizing 18 bit word lengths, a single hardware multiplier on the Virtex 4 FPGA is able to perform the 18x18 bit multiplications in just 1 clock cycle. Minimizing hardware utilization is important since many multiplications are computed throughout the decoder. However, accuracy decreases due to the decreased bit word lengths. Figure 5.26 compares the resulting Matlab BER with the hardware BER quantized to 18 bit words.

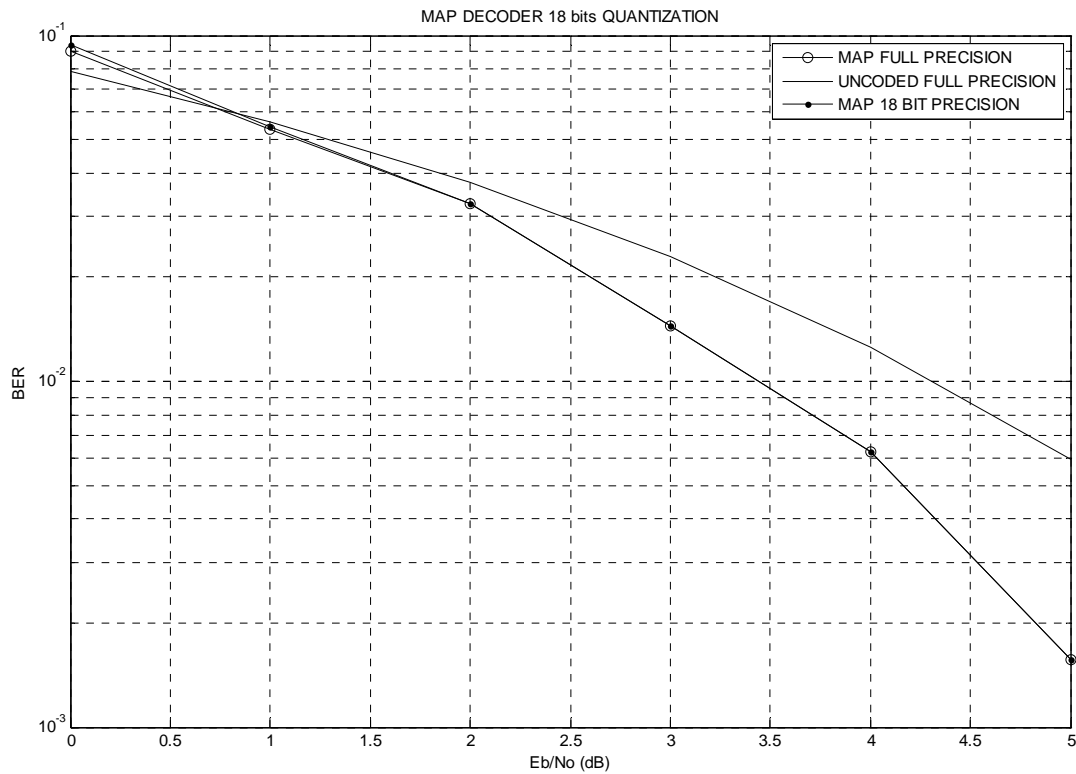


Figure 5.26: MAP BER 18 bit words

Figure 5.26 shows that the 18 bit words approximate the full precision BER with minimal degradation. However, general over many stages of decoder computation, the

accuracy of the fixed point quantized data gradually decreases. Therefore, it is important to minimize this degradation by increasing the word length.

In the second BER simulation, 32 bit word lengths were chosen to increase accuracy of the hardware computation. However, hardware utilization increases with increased word length. For example, in order to perform a 32x32 bit multiplication, four hardware multipliers in the Virtex 4 are required in a cascaded architecture. Figure 5.27 compares the Matlab BER with the hardware BER now quantized to a 32 bit word length.

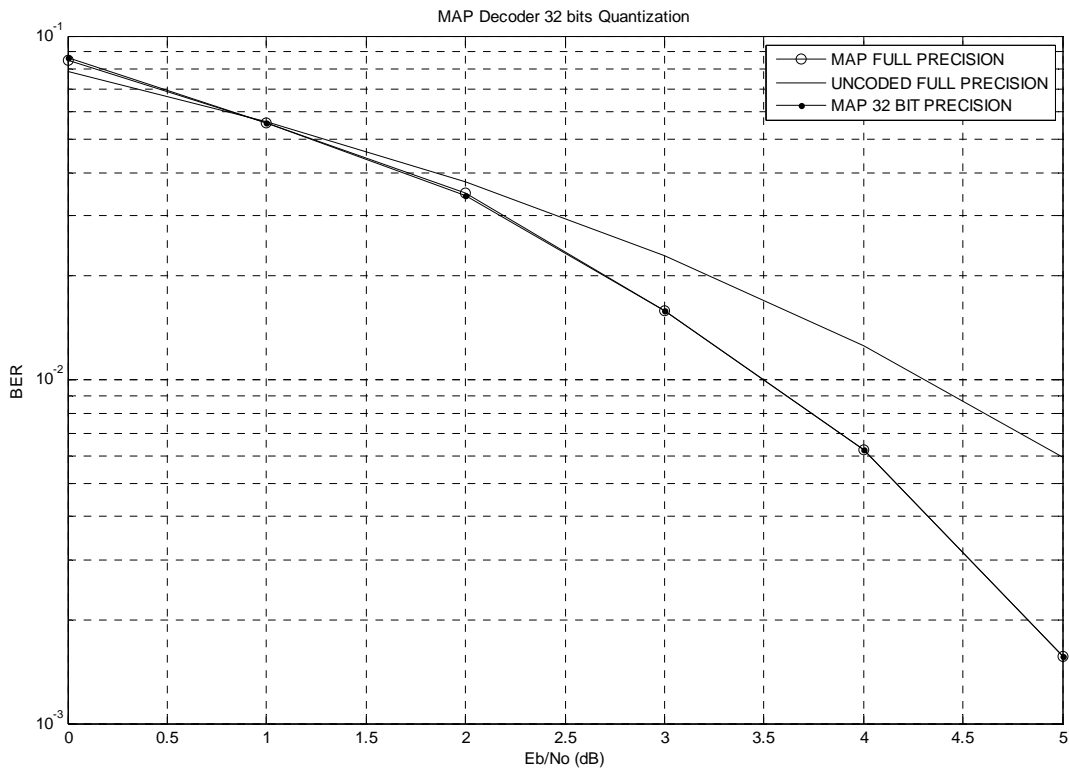


Figure 5.27: MAP BER 32 bit words

A comparison of Figure 5.26 and Figure 5.27 shows that the 32 bit word length data produces an even closer BER to the Matlab BER than the 18 bit word length data (especially at greater E_b/N_0) at the expense of increased hardware utilization. Thus, a

tradeoff is required between the accuracy and utilization of the hardware design. The hardware utilization for both of these MAP decoder designs is discussed in Chapter 5.4.

5.4 PCCC Decoder

As described in Chapter 2.11, PCCC decoding is an iterative decoder that utilizes two MAP decoders that work in concert to produce extrinsic information for updating the a-priori probability of a data bit 0 and 1. Therefore, constructing the PCCC decoder is accomplished by connecting two MAP decoders with interleavers and deinterleavers. The interleavers and deinterleavers communicate the extrinsic information to the appropriate MAP decoder.

An important aspect of the PCCC decoder design is maintaining a full FPGA hardware pipeline in order to maximize throughput. The PCCC decoder as part of this research is therefore designed in a manner to permit free flowing computations and utilize the hardware reasonably completely and efficiently.

Another important aspect of the iterative decoder is the need for intelligent stopping rules that control the process [10, 24]. Setting the iterative PCCC decoder to run for a fixed number of iterations is a rather primitive stopping rule that can produce decreased system performance. Thus, both primitive and more advanced stopping rules are evaluated and compared based on their BER performance and resource utilization.

5.4.1 Extrinsic Information

During the decoding process as discussed above, each MAP decoder produces a soft output known as the extrinsic information that is then utilized at each iteration as

feedback for the soft decision process. The computation of the extrinsic information is performed with a single multiplication and two subtractions. Specifically, the multiplication of the noise in the process is independent of the subtractions. Therefore, these two computations can be performed in parallel to increase system throughput. Implementation of the functional and Xilinx SG hardware implementation of the extrinsic information is shown in Figure 5.28 and Figure 5.29.

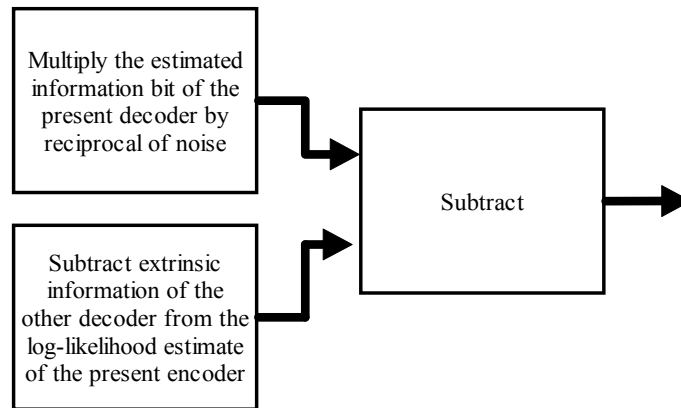


Figure 5.28: Extrinsic Information Functional Description

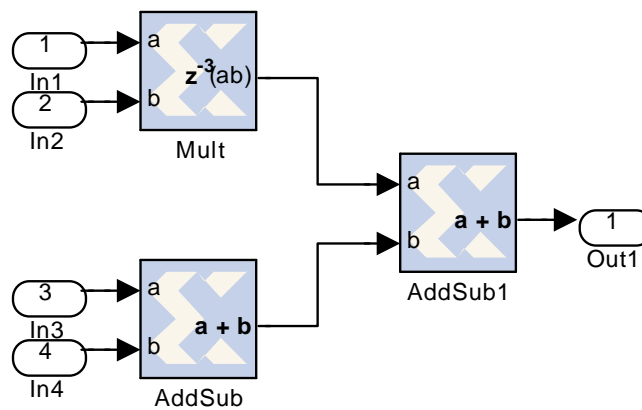


Figure 5.29: Extrinsic Information Xilinx SG Implementation

The computed extrinsic information is then interleaved or deinterleaved and passed to the next MAP decoder for updating the a-priori probability estimate (the interleaver structure is shown in Chapter 2.5). After each iteration, the a-priori-probability estimate becomes increasingly more accurate.

5.4.2 Stopping Rules

Turbo decoding is an iterative process that processes the received data until it is controlled to stop [8, 10, 24]. Although the estimate of the noise corrupted received symbol improves with each iteration.

However, each iteration adds to the system latency and overall hardware power consumption. The PCCC limiting factor then is the number of iterations that are required to produce a reliable estimate. A brief description of some basic stopping rules are as follows [31]:

- 1) Magic Genie: This rule assumes knowledge of the transmitted data. It stops at an iteration for which hard decision making will yield all correct symbols. This is not a practical system but rather used as a performance benchmark for other systems.
- 2) Fixed Number of Iterations: A predetermined number of iterations is always performed on every data frame. Typically this predetermined number of iterations produces a desirable BER and can be found by trial and error. However, the decoder processes the entire number of iterations even if the received frame is already corrected, which is inefficient.

3) H1: After each iteration, both decoders make a hard decision and compare the results. When both decoders completely agree on the data frame, the iterations are stopped.

4) LCT (LOG Convergence Threshold) : After each iteration, the soft decision is compared to a threshold. When a predetermined number of data points in the frame are above the threshold, the iterations are stopped. This takes advantage of the diverging nature of the log-likelihood function.

The Magic Genie is unrealistic and is only useful for experimental applications, because it requires a complete knowledge of the transmitted bits. The Magic Genie rule, however, is useful for benchmarking other stopping rules. In this research, the Magic Genie is the rule by which other stopping rules are compared. The stopping rules are also compared to each other in terms of hardware utilization and average number of iterations.

The fixed iterations rule presets the number of iterations before decoding begins. It is a simple stopping rule to implement in hardware because of its lack of functional complexity. A fixed iteration stopping controller can be easily implemented as a single up-counter in hardware. As the final bit of each frame exits the decoder, the up-counter is incremented by one. After a predetermined number of iterations, the up-counter effectively disables the PCCC decoder.

The H1 rule is a hard stopping rule that utilizes the power of both MAP decoders in the PCCC decoder [10, 24]. In FPGA hardware, the H1 rule is implemented by a logical exclusive or (XOR) gate and an up-counter. Specifically, the output of the first MAP decoder is buffered and then XOR with the output of the second MAP decoder. The up-counter is initialized to zero for each frame of data. Each time the XOR produces

a logic 1 the MAP decoders do not agree, the up-counter is incremented by one. The rule only stops the iteration when the up-counter finishes with a count of zero (no differences between the two MAP decoders). Shown in Figure 5.30 and Figure 5.31 is the functional and Xilinx SG hardware description of the H1 rule.

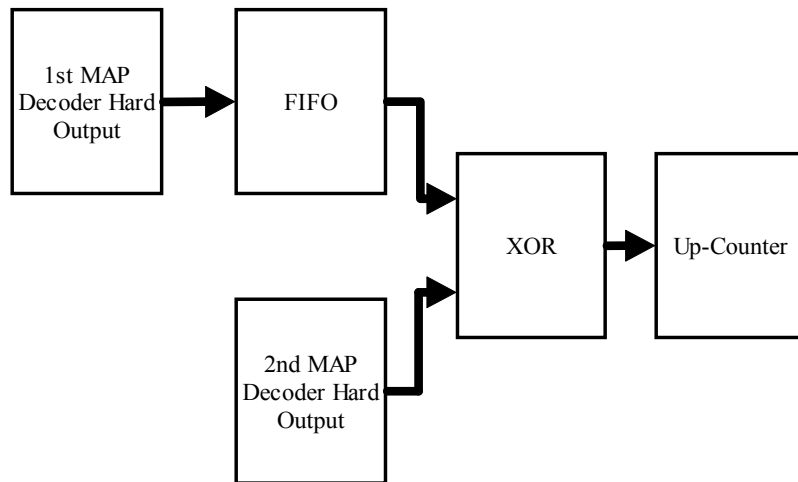


Figure 5.30: H1 Stopping Rule Functional Description

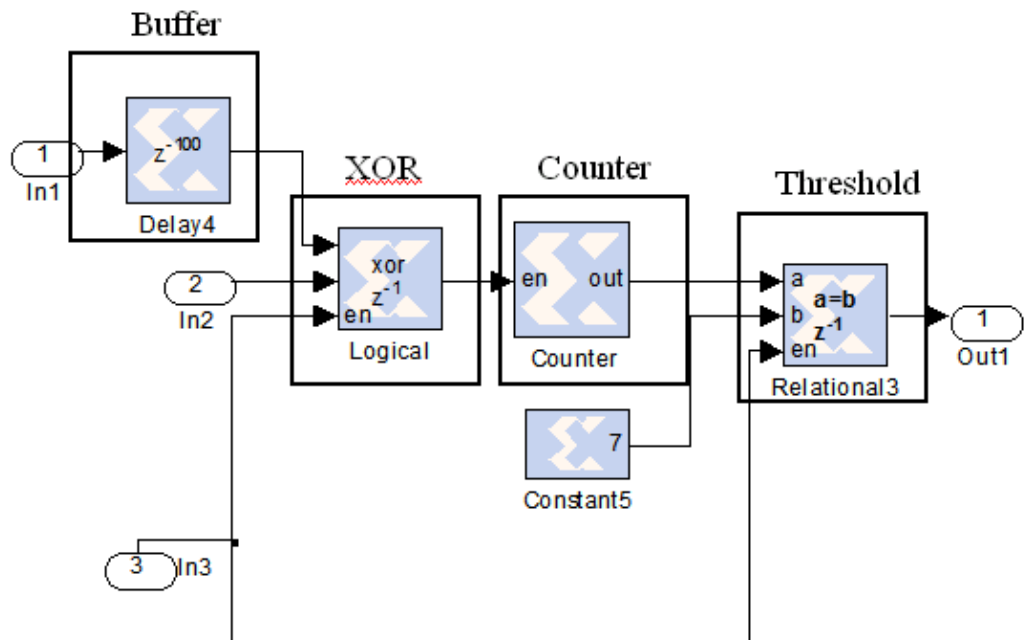


Figure 5.31: H1 Stopping Rule Xilinx SG Implementation

The LCT stopping rule is a soft estimate rule that designed to address the natural log converging behavior of the PCCC decoder. It has been shown that as the E_b/N_o increases, the iterations required by the decoder decreases [10, 24]. This behavior is due to the rapidly increasing performance of the decoder as E_b/N_o increases. After each iteration, the MAP decoder LLR diverges away from zero and converges to some estimate.

Other well known soft decision rules compute the average or the minimum estimate for a data frame [14, 38]. These rules, however, are not very configurable in hardware. Therefore, the LCT stopping rule is designed to compare each estimation with two threshold values and is easily configurable. In general if a predetermined number of bits per frame (ratio threshold) reach the predetermined log value (log threshold), the iteration is stopped.

For example, the Log threshold may be set at 10 and the ratio threshold may be set at 90%. This means that if 90% of the estimates reach the threshold of 10, then the decoder is stopped. As expected, the bit estimate divergence is faster as E_b/N_o increases, and therefore the bits reach the predetermined threshold in a smaller number of iterations. The hardware design of the LCT rule is accomplished by comparing the LLR to a threshold and then incrementing an up counter when they exceed the Log magnitude and ratio thresholds. Shown in Figure 5.32 and Figure 5.33 is a functional and Xilinx SG hardware implementation of the LCT stopping calculator.

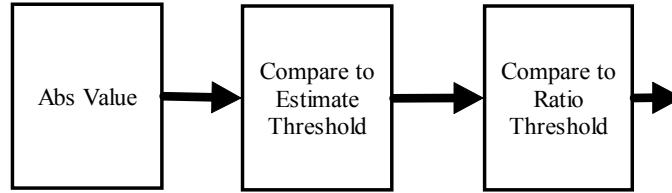


Figure 5.32: LCT Stopping Rule Functional Description

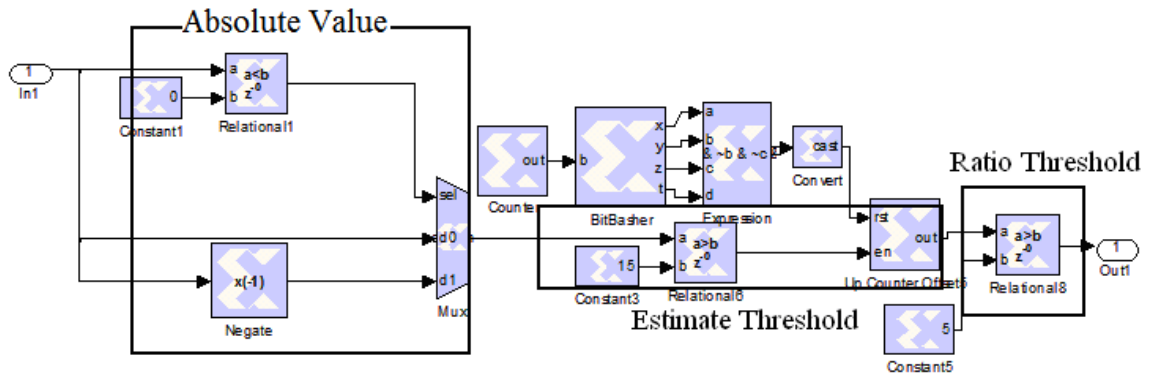


Figure 5.33: LCT Stopping Rule Xilinx SG Implementation

For comparing the stopping rules, each was executed to determine the average number of iterations required at a given E_b/N_o , for the same PCCC decoder and received bits. The fixed iterations, H1 and LCT rules are compared to the average number of iterations of the Magic Genie rule. Assuming the Magic Genie rule is the optimal stopping rule, the other rules are assessed by how close they mimic its behavior over a specific range of E_b/N_o .

Shown in Figure 5.34 through Figure 5.38 are the average numbers of iterations for a PCCC decoder over 100 independent trials for the three stopping rules. It is shown that the average number of iterations for the LCT rule is similar to that of the other three rules.

The LCT rule has similar performance metrics as that of the other proven stopping rules. More specifically, it is shown that the LCT stopping rule utilizes the powerful error correcting capabilities of the PCCC decoder at high E_b/N_o . This intelligence is observed by the fact that the LCT rule decreases its average iterations rather rapidly as the E_b/N_o increases and only a small amount of iterations are needed for a significant correction.

Furthermore, the LCT stopping rule is seemingly better than the Magic Genie rule for low E_b/N_o , because it recognizes that a large number of iterations do not significantly improve performance. Therefore, the LCT rule compares rather favorably in terms of the average iteration performance and hardware utilization to the other stopping rules.

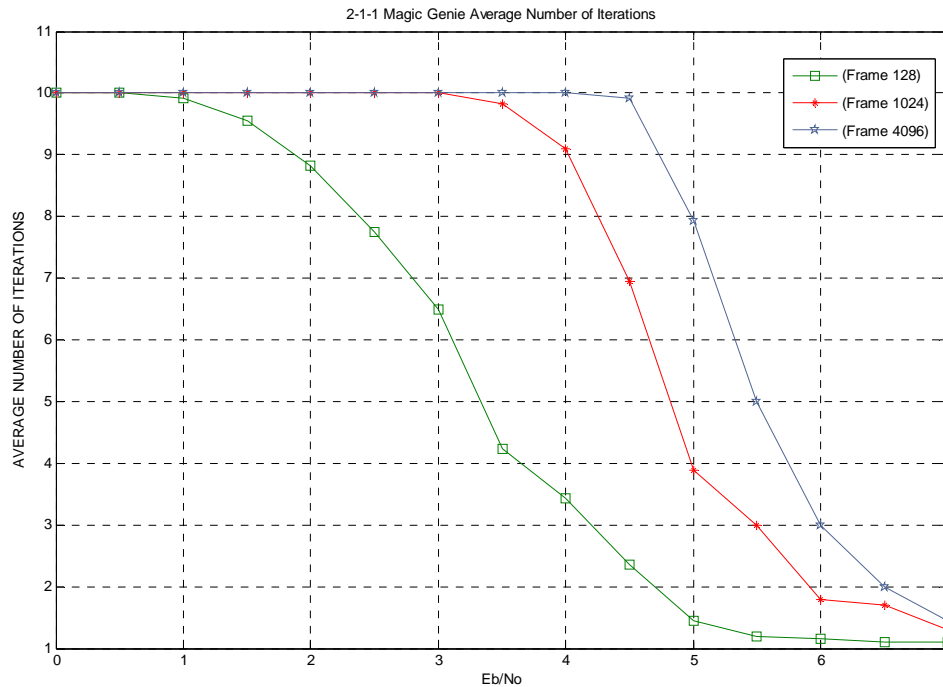


Figure 5.34: Average Number of Iterations Magic Genie

Figure 5.34 shows that as the frame size increases, the number of iterations increases at a given E_b/N_o . This is primarily due to the fact that the Magic Genie requires all of the bits in the frame to be correct before it stops iterating (larger frames take longer to correct). For example, the large 4096 frame does not start to decrease until the E_b/N_o relatively significant in the range of 5 to 6 dB [10, 24].

