For anything worth having one must pay the price;
and the price is always work, patience, love, self-sacrifice.
No paper currency, no promises to pay, but the gold of real service.
John Burroughs, 1837 - 1921

University of Alberta

A Low Power Parallel Processor Implementation of a Turbo Decoder

by

Marco Alejandro Castellón ©

A thesis submitted to the Faculty of Graduate Studies and Research in partial fulfillment of the requirements for the degree of **Master of Science**.

Department of Electrical and Computer Engineering

Edmonton, Alberta
Spring 2006

Your file Votre référence
ISBN: 0-494-13799-1
Our file Notre référence
ISBN: 0-494-13799-1

Para mi preciosa esposa Rosa, quien ha sido mi
fortaleza en mis momentos débiles; y para
nuestros hijos Gabriel Alejandro y Marco Angelo,
quienes me inspiran a ser un mejor hombre.

Abstract

A novel parallel decoding algorithm for turbo codes is presented, along with its implementation on an embedded Single-Instruction Stream, Multiple-Data Streams (SIMD) processor. The novelty of the parallel algorithm is the simultaneous computation of state metrics and log-likelihood ratios for all trellis stages in the constituent decoder. The results are then interleaved prior to parallel decoding in the subsequent constituent decoder. Implementation of the constituent decoder using the massively parallel SIMD Array Processor of the Atsana Semiconductor J2210 Media Processor achieves speedup factors of 10 or greater for data packet sizes in excess of 512 data symbols when compared to its sequential counterpart as executed by an ARM922T$^{\mathrm{TM}}$ processor. The bit error rate performance of the parallel processor turbo decoder implementation lies within 0.1 dB from that of the floating-point reference. The Processors-In-Memory architecture of the SIMD array processor offers a 24% reduction in energy consumption when compared to the low-power ARM922T$^{\mathrm{TM}}$ core.

Acknowledgements

I am grateful for the donation of development tools and for support provided for this research work by Atsana Semiconductor Corporation.

I would like to thank Fengqin Zhai from the Communications Research Laboratory for helping me understand turbo decoding and for providing source code that served as a stepping stone for my research. Thanks to Yan Xin, for his technical support and for helping to set up simulations in the Communications Research Lab. A special thank you to Dr. Vincent Gaudet for always being available to answer my questions on turbo codes and computer architecture. I am very grateful for the assistance provided by Christian Giasson and John Koob with all my LaTeX related questions. Many thanks to Jeff Mrochuk from Atsana Semiconductor Corporation whose expert technical support proved to be the key in getting applications working on the J2210 Customer Evaluation board. A special acknowledgement to my friend Aaron Hughes for reviewing and improving the Visual C++ code for the simulator application and for his constant encouragement during the writing of this thesis. I thank and acknowledge my great friend Dennis Bland for his never ending support and for providing a valuable contribution to my research by letting me use his lab equipment and tools. I would also like to express my sincere gratitude to Steve Knish for helping me learn more about myself during my experience as a graduate student.

I am specially grateful for the opportunity to work under the supervision of Dr. Ivan Fair and Dr. Duncan Elliott. Thank you for taking the big chance of having me as your student and for all the valuable advice and feedback which made the completion of this work possible. I have learned to be a better professional through your example.

My most heartfelt thanks go to my family, this work is as much your accomplishment as it is mine. All my love and appreciation to my wife Rosa and our children Gabriel Alejandro and Marco Angelo for enduring the sacrifice of being away from our home, and for their encouragement, moral support and patience throughout my degree. Words are not enough to thank our parents for all their support; may God bless you always for all your kindness.

Last but most certainly not least, I thank and praise our Lord Jesus Christ and our Blessed Mother Mary for the many blessings that I have experienced during this period of my life.

Table of Contents

1 Introduction **1**

1.1 Overview 1

1.2 Thesis Organization 5

2 Background and Fundamentals **7**

2.1 SIMD Computers 8

2.2 Processor-In-Memory Architectures 11

2.3 Development Platform 13

2.3.1 J2210 Media Processor 14

2.3.2 Software Development Environment 15

2.4 Iterative Decoding of Turbo Codes 16

2.5 Decoding Algorithms for Turbo Codes 18

2.5.1 MAP Algorithm 19

2.5.2 Log-MAP and Max-Log-MAP Algorithms 23

2.6 Data Parallel Turbo Decoder Implementations 26

2.6.1 Software-Based Decoders 27

2.6.2 VLSI Implementations 28

2.7 Fixed-Point Concepts 28

2.7.1 Number Representation 28

2.7.2 Arithmetic Operations Considerations 30

2.8 Summary 32

3 Fixed-Point Decoder Implementation **33**

3.1 The UMTS Turbo Code 34

3.2 Fixed-Point Turbo Decoder Architecture 36

3.3 Simplification of Decoding Algorithm 38

3.3.1 Log-Add Kernel Approximation 38

3.3.2 Normalization of State Metrics 40

3.3.3 Backward Recursion Initialization 41

3.4 Finite Precision Analysis 42

3.4.1 Quantization Method 42

3.4.2 Effect of Fractional Length 43

3.4.3 Density Evolution Analysis 45

3.4.4 Early Saturation of Extrinsics 54

3.5 Fixed-Point Turbo Decoder Simulations 58
3.5.1 BER Performance 58
3.6 Summary 61

4 Data Parallelism in Turbo Decoding 63
4.1 Turbo Code Trellis Analysis 64
4.1.1 Efficient Trellis Structure to SIMD Hardware Mapping . . . 67
4.1.2 Memory Requirements 68
4.2 Parallel Window ONE Algorithm 69
4.2.1 Convergence of α and β State Metrics 71
4.3 Parallelism and Efficiency 72
4.4 Summary 78

5 Parallel Processor Decoder Implementation 79
5.1 Sequential Interleaver 80
5.2 Implementation using the J2210 Array Processor 81
5.2.1 Inter-Processor Communication Topology 81
5.2.2 Description of Algorithm Implementation 85
5.3 Communication with the Host Processor 89
5.4 BER Performance Evaluation 90
5.4.1 Simulator and Experimental Method 90
5.4.2 Scalability of Parallel Algorithm 91
5.4.3 Convergence Steps Analysis 95
5.5 Execution Time and Speedup 100
5.6 Power and Energy Considerations 107
5.7 Summary 111

6 Conclusions 113
6.1 Future Research Directions 115

Bibliography 118

A Turbo Encoder/Decoder C++ Source Code 123
A.1 Fixed-Point Data Type Definition 123
A.2 UMTS Interleaver Generator Method 131
A.3 Turbo Encoder/Decoder Class 134

List of Figures

1.1 Standard Turbo Encoder. 2

2.1 SIMD Architecture Block Diagram. 8
2.2 Grid connected inter-processor communication networks. 10
2.3 J2210 Customer Evaluation Board. 14
2.4 J2210 Media Processor Block Diagram. 15
2.5 Conventional Turbo Decoder Architecture. 17
2.6 The Soft-Input Soft-Output (SISO) module. 19
2.7 Meaning of Notations. 20
2.8 Correction functions for Log-MAP, constant-Log-MAP and linear-Log-MAP algorithms. 26
2.9 Binary fixed point number representation. 29
2.10 Range of representable numbers 30

3.1 UMTS turbo encoder. 35
3.2 Fixed-Point Turbo Decoder Block Diagram. 36
3.3 Trellis tail for UMTS turbo code. 41
3.4 Fixed-point BER performance with $0 \leq \text{FWL} \leq 2$. 44
3.5 Fixed-point BER performance with FWL = 3. 44
3.6 BER performance with unnecessarily long FWL. 45
3.7 Density Evolution of $\Lambda_{o1}(x_k)$, E_b/N_0 = 0.6dB. 46
3.8 Density Evolution of $L^2_{(a+s)}(x_k)$, E_b/N_0 = 0.6dB. 47
3.9 Density Evolution of $\Lambda_{o2}(x_k)$, E_b/N_0 = 0.6dB. 48
3.10 Density Evolution of $L_{e2}(x_k)$, E_b/N_0 = 0.6dB. 49
3.11 Density Evolution of $\Lambda_{o1}(x_k)$, E_b/N_0 = 2.0dB. 50
3.12 Density Evolution of $L^2_{(a+s)}(x_k)$, E_b/N_0 = 2.0dB. 51
3.13 Density Evolution of $\Lambda_{o2}(x_k)$, E_b/N_0 = 2.0dB. 52
3.14 Density Evolution of $L_{e2}(x_k)$, E_b/N_0 = 2.0dB. 53
3.15 Early saturation of extrinsics, 2 iterations. 56
3.16 Early saturation of extrinsics, 4 iterations. 56
3.17 Early saturation of extrinsics, 6 iterations. 57
3.18 Early saturation of extrinsics, 8 iterations. 57
3.19 Fixed-point Turbo Decoder BER performance, 2 iterations. 59
3.20 Fixed-point Turbo Decoder BER performance, 4 iterations. 60

3.21 Fixed-point Turbo Decoder BER performance, 6 iterations. 60
3.22 Fixed-point Turbo Decoder BER performance, 8 iterations. 61

4.1 Trellis Butterfly with State Metrics. 64
4.2 Adjacent Trellis Sections for RSC Code of UMTS Turbo Code. . . . 66
4.3 PE allocation. 67
4.4 Parallel variable distribution per PE. 68
4.5 Parallel Decoding Method with respect to Time. 70
4.6 Effect of Code Parameters on Speedup. 76
4.7 Efficiency for different Code Parameters. 77