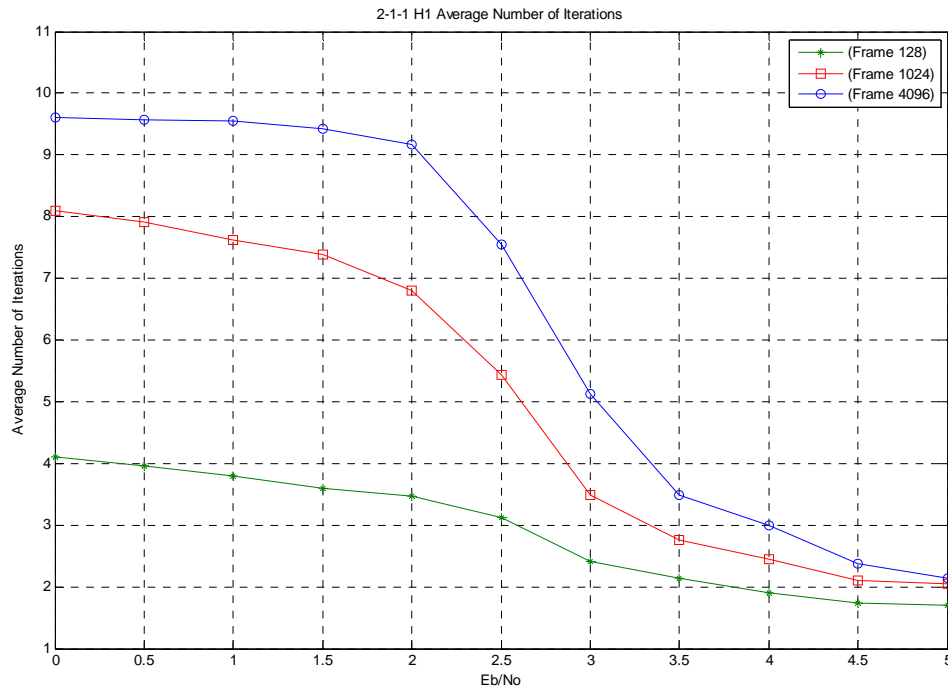


Figure 5.35: Average Number of Iterations H1

It is shown in Figure 5.35 that the H1 rules starts off with a low average number of iterations and then decreasing at an exponential rate when it approaches an E_b/N_o of two (when the decoder reaches its potential). Higher state decoders would reach their potential at much lower E_b/N_o ratios.

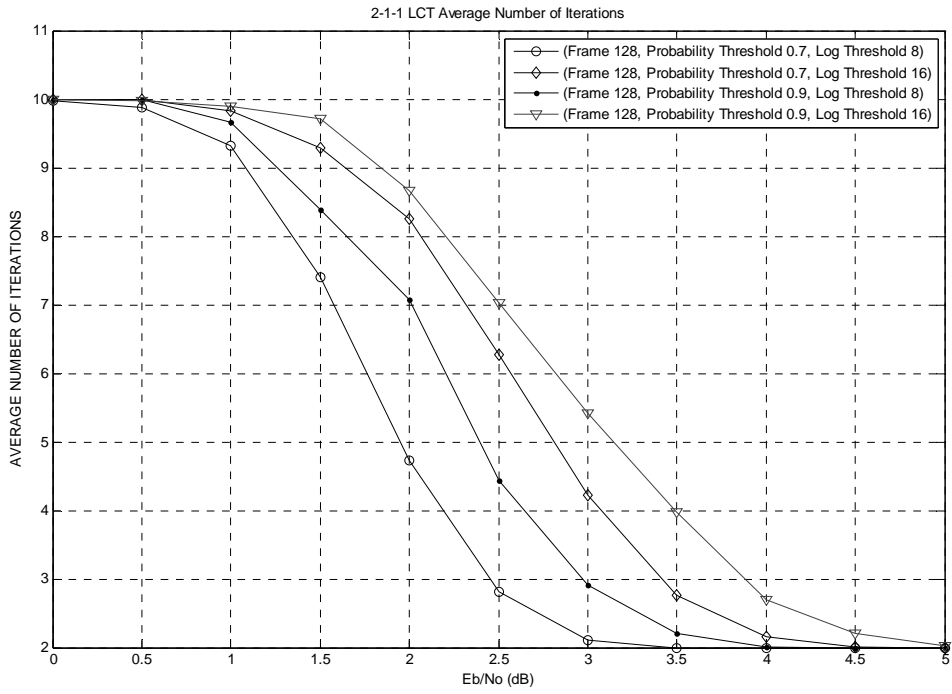


Figure 5.36: Average Number of Iterations LCT 128 Frame

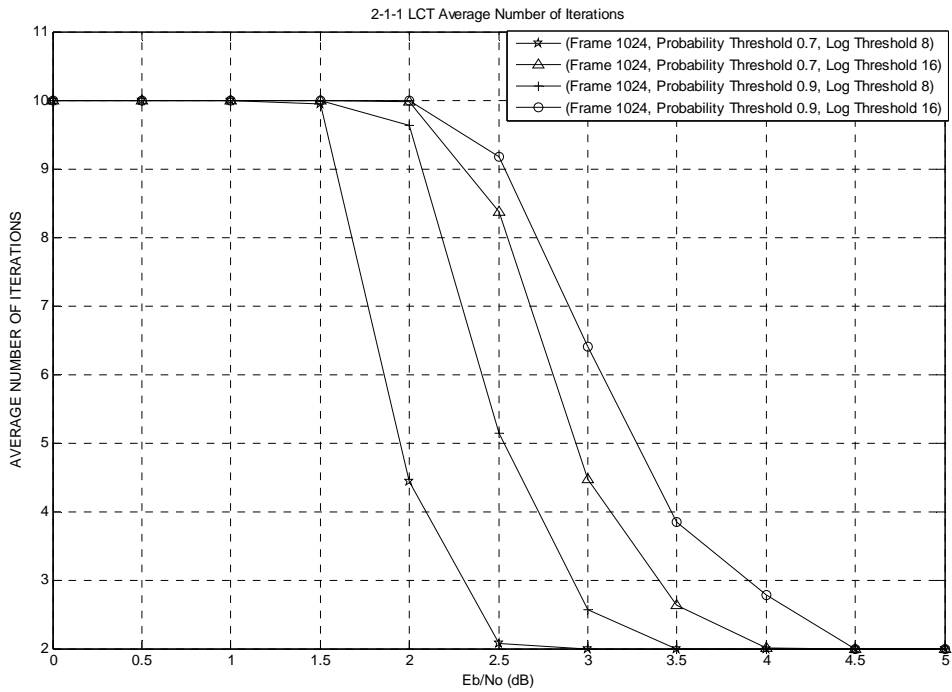


Figure 5.37: Average Number of Iterations LCT 1024 Frame

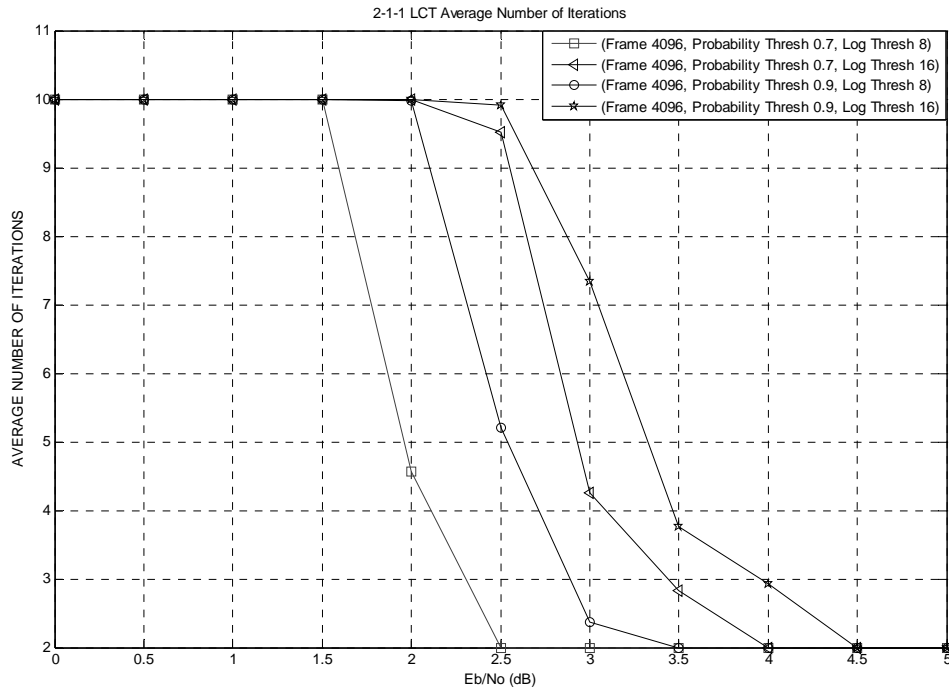


Figure 5.38: Average Number of Iterations LCT 4096 Frame

Figure 5.36, Figure 5.37 and Figure 5.38 shows that the LCT rule behavior is determined by two operational parameters. As the ratio threshold value, Log threshold or frame size increases, the average number of iterations for any E_b/N_0 ratio increases. This behavior is due to the LLR not reaching the Log threshold value until the E_b/N_0 is relatively high. Similar behavior is found for the probability threshold, due to the number of bits in the frame not passing the set threshold until E_b/N_0 is relatively high. It is shown that the LCT rule can closely mimic the Magic Genie Rule by changing the two threshold values. In general, the LCT rule can be designed to decrease its average number of iterations to any specific rate. The rate of decrease is directly dictated by the log and

probability thresholds. Thus, since the LCT rule has two configurable thresholds and it can be more precisely tuned to a specific decoder structure.

5.4.3 PCCC Performance Evaluation

An important aspect of this research is an exacting performance evaluation of the PCCC FPGA hardware design. Specifically, the FPGA hardware is evaluated in three distinct areas.

The first evaluation is a BER comparison between the FPGA hardware for 18 and 32 bit word lengths and the Matlab full precision simulation. The BER comparison illustrates how the design and quantization affects the hardware PCCC decoder.

The second evaluation is the throughput comparison between the hardware for two decoder configurations and a sequential C-language application program. The throughput comparison shows how the parallel and pipelined design of the FPGA hardware increases the throughput as compared to that of a sequential microprocessor, even though the latter is executing at a significantly greater clock rate.

The third evaluation is the FPGA hardware synthesis comparison for two hardware configurations and two bit word lengths. The synthesis comparison shows how the hardware utilization is affected by precision requirements that dictate the bit word lengths.

Furthermore these evaluations show the manner in which the decoder configuration of the FPGA hardware affects the resource utilization and system throughput. Specifically, it is shown that the FPGA resource utilization increases greatly with a design that features increased precision.

5.4.3.1 PCCC Hardware versus Matlab

The design of the PCCC decoder is implemented in fixed point FPGA hardware where the number of total bits, integer bits and fractional bits are fixed for any given computation. Furthermore, the design consists of converging computations such as division, logarithms and exponentiations that produce only estimated results. The fixed point arithmetic and the estimated computations generate inherent errors that affect the BER performance.

In this Chapter, a comparison is shown between the fixed point hardware BER and the full precision floating point Matlab BER. Specifically, the affects of varying the fixed point word length are shown. Varying the fixed point word length has a deleterious affect on the BER and also on the FPGA hardware resource utilization. For example, as the word length increases, the BER improves, but the resource utilization increases.

In contrast, as the word length decreases, the hardware utilization improves, but the BER suffers. Shown in Figure 5.39 through Figure 5.44 is a BER performance comparison between the Matlab floating point precision and the fixed point FPGA hardware precision for 18 and 32 bit word lengths.

In the first comparison, as shown in Figure 5.39, Figure 5.40 and Figure 5.41, 32 bit word lengths are chosen to provide the decoder with increased precision. This word length is also chosen because it is a common precision standard for a short representation in many programming languages, such as C although extended formats are often 64-bit floating point. It is shown that 32 bit word lengths allow the hardware to closely approximate the BER performance of the higher precision Matlab simulation without a significant difference.

However, one detriment in the implementation of 32 bit word lengths is that FPGA resource utilization increases. Specifically, the hardware multipliers on the Virtex 4 FPGA facilitate an 18 by 18 bit multiplier, but a 32 by 32 bit multiplication would require the inefficient utilization of four multipliers. This detriment is discussed in detail in Chapter 5.5.

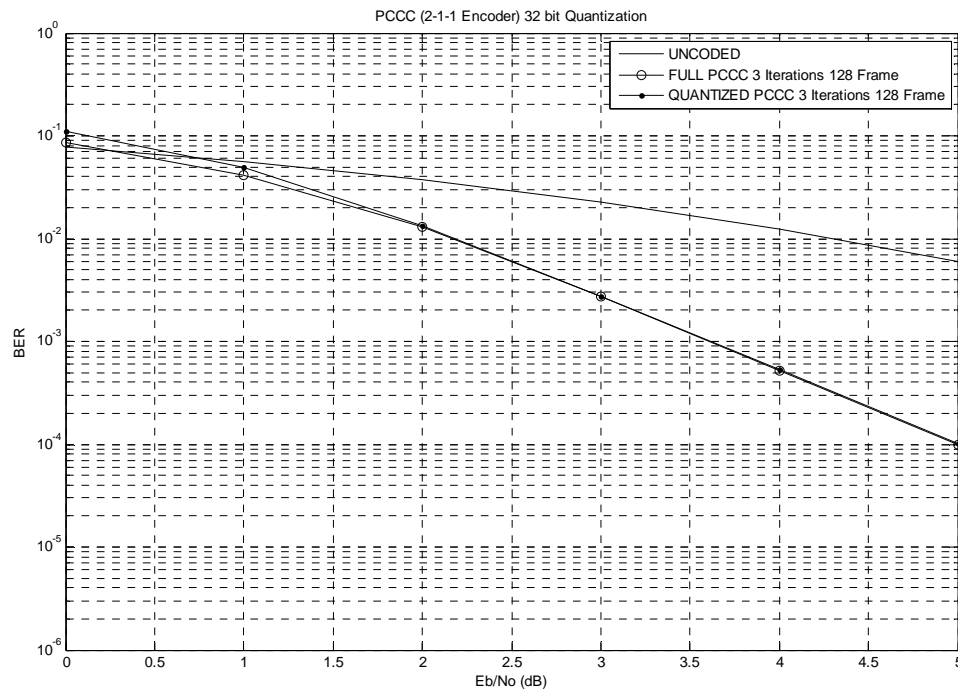


Figure 5.39: BER 32 bit Quantization 128 Frame

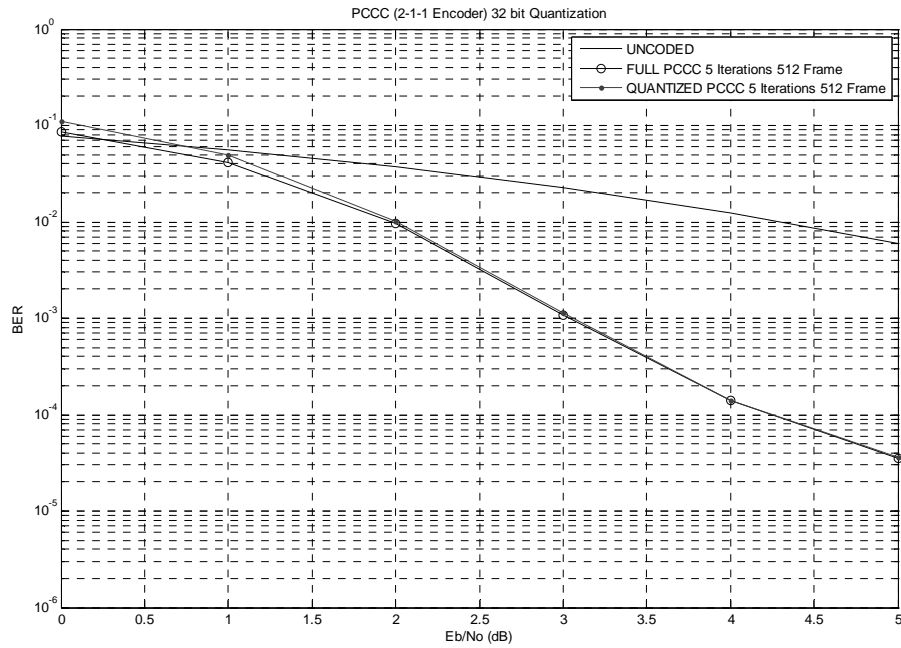


Figure 5.40: BER 32 bit Quantization 512 Frame

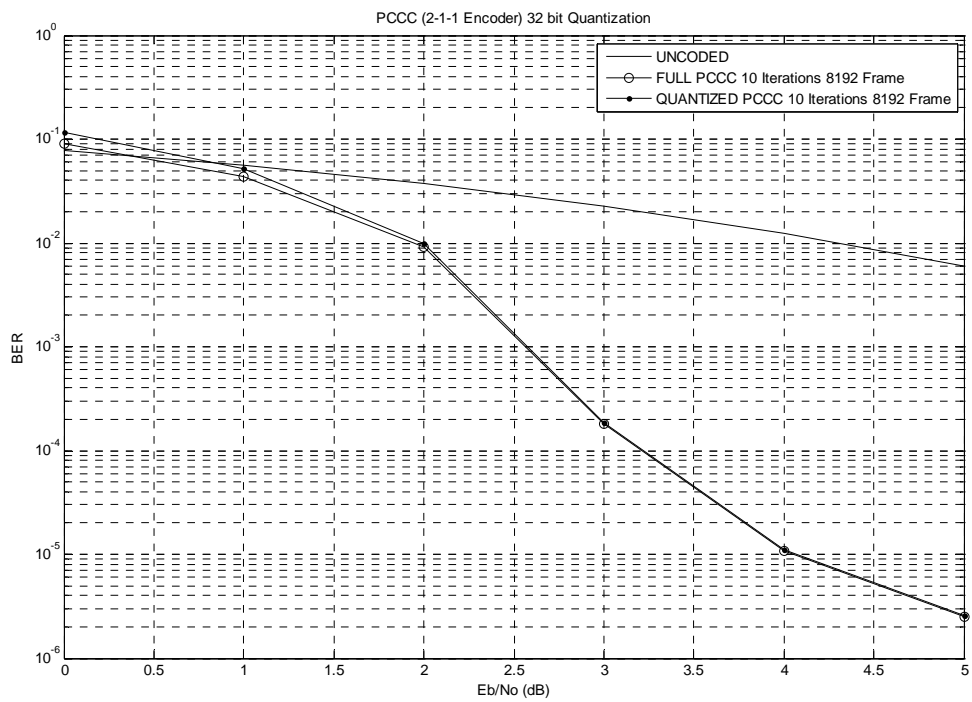


Figure 5.41: BER 32 bit Quantization 8192 Frame

In the second comparison as shown in Figure 5.42, Figure 5.43 and Figure 5.44, 18 bit word lengths were chosen to provide the decoder with reduced hardware utilization at the expense of decreased precision. This word length was chosen because the Virtex 4 FPGA hardware multipliers facilitate an 18 by 18 bit multiplication directly. It is shown that 18 bit word lengths allow the hardware to approximate the BER of the higher precision Matlab simulation at the expense of decreased performance. FPGA hardware resource utilization, however, is decreased because each multiplication utilizes only a single hardware multiplier.

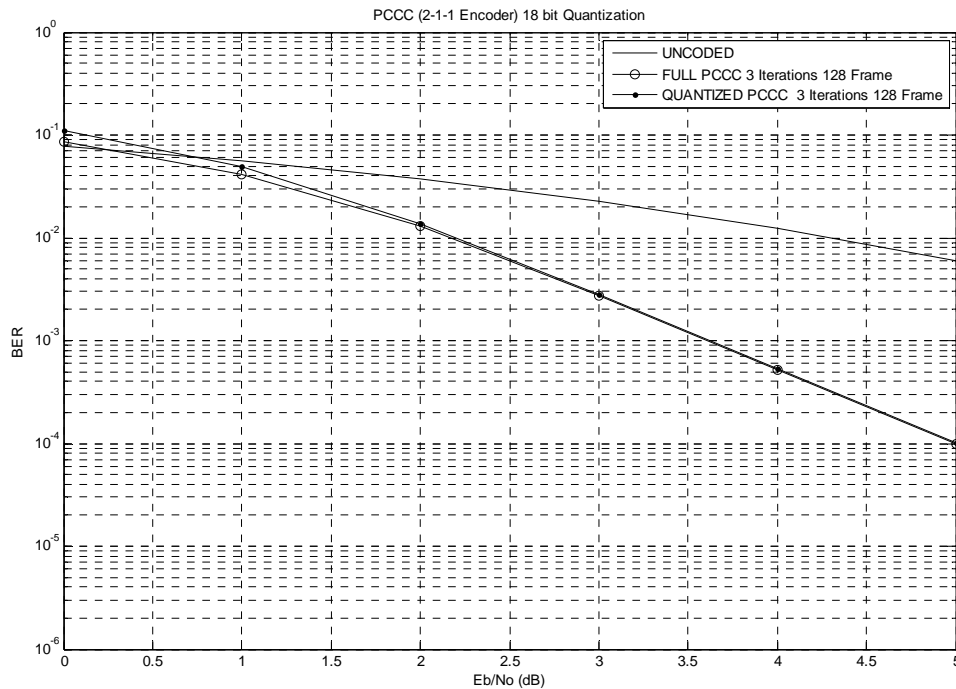


Figure 5.42: BER 18 bit Quantization 128 Frame

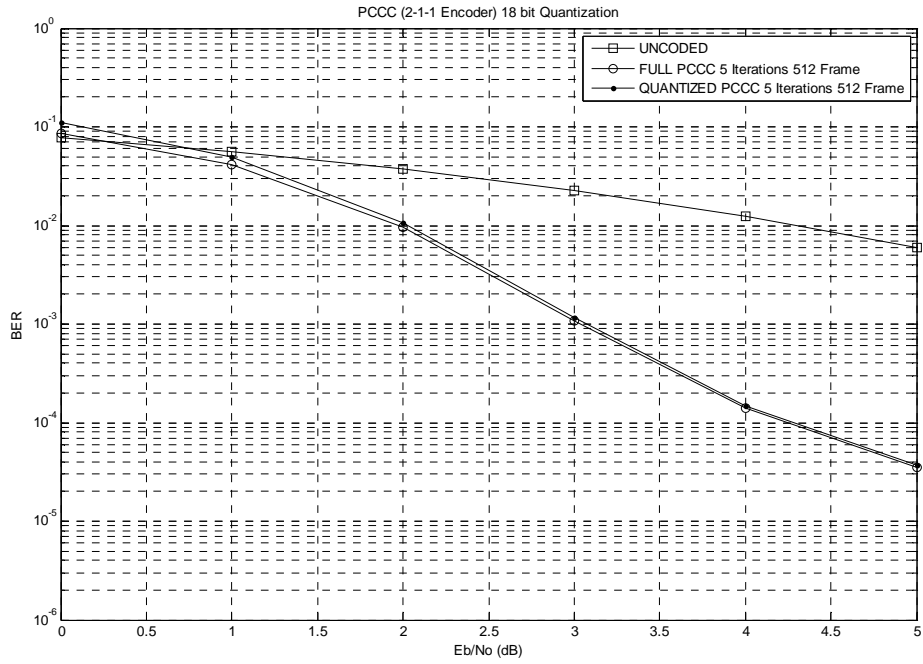


Figure 5.43: BER 18 bit Quantization 512 Frame

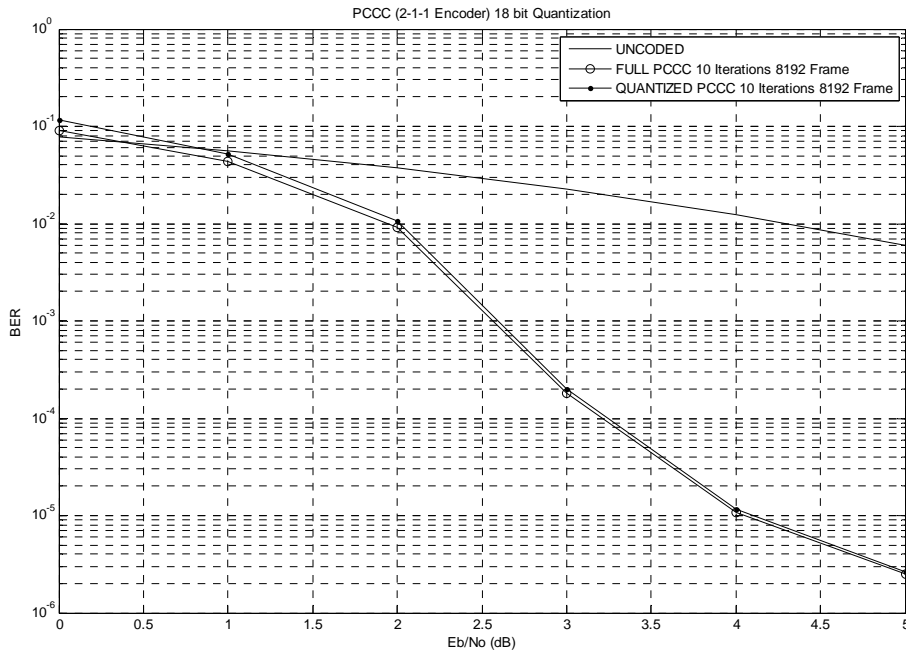


Figure 5.44: BER 18 bit Quantization 8192 Frame

A significant hardware design tradeoff is thus apparent between precision and FPGA hardware resource utilization. In general, increased precision is obtained at the expense of increased hardware utilization. Inversely, decreased hardware utilization is obtained at the expense of decreased precision. A salient result of this research is that the 18 bit representation is adequate and reasonable because it significantly decreases the FPGA hardware resource utilization while at the same time not significantly affecting BER performance.

5.4.3.2 FPGA Hardware versus C-Language Program

PCCC decoding is a computationally intensive iterative procedure that greatly limits the system throughput. Increasing the throughput is essential for some applications that require high bit rates. Therefore this Chapter is focused on designing the PCCC decoder in various configurations to demonstrate the inherent tradeoff between hardware utilization and system throughput. The throughput of the FPGA hardware is then compared to the throughput of a sequential C-Language program executing on a dual core 2.5 GHz Pentium processor.

In a traditional sequential processor software based design, the throughput of the system is limited only by the clock speed [13]. This dependence is due to the fact that traditional sequential processors cannot perform parallel computations and therefore cannot increase the throughput by any means other than increasing the clock speed and efficient programming techniques.

In FPGA hardware, however, system throughput can be increased by performing independent computations in parallel and maintaining a full pipeline. Maintaining a full pipeline is important for ensuring that all hardware resources are being utilized at all times [13]. In a pipelined architecture, it is possible for multiple data frames consisting of thousands of bits to be operated on simultaneously, thus efficiently utilizing all of the available FPGA hardware.

In this research it is shown that the system throughput increase due to pipelining is directly related to the number of MAP decoders in the PCCC decoder. The latency of each MAP decoder is determined by the length of the data frame. Specifically, each MAP decoder, due to the backward recursion, has a latency of approximately two data frames. Furthermore, each interleaver/deinterleaver has a latency of one data frame. Therefore, the overall throughput of the system is directly related to the number of frames that are being processed in the pipeline. When the pipeline is full, no new data can enter until the iterations are complete. Therefore, it is beneficial for the pipeline to be large in order to process multiple frames at any given time.

The MAP decoders in the PCCC decoder are identical to one another and are redundant. Therefore a traditional design that utilizes two MAP decoders is not efficient and a comparison is made as part of this research between the PCCC decoder with one MAP decoder and the PCCC decoder with two MAP decoders.

In the first PCCC design of this research, the PCCC decoder consists of a single MAP decoder, interleaver and deinterleaver [15]. This design minimizes the hardware resource utilization at the expense of throughput. Shown in Figure 5.45 is the functional structure of the PCCC decoder.

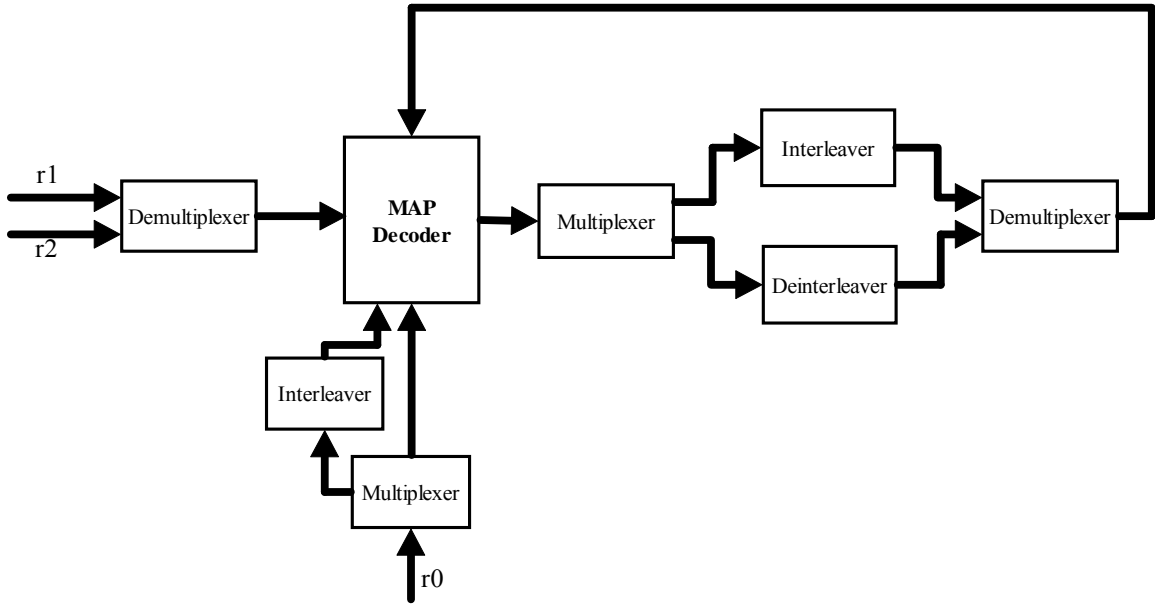


Figure 5.45: PCCC Decoder Single MAP Functional Description

In the second and more traditional design, the PCCC decoder consists of dual MAP decoders, one interleaver and one deinterleaver. This traditional design maximizes throughput at the expense of increased hardware resource utilization [15]. Shown in Figure 5.46 is the functional structure of the second PCCC decoder.

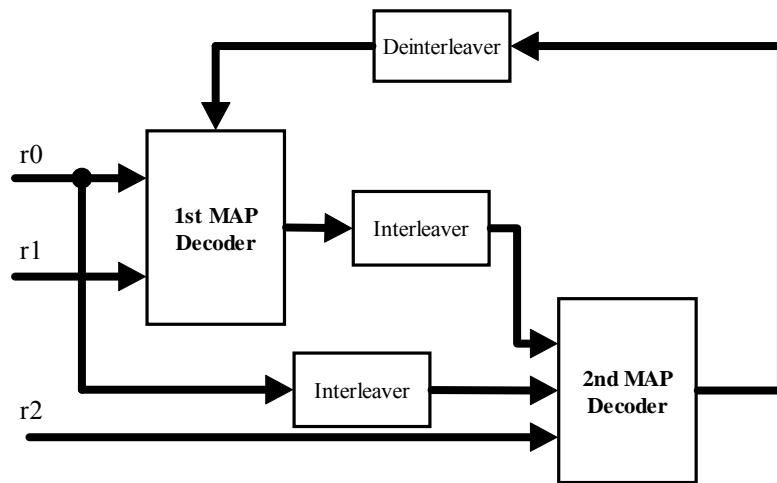


Figure 5.46: PCCC Decoder Dual MAP Functional Description

A comparison between the latency and throughput for both the first and second structures of the PCCC decoder is shown in Figure 5.47 and Figure 5.48. The second structure has approximately twice the throughput as the first structure for any given number of iterations. Also, it is shown that the size of the data frame does not significantly alter the throughput.

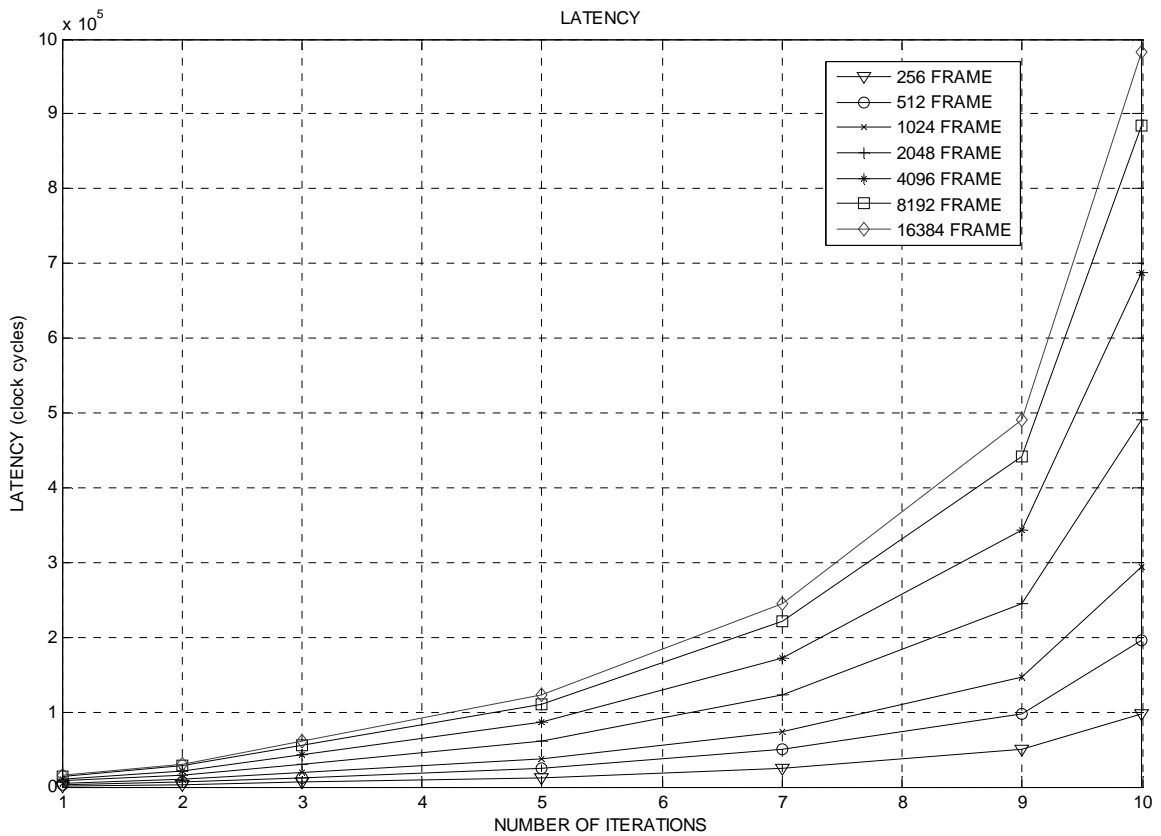


Figure 5.47: PCCC Decoder Latency

Figure 5.47 shows that the latency is dictated by the frame size and the number of iterations in the decoder. As the frame size increases, the latency increases. Similarly as the number of iterations increase, the latency increases. The number of iterations is

seemingly the primary factor in system latency. This is demonstrated in Figure 5.47, because the latency increases exponentially with a linear increase in iterations.

Since the latency increases exponentially it is expected that the throughput will decrease exponentially with a linear increase in the number of iterations and is demonstrated in Figure 5.48. Note also that the frame size does not significantly alter the throughput.

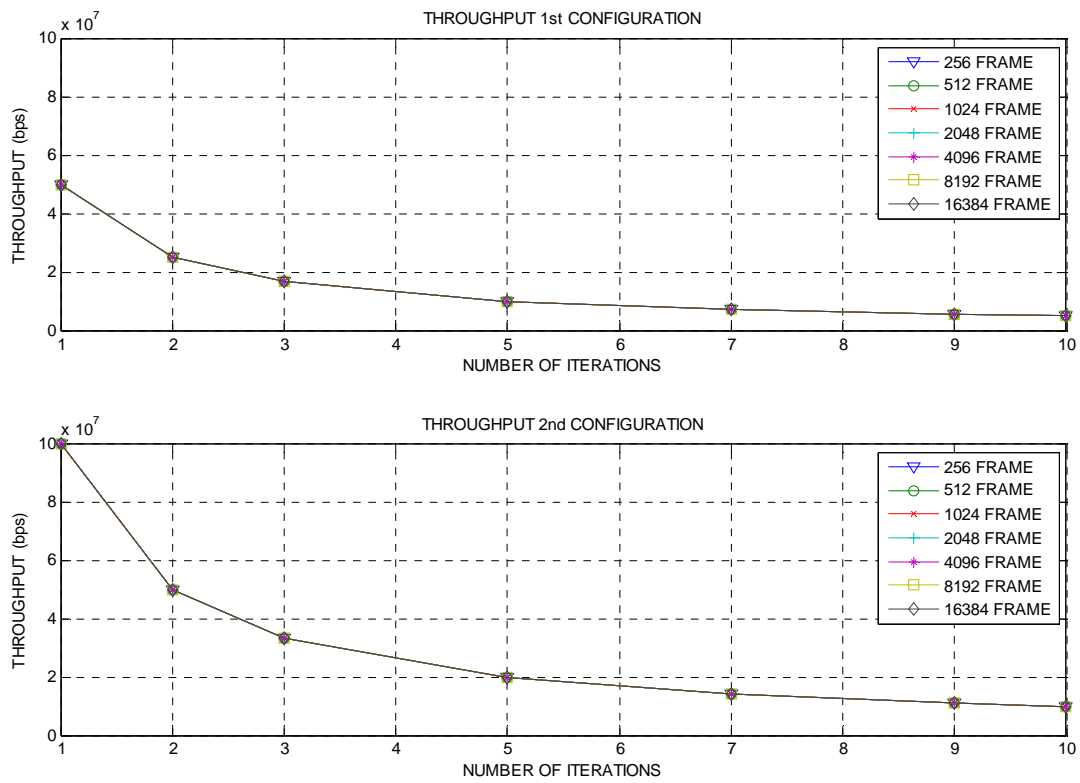


Figure 5.48: PCCC Decoder Throughput Single MAP vs Dual MAP

Another salient comparison can be made between the FPGA hardware utilization throughput and the sequential processor (CPU) implementation, as shown in Table 5.1.

The hardware was implemented on a Xilinx SX35 Virtex 4 FPGA executing at a clock

speed of 100 MHz. The sequential processor was implemented on a dual core Pentium processor executing at 2.5 GHz [13].

Table 5.1: Hardware vs. C-Language Program Throughput

Frame Size	1 Iterations	3 Iterations	7 Iterations
FPGA 256	100 Mbs	32.45 Mbs	13.91 Mbs
CPU 256	195 Kbs	27 Kbs	9.8 Kbs
FPGA 2048	100 Mbs	33.21 Mbs	14.24 Mbs
CPU 2048	193 Kbs	26 Kbs	9.7 Kbs
FPGA 8192	100 Mbs	33.30 Mbs	14.27 Mbs
CPU 8192	190 Kbs	25 Kbs	9.5 Kbs

Table 5.1 demonstrates that the FPGA hardware design, although running at a small fraction of the clock speed of the sequential processor is significantly faster [13]. It is also shown that for iterations greater than one, an increase in frame size increases the throughput slightly.

The pipeline for the two MAP decoder structure introduces a six frame delay with the addition of a small delay attributed by hardware components in the design. Therefore, the pipeline delay is slightly larger than six data frames. This added delay is what causes the throughput difference between frame sizes. Thus, larger data frames are not as susceptible to this extra added delay since the size of the frame is much larger than the delay itself.

Note that the traditional two MAP decoder PCCC configuration is not an intelligent design as discussed earlier. It is equivalent to two PCCC decoders as in the first configuration. Thus, by adding a second MAP decoder, the throughput is doubled as expected. Therefore, the first and obviously more efficient PCCC configuration utilizes only half the resources while still obtaining a relatively high throughput.

5.5 FPGA Synthesis Results

FPGA hardware utilization is an important aspect of hardware design. As discussed in the previous Chapters, a tradeoff exists between hardware utilization, throughput and precision. Although more FPGA hardware consumes more power when executing, it unfortunately also increases development time. In this research, the PCCC encoder and decoder are implemented on a Xilinx Virtex 4 SX35 FPGA. As described in Chapter 3.13, the Xilinx Virtex 4 FPGA is a fine grained architecture that comprises logic blocks and hardware multipliers that provides an optimal platform for implementing the PCCC decoder.

The hardware design of various systems, such as the PCCC encoder, PCCC decoder, interleaver, deinterleaver and stopping rule calculators have already been described in Chapter 5. These designs have been evaluated based on design complexity, BER performance and throughput. To complete the evaluation of the system, the Xilinx SG hardware design is synthesized to the Virtex 4 SX35 FPGA. In synthesizing each subsystem, the hardware design implemented in this research is evaluated based on the resource utilization in a fine grained FPGA. The subsystem synthesis is shown in Table 5.2 through 5.8.

In particular, the resource utilized between the MAP and LOG-MAP-LLR are compared. This comparison is made because the LOG-MAP-LLR decoder which computes the LLR rather than estimating it as the LOG-MAP, can increase BER performance. Thus, the apparent vast hardware utilization differences between the two equivalent algorithms are shown.

The first sub-system is the PCCC encoder. As discussed in Chapter 5.1, the PCCC encoder comprises XOR logic, BRAM and a random interleaver. Table 5.2 is the hardware utilization for the PCCC encoder. It is shown that since the PCCC encoder is a rather simplistic structure, it does not consume many resources. The only resources that will significantly vary is the number of registers and the number of BRAMs in the random interleaver. These resources will vary as the size of the data frame increases (a single Virtex 4 BRAM can support up to 18 Kb). For example, for a 18 bit word implementation, a single BRAM can support a frame size of up to 1 K.

Table 5.2: PCCC Encoder Synthesis Results

Encoder	Multipliers	BRAM	Adders
PCCC Encoder	0 of 192	1 of 192	0

As discussed, interleavers and deinterleavers are used to exchange data between the MAP decoders within the PCCC decoder and are designed with BRAMs and LFSR logic. Thus, as shown in Table 5.3, the interleaver and deinterleaver have minimal resource utilization. As in the PCCC encoder, the number of BRAMs will increase as the frame size increases.

Table 5.3: PCCC Interleaver Synthesis Results

Int/Deint	Multipliers	BRAM	Adders
Interleaver	0 of 192	1 of 192	0
Deinterleaver	0 of 192	1 of 192	0

The MAP decoder is the functional center of the PCCC decoder. As discussed in Chapter 5.3, the MAP decoder comprises four unique computations. Each of the

computations contains various multipliers, BRAMs, adders, subtractors and logic blocks. The distance metric and log-likelihood functions are the more complex computations of the MAP decoder. Specifically, the distance metric and log-likelihood functions comprise multiplications, additions, buffering, subtractions, exponentiations and logarithms. In each of the four computations, the number of multipliers and BRAMs will increase as the frame size increases and the number of decoding states increases.

A single BRAM can support a 1Kb frame for 18 bit words. Thus, a 16K frame size would require 16 BRAMs. The number of decoding states also increases by powers of two. The number of BRAMS and multipliers in the forward and backward recursion computations will double with each increment of the decoding states. Furthermore, the number of multipliers is shown to quadruple between the 18 and 32 bit implementation [14].