5.1 A CU and its Nearest Neighbors. 82
5.2 West-East Data Transfer inter-PE Communication Pattern. 83
5.3 North-South Data Transfer inter-PE Communication Pattern. 84
5.4 Row-wise to Column-wise re-ordering of data. 84
5.5 Emulation of Linear Array network using a 2-D Mesh. 85
5.6 Turbo Encoder and AWGN Channel Simulator GUI. 91
5.7 BER performance vs. Frame Length, 4 Turbo Iterations. 93
5.8 BER performance vs. Frame Length, 7 Turbo Iterations. 94
5.9 BER performance vs. Frame Length, 10 Turbo Iterations. 94
5.10 Bit Error Rate versus Number of Convergence Steps (Short Frame Size). 96
5.11 Effect of Decoder Iterations (Short Frame Size). 97
5.12 BER versus Decoder Iterations versus Convergence Steps (Short Frame Size). 97
5.13 Bit Error Rate versus Number of Convergence Steps (Long Frame Size). 98
5.14 Effect of Decoder Iterations (Long Frame Size). 99
5.15 BER versus Decoder Iterations versus Convergence Steps (Long Frame Size). 99
5.16 Execution Time and Speedup Factor of Single Constituent Decoder. 104
5.17 Execution Time and Speedup Factor of Turbo Decoder. 106
5.18 J2210 Customer Evaluation Board power distribution. 107
5.19 Experimental Setup using E3647A Power Supply. 108

List of Tables

3.1 UMTS Turbo code Parameters 35
3.2 Turbo Decoder Fixed-Point Word Configuration. 59

4.1 Operations per Trellis Column in Sequential Log-MAP Algorithm. . 73
4.2 Operations per PE in PW-ONE Algorithm. 74

5.1 Memory Accesses and Interleaver Run-Times 102
5.2 Execution Time of PW-ONE for Different Topologies 102
5.3 Speedup Factor Comparison for Constituent SISO Decoder 103
5.4 Speedup Factor Comparison for Turbo Decoder 105
5.5 Raw Current Consumption Measurements 110
5.6 Power Estimates for Individual Subsystems 110
5.7 Energy Consumption Comparison 111

Nomenclature

List of Acronyms

AC Array Controller

AHB Advanced High-performance Bus

ALU Arithmetic Logic Unit

AP Array Processor

API Applications Programming Interface

APP *a posteriori* probability

ASIC Application Specific Integrated Circuit

AWGN Additive White Gaussian Noise

BER bit error rate

BPSK binary phase shift keying

CEB Customer Evaluation Board

CIU CMEM Interface Unit

CU Computational Unit

DMA Direct Memory Access

DSP digital signal processor

FEC forward error correction

FPGA Field Programmable Gate Array

LDPC low density parity check

LLR log-likelihood ratio

LSB least significant bit

LUT	look-up table
MSB	most significant bit
PCCC	parallel concatenated convolutional code
PE	processing element
PIM	Processors-In-Memory
RSC	recursive systematic convolutional
SEL	SIMD Engine Language
SIMD	Single Instruction Stream, Multiple Data Streams
SISO	soft-input/soft-output
SNR	signal-to-noise ratio
SoC	System-on-Chip
SOVA	soft-output Viterbi algorithm
UMTS	Universal Mobile Telecommunications System
VLSI	Very Large Scale Integration

Chapter 1

Introduction

The advent of third generation (3G) mobile telephone systems has brought forth an increase in the available bandwidth to support high data rate applications such as multimedia messaging and large file transfer. The characteristic low signal-to-noise ratio (SNR) of wireless channels requires the use of advanced forward error correction (FEC) coding to guarantee the fast and reliable delivery of wireless connections to the Internet and the support of data applications for mobile devices. Turbo codes have been adopted as one of the preferred methods of forward error correction in 3G wireless systems because they can achieve a level of performance that comes closer to the theoretical bounds than more conventional coding techniques [1]. This thesis investigates the implementation of a simple, low-power turbo decoder design suitable for wireless applications.

1.1 Overview

Turbo codes, introduced in 1993 by Berrou *et al* [1], are powerful error correcting codes with performance that approaches the Shannon bound at low SNR levels. The many research efforts aimed at the development of decoding algorithms for turbo codes, as well as advances in digital signal processor (DSP) and Very Large Scale Integration (VLSI) technologies, has allowed them to become practical for use in real-world applications. One key example is their use for medium to high data rate transmission in the Universal Mobile Telecommunications System (UMTS) specification [2], as standardized by the Third-Generation Partnership Project (3GPP)[1].

Turbo codes are parallel concatenated convolutional codes (PCCCs). The block diagram of a standard turbo code encoder is depicted in Figure 1.1. The turbo encoder is composed of two identical recursive systematic convolutional (RSC) encoders separated by a pseudo-random interleaver. Parallel concatenation means that the two encoders operate on the same data stream, but the lower encoder receives the data after it has been permuted by an interleaver. The purpose of the interleaver is to spread out bit information over large blocks of data. The data stream and the outputs of the encoders are concatenated to form the overall encoder output. A puncturing mechanism may be used for applications where a code with rate $1/2$ or higher is preferred. The role of the puncturer is to periodically delete selected bits to reduce coding overhead. In the case of iterative decoding, such as the one employed for turbo codes, it is preferable to delete only parity bits as indicated in Figure 1.1. For example, to achieve a rate of $1/2$, the selection of the parity bit may alternate between the upper encoder and the lower encoder for each transmit cycle.

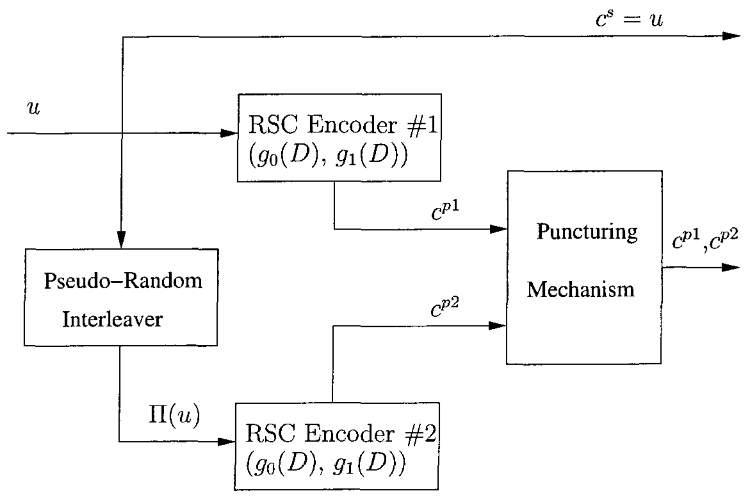


Figure 1.1: Standard Turbo Encoder.

[1]3GPP is the collaboration agreement between a number of telecommunications standards bodies in charge of producing globally applicable technical specifications and technical reports for third generation mobile systems.

A significant issue regarding turbo codes that remains under active investigation is the development of a simpler, low-power implementation of the iterative decoding procedure. Common implementations of turbo decoders are: software based algorithms [3][4], commercially available IP cores for use on a digital Application Specific Integrated Circuit (ASIC) or Field Programmable Gate Array (FPGA) [5][6], and analog VLSI decoders [7]. Dedicated hardware under software control can offer an elegant and effective solution such as in the case of the TMS320C6416T DSP that integrates a programmable, 3GPP compliant Turbo Decoder Co-Processor (TCP) [8]. It is also important to recognize that real-time turbo decoders used in embedded applications are likely to be implemented using fixed-point arithmetic.

This thesis discusses the novel implementation of a turbo decoder for execution on a Single Instruction Stream, Multiple Data Streams (SIMD) array processor with Processors-In-Memory (PIM) technology. The research project is divided into two stages.

The first stage of the research focuses on the development of a fixed-point, sequential implementation of the Log-MAP algorithm. The objective of this stage is to quantify the minimum fixed-point word length required for internal variables of the constituent decoders as well as for the extrinsic information exchanged between these constituent decoders. Results obtained indicate that it is possible to use a maximum word length size of 8 bits, consisting of 2 fractional bits and 5 integral bits, across the entire architecture of the turbo decoder to achieve bit error rate (BER) performance that lies within 0.1 dB of the floating-point implementation performance. From a hardware point of view, the use of fixed-point arithmetic is advantageous because simple integer hardware can be used to carry out the computations (most of them additions) of the Log-MAP algorithm. The reduced fixed-point word size lends itself nicely to a pipelined VLSI design. From a software perspective, minimizing the fixed-point word length to the size stated above may not seem very relevant since in practice one would like to take full advantage of the available native integer word size of the microprocessor or DSP architecture selected to execute the algorithm. However, there also exists the potential of using a fixed-point DSP whose instruction set includes SIMD extensions. In such a case, smaller word sizes allow the manipulation of more data elements by the SIMD instructions. Smaller fixed-point word sizes are also important when one considers a massively parallel processor that may consist of processing elements with a bit-serial architecture or whose Arithmetic Logic Units (ALUs) require operands with a short word length.

The second stage of the project explores the data parallelism available in the Log-MAP algorithm by analyzing the regular trellis structure of systematic convolutional codes. The main goal is the parallel processor implementation of the algorithm for execution on commercially available hardware. Development is targeted to the Array Processor in the J2210 Media processor from Atsana Semiconductor Corporation[2]. This particular array processor consists of an array of simple computational units tightly coupled with a memory core. More details about the hardware are provided in section 2.3.

A method is developed such that the operation of the constituent decoders in the turbo decoder is parallelized and can be executed by a SIMD array processor. The basis for the parallel method is the concept of the *convergence step*, where each processing element in the array processor simultaneously operates on the data that corresponds to an individual trellis stage of the code, to calculate the state metrics needed to compute the log-likelihood ratios. Empirical results demonstrate that BER performance equivalent to that of the sequential forward/backward algorithm is achieved when the number of convergence steps is set to approximately seven times the constraint length of the code. The applicability of this observation to data packets of variable length was verified with the limited number of processing elements in the Array Processor of the J2210 Media processor by using a windowing technique.