Table 5.4: MAP Decoder Synthesis Results

MAP Decoder	Multipliers 18 bits	Multipliers 32 bits	BRAM	Adders
Distance Metric	20 of 192	80 of 192	8 of 192	12
Forward Recursion	4 of 192	16 of 192	0 of 192	2
Backward Recursion	4 of 192	16 of 192	6 of 192	2
Log-Likelihood	36 of 192	144 of 192	0 of 192	18
Total	64 of 192	256 of 192	14 of 192	34

Similarly, the hardware utilization of the log domain LOG-MAP-LLR decoder is shown in Table 5.5. A comparison of Table 5.4 and Table 5.5 shows that the LOG-MAP-LLR decoder utilizes less than half the hardware multipliers and a third of the BRAMs as the MAP decoder. This reduction in multipliers and BRAMs is at the expense of the

relatively abundant hardware adder. Thus, as expected, the LOG-MAP-LLR decoder consumes far less of the crucial hardware components as compared to the MAP decoder.

Table 5.5: LOG-MAP-LLR Decoder Synthesis Results

LOG-MAP-LLR Decoder	Multipliers 18 bits	Multipliers 32 bits	BRAM	Adders
Distance Metric	20 of 192	80 of 192	8 of 192	12
Forward Recursion	0 of 192	0 of 192	0 of 192	4
Backward Recursion	0 of 192	0 of 192	6 of 192	4
Log-Likelihood	32 of 192	128 of 192	8 of 192	24
Total	52 of 192	208 of 192	22 of 192	44

The stopping rule calculators are the control element of the PCCC decoder. Each of the three evaluated stopping rules is designed based upon basic FPGA logic blocks and up-counters. As shown in Table 5.6, the designed LCT stopping rule compares adequately for hardware utilization against the more established rules, although none of these rules consumes significant resources.

Table 5.6: Stopping Rules Synthesis Results

Stopping Rules	Multipliers	BRAM
Fixed	0 of 192	0 of 192
H1	0 of 192	0 of 192
LCT	0 of 192	0 of 192

As described in Chapter 5.4, the PCCC decoder is implemented in two distinct configurations. The first distinct configuration utilizes a single decoder (MAP and LOG-MAP-LLR), single interleaver and single deinterleaver. Thus, the first configuration

utilizes minimal FPGA hardware resources. The resource summary of the first PCCC configuration is shown in Table 5.7 and Table 5.8.

Table 5.7: PCCC First Configuration Synthesis Results

Hardware	Multipliers 18 bits	Multipliers 32 bits	BRAM	Adders
PCCC 1st Configuration MAP	64 of 192	256 of 192	14 of 192	34
PCCC 1st Configuration (LOG-MAP-LLR)	52 of 192	208 of 192	22 of 192	44

The minimization of resources in the first configuration, however, is accomplished at the expense of decreased throughput [15]. In the second configuration, the PCCC decoder utilizes two MAP decoders, a single interleaver and a single deinterleaver.

Table 5.8: PCCC Second Configuration Synthesis Results

Hardware	Multipliers 18 bits	Multipliers 32 bits	BRAM	Adders
PCCC 2 nd Configuration MAP	128 of 192	512 of 192	28 of 192	72
PCCC 2 nd Configuration (LOG-MAP-LLR)	104 of 192	416 of 192	44 of 192	88

The second PCCC configuration consumes a large number of FPGA resources, approximately twice that of the first configuration. The resource utilization for the second configuration is in fact larger than that for the Xilinx Virtex 4 FPGA in at least

the areas of total multipliers and total logic slices. Specifically, the second configuration utilizes approximately twice the number of crucial resources as the first configuration.

This research concludes that the LOG-MAP-LLR is reasonable to implement in an FPGA because of the lack of multipliers and underflow correction. This leads to a minimal use of hardware multipliers, most of which are committed to computing the LLR instead of merely estimating it. Shown in Table 5.9 is the PCCC first configuration based on the LOG-MAP-LLR decoder. It is shown that the number of adders is the only resource that increases significantly with an increase in the number of states.

Table 5.9: PCCC First Configuration Varying States

PCCC 1 st Configuration (LOG-MAP-LLR)	Multipliers 18 bits	BRAM	Adders
2 States	52 of 192	22 of 192	44
4 States	56 of 192	32 of 192	60
8 States	64 of 192	52 of 192	92
16 States	80 of 192	92 of 192	156

CHAPTER 6

CONCLUSION AND FUTURE WORK

6.1 Conclusion

The completed work for this research includes the hardware design implementation and verification of a 1) MAP based Turbo decoder, 2) Hybrid LOG-MAP-LLR Turbo decoder, 3) Single MAP decoder based Turbo Decoder and 4) Log convergence stopping rule calculator LCT.

1) MAP based Turbo Decoder

The hardware implementation of the MAP based Turbo decoder design is compared to the Matlab simulations in terms of BER performance in order to verify the functionality and quantization effects. It is found that 18 bit implementation utilized less hardware than the 32 bit implementation and did not greatly diminish the BER performance of the decoder. Thus, the 18 bit implementation is preferred over the 32 bit implementation.

The MAP based Turbo decoder is also compared to the relatively simplistic LOG-MAP based Turbo decoder and innovative LOG-MAP-LLR based Turbo decoder in terms of FPGA resource utilization. It is shown that the MAP based Turbo decoder utilized the most resources, and in particular a large number of multipliers. The MAP based Turbo decoder is based on the traditional MAP algorithm which comprises complex computations such as exponentiations and logarithms. These complex computations require a large number of critical resources to implement. Furthermore, multiplications in the forward recursion, backward recursion and LLR are dependent on

the number of states. Thus, a large number of hardware multipliers are required when implementing a Turbo decoder with an increased number of states.

2) LOG-MAP-LLR based Turbo Decoder

Specifically, the LOG-MAP-LLR performs the distance metric, forward recursion and backward recursion in the same manner as the LOG-MAP decoder. However, the LLR is computed outright and is not estimated by the maximum function. The LLR was chosen rather than the other metrics because the costly hardware logarithms are not dependent on the number of states. Thus, only two logarithms are required regardless of the number of states.

The LOG-MAP-LLR based Turbo decoder proved to be a reasonable tradeoff between the accuracy of the MAP and the relative simplicity of the LOG-MAP. The LOG-MAP-LLR benefits from the reduced multiplier consumption attained by estimating the forward and backward recursions in the LOG domain while attaining a slight increase in accuracy by actually computing the LLR.

The resource utilization of both the MAP and LOG-MAP-LLR Turbo decoder are shown to be affected primarily by the bit word length. This is due to a low bit word multiplication that is supported in the Virtex 4 embedded multipliers. Other than the large usage of FPGA multipliers, utilization of BRAM and logic slices are not significant. In general, the commitment of multipliers is the primary reason that the LOG-MAP algorithm is relied upon. As the complexity and power consumption of hardware multipliers decreases with the increase in the fine grained FPGA technology, the resource utilization of multipliers becomes even less significant.

3) Single MAP decoder based Turbo decoder

The hardware design of the MAP based Turbo decoder is designed with throughput performance as a primary asset. It is found that the number of independent computations and pipeline length directly affects the system throughput. The traditional dual MAP Turbo decoder is compared to an innovative single MAP Turbo decoder. The single MAP Turbo decoder removes the redundant hardware that is unnecessary for decoding. By only utilizing a single MAP decoder and accompanying buffers, the single MAP Turbo decoder is able to perform decoding with utilizing only half the hardware resources as the dual MAP Turbo decoder. This hardware decrease, however, comes at the expense of a decrease in throughput. The single MAP Turbo decoder is, however, shown to have reasonable throughput capabilities that would satisfy many communications systems.

In the MAP based Turbo Decoder, it is shown that as the number of states increases, the system throughput does not decrease. This is because the states are independent of each other and are computed in parallel. Thus, as the number of states increase, the throughput difference between the hardware and the software (Processor) implementation becomes significant. This is because the sequential processor implementation throughput is largely dependent on the number of states in the decoder. Thus, the processor shows a throughput decrease with an increase in the number of states whereas the FPGA hardware throughput is unaffected.

4) LCT stopping rule

The innovative LCT stopping rule is designed to take advantage of the fact that the LLR function diverges more rapidly as the E_b/N_0 increases. It is shown that the LCT

could be configured to closely mimic the performance of even the Magic Genie rule and is affected by the choice of the log threshold value and bits per frame threshold value. The hardware utilization of the LCT rule was shown to be approximately equivalent to the fixed iteration and H1 rules. An important feature of the LCT rule is its increased configurability. The LCT rule can be configured by the two threshold values to better accommodate different decoder structures and channels. The H1, Magic Genie and other soft stopping rules are not as versatile since they do not implement a bits per frame threshold.

6.1 Future Work

This research can be continued by exploring the hardware implementation of other Turbo code decoding algorithms. As more advanced FPGA architectures become available, the hardware designs of these decoding algorithms can be implemented to more accurately optimize hardware utilization and throughput. For example, with wider bit widths and faster multipliers the accuracy and throughput of the Turbo decoder can be increased. These types of advanced multipliers are found, for example, on the state of the art Xilinx Virtex 5 FPGA. Turbo code hardware implementations on the Virtex 5 would be an interesting extension to this research.

In general hardware design of complex algorithms should be performed by focusing the design to the inherent strengths of the FPGA that is to be utilized. For example, if a particular FPGA architecture has multipliers of certain speeds and bit accuracies, the hardware should be designed to optimize their use. Thus, the fine grained resources on the FPGA should directly dictate how the hardware is designed.

REFERENCES

- [1] Bahl R., "Optimal Decoding of Linear Codes for Minimizing Symbol Error Rate", *IEEE Transactions on Information Theory*, March 1974, pp 284-287.
- [2] Barbulescu S., "Turbo Codes: a tutorial on a new class of powerful error correcting coding schemes", *Institute for Telecommunications Research University of South Australia*, October 1998.
- [3] Barbulescu S., "Turbo Codes 2000", *ITU Telecommunications Standardization Sector*, January 2001.
- [4] Benedetto S., "Serial Concatenation of Interleaved Codes: Performance Analysis, Design, and Iterative Decoding", *TDA Progress Report Communications Systems and Research Section*, August 1996.
- [5] Benedetto S., "Soft Output Decoding Algorithms in Iterative Decoding of Turbo Codes", *JPL, Pasadena, CA, TDA Progress Report 42-122*, August 15, 1995, pp56-65.
- [6] Berrou C., "Near Shannon Limit Error-Correcting Coding and Decoding: Turbo-Codes (1)", *Integrated Circuits for Telecommunication Laboratory*, 1993.
- [7] Bhatt T., "Matlab as a Development Environment for FPGA Design", *Nokia Research Center, Irving Texas*, June 2005.
- [8] Blackert W. J., "Turbo Code Termination and Interleaver Conditions", *Electronics Letters*, vol. 31, no. 24, November 23, 1995, pp 2082-2084.
- [9] Chandran N., "Bridging the Gap Between Parallel and Serial Concatenated Codes", *Wireless Communications Research Laboratory West Virginia University*.
- [10] Colavito L., "Characteristics of Stopping Rules in Iterative Decoding", *Masters Thesis Engineering*, January 2002.
- [11] Divsalar D., "Turbo Codes for Deep Space Communications", *JPL, Pasadena, CA, TDA Progress Report 42-120*, February 1995, pp 29-39.
- [12] Esposito R., "Digital Signal Processing: A Hardware-Based Approach", *ASEE Proceedings of the Middle Atlantic Section Fall Conference*, 2007.
- [13] Esposito R., "Parallel Architecture Implementation of a Reliable (k, n) Image Sharing Scheme", *IEEE International Conference on Parallel and Distributed Systems*, 2008.
- [14] Esposito R., "MAP Decoder Implemented on an FPGA", Submitted to the *IEEE Global Communications Conference*, 2009.

- [15] Esposito R., "Single MAP Decoder based Turbo Decoder Implemented on an FPGA", Submitted to the *IEEE Sarnoff Symposium*, 2009.
- [16] Giulietti A., "Turbo Codes: Desirable and Designable", *Kluwer Academic Publishers*, ISBN: 1-4020-7660-6, 2004.
- [17] Gross W. J., "Simplified MAP Algorithm Suitable for Implementation of Turbo Decoders", *Electronics Letters*, vol. 34, no. 16, August 6, 1998.
- [18] Hagenauer J., "Iterative Decoding of Binary Block and Convolutional Codes", *IEEE Transactions on Information Theory*, vol. 42, no. 2, March 1996, pp 429-445.
- [19] Haykin S., "Communication Systems", *Wiley and Sons, Inc. Publisher*, ISBN: 0-471-17869-1, 2001.
- [20] Haykin S., "Modern Wireless Communications", *Prentice-Hall, Inc. Publisher*, ISBN: 0-13-022472-3, 2005.
- [21] Hu W., "A Turbo Code Benchmark and its Performance Exploration on ILP processors", *Department of Electrical Engineering, University of California*, Los Angeles California, 2001.
- [22] Kerouedan S., "Block Turbo Codes: Towards Implementation", *IEEE 8th International Conference on Electronic Circuits and Systems*, 2001.
- [23] Kim B., "Reduction of the Number of Iterations in Turbo Decoding Using Extrinsic Information", *Proceedings of IEEE TENCON 99*, Inchon, South Korea, 1999, pp 494-496.
- [24] Matache S., "Stopping Rules for Turbo Decoders", *JPL, Pasadena, CA, TDA Progress Report 42-142*, August 15, 2000, pp 1-22.
- [25] Ngo T., "Fixed Point Implementation for Turbo Codes", *Department of Electrical Engineering, University of California*, Los Angeles California, 1999.
- [26] Parahami B., "Computer Arithmetic Algorithms and Hardware Designs", *Oxford University Press Publisher*, ISBN: 0-19-512583-5, 2000.
- [27] Proakis J., "Contemporary Communication Systems", *Thomson Publisher*, ISBN: 0-534-40617-3, 2004.
- [28] Robertson P., "A Comparison of Optimal and Sub-Optimal MAP Decoding Algorithms Operating in the Log Domain", *Institute for Communications Technology, German Aerospace Research Establishment IEEE*, 1995.

- [29] Rosa A., "Implementation of a UMTS Turbo Decoder on a Dynamically Reconfigurable Platform", *IEEE Computer Aided Designs of Integrated Circuits and Systems*, 2004.
- [30] Sabir M., "A Real-Time Embedded Software Implementation of a Turbo Encoder and Soft Output Viterbi Algorithm Based Turbo Decoder", *Dept. of Electrical and Comp. Eng., The University of Texas, Austin, TX*.
- [31] Shannon C., "A Mathematical Theory of Communications", *Bell Systems Technical Journal*, vol. 27, July 1948, pp 379-423, pp 623-656.
- [32] Shibutani A., "Complexity Reduction of Turbo Decoding", *Vehicular Technology Conference, VTC 1999*, vol. 3, pp 1570-1574.
- [33] Sum Chuen Ho M., "Serial and Parallel Concatenated Turbo Codes", PhD *Engineering Telecommunications*, November 2002.
- [34] Thul M., "FPGA Implementation of Parallel Turbo Decoders", *IEEE 17th Symposium on Integrated Circuits and Systems Design*, 2004.
- [35] Thul M., "A Highly Parallel Decoder for Turbo Codes", *IEEE International Symposium on Information Theory*, 2002.
- [36] Valenti M., "Turbo Codes and Iterative Processing", *IEEE New Zealand Wireless Communications Symposium*, November 1998.
- [37] Valenti M., "The UMTS Turbo Code and an Efficient Decoder Implementation Suitable for Software-Defined Radios", *IEEE Symposium on Personal, Indoor, and Mobile Communications*, October 2001.
- [38] Vucetic B., "Turbo Codes: Principles and Applications", *Kluwer Academic Publishers, ISBN: 0-7923-7868-7*, 2000.
- [39] Xilinx., "Xilinx ChipScope Pro", <http://www.xilinx.com>, 2006
- [40] Xilinx., "Xilinx ISE WebPack 8.2i", <http://www.xilinx.com>, 2006.
- [41] Xilinx., "Xilinx ML402 Evaluation Platform Users Guide", <http://www.xilinx.com>, 2006.
- [42] Xilinx., "Xilinx System Generator", <http://www.xilinx.com>, 2006.
- [43] Xilinx., "Xilinx Virtex User Guides", <http://www.xilinx.com>, 2006.
- [44] Yoon S., "A Parallel MAP Algorithm for Low Latency Turbo Decoding", National Science Foundation, February 2002.

[45] Yufei W., “Turbo Code Demo”, *Virginia Tech Research Lab*, June 1999, <http://www.ee.vt.edu/~yufei/turbo>.

APPENDIX
CONFERENCE PUBLICATIONS

Parallel Architecture Implementation of a Reliable (k, n) Image Sharing Scheme

Robert Esposito, John Mountney, Li Bai, Dennis Silage
Temple University

Department of Electrical and Computer Engineering
resposit@temple.edu, jmm@temple.edu, lbai@temple.edu, silage@temple.edu

Abstract

This paper presents a hardware implementation of a secure and reliable k -out-of- n threshold based secret image sharing method. The secret image is divided into n image shares so that any k image shares are sufficient to reconstruct the secret image in a lossless manner, but $(k-1)$ or fewer image shares cannot reveal anything about the secret image. This secret sharing method comprises multiple independent computations which are conducive to parallel processing architectures. Fine-grained field programmable gate array (FPGA) architectures are the near optimal hardware platform for performing parallel processing. This paper illustrates the design and implementation of the secret image sharing method for 8-bit grayscale images on an FPGA which enhances execution time. On average, it was found that the FPGA executes image sharing and reconstruction approximately 300 times faster than a microprocessor operating on the same image.

1. Introduction

In many fields, secure protection and efficient storage of sensitive information is an important concern. Encryption techniques are a popular approach to ensuring the secrecy of sensitive information. It is well known, however, that most encryption techniques suffer from a single-point-of-failure. For example, if the decryption key is lost, stolen or corrupted, the encrypted content can not be decrypted properly. To address this vulnerability, Bai [2] developed a (k, n) secret sharing technique based on matrix projection, wherein the secrets are

divided into n parts, with any k parts needed to construct the data, but $(k-1)$ or fewer parts cannot reveal the secrets. By dividing the decryption key into multiple parts, a single-point-of-failure is avoided.

The (k, n) secret sharing technique includes many computations which are independent and thus are capable of being executed in parallel. Traditional software systems are not able to take advantage of this independence, because they are determined by the operation of a microprocessor. The computational architecture of such a processor thus does not optimize the execution time of the intrinsically parallel algorithm of the (k, n) secret image sharing technique suggested by Bai [2]. In general, micro-processors maintain large overhead for handling floating-point computations. Floating point requirements are not necessary in most cryptographic computation. Consequently, we can fully utilize a hardware-based platform to handle fixed point integer operations.

A processing architecture conducive to parallelization is the fine-grained field programmable gate array (FPGA). An FPGA is a reconfigurable device comprised of logic blocks and programmable interconnections. These logic blocks and embedded arithmetic components can be configured to produce combinational and sequential logic constructs capable of parallel processing.

This paper demonstrates a parallel hardware design focused on minimization of the execution time of the matrix projection based (k, n) secret image sharing algorithm as suggested by Bai [2]. Independent computations are performed in parallel on a Xilinx fine-grained FPGA whenever feasible. This FPGA hardware implementation allows real-time processing of digital images.

This paper is organized with a brief review of the secret image sharing and reconstruction algorithm in Section 2; the FPGA implementation is described in Section 3; benchmarking the FPGA hardware implementation versus a microprocessor is described in Section 4; finally, Section 5 provides the conclusions and implications of this work.

2. Review of (k, n) Image Secret Sharing

Shamir [5] was attributed in developing one of the first secure (k, n) threshold-based secret sharing schemes (SSS) in 1979. Many other SSSs have since been proposed. Among them Bai [1] developed an SSS using matrix projection. The image sharing scheme is described here is referenced to a specific example. For this analysis then, the secret matrix to be shared is:

$$S = \begin{bmatrix} 2 & 3 & 1 & 2 \\ 5 & 4 & 6 & 1 \\ 8 & 9 & 7 & 2 \\ 3 & 4 & 1 & 2 \end{bmatrix}$$

Therefore, the sharing scheme is described as follows:

1. Construct Secret Shares from a secret matrix S of size $m \times m$

- Construct a $m \times k$ random matrix A of rank k where $m > 2(k-1)-1$,
- Choose n linearly independent $k \times 1$ random vectors x_i ,

- Calculate share $v_i = (A \times x_i)(\text{mod } p)$

where p is prime

for $1 \leq i \leq n$.

- Compute Projection Matrix using A and p
 $S = (A(A'A)^{-1}A')(\text{mod } p)$,

where A' is the transpose of A

- Solve $R = (S - S)(\text{mod } p)$,

- Destroy A, x_i, S, S , and
- Distribute n shares v_i to n participants and make matrix R publicly known.

To construct the shares, we choose, for example, a 4×2 random matrix A of rank 2:

$$A = \begin{bmatrix} 10 & 1 \\ 7 & 2 \\ 8 & 4 \\ 1 & 1 \end{bmatrix}$$

The value of $m = 4$ and $k = 2$ satisfies the condition of secret sharing where $m > 2(k-1)-1$. Choose $n = 4$ linearly independent vectors as:

$$x_1 = \begin{bmatrix} 1 \\ 17 \end{bmatrix}, \quad x_2 = \begin{bmatrix} 1 \\ 7 \end{bmatrix}, \quad x_3 = \begin{bmatrix} 1 \\ 1 \end{bmatrix}, \quad x_4 = \begin{bmatrix} 1 \\ 9 \end{bmatrix}$$

Next, compute $v_i = (A \times x_i)(\text{mod } 251)$ for $i=1, 2, 3$, and 4:

$$v_1 = \begin{bmatrix} 27 \\ 41 \\ 76 \\ 18 \end{bmatrix}, \quad v_2 = \begin{bmatrix} 17 \\ 21 \\ 36 \\ 8 \end{bmatrix}, \quad v_3 = \begin{bmatrix} 11 \\ 9 \\ 12 \\ 2 \end{bmatrix}, \quad v_4 = \begin{bmatrix} 19 \\ 25 \\ 44 \\ 10 \end{bmatrix}$$

The projection matrix S , is equal to:

$$S = (A(A'A)^{-1}A')(\text{mod } 251) = \begin{bmatrix} 53 & 87 & 88 & 175 \\ 87 & 137 & 199 & 100 \\ 88 & 199 & 119 & 46 \\ 175 & 100 & 46 & 195 \end{bmatrix}$$

Then the remainder matrix R , is equal to:

$$R = (S - S)(\text{mod } p) = \begin{bmatrix} 200 & 167 & 164 & 78 \\ 169 & 118 & 58 & 152 \\ 171 & 61 & 139 & 207 \\ 79 & 155 & 206 & 58 \end{bmatrix}$$

Matrix R is then made public and utilized with any two of the four secret shares

v_1, v_2, v_3 and v_4 for reconstructing the secret matrix S . Further details on image reconstruction are described in Bai [1].