This research project has demonstrated that the massively parallel, processors-in-memory architecture of the SIMD array processor provides a platform that offers computational and power consumption advantages over a high-performance ARM922T™ microprocessor core that is a popular choice for embedded, low-

[2]As of August 24, 2005 Mtekvision Co. Ltd. has acquired all assets and technology of Atsana.

power applications. Additionally the use of a commercially available System-on-Chip (SoC) with an embedded array processor to validate the functionality of the data parallel turbo decoding algorithm reveals that SIMD cores tightly coupled with memory used in the design of a cellular handset may offer benefits meeting the bandwidth and battery life demands in 3G systems.

1.2 Thesis Organization

Six chapters, including this introductory chapter, comprise this thesis.

Chapter 2 presents background information related to the iterative operation of turbo decoders and the decoding algorithms used by the constituent soft-input/soft-output (SISO) decoders. This chapter also discusses basic concepts of fixed-point arithmetic and presents background material on SIMD computers and Processors-In-Memory architectures. It also includes an overview of the J2210 Media Processor and the development tools to generate application software for it.

Chapter 3 provides details of a fixed-point implementation of the sequential turbo decoding algorithm. Explanation is provided for the selection of the constant-Log-MAP algorithm over other methods, such as the use of look-up tables or the Linear-Log-MAP algorithm, for the approximation of the correction function of the log-add kernel in the Log-MAP algorithm. This chapter also describes the criteria for selecting a combination with the smallest possible number of fractional and integral bits in the fixed-point word that minimizes the degradation in performance of the overall turbo decoder. Empirical results are used to arrive at the minimum fixed-point word size that results in negligible loss in BER performance when compared to the baseline floating point implementation.

Chapter 4 explores the data parallelism of the Log-MAP algorithm by examining the structure of the trellis that describes the turbo code. The requirements for the number of processing elements and the inter-processor communication network in a generic SIMD array processor are examined. It also discusses the distribution of data and variables in the localized memory for all processing elements.

Chapter 5 presents the details of the implementation that targets the Array Processor in the J2210 Media processor. It presents a comparative analysis of results related to BER performance, processing time and power consumption between the sequential, fixed-point implementation running on the embedded ARM922T^{TM} core and the parallel implementation executed by the Array Processor.

Chapter 6 provides conclusions and states the main contributions made by this research. It also proposes further research such as modifications to the array processor architecture, specifically related to the inter-processor communications network, to further increase performance. It suggests the implementation of a dedicated communication network that can take over the role of the interleaver.

Chapter 2

Background and Fundamentals

This chapter begins by describing fundamental architectural features of SIMD multiprocessors. Section 2.2 elaborates even further by introducing examples of array processors that have been developed to take advantage of the aggregate memory bandwidth found inside memory chips by tightly coupling the processing elements with the memory core.

Section 2.3 provides a brief overview of the hardware and software components of the development platform used in this research project.

Sections 2.4 and 2.5 describe the decoding procedure for turbo codes and present the relevant equations of the decoding algorithms used by the constituent decoders. The complexity of the decoding algorithm is reduced when its operations are performed in the logarithmic domain. The different approximation methods for the correction function of the *max** operator are also discussed. Some sample turbo decoder implementations where a small degree of data parallelism is exploited are presented in section 2.6.

The chapter concludes by discussing basic concepts of fixed-point arithmetic.

2.1 SIMD Computers

It is possible to develop the parallel counterpart of a sequential algorithm given a specific target architecture. Parallel processing is a concept where many computational units share the workload of a processing job. This idea dates back to the early days of electronic computers when it was recognized that parallel processing offers performance gains that go beyond the computational capacity of the individual units [9]. A category of parallel computers is Single-Instruction stream, Multiple-Data streams (SIMD) computers. The term SIMD was introduced by Flynn in his classic taxonomy of computer architectures in 1966 [10]. This classification is based on the manner of instruction and data distribution in a computer architecture. A common architectural block diagram of a SIMD machine that uses the distributed memory model is illustrated in Figure 2.1. It consists of an array of processing elements (PEs), localized memory, an array controller, a host computer, and I/O for external communication.

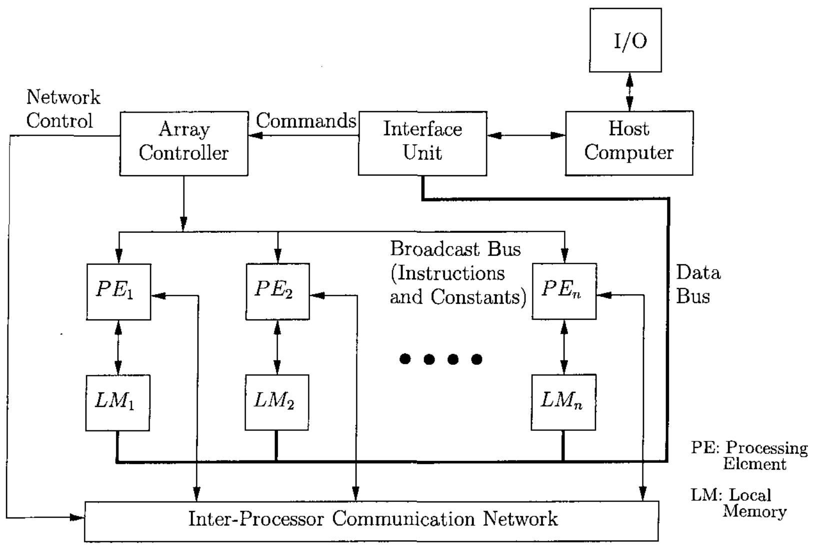


Figure 2.1: SIMD Architecture Block Diagram.

The multiple PEs in a SIMD computer execute the same instruction on their own data set which may or may not be different from that of their neighbors. This type of parallelism signifies that SIMD architectures are best suited for computation

on arrays of data common in applications such as signal processing.
A SIMD computer exploits spatial parallelism rather than temporal parallelism as in a pipelined computer. SIMD arrays can have a large number of processors if their computational units have reduced complexity. The loss in performance per processor is made up by the increase in the number of processors that can be implemented in the same space [11]. A SIMD architecture with simple bit-serial processors can implement fine grain parallelism by assigning a single data item to each processor. The relative simplicity of SIMD architectures allows them to claim some advantages over other parallel architectures, including:

- Simplicity of concept, programming, and synchronization.
- Regularity of structure.
- Scalability of size and performance.
- Straightforward applicability in a number of fields which demand this type of parallelism.

The PEs of an array processor are interconnected by an inter-processor communication network which performs inter-PE data communications such as broadcast, point-to-point communication (shifting) and combine operations. Inter-processor communication in SIMD computers can be autonomous where the destination of the data packet is specified by each sender, for example the hypercube network of Thinking Machines' CM-2, or uniform (also known as patterned communication) where inter-PE data communication is performed in lockstep, synchronized in hardware by the array controller. Uniform inter-processor communication was popular with grid-connected SIMD computers such as the Illiac IV and the Goodyear MPP, and for cost and area reasons [12] it is used in array processors with PIM technology. Synchronized inter-processor communication operations make SIMD computers efficient in exploring spatial parallelism in large arrays of data, and the use of a particular inter-processor communication network will determine which applications will run efficiently.

Popular inter-processor communication networks include:

- Linear Array.
- Mesh connected networks including NEWS net, Torus and Toroid.
- Hypercube.

Examples of linear array and 2-D mesh networks can be observed in Figure 2.2. Grid connected inter-processor communication networks are described with more detail in [13] and communication properties of interconnection networks are covered in [10].

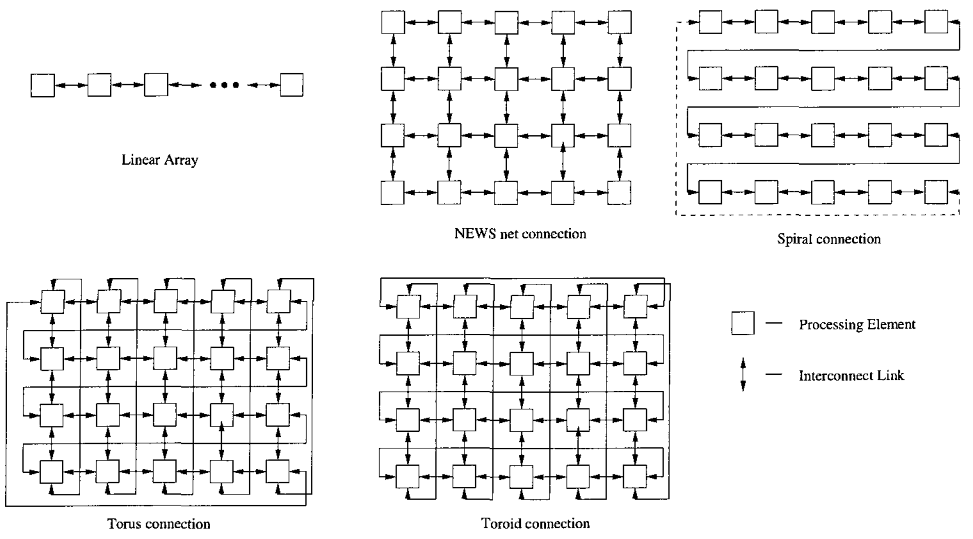


Figure 2.2: Grid connected inter-processor communication networks.

Successfully deployed SIMD machines date back to the Illiac IV introduced in 1968 [9] [10]. Some examples of these conventional SIMD systems include: Goodyear Corporation's MPP (Massively Parallel Processor) in the 1980s, Thinking Machines' CM-1 and CM-2 models, MasPar MP series and Active Memory Technology's (AMT) DAP machines from the early 1990s. The CM-2 and MasPar MP-1 were array processors that targeted a wide range of applications in scientific computing, however, they suffered from limited functionality, low processor utilization in some cases, and unsuitability for some important problems [9][12].

A significant factor that prevented the widespread use of SIMD array processors was their high price. This meant that they served a narrow market of universities and government agencies that could afford them.

The idea of exploiting data parallelism in multimedia applications has resulted in the integration of SIMD instructions to the instruction set of modern high performance microprocessors and DSPs. When the width of the majority of the variables for a given application is smaller than the width of the microprocessor registers, there exists the opportunity for parallel operation on the data. In essence, multiple values can be manipulated by utilizing the same wide data path. For example, consider a 64-bit register. The data can be either a single 64-bit value in the native precision of the processor, two 32-bit values, four 16-bit values, or eight 8-bit values. In this example, the latter three formats are considered SIMD representations. SIMD architectures found in modern high performance microprocessors include:

- Sun Microsystems' Visual Instruction Set (VIS^{TM}) in the UltraSPARC^{TM} processor.
- Intel's Pentium® Matrix Manipulation eXtensions (MMX^{TM}).
- Intel's Streaming SIMD Extensions (SSE) and SSE2.
- Motorola's AltiVec^{TM} as SIMD extension to the PowerPC architecture.
- AMD's Direct3D and 3D-Now technology.

Features and characteristics for most of the above SIMD extensions are summarized in detail in [14].

2.2 Processor-In-Memory Architectures

In recent years SIMD architectures have emerged that are implemented around memory cores. Such technology is known as logic-enhanced memories or Processor-In-Memory (PIM) systems. These SIMD architectures are massively parallel and are designed to take advantage of the high internal memory bandwidth by placing

SIMD processors inside memory devices. It has been shown that there are three to four orders of magnitude more bandwidth available within the memory chip than what is available externally [12]. The combination of high data bandwidth and the fine grain parallelism possible with a SIMD architecture permits data-parallel operations to be performed in memory. The regular structure of SIMD architectures facilitates the addition of array processors in memory [15], thereby providing an economic solution for parallel processing. These systems offer great benefit and increased computational performance for applications that manipulate data sets with a high degree of parallelism. Examples of these applications include image, video and signal processing.

In [16] a review and survey of SIMD processor arrays for image and video processing is presented. Emphasis is given to those designs whose logic circuits are embedded in the SRAM or DRAM memory process. Dillen in [14] also briefly describes four PIM array processors and elaborates on the DSP-RAM architecture. Some of the most relevant PIM-style architectures are:

- Terasys.
- Computational RAM (C•RAM).
- Integrated Memory Array Processor (IMAP)
- EXECUBE
- DSP-RAM

Only a few details for the some of array processors mentioned above will be presented here.

Terasys, a massively parallel array processor, was developed by researchers at the Supercomputing Research Center (SRC) in the early 1990s [17]. The SIMD array consisted of 32K bit-serial processing elements, where each had access to a 2-Kbit column of SRAM as their local memory. The communication network for the processing array consisted of a global OR network, a partitioned OR network, and a parallel prefix network.

C•RAM, developed at the University of Toronto [12], integrates bit-serial processing elements at the memory sense amplifiers of SRAM or DRAM cores. The processing element ALU is a 8-1 multiplexor, where the 3 inputs to the ALU are a X register, a Y register, and the data memory. This ALU design is capable of implementing 256 functions or operations. Inter-processor communication in C•RAM is accomplished via a linear network. C•RAM, like many other SIMD systems, requires a host processor to handle the non-data-parallel operations. A bit-parallel version of C•RAM is described in [18]. A commercial variant, the AX256 graphics accelerator from Accelerix Inc., is a C•RAM architecture with 4096 PEs. Work was done in [11] to integrate this commercial product into an embedded system.

DSP-RAM is a moderately parallel processor that consists of processing elements built around a 16-bit multiply-accumulate (MAC) unit. The more sophisticated PE architecture allows a smaller degree of parallelism, typically ranging from 64 to 256 PEs. DSP-RAM targets algorithms that exhibit moderate levels of parallelism and results in reduced latency of the multiplication operation compared to bit-serial processors. A quantitative analysis of DSP-RAM has been carried out in [14] and includes suggestions for improvement of the architecture.

2.3 Development Platform

The development platform selected for the implementation of this research project is the J2210 Evaluation Kit that was provided by Atsana Semiconductor Corporation. The J2210 Evaluation Kit consists of a hardware platform and a software development environment. The main hardware component is the Customer Evaluation Board (CEB). The CEB contains the necessary on-board hardware to access and control the J2210 Media Processor. Figure 2.3 shows a picture of the evaluation board. The software environment consists of a set of tools needed to develop firmware for the SIMD array processor of the J2210. These software tools allow the user to program the array processor from two perspectives: a) for simulation only and b) for generation of embedded firmware that is downloaded to the hardware target. These tools are described in detail in [19]. The opportunity to evaluate and

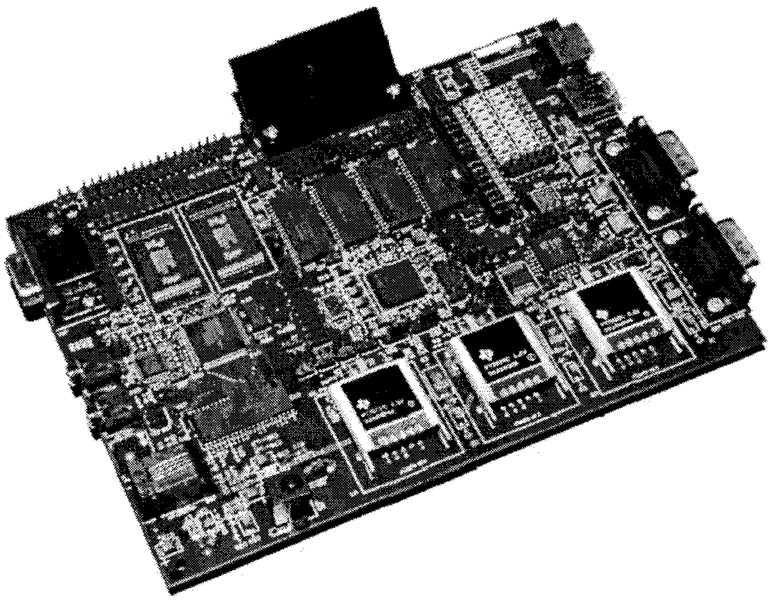


Figure 2.3: J2210 Customer Evaluation Board.

verify the functionality of SIMD algorithms on commercially available hardware is an important step in bringing the parallel processor implementation of a turbo decoder into a real-world application.

2.3.1 J2210 Media Processor

The J2210 Media Processor from Atsana Semiconductor Corporation is a System-on-Chip (SoC) that combines a general purpose ARM922T$^{\text{TM}}$ RISC microprocessor, a fully programmable SIMD array processor for low power multimedia processing, and a set of peripherals to support multimedia applications. A block diagram of the J2210 is portrayed in Figure 2.4. The architecture of the Array Processor (AP) delivers more processing capability that conventional DSP solutions while consuming less power. The SIMD array processor in the J2210 is tightly coupled with a memory core, in other words, it is a Processor-In-Memory architecture. One may consider the architecture of the Array Processor to be similar to the bit-parallel computational RAM (C•RAM) architecture presented in [18]. The Array Processor consists of 96 Computational Units (CUs), each with 4KBytes of local me-

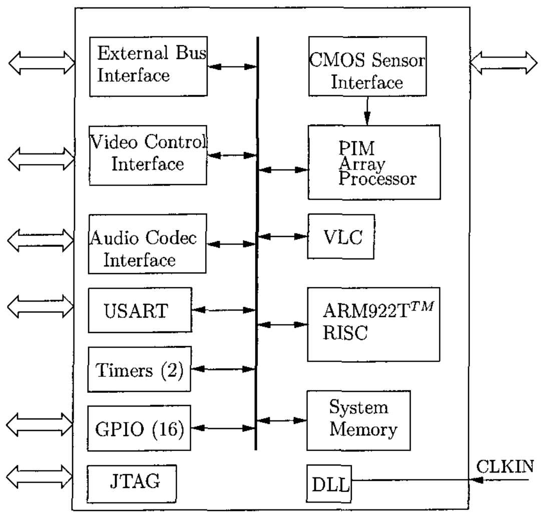


Figure 2.4: J2210 Media Processor Block Diagram.

mory. The CUs are arranged in a 4 x 24 grid with an inter-processor communication network that enables each CU to obtain data from its eight nearest neighbors. The SIMD processor uses uniform memory addressing and uniform inter-processor communication. There is an Array Controller that is in charge of issuing and sequencing micro-instructions to the Array Processor. The ARM922T™ processor behaves as the system host; it is in charge of sending commands to the Array Controller, and it controls the movement of data in and out of the Array Processor since it too has access to the memory of the SIMD array.

2.3.2 Software Development Environment

The generation of software/firmware for the J2210 Media Processor involves three levels of programming, however, only two out of the three are commonly used in practice. At the lowest level is Micro-code Programming. Regular users are not exposed to this level of programming because it requires detailed knowledge of the architecture of the CUs and the Array Processor in general. The development in this research project relies heavily on the provided micro-code and no new micro-instructions are introduced.

At the middle level is the Array Processor Programming. This level of programming requires the developer to become familiar with the SIMD Engine Language (SEL) and with the Array Processor Software Development Kit (SDK). The role of AP Programming is to implement the algorithms that have a parallel nature so that they execute faster and use less power. These algorithms are written in SEL and are known as AC Commands. SEL is a C-like proprietary language developed by Atsana; it is described in detail in [19]. Developers at this stage in the process make extensive use of the SEL compiler and the AP simulator and visual debugger provided with the Array Processor SDK.

At the highest programming level is the development of software for the ARM host processor. Software at this stage is developed with C/C++ by using the tools available through the ARM Developer Suite.