Since matrix R is necessary for reconstruction, it represents a single-point-of-failure in the algorithm. If R is lost or corrupted, image reconstruction can not be performed properly. In order to overcome this limitation, Bai [2] proposed another method for distributing the R matrix into four image shares based on the Thein [6] image SSS. The value pairs of the R matrix are utilized to compute the four image shares as follows:

$$SH_i = [v_i \ G_i] \text{ where } i=1, 2, 3 \text{ and } 4$$

$$G_i = \left[g_1^{(i)} \ g_2^{(i)} \ \dots \ g_{\left\lfloor \frac{m}{k} \right\rfloor}^{(i)} \right] \text{ for}$$

$$g_t^{(i)}(j) = I(tk+1, j) + \dots + I(tk+k-1, j)r_t^{(k-1)} \pmod{251}$$

$$\text{where } 1 \leq t \leq \left\lfloor \frac{m}{k} \right\rfloor \text{ and } 1 \leq j \leq m$$

For example, the R matrix previously computed would be divided into the following four image shares:

$$SH_1 = \begin{bmatrix} 27 & 116 & 242 \\ 41 & 36 & 210 \\ 76 & 232 & 95 \\ 18 & 234 & 13 \end{bmatrix}, SH_2 = \begin{bmatrix} 17 & 32 & 69 \\ 21 & 154 & 111 \\ 36 & 42 & 51 \\ 8 & 138 & 71 \end{bmatrix},$$

$$SH_3 = \begin{bmatrix} 11 & 199 & 147 \\ 9 & 21 & 12 \\ 12 & 103 & 7 \\ 2 & 42 & 129 \end{bmatrix}, SH_4 = \begin{bmatrix} 19 & 115 & 225 \\ 25 & 139 & 164 \\ 44 & 164 & 214 \\ 10 & 197 & 187 \end{bmatrix}$$

Any two out of the four image shares can then be utilized to recompute the R matrix for secret image reconstruction. Thus, up to two of the four image shares can be lost or corrupted, and image reconstruction will still be possible. Furthermore, in the previous example, the secret matrix is a small 4×4 matrix. If the image to be shared is larger than 4×4 , the image is broken down into multiple 4×4 images that are able to reconstruct the larger image.

To reconstruct the original image, in one example, v_1 and v_2 are concatenated to produce matrix V , although any two out of the four would suffice:

$$V = \begin{bmatrix} 27 & 17 \\ 41 & 21 \\ 76 & 36 \\ 18 & 8 \end{bmatrix}$$

The size $m \times m$ reconstruction projection matrix S is then computed by utilizing V , and the remainder matrix R is computed by reverse solving for G_1 and G_2 .

$$S = (V(V'V)^{-1}V') \pmod{p}$$

In order to compute the secret matrix S , the remainder matrix and projection matrix are added together.

$$S = (R+S) \pmod{p}$$

Reconstructing the secret matrix S is computationally similar to computing the secret shares.

3. Hardware Implementation of Image Secret Sharing and Reconstruction

As described in Section II, sharing an image and reconstructing an image involves multiple independent computations. These computations may be implemented on the parallel architectures of a fine-grained FPGA in order to decrease execution time and thus increase throughput. The hardware implementations of each of the algorithmic steps in Section II are now described as five stages.

3.1 First Stage

The first stage in sharing a secret image is computing the random A matrix and the linearly independent vectors x_i :

- Construct a random $m \times k$ matrix A of rank k where $m > 2(k-1)-1$,
- Choose n linearly independent $k \times 1$ random vectors x_i ,

These computations are independent and thus are able to be parallelized.

Choosing the random A matrix has a consequence that can severely limit computational throughput. Matrix A must have rank k , and $(A'A)$ must have a non-zero determinant. A non-zero determinant is necessary in order to compute the matrix inversion in the projection computation $(A'A)^{-1}$.

It would be inefficient to generate a random A matrix and then compute its determinant to prove invertability. If it is found to be non-invertible, then the process would have to start over by generating another random matrix which might also be found to be non-invertible (thus wasting time). Similarly, it would be inefficient to choose random vectors x_i and then prove that they are linearly independent.

In order to overcome these potential problems, multiple random matrices A and multiple sets of linearly independent vectors x_i are stored in external memory for quick access. For example, the processing hardware chooses a matrix A at random from multiple pre-computed possibilities to ensure they are invertible. Likewise, in parallel, a set of vectors x_i are chosen at random from multiple pre-computed possibilities.

The memory address where they are located is then specified by a random number generator to ensure that the matrices and vectors are chosen at random. A random number generator is implemented by a linear feedback shift register (LFSR) with a pre-determined number of states equal to the number of matrices and vectors stored in memory. Each state of the LFSR corresponds to a different matrix or vector. Thus the security of the system is not compromised, because an external memory can easily store many thousands of possibilities to randomly choose from. Implementation of these two parallel steps is shown in Figure 1.

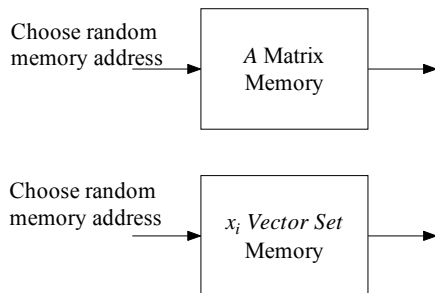


Figure 1. Choosing A matrix and x_i vectors

3.2 Second Stage

The second stage in sharing a secret image is computing the secret share vectors v_i and the $m \times m$ projection matrix S :

- Compute shares $v_i = (A \times x_i) \pmod{p}$ for $1 \leq i \leq n$.
- Compute $S = (A(A'A)^{-1}A') \pmod{p}$,

These computations are also independent and thus are able to be parallelized (bullets indicate independence). Shown in Figure 2, is a block diagram illustrating the two parallel computations.

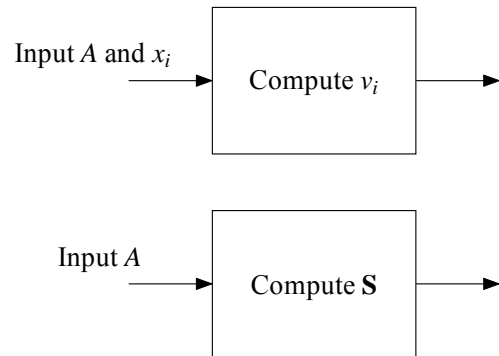


Figure 2. Computing v_i and S

All four v_i shares are computed in parallel since each v_i share is independent from the next. It should also be noted that matrix multiplication, is yet another layer of parallelization. Each and every element of a resulting product vector can be computed in parallel, thereby further decreasing the execution time of the algorithm. Thus, the parallel computation of the v_i shares is as follows:

In parallel compute:

$$v_1 = (A \times x_1) \pmod{251}$$

$$v_2 = (A \times x_2) \pmod{251}$$

$$v_3 = (A \times x_3) \pmod{251}$$

$$v_4 = (A \times x_4) \pmod{251}$$

In parallel for each $(A \times x_i)$ compute:

$$A_{11}x_1 + A_{12}x_2$$

$$A_{21}x_1 + A_{22}x_2$$

$$A_{31}x_1 + A_{32}x_2$$

$$A_{41}x_1 + A_{42}x_2$$

When computing projection matrix \mathbf{S} , a computation order must be followed. First, the modular inverse of $(A'A)$ must be computed [7]. Furthermore, a parallel algorithm for computing the modular inverse of a matrix is presented by Obimbo [4]. Obimbo's technique takes advantage of the fact that rows in a matrix are independent from one another. Obimbo's technique first concatenates an identity matrix on the left side of $(A'A)$. If the elements in the matrix are equal to zero, then the rows are exchanged. If the elements in the matrix are not equal to 1, then the modular inverse of each element is computed using the extended greatest common divisor (GCD) algorithm [7].

GCD for eight bit gray scale pixel values is easily computed by a small hardware lookup table containing just 256 entries. Then, in a parallel process for each row, a multiplication and subtraction is performed. Once the right half of the matrix is in reduced row echelon form (RREF) [8], the modular inverse of $(A'A)$ is complete. Sequential Multiplications with matrices A and A' respectively followed by a modulo computation, complete the projection \mathbf{S} .

Computing the modulo of integer values within the hardware, such as during projection, is performed in a comparison/subtraction pipeline. The pipeline successively subtracts multiples of the modulus number in powers of two, until the outcome is less than the modulus number itself. By subtracting multiples of the modulus number, rather than the number itself, the pipeline latency is decreased. An example of this computation is as follows:

$$\text{mod}(895, 251)$$

$$1) 895 - 502 = 393$$

$$2) 393 - 251 = 142$$

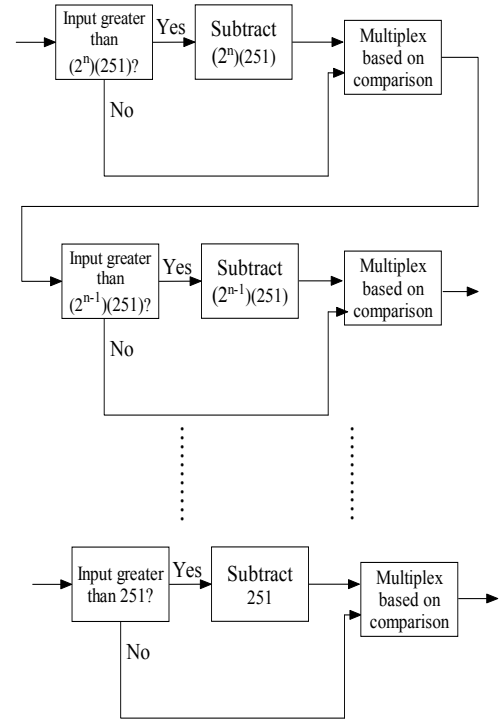


Figure 3. Pipeline for computing modulo

A generic version of the pipelined hardware architecture for modulo computations is shown in Figure 3, wherein each stage of the pipeline includes a comparator, subtraction and a multiplexer.

3.3 Third Stage

The third stage in sharing a secret image is computing the remainder matrix R :

- Compute $R = (S - \mathbf{S})(\text{mod } p)$

It should also be noted that matrix subtraction, is yet another layer of parallelization. Each and every element of a resulting difference matrix is independent and is therefore computed in parallel. Thus, the computation of the subtraction $(S - \mathbf{S})$ is as follows:

In parallel compute:

$$\begin{array}{ll}
(S_{11} - S_{11})(\text{mod } 251) & (S_{12} - S_{12})(\text{mod } 251) \\
(S_{13} - S_{13})(\text{mod } 251) & (S_{14} - S_{14})(\text{mod } 251) \\
(S_{21} - S_{21})(\text{mod } 251) & (S_{22} - S_{22})(\text{mod } 251) \\
(S_{23} - S_{23})(\text{mod } 251) & (S_{24} - S_{24})(\text{mod } 251) \\
(S_{31} - S_{31})(\text{mod } 251) & (S_{32} - S_{32})(\text{mod } 251) \\
(S_{33} - S_{33})(\text{mod } 251) & (S_{34} - S_{34})(\text{mod } 251) \\
(S_{41} - S_{41})(\text{mod } 251) & (S_{42} - S_{42})(\text{mod } 251) \\
(S_{43} - S_{43})(\text{mod } 251) & (S_{44} - S_{44})(\text{mod } 251)
\end{array}$$

3.4 Fourth Stage

The fourth stage in sharing a secret image is computing the secret image shares SH_i from the remainder matrix R :

- $SH_i = [v_i, G_i]$ where $i=1, 2, 3$ and 4

All four G_i image shares are computed in parallel since each G_i is independent from the next. Furthermore, each element within G_i is also independent, and thus is computed in parallel.

$$\begin{array}{l}
G_i(1,1) = (R(1,1) + c_i R(1,2))(\text{mod } 251) \\
G_i(1,2) = (R(1,3) + c_i R(1,4))(\text{mod } 251) \\
G_i(2,1) = (R(2,1) + c_i R(2,2))(\text{mod } 251) \\
G_i(2,2) = (R(2,3) + c_i R(2,4))(\text{mod } 251) \\
G_i(3,1) = (R(3,1) + c_i R(3,2))(\text{mod } 251) \\
G_i(3,2) = (R(3,3) + c_i R(3,4))(\text{mod } 251) \\
G_i(4,1) = (R(4,1) + c_i R(4,2))(\text{mod } 251) \\
G_i(4,2) = (R(4,3) + c_i R(4,4))(\text{mod } 251)
\end{array}$$

where c_i = random number for each G_i

This parallel process is furthermore shown in Fig. 4, wherein the four G_i matrices are computed and then concatenated with the four v_i vectors.

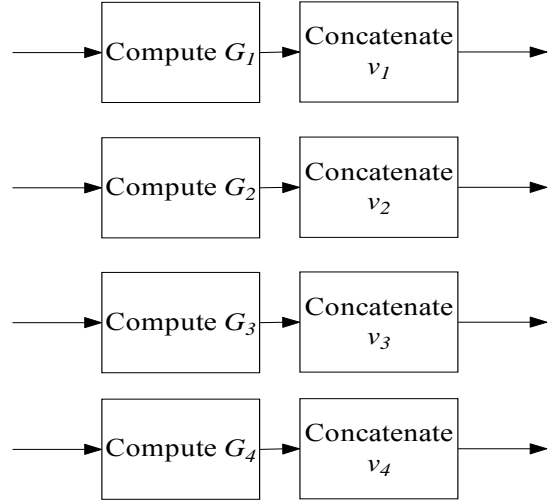


Figure 4. Computing shares SH_i

3.5 Fifth Stage

The fifth stage, is image reconstruction which entails computing in parallel, the projection matrix (similar to the second stage) and the remainder matrix. The remainder matrix is computed from the values of any two of the secret shares, for example, G_1 and G_2 .

All of the pairs within the rows of R are independent from one another, and thus may be computed in eight parallel computations:

$$\begin{array}{ll}
R(1,2) = \frac{G_2(1,1) - G_1(1,1)}{c_2 - c_1} & R(1,1) = G_1(1,1) - c_1 R(1,2) \\
R(1,4) = \frac{G_2(1,2) - G_1(1,2)}{c_2 - c_1} & R(1,3) = G_1(1,2) - c_1 R(1,4) \\
R(2,2) = \frac{G_2(2,1) - G_1(2,1)}{c_2 - c_1} & R(2,1) = G_1(2,1) - c_1 R(2,2) \\
R(2,4) = \frac{G_2(2,2) - G_1(2,2)}{c_2 - c_1} & R(2,3) = G_1(2,2) - c_1 R(2,4)
\end{array}$$

$$R(3,2) = \frac{G_2(3,1) - G_1(3,1)}{c_2 - c_1} \quad R(3,1) = G_1(3,1) - c_1 R(3,2)$$

$$R(3,4) = \frac{G_2(3,2) - G_1(3,2)}{c_2 - c_1} \quad R(3,3) = G_1(3,2) - c_1 R(3,4)$$

$$R(4,2) = \frac{G_2(4,1) - G_1(4,1)}{c_2 - c_1} \quad R(4,1) = G_1(4,1) - c_1 R(4,2)$$

$$R(4,4) = \frac{G_2(4,2) - G_1(4,2)}{c_2 - c_1} \quad R(4,3) = G_1(4,2) - c_1 R(4,4)$$

where c_1 and c_2 = random number for G_1 and G_2

A parallel matrix addition is then performed to recover the original secret. This is performed similarly to the subtraction as shown in the third stage.

$$S = (R + S)(\text{mod } p)$$

4. Hardware Benchmarking

The primary reason for implementing secret image sharing in the FPGA hardware is to decrease execution time. Execution time is of particular concern when dealing with large images that are required to be processed in real-time.

In traditional software systems, the image shares are computed by a microprocessor. Thus it is instructive to compare the execution time of the FPGA hardware implementation against the apparent execution time of a software based system. The following benchmarking results are a comparison of the Matlab/Xilinx System Generator FPGA hardware implementation executing on a Xilinx Virtex-4 SX35 FPGA at a clock frequency of 100 MHz, and a C language implementation executing on a microprocessor utilizing an Intel 3.2 GHz Dual Core Pentium [3, 10, 11]. The execution time of the FPGA hardware and traditional software implementation for the secret image sharing algorithm utilizing various generic image sizes are shown in Table 1.

It is apparent that the FPGA hardware implementation has significantly reduced execution time compared to that of the microprocessor system. The decrease in execution time is primarily due to the parallel computations and the intrinsic pipelined architecture of the fine-grained FPGA. FPGA hardware in the pipelined architecture allows the simultaneous computation of multiple data sets.

Furthermore, the high speed I/O capabilities of the hardware platform, such as Ethernet and USB ports, facilitate high speed data communication to the Virtex 4 FPGA.

Table 1. Hardware versus Software Execution Time for Secret Share Computation

Image Size (pixels)	Number of 4x4 shares	C-code execution time (μ s)	Hardware execution time (μ s)	Hardware Times Faster
128x128	4 096	3 300	11	300x
256x256	16 384	13 000	42	310x
512x512	65 536	52 000	165	315x
1024x1024	262 144	210 000	657	320x

For example, after a computational unit of the FPGA hardware (for example, a multiplier) produces a result, it is then available to be utilized to produce a result for the next data set. Thus, at any given time, the pipeline is computing multiple data sets (or multiple image shares).

The FPGA hardware pipeline has an observed initial delay of 124 clock cycles. This means that it takes 1024 ns at 100 MHz for the first four image shares to be computed. However, once the FPGA hardware pipeline is loaded, every single clock cycle thereafter, or every 10 ns at 100 MHz, produces four new image shares. Thus, the hardware will show increased performance as the image size gets larger.

Even more significant is the fact that the FPGA hardware here is only executing at clock frequency of 100 MHz, whereas the microprocessor (albeit a dual core processor) is running at 3.2 GHz or 32 times faster. Even with a significantly faster clock, the microprocessor software is agonizingly slow as compared to the FPGA hardware.

The Xilinx ML402 development board is the platform of choice for this research [9]. The development board comprises a Xilinx Virtex-4 SX35 FPGA with 192 embedded hardware multipliers. A summary of the important FPGA resources utilized to implement this algorithm is shown in Table 2.

Table 2. Summary of Resource Utilization (Virtex-4 SX35)-Secret Share Computation

Embedded Multipliers	91 out of 192	47%
Block RAM	4 out of 192	2%
Slices	11434 out of 15360	74%

A similar comparison is shown in Table 3 for image reconstruction. It should be noted that the initial pipeline delay for reconstruction is the same as the image sharing (124) due to the latency of the matrix projection computation. As expected, the throughput of the parallel computations on the FPGA shows a significant improvement over the conventional microprocessor. In addition, Table 4 displays the resource utilization for image reconstruction implemented with the Virtex-4 SX35. It is shown that the resource utilization in image reconstruction is similar to the resource utilization for image sharing, with the exception of the embedded multipliers.

Table 3. Hardware versus Software Execution Time for Image Reconstruction

Image Size (pixels)	Number of 4x4 shares	C-code execution time (μ s)	Hardware execution time (μ s)	Hardware Times Faster
128x128	4 096	3 200	11	291x
256x256	16 384	12 600	42	300x
512x512	65 536	50 700	165	307x
1024x1024	262 144	205 000	657	312x

Table 4. Summary of Resource Utilization (Virtex-4 SX35)-Image Reconstruction

Embedded Multipliers	78 out of 192	40%
Block RAM	4 out of 192	2%
Slices	11425 out of 15360	73%

5. Conclusion

We proposed a fine-grained FPGA hardware implementation of a (k, n) secret image sharing and reconstruction scheme [2]. Image shares and image reconstruction for a large image may now be computed in real-time by the parallel

hardware. A significant decrease in execution time was realized as compared to the conventional microprocessor implementation of the algorithm. The decrease in execution time can be primarily attributed to the parallelization of the algorithm and the pipelined architecture of the FPGA. In general, any algorithm that has independent computations would greatly reduce its execution time by implementing the parallel techniques discussed in the sections above.

6. References

- [1] L. Bai, "A Strong Ramp Secret Sharing Scheme using Matrix Projection," presented at the Second International Workshop on Trust, Security and Privacy for Ubiquitous Computing, 2006.
- [2] L. Bai, "A Reliable (k,n) Image Secret Sharing Scheme," presented at the Second International Symposium on Dependable, Autonomic and Secure Computing, 2006.
- [3] Mathworks, "Matlab", available at <http://www.mathworks.com>, last accessed May 2008.
- [4] C. Obimbo, "A Parallel Algorithm for determining the inverse of a matrix for use in block cipher encryption/decryption," Published On-line, Department of Computing and Information Sciences, University of Guelph, 2007.
- [5] A. Shamir, "How to share a secret," Communications of the ACM, vol.22, no.11, pp. 612-613, Nov. 1979.
- [6] C.Thein and J. Lin, "Secret image sharing," Computers and Graphics, vol. 26, no. 5, pp. 765-770, 2002.
- [7] Vanderbilt University, available at <http://vanets.vuse.vanderbilt.edu/~xue/cs291fall06/hint.pdf>, last accessed May 2008.
- [8] Wikipedia, available at http://en.wikipedia.org/wiki/Reduced_row_echelon_form#Reduced_row_echelon_form, last accessed August, 2008.
- [9] Xilinx "ML402", available at http://www.xilinx.com/support/documentation/boards_and_kits/ug080.pdf, last accessed May 2008.

- [10] Xilinx, “System Generator”, available at http://www.xilinx.com/support/documentation/sw_manuals/sysgen_bklist.pdf, last accessed May 2008.
- [11] Xilinx, “Virtex 4”, available at http://www.xilinx.com/support/documentation/user_guides/ug070.pdf, last accessed May 2008.

Digital Signal Processing: A Hardware-Based Approach

Robert Esposito
Electrical and Computer Engineering
Temple University

Introduction

Teaching Digital Signal Processing (DSP) has included the utilization of a simulation tool (ST) for student projects and homework. The leading ST in academia is MATLAB by MathWorks. MATLAB is a vector based environment that is conducive to DSP simulation. Specifically, filter design is simulated utilizing a C-like code. Students are able to enter a filter design as a discrete time sequence or a discrete transfer function. MATLAB has built in functions that generate deterministic and non-deterministic signals which can then be inputted to the designed filter. The output of the filter can then be analyzed in the time-domain or frequency-domain utilizing other MATLAB functions. MATLAB, however, is not conducive to teaching the structural aspects of filter design.

Simulink is a block based design system that provides a graphical environment and a customizable set of block libraries that allows the user to simulate and test a variety of systems such as digital filters¹. Simulink has an extensive DSP library that contains blocks for implementing everything from signal generation to adaptive filtering. Even though Simulink is a more realistic implementation environment than MATLAB, it is still purely simulation. Realizing the need for users to be able to perform real hardware implementations, MathWorks collaborated with Xilinx to produce System Generator (SG). SG is a group of extensive libraries that are included in Simulink. The SG libraries include hardware based blocks that interact with traditional Simulink blocks. Therefore, hardware designs can be synthesized, downloaded to a Xilinx Field Programmable Gate Array (FPGA) and then compared in real time to their Simulink simulation counterparts.

Simulink IIR Filter Design

IIR filter design is a standard topic in any DSP course. Specifically, second order lowpass, highpass, bandpass and bandstop filters can be implemented and analyzed. Shown in equation 1, is a well known transfer function of a second order notch IIR filter².

$$H(z) = \frac{K(1 - 2\beta z^{-1} + z^{-2})}{1 - \beta(1 + \alpha)z^{-1} + \alpha z^{-2}} \quad (1)$$

In the transfer function of equation 1, α determines the 3-dB bandwidth, β determines the center frequency and K determines the maximum magnitude value. This filter can be implemented in Simulink in various ways. One implementation method is the utilization of a built in IIR filter block where the user can specify the numerator and denominator coefficients of the transfer function. Shown in figure 1, however, is a discrete time equation implementation of the notch IIR filter in equation 1 with $\alpha = .1$, $\beta = .2$, $K = .55$. In this design, a white noise sequence is

sampled at 10 000 samples per second and inputted to the notch filter. Its output spectrum is then analyzed using a fast Fourier transform (FFT) block.

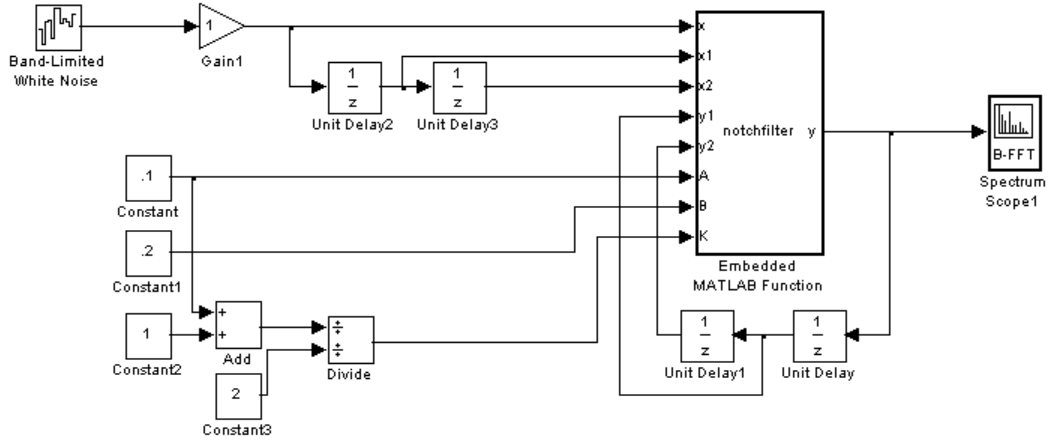


Figure 1: Simulink Notch Filter

The notch filter of equation 1 is implemented using an embedded MATLAB function and delay blocks as shown in figure 1. This implementation is a direct implementation of the discrete time equation that represents the transfer function as shown in equation 2.

$$y[n] = \beta(1 + \alpha)y[n - 1] - \alpha y[n - 2] + Kx[n] - 2K\beta x[n - 1] + Kx[n - 2] \quad (2)$$

System Generator IIR Filter Design

The Simulink implementation of the IIR filter as shown in figure 1, does not take into account structure. Shown in figure 2, is a well known structure for IIR filters is Direct Form II¹. Direct Form II is a canonic IIR filter structure wherein the number of delays is equal to the order of the filter. Canonic structures are important to minimize hardware components.

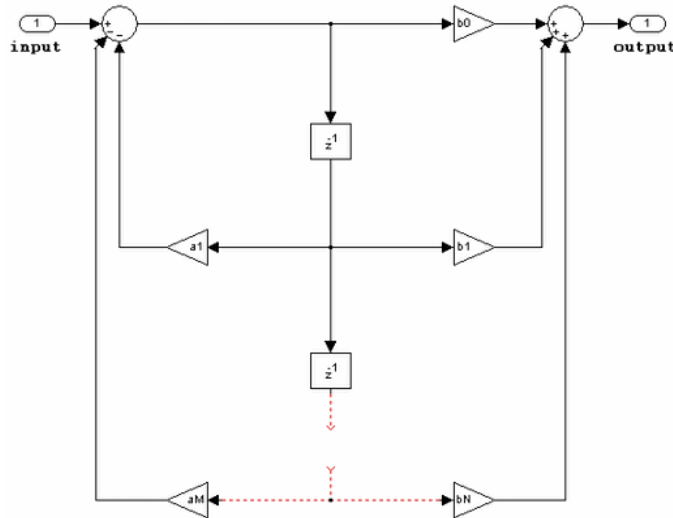


Figure 2: Generalized Direct Form II IIR Structure

A Direct Form II structure that represents the notch filter in equation 1 can be implemented utilizing hardware blocks in SG³. Specifically, the only SG hardware blocks that will be needed are delays, adders and multipliers. Shown in figure 3, is the SG implementation of the Direct Form II structure.

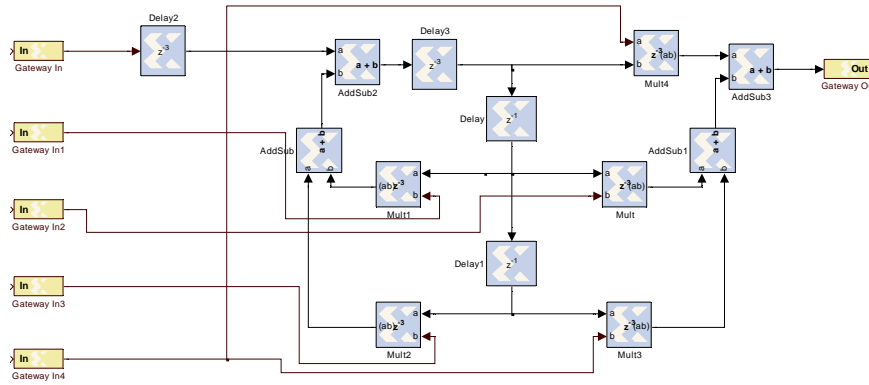


Figure 3: SG Implementation of Direct form IIR Filter

The Gateway-In blocks are essentially analog to digital converters (ADC). The Gateway-In blocks sample and quantize signals from the Simulink environment so they can be processed by digital SG hardware blocks. The Gateway-Out block is the opposite of the Gateway-In block. The Gateway-Out acts as a digital to analog converter (DAC) for outputting signals to be analyzed back to the Simulink environment.

One distinct difference between the Simulink structure in figure 2 and the SG structure in figure 3 is the addition of two extra delays (delay2 and delay3). The structure in figure 2 assumes that multipliers a1-aM and b1-bN have a latency of zero. In hardware design, however, blocks sometimes have latency which can throw off the synchronization of the data pipeline. Since SG multipliers Mult1, Mult2, Mult3, Mult4 and Mult5 each have a default latency of 3 samples, it is necessary to add delay2 and delay3 which have a latency of 3 in order to maintain the integrity of equation 2. Furthermore, unlike the structure in figure 2, quantization error becomes an important issue. Specifically, each SG hardware block of figure 3 has a finite wordlength. Before setting the fixed number of bits for each block, the user must have knowledge of the range of data points being used in the design. Knowledge of the data range allows for optimal setting of integer and fractional bits.

ML402 Board

After the initial design, the next logical step is to implement and verify the functionality of the SG hardware design on an FPGA. The ML402 is a development platform used for hardware verification that includes a Xilinx Virtex 4 FPGA, push buttons, slide switches, LEDs and an LCD. Shown in figure 4, is the ML402 board⁴.



Figure 4: ML402 Board

An important feature of the ML402 is the Xilinx Virtex 4 FPGA⁵. The Virtex 4 can come in a variety of device packages. Specifically, one device package is the Virtex 4 XC4VSX35. The XC4VSX35 is a DSP focused device that comprises 192 embedded multipliers, 192 18KB of block RAM and 34,560 logic cells. This is the device that was targeted in this publication.

Hardware Analysis and Verification

Hardware synthesis is the process of translating and mapping a hardware design into a targeted architecture. The SG implementation of an IIR filter as shown in figure 3 must be synthesized to the XC4VSX35 FPGA of the ML402 board. Shown in figure 5, is a screen shot of the SG window wherein synthesis options are selected.

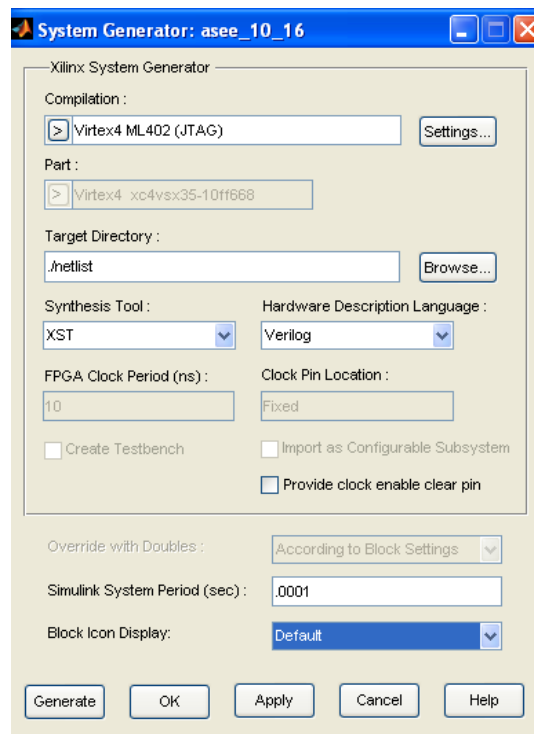


Figure 5: SG Synthesis Options

In figure 5, it is shown that the compilation device is selected as the Virtex 4 ML402 board. This selection will allow Simulink to pass signals and retrieve signals from the ML402 board during a

process called Hardware Co-Simulation. Hardware Co-Simulation is a process wherein the synthesized design is running on the hardware board while the Simulink simulation model is running on the PC. This allows the output of the hardware implementation to be simultaneously compared with the output of the simulation for verification purposes.

When synthesis is complete, SG provides a report which details the amount and type of FPGA resources that were needed to implement the design. The IIR filter design of figure 3 utilized 20 embedded multipliers (10% of available multipliers) and 1100 hardware slices (7% of available slices). The design utilized 20 embedded multipliers instead of the five shown in the design because of the data wordlength. For this particular synthesis, the data wordlength was set at 32 bits for each block. On the Virtex 4, however, each embedded multiplier can only perform an 18-bit by 18-bit multiplication. Therefore, in order to produce a 32-bit by 32-bit multiplication, four 18-bit multipliers were utilized for each fully pipelined multiplication. This multiplier pipeline also increased the minimum latency of each multiplication to 5 clock cycles. Shown in figure 6, is the hardware co-simulation setup after a successful synthesis. The synthesized block is shown on the bottom wherein it is connected to the Simulink inputs and outputs in the same manner as the other models.

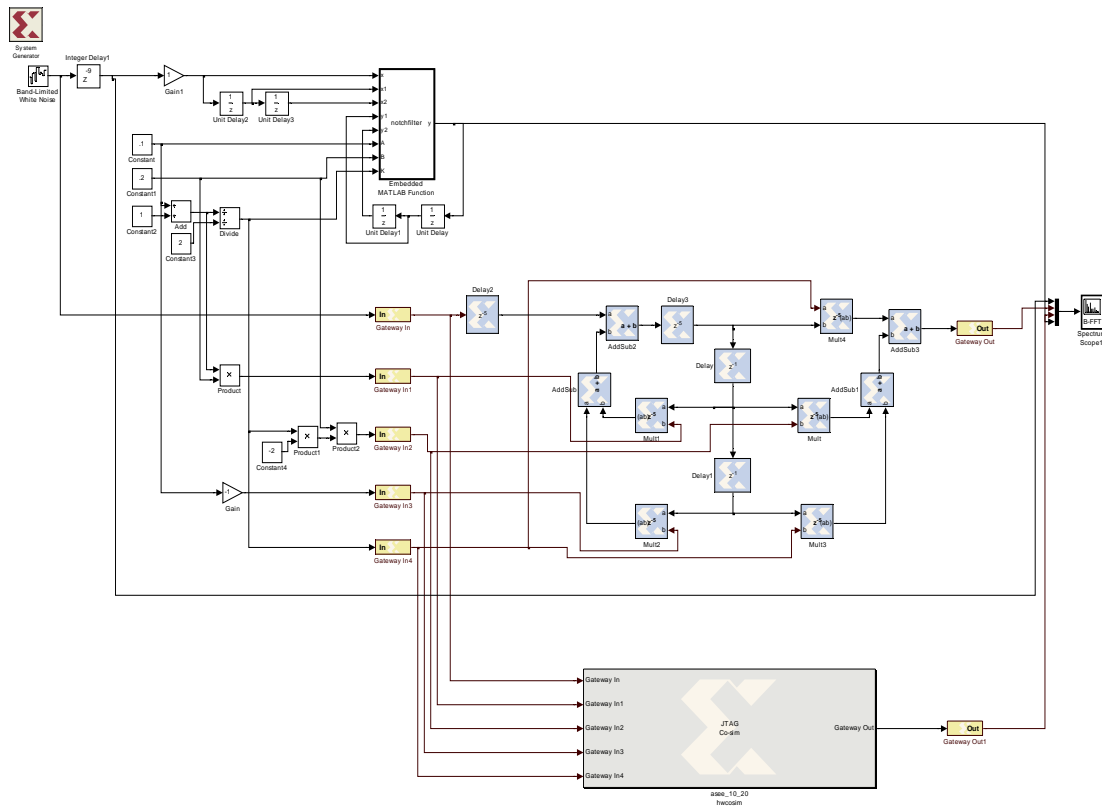


Figure 6: Simulink Model (Top), Un-Synthesized Hardware (Middle), Synthesized Hardware (Bottom)

After the synthesized block has been connected in the model, hardware co-simulation can be executed. During co-simulation, the synthesized design is downloaded to the target device (ML402 Virtex 4). The design then runs on the FPGA and communicates with the host PC via a JTAG cable. JTAG cable is a standard communication protocol that is widely used in Xilinx

development boards. JTAG allows Simulink to pass the inputs through the Gateway-In to the Virtex 4. The Virtex 4 then performs the notch filter as designed and passes the output back to Simulink through the Gateway-Out. This provides a means for comparing the simulation output with the actual hardware output. Shown in figure 7 and 8, are comparisons of the spectrums of the Simulink white noise input, Simulink notch filter output and hardware notch filter output. Figures 7 and 8 have data wordlengths of 8 bits and 12 bits respectively. The spectrum output of the 8-bit hardware implemented notch filter in figure 7 does not perform as well as the Matlab simulation. The spectrum output of the 12-bit hardware implementation notch filter in figure 8, however, performs similar to the Matlab Simulation.

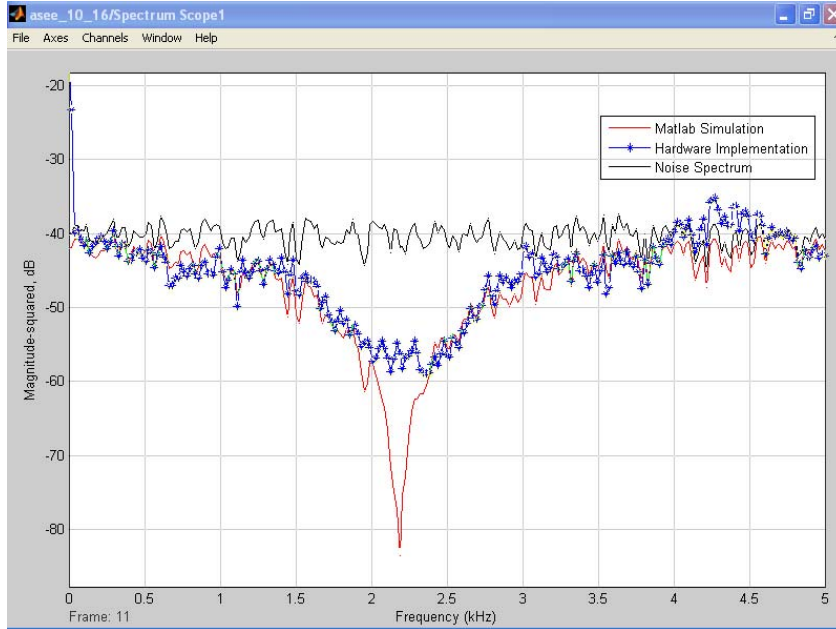


Figure 7: Filter Output Spectrum (8-bit word)

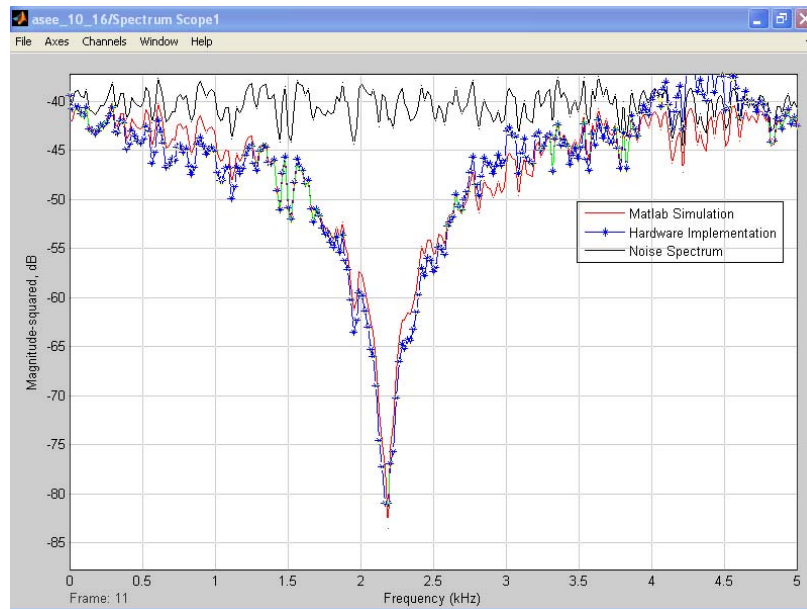


Figure 8: Filter Output Spectrum (12-bit word)

Conclusion

Teaching and understanding pertinent topics of Digital Signal Processing can no longer be limited to computer based simulation. Particularly, analysis of multirate filtering and filter structures can benefit from a hardware based approach. Hardware implementation of fundamental DSP topics introduces the student to effects that are oblivious to computer simulation. Specifically, finite wordlength effects, floating point vs. fixed point number representation and pipeline synchronization. Xilinx System Generator provides an easy to use software/hardware hybrid platform for basic to advanced hardware designs. Furthermore, students are not required to study hardware description language, which translates to a fluent introduction of System Generator into the course.

References

- [1] Sanjit K. Mitra, "Digital Signal Processing: A Computer Based Approach", McGraw Hill, 2006.
- [2] MathWorks, Simulink, <http://www.mathworks.com/products/simulink/>.
- [3] Xilinx, System Generator, http://www.xilinx.com/ise/optional_prod/system_generator.htm.
- [4] Xilinx, Virtex 4, http://www.xilinx.com/products/silicon_solutions/fpgas/virtex/virtex4/index.htm.
- [5] Xilinx, ML402, http://www.xilinx.com/products/boards/ml402/reference_designs.htm.

MAP DECODER IMPLEMENTED ON A FINE GRAINED FPGA

Robert Esposito, Dennis Silage
Temple University
Department of Electrical and Computer Engineering
resposit@temple.edu, silage@temple.edu

Abstract

This paper presents a hardware implementation of a maximum a-posteriori probability (MAP) decoder which is the computational core of Turbo Decoding. MAP decoding comprises complex computations such as exponentiations, logarithms and numerous multiplications and divisions. Thus, the MAP algorithm is typically converted to the log domain which greatly reduces its inherent complexity. However, in the log domain MAP, the logarithm of the sum of exponentials are typically estimated by a maximum function and corrected by a Jacobian function. The Jacobian correction is a complex function that is typically estimated by a look-up table. Thus, the performance of the log domain MAP decoder is somewhat compromised because of the correction. With the increase in the computational prowess of the fine grained field programmable gate array (FPGA), the implementation of the uncompromised MAP decoder is possible, thereby demonstrating an increase in performance.

1. Introduction

In telecommunications, bit error correction is important in assuring reliable digital communication [2, 3, 4]. Convolutional coding is a channel coding technique that has been utilized to provide error correcting capabilities by including additional parity bits to the transmission. One algorithm utilized to decode convolutional codes is maximum a-posteriori probability (MAP) decoding [1]. MAP is a salient bit error correcting algorithm that finds the most probable path through the trellis for the particular convolutional code utilized.

MAP decoding has been shown to be the computational core of some Turbo Coding communications systems [6]. Typically Turbo Codes utilize two MAP decoders coupled together in a recursive structure. After a half iteration, each MAP decoder makes a soft estimate of the transmitted bits. This estimation is then utilized to adjust the a-priori

probability of the next MAP decoder in the Turbo decoder. After numerous iterations, a hard estimation of the transmitted bit is made.

MAP decoding comprises numerous complex computations such as exponentiations, logarithms, multiplications and division. Typically, the MAP decoding algorithm is converted to the log domain where multiplications become simpler addition operations. Furthermore, the computation of the logarithm of a sum of exponentiations is estimated by a maximum operation and corrected to a degree by a Jacobian function. However, the estimation of the Jacobian function by a look-up table results in a decrease in error correction performance.

This paper demonstrates a parallel hardware design in a fine grained field programmable gate array (FPGA) and focuses on implementing the exact complex architecture of the MAP decoder in order to realize its full potential. Independent computations are performed in parallel on a Xilinx Virtex 4 fine FPGA wherever possible. Furthermore, complex functions are dealt with by utilizing techniques to maximize system throughput, facilitating real-time MAP decoding.