2.4 Iterative Decoding of Turbo Codes

In section 1.1 turbo codes were described as parallel concatenated convolutional codes with encoders formed usually by two constituent systematic encoders joined through an interleaver. Assuming no puncturing, coded symbols of the UMTS turbo code consist of one systematic bit followed by two parity bits. A typical turbo decoder has a constituent decoder corresponding to each of the constituent encoders. Since the decoder knows the interleaving pattern used by the encoder, it is capable of generating the proper systematic bit order for each of the constituent decoders. In order to obtain the best possible estimate of the original message, the constituent decoders share the results of their calculations by using *iterative feedback decoding*. The component decoders output soft-bit information typically represented as (*a posteriori*) *log-likelihood ratios* (LLRs) of the form

$$\Lambda_k = \ln \frac{P[u_k = 1|\mathbf{y}]}{P[u_k = 0|\mathbf{y}]} \tag{2.1}$$

where u_k is the source data bit and $\mathbf{y}$ is the data sequence observed by the receiver.

The log-likelihood ratios (LLRs) produced by a *soft-input/soft-output* (SISO) decoder consist of three components of information about the data bit u_k: 1) the systematic channel observation denoted as y_k; 2) the information derived from the other constituent decoder which is used as *a priori* information and is denoted L_a, and 3) the *extrinsic* information which is new information generated by the current stage of decoding, denoted as L_e. The overall LLR can be then be written:

$$\Lambda_k = L_c y_k + L_a + L_e \tag{2.2}$$

In order to prevent correlations, it is imperative that only the extrinsic information is passed from one constituent decoder to the other. Therefore, the *a priori* data input to the subsequent decoder is computed by subtracting the systematic channel input and the *a priori* data input for the current decoder.

The architecture for a conventional turbo decoder is displayed in Figure 2.5. Note that all the values received from the channel are multiplied by the factor L_c. The term L_c is known as the channel reliability because it provides an indication of how much weight the channel symbols have on the LLR[20]. For example, a channel with a higher SNR has a higher channel reliability, therefore, more weight is put on the channel observations. The expression for L_c when binary phase shift keying modulation is used in an Additive White Gaussian Noise (AWGN) channel is defined in [20], while a more generalized version for fading channels is given in [21].

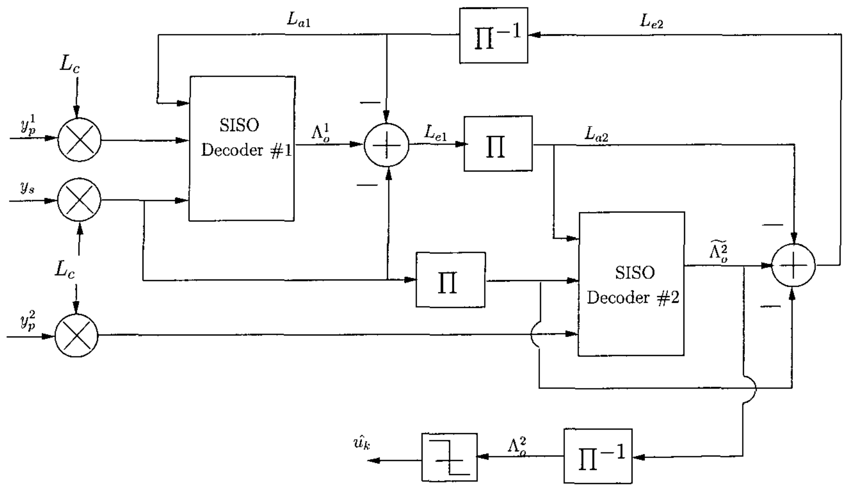


Figure 2.5: Conventional Turbo Decoder Architecture.

In the following, boldface type is used to represent vectors that contain values associated with an entire block of data. The first constituent decoder receives the systematic channel symbols $\mathbf{y}_s$, the channel symbols associated with the parity from the first encoder $\mathbf{y}_p^1$, and *a priori* information $\mathbf{L}_{a1}$ derived from the output of the second constituent decoder. The first decoder generates the LLR Λ_o^1. The extrinsic information of the first decoder $\mathbf{L}_{e1}$ is found by subtracting the scaled systematic and *a priori* inputs from the output of this decoder. The extrinsic information is permuted by an interleaver, and used as *a priori* information for the second decoder. The second decoder also receives the scaled and permuted systematic channel observations $\Pi(\mathbf{y}_s)$ and scaled observations of the parity bits from the second encoder $\mathbf{y}_p^2$. The second constituent decoder produces the LLR $\widetilde{\Lambda_o^2}$ from which the extrinsic information $\mathbf{L}_{e2}$ is derived. The extrinsic information generated by the second decoder is deinterleaved and becomes the *a priori* input to the first decoder. After a pre-defined number of iterations, the final estimate of the original data bit is found by deinterleaving and hard-limiting the output of the second constituent decoder according to the following expression:

$$\hat{u}_k = \begin{cases} 1, & \Lambda_o^2 \geq 0 \\ 0, & \Lambda_o^2 < 0 \end{cases} \tag{2.3}$$

2.5 Decoding Algorithms for Turbo Codes

A decoding algorithm that accepts *a priori* soft-information at its input and generates *a posteriori* information is referred to as a SISO decoding algorithm. Some of the SISO algorithms most commonly used in turbo decoding are described in this section. The concept of the SISO module and its variations is introduced in [22]. As an example, Figure 2.6 shows the SISO module for a rate 1/2 RSC code. It accepts three inputs: the systematic observations λ_s, the parity observations λ_p, and the *a priori* information λ_a which, in the case of a turbo decoder, is derived from the output of the other constituent decoder. The output of the SISO module is the LLR Λ_o as defined by (2.1).

The SISO algorithms presented in this section belong to the maximum *a posteriori* (MAP) family of decoding algorithms most commonly used in turbo decoding. It is important to note that there also exists a decoding algorithm based on a modification to the Viterbi algorithm that produces soft-decision outputs. This algorithm is known as the soft-output Viterbi algorithm (SOVA) and its details are described in [23].

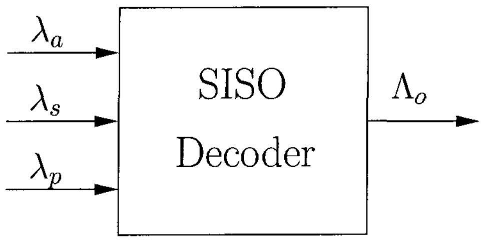


Figure 2.6: The Soft-Input Soft-Output (SISO) module.

2.5.1 MAP Algorithm

In order to make the presentation of the MAP algorithm understandable, it is necessary to introduce clear notation. For a constituent RSC encoder with constraint length K_c (and therefore memory $m = K_c - 1$), the following notation is used with reference to Figure 2.7. Assume that the time instant of interest is the kth interval:

1. The block of N source information bits that is encoded by the encoder is denoted by $\mathbf{u} = \{u_0, u_1, \cdots, u_{N-1}\}$
2. The noisy estimates of the encoded bits received by the decoder are denoted by $\mathbf{y} = \{(y_0^s, y_0^p), (y_1^s, y_1^p), \cdots, (y_{N-1}^s, y_{N-1}^p)\}$
3. S_k is the generic state at time k, belonging to the set $S = \{S^0, \cdots, S^{2^m-1}\}$
4. S_{k-1} is one of the precursor states of S_k, more precisely the one defined by the information symbol u_{k-1} in the transition $S_{k-1} \rightarrow S_k$
5. S_{k+1} is one of the successors states of S_k, more precisely the one defined by the information symbol u_k in the transition $S_k \rightarrow S_{k+1}$

6. A parity coded symbol c_k^p is associated with each transition in the trellis[1]. This coded symbol depends on the state from which the transition originates and on the information symbol u_k determining that transition.

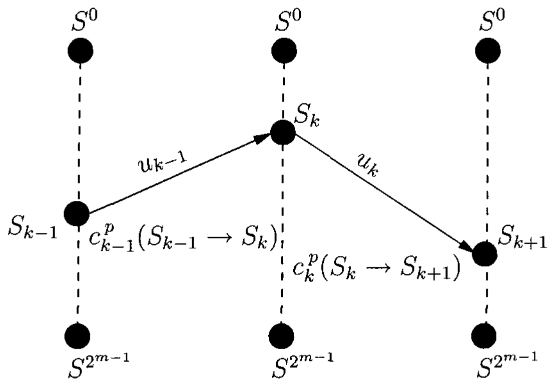


Figure 2.7: Meaning of Notations.

The symbol-by-symbol MAP algorithm, originally described in the late 1960's [24] and re-introduced in 1974 [25], was generally overlooked for use in decoding convolutional codes in favor of the less complex Viterbi algorithm. The MAP algorithm calculates the *a posteriori* probability (APP) that the original information bit was a 0 or a 1 given the channel observation $\mathbf{y}$. Since the outputs produced by the MAP algorithm can be considered as soft decisions, it is a solution for use in turbo decoding. The MAP algorithm consists of a forward and a backward recursion and is applied on a block of N received symbols corresponding to a trellis with a finite number of stages N. Once the APP values have been obtained for the desired quantity, a hard decision is made by selecting the quantity with the highest probability. However, in turbo decoding, hard decisions are not made until the iterations are complete so that soft information can be exchanged between constituent decoders. It can be observed from equation (2.1) that the APPs calculated by the MAP algorithm of the constituent decoders are those of the message bits, $P[u_k = 1|\mathbf{y}]$ and

[1]Since the encoder is systematic, the systematic coded symbol c_k^s is the same as the information symbol u_k

$P[u_k = 0|\mathbf{y}]$. The APPs can be expressed as

$$P[u_k = i|\mathbf{y}] = \sum_{S_k} \lambda_k^i(S_k \rightarrow S_{k+1}) \qquad i \in \{0,1\} \tag{2.4}$$

where the term $\lambda_k(S_k \rightarrow S_{k+1})$ is the probability of each valid state transition given the noisy channel observation $\mathbf{y}$. Using the definition of conditional probability this term can be expressed in the following manner:

$$\lambda_k(S_k \rightarrow S_{k+1}) = P[S_k \rightarrow S_{k+1}|\mathbf{y}] = \frac{P[S_k \rightarrow S_{k+1}, \mathbf{y}]}{P[\mathbf{y}]}. \tag{2.5}$$

As will be shown later, the denominator of the expression in equation (2.5) is a constant and cancels out, therefore, the numerator is more commonly used instead of $\lambda_k(S_k \rightarrow S_{k+1})$. Applying properties of the Markov process, the numerator can be partitioned into the following components:

$$P[S_k \rightarrow S_{k+1}, \mathbf{y}] = A_k(S_k)\Gamma_i(S_k \rightarrow S_{k+1})B_{k+1}(S_{k+1}), \tag{2.6}$$

where

$$A_k(S_k) = P[S_k, (y_0, \cdots, y_{k-1})] \tag{2.7}$$
$$\Gamma_i(S_k \rightarrow S_{k+1}) = P[S_{k+1}, y_k|S_k] \tag{2.8}$$
$$B_k(S_k) = P[(y_{k+1}, \cdots, y_{N-1})|S_{k+1}]. \tag{2.9}$$

The branch metric associated with the transition $S_k \rightarrow S_{k+1}$ is $\Gamma(S_k \rightarrow S_{k+1})$, and it is determined by properties of the channel and the encoder. It is expressed as

$$\Gamma(S_k \rightarrow S_{k+1}) = P[u_k]P[y_k|x_k], \tag{2.10}$$

where u_k and x_k are the source bit and the channel input bit associated with the state transition $S_k \rightarrow S_{k+1}$. Note that if the state at the k^{th} interval is not connected to the state at the next time interval, then the above probability is zero. In equation (2.10), the first term on the right hand side is *a priori* information, and the second term is a function of the modulation and channel model.

The probability $A(S_k)$ is calculated according to the forward recursion

$$A(S_k) = \sum_{S_{k-1}} A(S_{k-1})\Gamma(S_{k-1} \rightarrow S_k). \tag{2.11}$$

Likewise, $B(S_k)$ is computed according to the backward recursion

$$B(S_k) = \sum_{S_{k+1}} B(S_{k+1})\Gamma(S_k \to S_{k+1}). \tag{2.12}$$

Once the APP of each state transition $P[S_k \to S_{k+1}|\mathbf{y}]$ is found, the message bit probabilities can be computed according to

$$P[u_k = 1|\mathbf{y}] = \sum_{S_1} P[S_k \to S_{k+1}|\mathbf{y}], \tag{2.13}$$

and

$$P[u_k = 0|\mathbf{y}] = \sum_{S_0} P[S_k \to S_{k+1}|\mathbf{y}] \tag{2.14}$$

where $S_1 = \{S_k \to S_{k+1} : u_k = 1\}$ is the set of all state transitions associated with a source bit of 1, and $S_0 = \{S_k \to S_{k+1} : u_k = 0\}$ is the set of all state transitions associated with a source bit of 0. The log-likelihood ratio then becomes

$$\Lambda_k = \ln \frac{\sum_{S_1} A(S_k)\Gamma(S_k \to S_{k+1})B(S_{k+1})}{\sum_{S_0} A(S_k)\Gamma(S_k \to S_{k+1})B(S_{k+1})}. \tag{2.15}$$

Note that the term $P[\mathbf{y}]$ in equation (2.5) cancels out in the calculation of the ratio, and therefore it can be ignored in all calculations.

The MAP decoding procedure can be broken down into the following steps:

a) Initialize A according to

$$A(S_0) = \begin{cases} 1, \text{ for } S_0 = 0 \\ 0, \text{ for } S_0 \neq 0 \end{cases} \text{ and } A(S_k) = 0 \; \forall \; k \neq 0 \tag{2.16}$$

b) Initialize time index variable $k = 1$

c) Compute Γ_i and A_k for all states S_k according to equations (2.10) and (2.11)

d) Increment k

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

f) Initialize B according to

$$B(S_N) = \begin{cases} 1, \text{ for } S_N = 0 \\ 0, \text{ for } S_N \neq 0 \end{cases} \text{ and } B(S_k) = 0 \; \forall \; k \neq N \tag{2.17}$$

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

h) Compute Γ_i and B_k for all states S_k according to equation (2.12)

i) Decrement k

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

k) Compute the log-likelihood ratio according to

$$\Lambda_k(u_k) = \ln \frac{\sum_{S_{k+1}} \sum_{S_k} \Gamma_1(S_k \to S_{k+1}) \cdot A(S_k) \cdot B(S_{k+1})}{\sum_{S_{k+1}} \sum_{S_k} \Gamma_0(S_k \to S_{k+1}) \cdot A(S_k) \cdot B(S_{k+1})} \tag{2.18}$$

2.5.2 Log-MAP and Max-Log-MAP Algorithms

The MAP algorithm is considered to be the optimum SISO decoding algorithm for minimizing the bit error rate when decoding a constituent code within a turbo code. However, the numerical representation of probabilities and the large number of multiplications make straightforward use of this algorithm unsuitable for practical implementation in both hardware and software for embedded systems. The computational cost of the MAP algorithm can be alleviated by performing the entire algorithm in the logarithmic domain, rather than waiting until the last step to take the logarithm of the likelihood ratio. The main benefit of operating in the logarithmic domain is that multiplication becomes addition. There exists one drawback when working in the log domain: the arithmetic operation of addition becomes more complex as can be observed in the *Jacobian logarithm*.

$$\begin{aligned} \ln(e^a + e^b) &= max(a,b) + \ln(1 + \exp(-|b-a|)) \\ &= max(a,b) + f_c(|b-a|) \end{aligned} \tag{2.19}$$

The above expression suggests that addition, when performed in the log domain, becomes a maximization operation followed by a correction function $f_c(\cdot)$. However, it is important to note that when a and b are not close in value, the result of the correction function is close to zero. Therefore, a usable approximation to the above expression is

$$\ln(e^a + e^b) \approx max(a,b). \tag{2.20}$$

This approximation is the basis for the Max-Log-MAP algorithm, one of the two versions of the MAP algorithm that operate solely in the log domain [26], with the other one being the Log-MAP algorithm. The two algorithms differ in how they compute addition in the log-domain. The Max-Log-MAP approximates addition as a maximization operation as per equation (2.20). It is this simplification that makes the Max-Log-MAP algorithm sub-optimal as is the case for SOVA. The Log-MAP algorithm computes addition as a maximization followed by a correction as per equation (2.19). This new operation is commonly referred to as $max^{\star}(\cdot)$.

$$max^{\star}(a,b) \triangleq max(a,b) + f_c(|b-a|) \tag{2.21}$$

The operation of the algorithm in the log domain results in new definitions for the branch and state metrics. Letting $\gamma(S_k \rightarrow S_{k+1})$ denote the natural logarithm of $\Gamma(S_k \rightarrow S_{k+1})$, one obtains

$$\gamma(S_k \rightarrow S_{k+1}) = \ln P[u_k] + \ln P[y_k|x_k]. \tag{2.22}$$

The calculation of the branch metric is greatly simplified when one considers the particular case of binary transmission, antipodal modulation with signal amplitudes such that $x_k \in \{A, -A\}$, and the AWGN channel model. In such a case, the systematic and parity inputs to the SISO decoder are likely to be in the form of log-likelihood ratios and not probabilities. In [22] it is shown that the input LLRs take on the following form

$$\Lambda_k^i = \frac{2A}{\sigma^2} y_k^i, \tag{2.23}$$

where i refers to the y_k symbol being either the systematic or the parity symbol. It is shown in [20] that the branch metric calculation reduces to

$$\begin{aligned} \gamma(S_k \rightarrow S_{k+1}) &= \ln P[y_k|x_k] + \ln \frac{P[u_k = 1]}{P[u_k = 0]} \\ &= \frac{2A}{\sigma^2} \cdot \left[y_k^s \cdot u_k + y_k^p \cdot c_k^p\right] + L_a, \end{aligned} \tag{2.24}$$

where

σ^2 is the noise variance.

L_a is the *a priori* LLR.

c_k^p is the coded parity symbol that corresponds to the information symbol u_k during the state transition $S_k \to S_{k+1}$.

The definition of the forward and reverse state metrics in the log domain can be stated in the following manner:

$$\ln A(S_k) = \alpha(S_k) = \max_{S_{k-1} \in \mathcal{A}}{}^{*} [\alpha(S_{k-1}) + \gamma(s_{k-1} \to s_k)] \tag{2.25}$$

and

$$\ln B(S_k) = \beta(S_k) = \max_{s_{k+1} \in \mathcal{B}}{}^{*} [\beta(S_{k+1}) + \gamma(S_k \to S_{k+1})] \tag{2.26}$$

where $\mathcal{A}$ is the set of states S_{k-1} with connections to state S_k, and $\mathcal{B}$ is the set of states S_{k+1} connected to state S_k. Note that when the Max-Log-MAP algorithm is utilized, $max^{\star}(a,b)$ becomes simply $max(a,b)$.