This paper is organized with a brief review of MAP decoding in Section 2; the FPGA implementation is described in Section 3; benchmarking the FPGA hardware implementation versus a sequential processor is described in Section 4; finally, Section 5 provides the conclusions and implications of this work.

2. Review of MAP Decoding

MAP decoding is based on a transmitted convolutional code which comprises an information bit and parity bits that are directly generated by the coder itself. The objective of the MAP decoder is to utilize the noise corrupted information and parity bits in order to correctly estimate the information bit. Initially, the MAP decoder assumes a-priori bit probability of 0.5 for a logic 1 and logic 0. For the received bits, the MAP process generates a soft output

in the form of a-posteriori probability. The MAP process utilizes the log-likelihood ratio based on the a-posteriori estimate [8]:

$$\Lambda(c_i) = \log \frac{P_r\{c_i = 1 | r\}}{P_r\{c_i = 0 | r\}}$$

r : received bit

c_i : hard bit estimate

The MAP process uses this soft output ratio and compares it to a threshold value of nominally zero to compute the hard bit estimate [8]:

$$c_i = \begin{cases} 1 & \text{if } \Lambda(c_i) \geq 0 \\ 0 & \text{otherwise} \end{cases}$$

Although the intent here is to compute the log-likelihood ratio and compare it to a threshold, it must be in a form that can be computed. It has been shown that the log-likelihood ratio can be written as [8]:

$$\Lambda(c_i) = \log \frac{\sum_{(l',l) \in B_i^1} \alpha_{t-1}(l') \gamma_t^1(l',l) \beta_t(l)}{\sum_{(l',l) \in B_i^0} \alpha_{t-1}(l') \gamma_t^0(l',l) \beta_t(l)}$$

where

l = present state

l' = previous state

t = time

Here, alpha is the joint probability of being in the present state given the received bits from time 1 to time t and is known as the forward estimator. Alpha starts from the beginning of the data frame trellis and estimates the optimal path to the end of the data frame trellis [8].

$$\alpha_t(l) = P_r\{S_t = l, r_t^t\}$$

Beta then is the conditional probability of receiving the bits from time t+1 given the present state and is known as the backward estimator. Beta is computed in the reverse of alpha and starts from the end of the data frame trellis and estimates the optimal path to the beginning of the data frame trellis.

$$\beta_t(l) = P_r\{r_{t+1}^t | S_t = l\}$$

Finally, gamma is the state transition probability. Gamma utilizes the Euclidean distance between the received noise corrupted symbols and the possible

transmitted symbols. Each symbol comprises an information bit and a parity bit [8].

$$\gamma_t^i(l',l) = P_r\{c_t = i, S_t = l, r_t^i | S_{t-1} = l'\}$$

The entire algorithm can be simplified and summarized as follows [8]:

1) Euclidean Distance:

- For all branches in the trellis calculate γ as

$$\gamma_t^i(l',l) = p_t(i) \exp\left(\frac{-d^2(r_t, x_t)}{2\sigma^2}\right) \text{ for } i = 0,1$$

where $p_t(i)$ is the apriori probability of each bit and $d^2(r_t, x_t)$ is the squared Euclidean distance between r_t and x_t

2) Forward Recursion:

- Initialize alpha

$$\alpha_0(0) = 1 \text{ and } \alpha_0(l) = 0 \text{ for } l \neq 0$$

and calculate $\alpha_t(l)$

for $t = 1, 2, \dots, \tau$ and $l = 0, 1, \dots, \text{LastState}$

where

$$\alpha_t(l) = \sum_{l'}^{\text{LastState}-1} \sum_{i \in \{0,1\}} \alpha_{t-1}(l') \gamma_t^i(l',l)$$

3) Backward Recursion:

- Initialize

$$\beta_\tau(0) = 1 \text{ and } \beta_\tau(l) = 0 \text{ for } l \neq 0$$

calculate $\beta_t(l)$

for $t = \tau-1, \dots, 1, 0$ and $l = 0, 1, \dots, \text{LastState}$

where

$$\beta_t(l) = \sum_{l'}^{\text{LastState}-1} \sum_{i \in \{0,1\}} \beta_{t+1}(l') \gamma_{t+1}^i(l, l')$$

3) Log Likelihood Ratio (LLR):

- Calculate

$\Lambda(c_t)$ for $t = 1, 2, \dots, \tau - 1$

where

$$\Lambda(c_t) = \log \frac{\sum_{l=0}^{\text{LastState}-1} \alpha_{t-1}(l) \gamma_t^1(l', l) \beta_t(l)}{\sum_{l=0}^{\text{LastState}-1} \alpha_{t-1}(l) \gamma_t^0(l', l) \beta_t(l)}$$

The MAP decoder as described by the equations above utilizes numerous multiplications and numerous complex functions, including the logarithm and the exponential. The implementation of alpha, beta, gamma and lambda in computational hardware is described in detail in Section 3.

3. Hardware Implementation of MAP Decoding

As described in Section 2, MAP decoding involves multiple independent computations. Each state in the MAP decoder is independent and thus may be computed in parallel. Also, alpha and beta are independent of each other and also may be computed in parallel. A block diagram showing the parallel nature of the hardware implementation is shown in Figure 1.

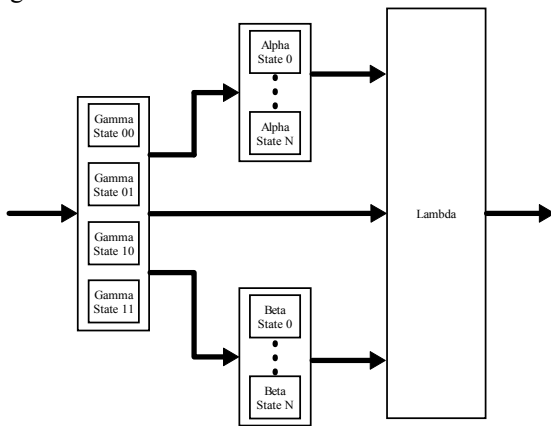


Figure 1. Overall MAP Decoder

An illustrative example of a rate 1/2 encoder is presented here. A rate 1/2 encoder requires that the decoder receives one parity bit for every information bit. The hardware implementations of each of the algorithmic steps for of alpha, beta, gamma and lambda in Section 2 are now described.

3.1 Gamma

Gamma computes the squared Euclidian distance between the received noisy symbols and the four

possible $(-1, -1)$, $(-1, +1)$, $(+1, -1)$, $(+1, +1)$ transmitted symbols in the rate 1/2 decoder. The hardware description of the gamma computation is shown in Figure 2.

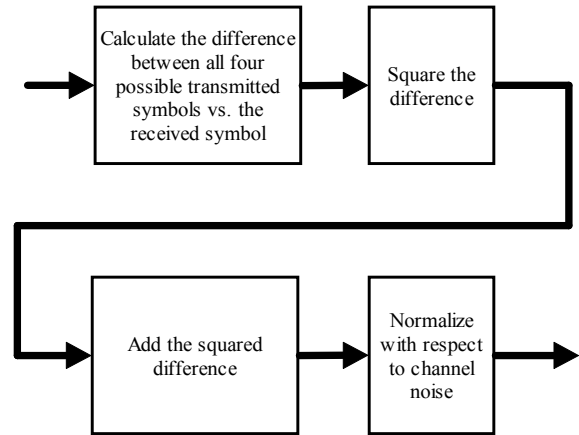


Figure 2. Gamma Euclidean Distance

Calculating the difference between the four possible transmitted symbols and the received symbol is performed with two parallel hardware subtractors for each symbol. Squaring the difference then is accomplished with two parallel hardware multipliers. Furthermore, adding the squared difference and normalizing with respect to the channel noise is accomplished by a single hardware adder and multiplier.

Exponentiation is the final and most computationally intensive step of the gamma calculation and is not an easy task to perform in hardware. As part of the presentation here, the hardware implementation of the exponential computation is performed using an indirect look-up table (LUT). An indirect LUT performs a specific computation before and after the look-up, which provides a benefit. For example, a direct LUT for 18 bit words would require 262,144 entries or 4,718,952 bits. This number of entries is too large to be reasonably accommodated in hardware. In order to reduce the amount of entries in the LUT to a reasonable number, an identity is used where:

$$e^{(A+B)} = e^A e^B$$

By breaking the 18-bit data word essentially into two 9-bit words, it is possible to use two smaller parallel LUTs of size 512 and a single multiplication. This reduces the number of LUT entries from 262,144 to a more reasonable 1024. It should be noted that the data word can be broken down into any number of bit

aggregates to decrease the number of LUT entries, however this requires more hardware multipliers. The hardware description of the exponentiation is shown in Figure 3.

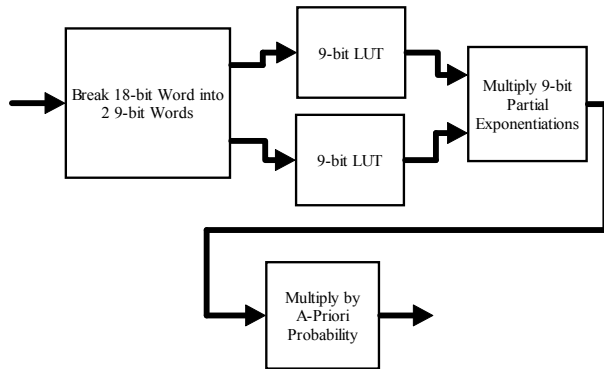


Figure 3. Gamma Exponential

Thus to implement the entire gamma computation for all four symbols in the rate 1/2 decoder in computational hardware, the following resources are required: 8 subtractors, 20 multipliers, 8 blocks of memory and 4 adders. Each stage of the gamma computation was performed in a single clock cycle, resulting in a total latency of 7 clock cycles.

3.2 Alpha/Beta

The forward recursion of alpha is the second calculation in the MAP decoding process. It is essentially a calculation that starts at the beginning and then proceeds to the end of the trellis for the entire data frame. As it proceeds through the trellis, the process utilizes the pre-calculated gamma values at each branch node and the previous alpha value to calculate a weighted path through the trellis. Each alpha value will have two possible paths at each node in the trellis (logic 0 and logic 1) and therefore will require two multiplications. Therefore, the number of multipliers and adders in the alpha computation are directly related to the number of states N in the decoder. This is because alpha can be implemented in parallel because of its state independence. The hardware description of the alpha computation is shown in Figure 4.

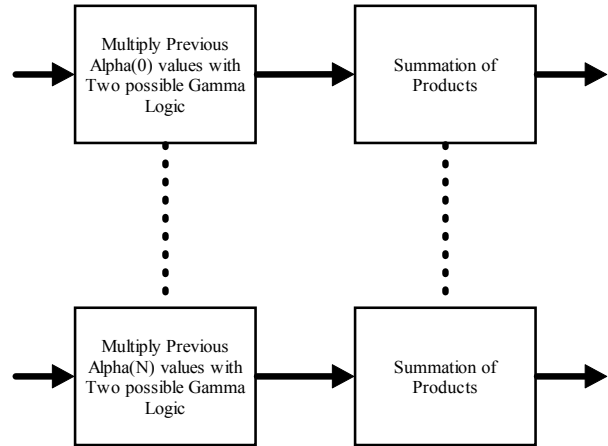


Figure 4. Alpha

Thus, to implement the alpha computation for all four symbols in the rate 1/2 decoder in computational hardware, the following resources are required: $2N$ multipliers and N adders. Each stage of alpha is performed in a single clock cycle, therefore resulting in a total latency of 2 clock cycles.

Since alpha is a recursive computation that multiplies probabilistic numbers between 0 and 1, it can be limited by data which bit underflows. As alpha gets smaller with each iteration, the data converges towards zero which quickly underflows in fixed bit computational hardware.

Note that alpha is utilized in the log-likelihood ratio (LLR) which is a ratio and can be normalized after each iteration in order to avoid the possibility of bit underflow. However, the present alpha value depends on the previous alpha value. This requires that the alpha value to be computed in just one clock cycle or the hardware system would have to perform up-sampling.

Up-sampling would reduce the throughput of the system by a factor equivalent to the sampling factor and therefore needs to be avoided. Thus, normalizing the alpha values can not be performed by latency prone division.

In this presentation, normalization is performed by a bit shifting scheme. First, the maximum of the alpha values is determined. Then the maximum value is shifted so that its first non-zero bit is in the most significant position. At the same time, the other alpha values are shifted by the same amount. This technique ensures that the multiplication of the alpha computation and normalization occur in just one clock cycle. The normalization scheme utilized is shown in Figure 5.

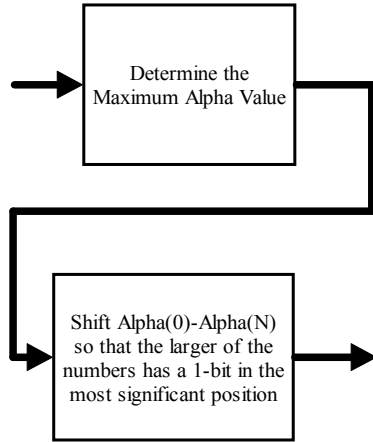


Figure 5. Recursive Normalization

Beta is the same computation as alpha with the exception of reversing the order of the data frame. Since beta works from the end of the frame to the beginning, it is necessary to buffer and then reverse all four gamma computations. This is accomplished with blocks of memory that act as a first-in last-out (FILO) buffer. The number of memory blocks required for the beta computation is $4+N$. Thus, the beta computation requires blocks of memory which are a hardware resource not required to compute alpha.

3.3 Lambda

The lambda log-likelihood ratio (LLR) is the soft estimate of the MAP decoder. The natural logarithm of a ratio using the alpha, beta and gamma values is calculated with respect to a logic 0 and logic 1.

After lambda is calculated, a hard decision is made with the threshold set at zero. Positive values are decided as logic 1 and negative values as logic 0. The LLR process consists of three steps. The first step is the multiplication and summation of alpha, beta and gamma. The only difference between the numerator and denominator in the computation is that lambda is calculated with respect to either a logic 0 input or logic 1 input.

The number of multipliers and adders in the lambda computation are directly related to the number of states in the decoder. This is because the numerator and denominator computation for lambda can be implemented in parallel because of its state independence. The hardware description of the numerator and denominator in the lambda computation is shown in Figure 6.

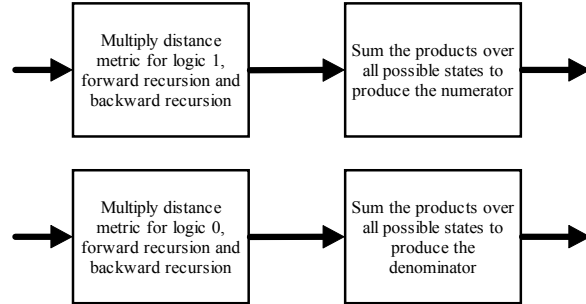


Figure 6. Numerator and Denominator of Lambda

Thus, to implement the numerator and denominator of the lambda computation, the following hardware resources are required: $4N$ multipliers and $2N$ adders. The multiplications and additions are performed in 3 clock cycles for each state. Because each state as well as the numerator and denominator are independent, the computation is performed with a total latency of only 3 clock cycles.

The second step of the LLR is more computationally intensive than the first. A possible implementation of the LLR is the division of the numerator and denominator obtained in the first step, followed by a logarithm of the ratio. However, this implementation has a possible fault in that the division may produce either a very large or a very small number. This is due to the fact that the numerator and denominator of the ratio is the probability that the bit is a logic one and logic zero respectively.

For an 18 bit example, the ratio may be $0.999/2^{18}$, which results in a very large number that would require 18 integer bits to represent. In another example, the ratio may be $2^{-18}/0.999$, which results in a very miniscule number that would require 18 fractional bits to represent. This extreme range requires a large number of integer as well as fractional bits which may be unwarranted in some fixed point hardware systems.

An alternative to this possible implementation is an implementation that is more efficient in a computational system with a fixed number of bits. This implementation employs the identity:

$$\log(A/B) = \log(A) - \log(B)$$

Therefore, in this more efficient implementation, the sequential division and then logarithm of the first implementation is replaced by two logarithms and a subtraction. Furthermore, the two logarithmic calculations may be computed in parallel since they are independent.

A major advantage to this implementation is that the extreme range required by the division in the initial implementation is avoided because the logarithm is computed separately for two numbers in the range 0-1.

Polynomial representation of a function is useful in computational hardware, because complex functions can be calculated with multiplications and additions [6]. The Taylor series representation is chosen for the computation of the natural logarithm needed to compute the LLR. In order to compute the Taylor series, the following hardware pipeline must be implemented. First, the value must be scaled in the range [0.5, 1) in order to obtain proper convergence of the series. Second, at least the first fourteen terms of the series is calculated in order to obtain a reasonably accurate result. Third, the result must be rescaled. Finally, the terms of the series must be accumulated. The hardware description of the logarithm is shown in Figure 7.

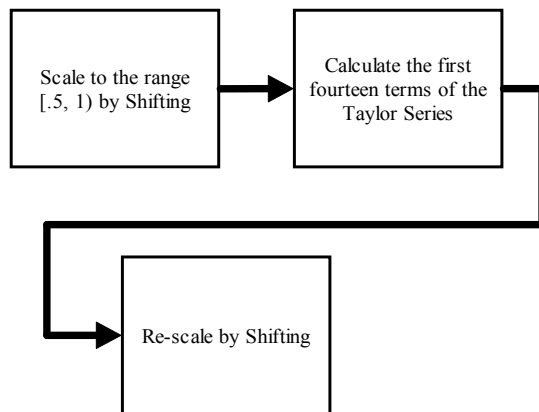


Figure 7. Taylor Series Lambda Logarithm

Certain terms in the Taylor series may be computed in parallel. Specifically, the third-fourth, fifth-eighth and ninth-fourteenth terms may be computed in parallel. This further decreases the hardware latency of the lambda computation.

Thus to implement the logarithm of the lambda computation, the following hardware resources are required: 18 multipliers and 15 adders. The multiplications and additions are independent from the states and therefore do not increase with an increase in the number of states. Because the numerator and denominator logarithms are independent, the computation is performed with a total latency of 7 clock cycles. Therefore, the entire lambda computation, consisting of numerator, denominator and the logarithm, is performed in 10 clock cycles.

4. Hardware Benchmarking

A salient reason for implementing the MAP decoding now in fine grained FPGA computational hardware is to obtain the error correcting capabilities of the algorithm while decreasing execution time. Execution time is of particular concern when dealing with large data sets.

In traditional software systems, the MAP algorithm may be computed by a sequential processor. Thus it is instructive to compare the execution time of the FPGA hardware implementation against that of the apparent execution time of a sequential processor system. The following benchmark results are a comparison of the Matlab/Xilinx System Generator FPGA hardware implementation executing on a Xilinx Virtex-4 SX35 device at a clock frequency of 100 MHz [5, 8, 9, 10] and a C language implementation executing on a Intel 3.2 GHz Dual Core Pentium processor. The execution times, as a throughput in Mb/sec, of the FPGA hardware and the traditional sequential processor implementation for the MAP decoder algorithm for various data frame sizes are shown in Table 1.

Table 1. FPGA Hardware versus Sequential Processor Throughput for MAP Decoder

Data Frame Size	Sequential Processor Throughput (Mb/sec)	FPGA Hardware Throughput (Kb/sec)	FPGA Hardware Apparent Increase
256	100	390	256x
2048	100	386	259x
8192	100	380	263x

It is apparent that the fine grained FPGA hardware implementation has significantly reduced the execution time compared to that of the processor system. The decrease in execution time is primarily due to the parallel computations and the intrinsic pipelined architecture of the FPGA.

The FPGA pipelined architecture facilitates the simultaneous computation of multiple data sets. Specifically, once the hardware pipeline is full, a MAP estimate is outputted every clock cycle. Furthermore, the high speed I/O capabilities of the FPGA hardware platform, such as Ethernet and USB ports, facilitate high speed data communication to the Virtex 4 FPGA [9, 10, 11].

The FPGA hardware pipeline has an observed initial delay of $2F + 15$ clock cycles where F is the size of the frame. This means that it requires approximately 164 μ s at 100 MHz for the pipeline to be filled. However, once the FPGA hardware pipeline is fully loaded with data, every single clock cycle thereafter produces a new MAP decoder bit estimate.

More significant is the fact that the FPGA hardware here is only executing at a clock frequency of 100 MHz, whereas the microprocessor (albeit a dual core processor) is running at 3.2 GHz or 32 times faster. Thus even with a significantly faster clock, the sequential processor is agonizingly slow as compared to the fine grained FPGA hardware.

The Xilinx ML402 development board is the platform of choice for this research. The development board comprises a Xilinx Virtex-4 SX35 FPGA with 192 embedded hardware multipliers [11]. A summary of the important FPGA multiplier, adder and block memory resources utilized to implement this MAP decoder algorithm for various frame sizes and states is shown in Table 2.

Table 2. Summary of Resource Utilization Virtex-4 SX35 MAP Decoder

Data Frame Size	2 States	8 States	16 States
256	54 Mult. 35 Add. 16 Mem.	102 Mult. 59 Add. 28 Mem.	166 Mult. 91 Add. 44 Mem.
2048	54 Mult. 35 Add. 32 Mem.	102 Mult. 59 Add. 56 Mem.	166 Mult. 91 Add. 88 Mem.
8192	54 Mult. 35 Add. 128 Mem.	102 Mult. 59 Add. 224 Mem.	166 Mult. 91 Add. 352 Mem.

The only configuration that does not apparently hardware synthesize on the Xilinx Virtex 4 SX35 FPGA is the 8 and 16 state configuration for the 8192 size frame. This is because the Xilinx Virtex 4 only has 192 blocks of memory (BRAM). However, other the 7 other configurations do fit on the Virtex 4 SX35 FPGA.

5. Conclusion

We proposed a fine grained FPGA hardware implementation of a MAP decoder. Even though the MAP algorithm is computationally complex, a recent fine grained FPGA such as the Xilinx Virtex 4 are capable of implementing such algorithms. Thus, in the present and especially in the future, expedient computations such as converting the MAP algorithm to the log domain may not have to be relied upon.

A significant decrease in execution time was realized for the fine grained FPGA as compared to the sequential processor implementation of the MAP decoder algorithm. The decrease in execution time and thus throughput can be primarily attributed to the

parallelization of the MAP algorithm and the pipelined architecture of the FPGA.