Once $\alpha(S_k)$ and $\beta(S_k)$ have been found for all states in the trellis, the LLR is calculated using the following equation:

$$\begin{aligned} \Lambda_k \;=\; & \max_{S_1}{}^{*} [\alpha(S_k) + \gamma(S_k \to S_{k+1}) + \beta(S_{k+1})] \\ & - \max_{S_0}{}^{*} [\alpha(S_k) + \gamma(S_k \to S_{k+1}) + \beta(S_{k+1})] . \end{aligned} \tag{2.27}$$

The operation of the Log-MAP and Max-Log-MAP algorithm proceeds with the forward and backward recursions as described for the MAP algorithm, except that these new initializations are used:

$$\alpha(S_0) = \begin{cases} 0, \text{ if } S_0 = 0 \\ -\infty, \text{ otherwise} \end{cases} \tag{2.28}$$

$$\beta(S_N) = \begin{cases} 0, \text{ if } S_N = 0 \\ -\infty, \text{ otherwise} \end{cases} \tag{2.29}$$

Considerable research effort has been dedicated towards finding low-complexity approximations to the correction function of the max^{*} operation. The most widely used are: a) look-up tables (LUTs) as described in [27], with [26] reporting excellent results with 8 stored values and $|b-a|$ ranging between 0 and 5; b) the threshold and constant approximation presented in [28] and which is commonly known as *constant-Log-MAP*; and c) the linear approximation of Cheng and Ottoson [29] which will be referred to as *linear-Log-MAP* algorithm. These approximations are ideal for fixed-point software and VLSI implementations of the Log-MAP algorithm. The plot of Figure 2.8 compares the exact correction function against the constant and linear approximations.

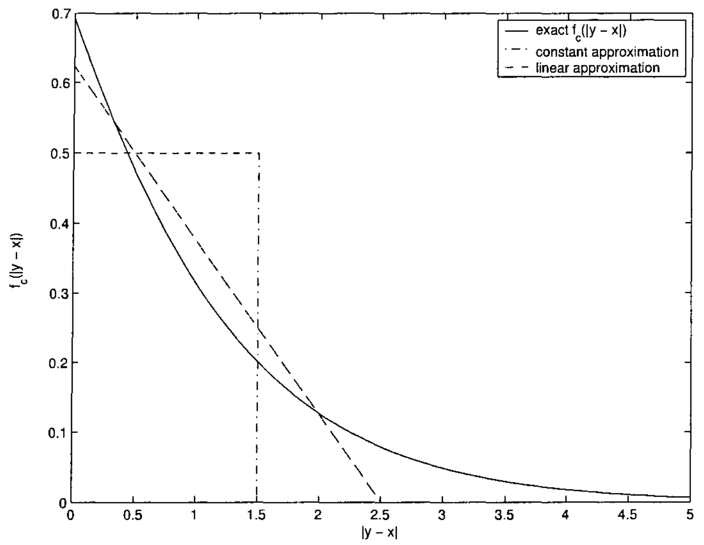


Figure 2.8: Correction functions for Log-MAP, constant-Log-MAP and linear-Log-MAP algorithms.

With respect to the relative performance of the approximations, the work in [30] shows that the performance of the constant-Log-MAP can be approximately 0.025 dB worse than the exact Log-MAP when evaluating an AWGN channel model over a range of SNR levels and with various frame sizes. In contrast, the linear-Log-MAP shows performance that is almost indistinguishable from the Log-MAP algorithm, with a performance within 0.01 dB from that of the exact computation.

2.6 Data Parallel Turbo Decoder Implementations

This section discusses previous work where it was found that performance improvements are obtained by taking advantage of the parallelism available in the MAP based trellis decoding algorithms.

2.6.1 Software-Based Decoders

It was previously stated in section 2.1 that newer, high-performance microprocessor and DSPs include enhancements to their instruction set by incorporating SIMD instructions. The implementation of a 3GPP turbo decoder described in [3] takes advantage of the SIMD features of the SP-5 SuperSIMD^{TM} DSP core from 3DSP Corporation to parallelize the computations within a frame to decode a single frame four times faster than the 32-bit fixed-point superscalar implementation for both the Max-Log-MAP and Log-MAP algorithms. The SIMD features of the SP-5 allow the 32-bit operational units of the DSP to perform a maximum of four 8-bit computations simultaneously. Their results show that with the 8-bit SIMD approach, the data rate obtained with 8 decoder iterations is 488 kbps for the Max-Log-MAP algorithm and 284 kbps for the Log-MAP algorithms when the SP-5 is operating at 250 MHz. Results were obtained through simulation on a cycle accurate simulator.

Another example of SIMD style operation of a turbo decoder implementation is reported in [4]. This particular implementation uses the SIMD instructions of a Texas Instruments TMS320C6416 DSP to perform decoding of four independent trellises using the Max-Log-MAP algorithm. Just as in the case of the SP-5 implementation, 8-bit fixed-point SIMD operations were performed in the decoding algorithm. However, their particular decoder relies on an adapted turbo code suitable for efficient parallel implementation. This means that the code is not necessarily a conventional convolutional turbo code, and it also requires a new interleaver design. Nevertheless, the data rate obtained with this implementation is 160 kbps with 8 decoder iterations and the DSP operating with a clock rate of 600 MHz.

A third architectural study is described in [31] where the performance of a Very Long Instruction Word (VLIW) DSP (ST120 from ST Microelectronics) is compared against that of an application-customized RISC core. It is stated that the ST120 DSP supports SIMD style operation by packing two 16-bit data elements into a 32-bit register. The data throughput in this case was measured to be 540 kbps for the Max-Log-MAP algorithm and approximately 200 kbps for the Log-MAP algorithm with 5 decoder iterations and the clock rate of the ST120 set to 200 MHz.

2.6.2 VLSI Implementations

Hardware implementations of turbo decoders rely mostly on the duplication of the functional units. This is the case for the work described in [32], where the Add-Compare-Select unit is replicated eight times to match the eight states of the trellis sections. In this manner, the forward and backward state metrics are computed simultaneously for a trellis stage. Another characteristic of hardware implementations is that, whether on-chip or off-chip, they require simple control from a DSP or microprocessor to handle data transport of input and output through an I/O interface. The main advantage of dedicated hardware is that much higher data throughput can be achieved with lower clock rates. For example, the authors of an implementation reported in [32] claim that it achieves a data rate of 2.1 Mbps at 42 MHz when prototyped on a FPGA, and that it could operate at 92 MHz when synthesized in a 0.25μm standard CMOS process.

2.7 Fixed-Point Concepts

Fixed-point data types are appealing for the development of digital signal processing algorithms as they can be used to perform fractional arithmetic with integer values. When considering hardware implementation, the use of fixed-point data types not only minimizes the amount of hardware needed to implement the functionality but also results in cost and power savings when compared to their floating-point counterparts.

2.7.1 Number Representation

Fixed-point numbers are characterized by their word length in bits, radix point, and whether they are signed or unsigned. They are a way of representing the integral and fractional bits of a real-valued quantity in an integer value. Figure 2.9 shows a common representation of a binary fixed-point number (either signed or unsigned) where

- b_i are the binary digits (bits).

- wl is the word length in bits.
- The most significant bit (MSB), represented by location b_{wl-1} takes the role of the sign bit for signed values.[2]
- The bits to the right of the radix point are the fractional bits.
- The bits to the left of the radix point form the *integer word length* (IWL).

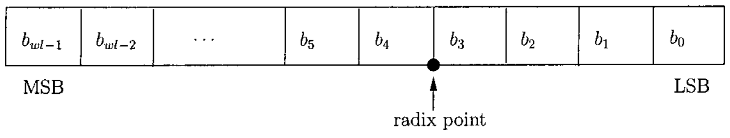


Figure 2.9: Binary fixed point number representation.

The radix point is used in the scaling of fixed-point numbers. For the special cases of integer only or fractional only number, the radix point is preset. In the case of signed or unsigned integer data, the radix point is assumed to be just to the right of the least significant bit (LSB). For unsigned fractional numbers, the radix point is to the left of the MSB, whereas for signed fractional data, it is just to the right of the MSB. In the more generalized case the location of the radix point is usually determined by the developer(s).

It is important to keep in mind that in performing arithmetic operations such as addition or subtraction, the adder or ALU hardware uses the same logic circuits regardless of the values of the scale factor. In other words, the hardware does not know about scaling. The arithmetic-logic-units perform signed or unsigned binary arithmetic as if the radix point was to the right of b_0.

Fixed-point numbers are encoded by applying scaling and quantization. The most common method of scaling is *Radix Point-Only Scaling*. With this method of scaling, the fixed-point numbers are encoded as follows:

$$R \approx \tilde{R} = 2^{-fraction\ length} \times Q \tag{2.30}$$

[2]Signed fixed-point numbers are usually represented using two's complement notation.

where

R is an arbitrarily precise real-world value.

$\tilde{R}$ is the approximate real-world value.

Q is an integer that encodes R.[3]

The *fraction length* (FL) is equal to the number of fractional bits.

The real-world value R is represented by the weighted sum of the bits in the encoded integer Q. The following expression demonstrates this concept for the case of a signed fixed-point quantity:

$$\tilde{R} = 2^{-fraction\ length} \cdot \left[-b_{wl-1} 2^{wl-1} + \sum_{i=0}^{wl-2} b_i 2^i \right]. \tag{2.31}$$

Two other important concepts to consider when it comes to the representation of fixed-point numbers are *Range* and *Precision*. The range is the span of numbers that can be represented for a given fixed-point word length. The range of representable numbers for a two's complement fixed-point number of word length wl is illustrated in Figure 2.10. The precision of a fixed-point number is the difference between

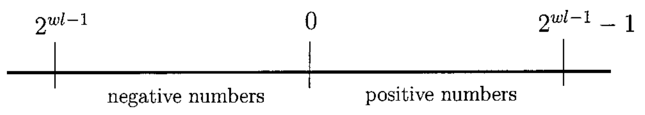


Figure 2.10: Range of representable numbers .

successive values that are representable, which is equal to the value of its least significant bit. The value of the least significant bit, and therefore the precision of the number, is determined by the number of fractional bits.

2.7.2 Arithmetic Operations Considerations

The development of algorithms where fixed-point arithmetic is used must include consideration of the following important factors.

[3]The quantization value Q is the binary number stored in memory and used by the hardware.

Overflow

The addition of two sufficiently large negative or positive numbers can produce a result with more bits on the most significant side than are available for representation in the IWL.