6. References

- [1] Bahl R., "Optimal Decoding of Linear Codes for Minimizing Symbol Error Rate", *IEEE Transactions on Information Theory*, March 1974, pp 284-287.
- [2] Berrou C., "Near Shannon Limit Error-Correcting Coding and Decoding: Turbo-Codes (1)", *Integrated Circuits for Telecommunication Laboratory*, 1993.
- [3] Haykin S., "Communication Systems", *Wiley and Sons, Inc. Publisher, ISBN: 0-471-17869-1*, 2001
- [4] Haykin S., "Modern Wireless Communications", *Prentice-Hall, Inc. Publisher, ISBN: 0-13-022472-3*, 2005.
- [5] Mathworks, "Matlab", available at <http://www.mathworks.com>, last accessed May 2008.
- [6] Parahami B., "Computer Arithmetic Algorithms and Hardware Designs", *Oxford University Press Publisher, ISBN: 0-19-512583-5*, 2000.
- [7] Shannon C., "A Mathematical Theory of Communications", *Bell Systems Technical Journal*, vol. 27, July 1948, pp 379-423, pp 623-656.
- [8] Vucetic B., "Turbo Codes: Principles and Applications", *Kluwer Academic Publishers, ISBN: 0-7923-7868-7*, 2000.
- [9] Xilinx "ML402", available at http://www.xilinx.com/support/documentation/board_s_and_kits/ug080.pdf, last accessed May 2008.
- [10] Xilinx, "System Generator", available at http://www.xilinx.com/support/documentation/sw_manuals/sysgen_bklist.pdf, last accessed May 2008.
- [11] Xilinx, "Virtex 4", available at http://www.xilinx.com/support/documentation/user_guides/ug070.pdf, last accessed May 2008.

SINGLE MAP DECODER FOR TURBO DECODING IMPLEMENTED ON A FINE GRAINED FPGA

Robert Esposito, Dennis Silage
Temple University

Department of Electrical and Computer Engineering
resposit@temple.edu, silage@temple.edu

Abstract

This paper presents a hardware implementation of a recursive structure Turbo decoder which is based on maximum a-posteriori (MAP) decoding. Turbo decoding comprises complex computations that require significant hardware resources such as multipliers, adders and block RAM (BRAM). However, in many hardware applications it is desirable to reduce the hardware resource utilization. Traditionally a Turbo decoder comprises two identical MAP decoders working in concert with one another to decode a given data frame. This paper addresses the inherent redundancy of the two MAP decoders and presents a Turbo decoder architecture utilizing a single MAP decoder effectively reducing the hardware utilization by half.

1. Introduction

In telecommunications, bit error correction is important in assuring reliable digital communication [3, 4, 6]. Convolutional coding is a channel coding technique that has been utilized in the past to provide error correcting capabilities by including additional parity bits to the transmission. One algorithm utilized to decode convolutional codes is maximum a-posteriori probability (MAP) decoding [1]. MAP is a salient bit error correcting algorithm that finds the most probable path through the trellis of the particular convolutional code utilized.

MAP decoding has been shown to be the computational core of some Turbo Coding communications systems [2, 7]. Typically Turbo Codes utilize two MAP decoders coupled together in a recursive structure. After a half iteration, each MAP decoder makes a soft estimate of the transmitted bits. This estimation is then utilized to adjust the a-priori probability of the next MAP decoder in the Turbo decoder. After numerous iterations, a hard estimation is made.

MAP decoding comprises numerous complex computations such as exponentiations, logarithms,

multiplications and division. Typically, the MAP decoding algorithm is converted to the log domain where multiplications become simpler addition operations. Furthermore, the computations of the logarithm of a sum of exponentiations is estimated by a maximum operation and corrected to a degree by a Jacobian function.

However, regardless of which decoding algorithm is utilized, numerous computational hardware resources are required. Hardware resources are directly dictated by the number of states in the decoder, which can be largely dependent on the application. Thus, some single field programmable gate array (FPGA) hardware platforms may not be able to accommodate a Turbo decoder. Demonstrated here is a Turbo decoder architecture that only utilizes a single MAP decoder to reduce hardware utilization.

This paper is organized with a brief review of the traditional Turbo decoder architecture in Section 2; the single MAP decoder based Turbo decoder is described in Section 3; benchmarking the FPGA hardware implementation versus a sequential processor is described in Section 4; finally, Section 5 provides the conclusions and implications of this work.

2. Review of Traditional Turbo Decoding

Turbo decoding is based on a transmitted convolutional code which comprises an information bit and parity bits that are directly generated by the encoder itself. The objective of the Turbo decoder is to utilize the noise corrupted information and uncorrelated parity bits in order to correctly estimate the information bit. The traditional Turbo decoder structure utilizes two MAP decoders and is shown in Figure 1 [7].

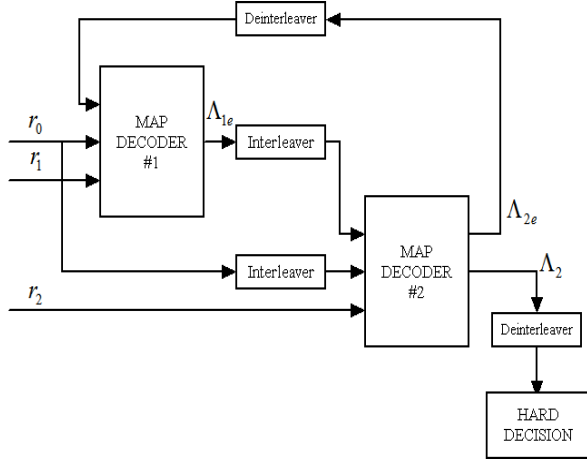


Figure 1. Dual MAP Decoder Turbo Decoder

In Figure 1, r_0 are the information bits, r_1 are the parity bits for the first decoder, r_2 are the parity bits for the second decoder, Λ_{1e} is the extrinsic information computed by the first decoder, Λ_{2e} is the extrinsic information computed by the second decoder and Λ_2 is the log-likelihood ratio of the second decoder. MAP decoding can be summarized as follows [7]:

1) Euclidean Distance:

- For all branches in the trellis calculate γ as

$$\gamma_i^j(l', l) = p_i(i) \exp\left(\frac{-d^2(r_i, x_i)}{2\sigma^2}\right) \text{ for } i = 0, 1$$

where $p_i(i)$ is the apriori probability of each bit and $d^2(r_i, x_i)$ is the squared Euclidean distance between r_i and x_i

2) Forward Recursion:

- Initialize alpha

$$\alpha_0(0) = 1 \text{ and } \alpha_0(l) = 0 \text{ for } l \neq 0$$

and calculate $\alpha_t(l)$

for $t = 1, 2, \dots, \tau$ and $\ell = 0, 1, \dots, \text{LastState}$

where

$$\alpha_t(\ell) = \sum_{\ell'}^{\text{LastState}-1} \sum_{i=(0,1)} \alpha_{t-1}(\ell') \gamma_t^i(\ell', \ell)$$

3) Backward Recursion:

- Initialize

$$\beta_\tau(0) = 1 \text{ and } \beta_\tau(l) = 0 \text{ for } l \neq 0$$

calculate $\beta_t(\ell)$

for $t = \tau - 1, \dots, 1, 0$ and $\ell = 0, 1, \dots, \text{LastState}$

where

$$\beta_t(\ell) = \sum_{\ell'}^{\text{LastState}-1} \sum_{i=(0,1)} \beta_{t+1}(\ell') \gamma_{t+1}^i(\ell', \ell)$$

4) Log Likelihood Ratio (LLR):

- Calculate

$$\Lambda(c_t) \text{ for } t = 1, 2, \dots, \tau - 1$$

where

$$\Lambda(c_t) = \log \frac{\sum_{\ell=0}^{\text{LastState}-1} \alpha_{t-1}(\ell') \gamma_t^1(\ell', l) \beta_t(l)}{\sum_{\ell=0}^{\text{LastState}-1} \alpha_{t-1}(\ell') \gamma_t^0(\ell', l) \beta_t(l)}$$

Initially, the MAP decoder assumes a-priori bit probability of 0.5 for a logic 1 and logic 0. For the received bits, the MAP decoder generates a soft output in the form of probabilistic extrinsic information that is passed to the next MAP decoder in the sequence. The extrinsic information is then used to update the a-priori bit probability utilized in the next MAP decoder. The extrinsic information can be summarized as follows:

1) Compute the extrinsic information for the first MAP decoder and pass it to the second MAP decoder to update the a-priori bit probability.

$$\Lambda_{1e}^{(r)}(c_t) = \Lambda_1^{(r)}(c_t) - \frac{2}{\sigma^2} r_{t,0} - \tilde{\Lambda}_{2e}^{(r-1)}(c_t)$$

2) Compute the extrinsic information for the second MAP decoder and pass it to the first MAP decoder to update the a-priori bit probability.

$$\Lambda_{2e}^{(r)}(c_t) = \Lambda_2^{(r)}(c_t) - \frac{2}{\sigma^2} \tilde{r}_{t,0} - \tilde{\Lambda}_{1e}^{(r)}(c_t)$$

With each iteration, the log-likelihood ratio (LLR) of each decoder becomes seemingly more accurate. After a number of iterations, a hard decision is made on the LLR of the second MAP decoder.

The dual MAP decoder architecture, as shown in Figure 1, is traditionally utilized in Turbo decoders. This redundancy utilizes approximately twice the

hardware resources that are necessary for Turbo decoding.

3. Single MAP Decoder Turbo Decoder

As described in Section 2, traditional Turbo decoding utilizes a dual MAP decoder architecture. Both of the MAP decoders perform identical computations and therefore have identical hardware. This redundant hardware increases the throughput of the system at the expense of costly and somewhat scarce hardware resources. However, this redundant hardware is not necessary to perform Turbo decoding. It is reasonable that a Turbo decoder could be configured with a single MAP decoder. Shown in Figure 2 is the overall single MAP decoder Turbo decoder.

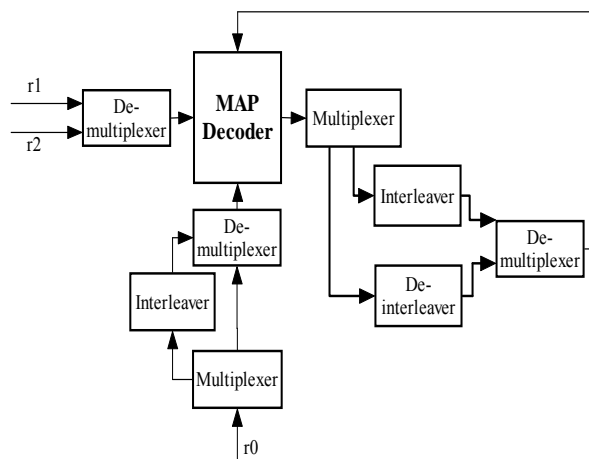


Figure 2. Single MAP decoder Turbo Decoder

As shown in Figure 2, the information bits and parity bits are input to the single MAP decoder via data multiplexers. The extrinsic information output of the single MAP decoder is fed back to the decoder via data multiplexers and data interleavers. The data multiplexers and de-multiplexers are necessary to route the input and output bits of the decoder through the proper hardware pipeline of interleavers and de-interleavers. In general, the single MAP decoder inputs and outputs are controlled as two phases (MAP1 and MAP2). The MAP1 and MAP2 phases are equivalent to the MAP Decoder #1 and MAP Decoder #2 of Figure 1, respectively. The detailed description of the Turbo decoder architecture, as shown in Figure 2, is described in Section 3.1 and Section 3.2.

3.1 MAP1 Phase

In the MAP1 phase, the single MAP decoder is equivalent to MAP Decoder #1 as in Figure 1. Specifically, the information bits r_0 are passed to the MAP decoder through a data multiplexer and a data demultiplexer. Likewise, the first parity bits r_1 are passed to the MAP decoder through a data demultiplexer. Thus, during the MAP1 phase, the single MAP decoder receives the information bits r_0 and parity bits r_1 similarly to MAP Decoder #1, as shown in Figure 1.

The extrinsic information produced by the MAP decoder is passed through a data multiplexer to a data interleaver. The interleaved extrinsic information is then fed back to the MAP decoder through a data demultiplexer similarly to the extrinsic information passing between MAP Decoder #1 and MAP Decoder #2, as shown in Figure 1. Thus, the MAP1 phase in the Turbo decoder as in Figure 2 is equivalent to a half iteration in the Turbo decoder as in Figure 1.

3.2 MAP2 Phase

In the MAP2 phase, the single MAP decoder is equivalent to MAP Decoder #2 as in Figure 1. Specifically, the information bits r_0 are interleaved and passed to the MAP decoder through an interleaver and a data demultiplexer. Likewise, the second parity bits r_2 are passed to the MAP decoder through a demultiplexer. Thus, during the MAP2 phase, the single MAP decoder receives the interleaved information bits r_0 and parity bits r_2 similarly to MAP Decoder #2 as in Figure 1.

The extrinsic information produced by the MAP decoder is passed through a data multiplexer to a data deinterleaver. The deinterleaved extrinsic information is then fed back to the MAP decoder through a data demultiplexer similarly to the extrinsic information passing between MAP Decoder #2 and MAP Decoder #1 as in Figure 1.

Once both the MAP1 and MAP2 phases are complete, the Turbo decoder has completed a full iteration. In both the MAP1 and MAP2 phases, the data frames input to the MAP decoder must be buffered to synchronize the iterations. The size of the buffer is equivalent to the size of the hardware pipeline times the data frame size. Thus, the single MAP decoder with the aid of buffers and multiplexers is able to perform the same function as the two MAP decoders in the traditional Turbo decoder. Data buffering is discussed in detail in chapter 4.

4. Hardware Benchmarking

The data throughput of the MAP decoder is dictated by the backward recursion computation. As discussed in Section 2, backward recursion requires the data frame to be completely received, reversed in time, computed and then re-reversed to its original received order. Thus, the MAP decoder has a latency of twice the frame size F of the data or $2F$ clock cycles.

Another factor that dictates the ultimate data throughput of the MAP decoder is the random interleaver and deinterleaver which have a latency equivalent to the frame size F of the data or F clock cycles. Thus, the traditional Turbo decoder as in Figure 1 has a latency of $6F$ clock cycles for each iteration. After the iterations are completed, the traditional Turbo decoder outputs six data frames.

Similarly, the single MAP decoder Turbo decoder as in Figure 2 has a latency of $6F$ clock cycles. However, the single MAP decoder Turbo decoder performs the data computation of two MAP decoders, and therefore only outputs half the data frames after the iterations are completed or three data frames.

Each of the three frames in the MAP1 phase are buffered in block RAM (BRAM) in the fine grained FPGA after each iteration so that three new frames in the MAP2 phase can be computed. The buffer must have a size of $3F$ to accommodate the three frames. Therefore, additional buffering of data frames is required in the single MAP decoder Turbo decoder architecture. Therefore, the single MAP decoder Turbo decoder presented here only utilizes half the hardware resources but only provides half of the overall throughput.

In traditional communications systems, Turbo decoding may be computed by a sequential processor. Thus may be also instructive to compare the execution time of the fine grained FPGA hardware implementation of the traditional Turbo decoder and single MAP decoder Turbo decoder against the apparent execution time of a sequential processor based system.

The following benchmark results are a comparison of the Matlab/Xilinx System Generator FPGA hardware implementation of the dual and single MAP Turbo decoder algorithm executing on a Xilinx Virtex 4 SX35 FPGA at a clock frequency of 100 MHz [5, 8, 9, 10], and a C language implementation of the dual MAP Turbo decoder algorithm executing on an Intel 3.2 GHz Dual Core Pentium microprocessor. The execution time, as a throughput in Mb/sec, of the FPGA hardware and sequential processor implementation for the Turbo decoding algorithm

utilizing various frame sizes and iterations are shown in Table 1.

Table 1. FPGA Hardware versus Sequential Processor Throughput for Dual and Single MAP Decoder Turbo Decoder

Frame Size:	1 Iteration	3 Iterations	7 Iterations
256:			
Dual MAP	100 Mb/sec	32.5 Mb/sec	13.9 Mb/sec
SingleMAP	50 Mb/sec	16.2 Mb/sec	7.0 Mb/sec
Processor	195 Kb/sec	27 Kb/sec	9.8 Kb/sec
2048:			
Dual MAP	100 Mb/sec	33.2 Mb/sec	14.2 Mb/sec
SingleMAP	50 Mb/sec	16.6 Mb/sec	7.1 Mb/sec
Processor	193 Kb/sec	26 Kb/sec	9.7 Kb/sec
8192:			
Dual MAP	100 Mb/sec	33.3 Mb/sec	14.3 Mb/sec
SingleMAP	50 Mb/sec	16.7 Mb/sec	7.1 Mb/sec
Processor	190 Kb/sec	25 Kb/sec	9.5 Kb/sec

As expected, the single MAP decoder Turbo decoder has approximately half the throughput of the traditional dual MAP decoder Turbo decoder in the fine grained FPGA hardware. However, it is also apparent that the FPGA hardware implementation has a significantly reduced execution time, or increased throughput, compared to that of the sequential processor system. The decrease in execution time is primarily due to the parallel computations and the intrinsic pipelined architecture of the fine grained FPGA. FPGA hardware in the pipelined architecture allows the simultaneous computation of multiple data sets. Furthermore, the high speed input/output (I/O) capabilities of the FPGA hardware platform, such as Ethernet and USB ports, can facilitate high speed data communication.

The FPGA hardware is only executing at clock frequency of 100 MHz, whereas the sequential processor (albeit a dual core processor) is running at 3.2 GHz or 32 times faster. Thus, even with a significantly faster clock, the sequential processor is agonizingly slow as compared to the fine grained FPGA hardware. Furthermore, the single MAP decoder Turbo decoder even at a large frame size (8192) and numerous iterations (7) attains a formidable throughput of over 7 Mb/sec with a clock of only 100 Mhz.

The Xilinx ML402 development board [8] is the platform of choice for this research. The development board comprises a Xilinx Virtex-4 SX35 FPGA with 192 embedded hardware multipliers [10]. A summary of the important FPGA multiplier, adder and block memory resources utilized to implement the traditional dual MAP decoder Turbo decoder and the single MAP

decoder Turbo decoder for various frame sizes and states is shown in Table 2 and Table 3.

Table 2. Summary of Resource Utilization (Virtex-4 SX35) Dual MAP Decoder Turbo Decoder

Data Frame Size:	2 States	8 States	16 States
256 (Dual MAP)	54 Mult. 35 Add. 16 Mem.	102 Mult. 59 Add. 28 Mem.	166 Mult. 91 Add. 44 Mem.
2048 (Dual MAP)	54 Mult. 35 Add. 32 Mem.	102 Mult. 59 Add. 56 Mem.	166 Mult. 91 Add. 88 Mem.
8192 (Dual MAP)	54 Mult. 35 Add. 128 Mem.	102 Mult. 59 Add. 224 Mem.	166 Mult. 91 Add. 352 Mem.

Table 3. Summary of Resource Utilization (Virtex-4 SX35) Single MAP Decoder Turbo Decoder

Data Frame Size	2 States	8 States	16 States
256 (Single MAP)	27 Mult. 18 Add. 11 Mem.	51 Mult. 30 Add. 17 Mem.	83 Mult. 45 Add. 25 Mem.
2048 (Single MAP)	27 Mult. 18 Add. 22 Mem.	51 Mult. 30 Add. 34 Mem.	83 Mult. 45 Add. 50 Mem.
8192 Single MAP)	27 Mult. 18 Add. 76 Mem.	51 Mult. 30 Add. 124 Mem.	83 Mult. 45 Add. 191 Mem.

The 8 and 16 state configurations for the 8192 size frame dual MAP decoder Turbo decoder configuration apparently can not be synthesized on the Xilinx Virtex 4 SX35 FPGA, as shown in Table 2. This is because the Xilinx Virtex 4 SX35 FPGA only has 192 blocks of memory (BRAM).

However, since the single MAP decoder Turbo decoder utilizes only half the available FPGA hardware resources, Table 2 shows that all of the configurations can be synthesized on the Virtex 4 SX35 FPGA. Furthermore, a comparison of Table 2 and Table 3 shows that the number of BRAMS utilized for the single MAP decoder are only slightly more than half that of the dual MAP decoder Turbo decoder. These additional BRAMS are due to the buffering of the three data frames required between the MAP1 phase and MAP2 phase of the decoder.

5. Conclusion

We proposed a fine-grained FPGA hardware implementation of a single MAP decoder Turbo decoder. Thus, the redundant hardware of the traditional dual MAP decoder Turbo decoder is eliminated. Decoders with large frame sizes and large numbers of states can now be implemented on FPGA systems with smaller hardware resources.

A hardware resource decrease of roughly one half was attained at the expense of a throughput decrease of roughly one half. Thus, in systems where hardware utilization is more of a concern than throughput, the single MAP decoder Turbo Decoder would be a reasonable choice.

6. References

- [1] Bahl R., "Optimal Decoding of Linear Codes for Minimizing Symbol Error Rate", *IEEE Transactions on Information Theory*, March 1974, pp 284-287.
- [2] Berrou C., "Near Shannon Limit Error-Correcting Coding and Decoding: Turbo-Codes (1)", *Integrated Circuits for Telecommunication Laboratory*, 1993.
- [3] Haykin S., "Communication Systems", *Wiley and Sons, Inc. Publisher, ISBN: 0-471-17869-1*, 2001
- [4] Haykin S., "Modern Wireless Communications", *Prentice-Hall, Inc. Publisher, ISBN: 0-13-022472-3*, 2005.
- [5] Mathworks, "Matlab", available at <http://www.mathworks.com>, last accessed May 2008.
- [6] Shannon C., "A Mathematical Theory of Communications", *Bell Systems Technical Journal*, vol. 27, July 1948, pp 379-423, pp 623-656.
- [7] Vucetic B., "Turbo Codes: Principles and Applications", *Kluwer Academic Publishers, ISBN: 0-7923-7868-7*, 2000.
- [8] Xilinx "ML402", available at http://www.xilinx.com/support/documentation/boards_and_kits/ug080.pdf, last accessed May 2008.
- [9] Xilinx, "System Generator", available at http://www.xilinx.com/support/documentation/sw_manuals/sysgen_bklist.pdf, last accessed May 2008.
- [10] Xilinx, "Virtex 4", available at http://www.xilinx.com/support/documentation/user_guides/ug070.pdf, last accessed May 2008.