Rounding

Also referred to as *Quantization.* The result of any operation on a fixed-point number is typically stored in a register that is longer than the number's original format. Rounding or quantization is used to determine what happens to the least significant bits if more bits of precision are required than are available when the result is put back into the original format.

Several methods to apply rounding/quantization and to handle overflow conditions are described in detail in [33].

The operations of addition and subtraction for fixed-point numbers are the same as for integers. Before the operation, care must be taken so that the fixed-point numbers have the same number of fractional bits; in essence, the radix point of the operands must be aligned. Overflow and underflow that may result from addition or subtraction of sufficiently large (negative or positive) numbers can be handled by saturation.

The multiplication of two signed fixed-point numbers will result in a number with the following format:

$$\mathrm{A}(IWL_1, FL_1) \times \mathrm{B}(IWL_2, FL_2) = \mathrm{C}(IWL_1 + IWL_2 + 1, FL_1 + FL_2). \qquad (2.32)$$

As can be observed in (2.32), the number of integer bits in the result is the sum of the integer bits in the two operands with one additional bit representing an extra sign bit. Also the number of fractional bits in the result is the sum of the fractional bits in the two operands. Depending upon the requirements for accuracy in the output, the number of fractional bits can be truncated after rounding. Different rounding techniques are available to ensure accuracy in the results after the truncation of fractional bits. A commonly used rounding method is *rounding to positive infinity*, where the most significant bit of the bits that are removed is added to the remaining bits [33].

2.8 Summary

The vector processing capabilities of SIMD computers and their ability to exploit spatial/data parallelism by using multiple processing units can result in computational performance improvements when targeting digital communication algorithms. PIM-style architectures can also provide benefits such as reduced power consumption by reducing the amount of off-chip data transfers. The SIMD computer selected to execute the parallel algorithm developed in this research project is the Array Processor found in Atsana Corporation's J2210 System-on-Chip. A brief description of the hardware and software components of the development platform was included for future reference.

Turbo decoding is the target application for SIMD implementation in this thesis, and details of the decoding algorithms used by the constituent SISO decoders were described in section 2.5. Sample turbo decoder implementations that exhibit a small degree of data parallelism were presented to provide background to previous work in this area.

Finally, fundamental concepts about fixed-point number representation and fixed-point arithmetic operations have been explained since practical, real-time turbo decoder implementations are likely to use fixed-point arithmetic.

Chapter 3

Fixed-Point Decoder Implementation

Floating-point precision is usually assumed in the implementation of software based turbo decoders in a research setting in order to better handle the numerical representation of quantities in the decoding algorithms. However, decoders used in real-time embedded applications are likely to be implemented using fixed-point arithmetic. This is also true for hardware implementations with FPGAs and VLSI technologies. For these decoders to be efficient and meet the demands of high data rate systems, they require low-complexity algorithms and small word sizes. The implementation of such a decoder is described in this chapter.

The main objective of a fixed-point implementation is to minimize the fixed word length so that the same integer range and fractional precision is utilized for similar quantities throughout the decoding algorithm.

Applying the idea of *early saturation*, where less bits are allocated to the fixed-point representation of the extrinsic information as compared to other variables, helps the fixed-point turbo decoder implementation to continue to converge with every iteration when no stopping criteria are used.

High level programming languages, such as C and C++, commonly used for embedded firmware development, provide integral data types that do not allow developers to specify configuration parameters to resemble a fixed-point representation. As described previously a fixed-point number is characterized by its word length in bits, the type of sign encoding, and the location of the radix point which in turn determines the fractional precision (number of bits to the right of the radix point) and the integer word length (IWL). A fixed-point data type has been developed using the C++ programming language to help determine the impact that the IWL and fractional word length (FWL) have on the accuracy and BER performance of the fixed-point turbo decoder implementation. The concepts used in the development of the fixed-point data type have been borrowed from similar but more elaborate implementation such as the ones used in the SystemC [33] and IT++ [34] libraries. Limitations of this fixed-point data type are:

- Saturating arithmetic, where the result of an arithmetic operation is clamped to the most positive or negative representable value, is the only overflow handling mode.
- Rounding to infinity is the only quantization mode available.
- Limited precision is used. The word length of the result of any operation or expression is not allowed to exceed 16 bits.

Details of the definition and implementation of the fixed-point data type are included in section A.1 of Appendix A.

This chapter begins with the description of the turbo code used in the UMTS specification. Working with a concrete turbo code makes development and evaluation of the turbo decoder simpler. The following sections include the description of the fixed-point turbo decoder architecture, details of and simplifications in the SISO decoding algorithm of the constituent decoders, and section 3.4 covers details of the finite precision analysis. Simulation results are summarized in section 3.5.

3.1 The UMTS Turbo Code

The development of the turbo decoder discussed in this thesis targets a specific turbo code in order to avoid the many factors that play a role in the performance of these codes. Such factors include: the generator polynomials of the constituent encoder, interleaver design, frame size, code rate (directly related to the puncturing mechanism), and trellis termination. The turbo code selected for the development described in this thesis is the turbo code used by the Universal Mobile Telecommunications Systems (UMTS). The most important parameters that describe the UMTS turbo code are listed in Table 3.1.

UMTS Turbo Code Parameters	
constraint length	4
feed-forward polynomial	15_0
feed-back polynomial	13_0
number of data bits N	$40 \leq N \leq 5114$
code rate R	1/3
interleaver type	Prime Block Interleaver

Table 3.1: UMTS Turbo code Parameters

The UMTS turbo encoder is depicted in Figure 3.1. The data is encoded by the first (upper) encoder in its natural order, and by the second encoder after being permuted by the interleaver. The interleaver is a matrix where the number of rows and columns depends on the size of the data frame. The scrambling of the data is performed by a set of intra-row and inter-row permutations in accordance with an algorithm that is described in the specification [2].

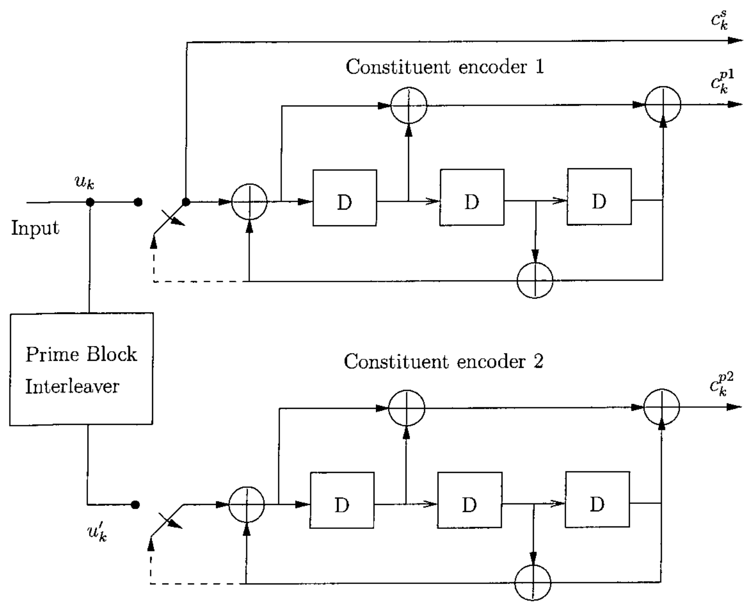


Figure 3.1: UMTS turbo encoder.

As shown in Figure 3.1, the UMTS turbo encoder also employs trellis termination by forcing both encoders back to the all-zero state at a cost of reduced code rate *R*. The two switches are in the up position until the end of the data frame, at which time they get thrown to the down position. Because the state of the two RSC encoders will usually be different after the data has been encoded, each encoder then independently generates the tail bits required to terminate itself. The tail bits are transmitted at the end of the encoded data frame.

3.2 Fixed-Point Turbo Decoder Architecture

The operation of the turbo decoder is described using the notation of Fig. 3.2, with a data frame of size *N*. Throughout the description of the operation of the turbo decoder it is assumed that the decoder operates on unscaled channel symbols y_k and that the operations are performed in the logarithmic domain. This assumption will be validated in section 3.3.1. The role of the "Early Saturation" block will be

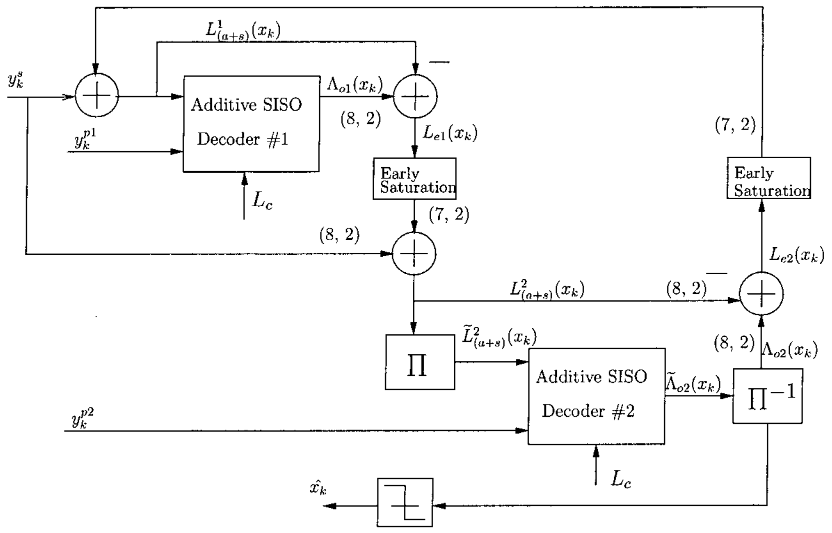


Figure 3.2: Fixed-Point Turbo Decoder Block Diagram.

ignored for now, but its relevance to the correct operation of the turbo decoder is discussed in section 3.4.4. The quantities $L_{e1}(x_k)$ and $L_{e2}(x_k)$ are the extrinsic information generated by constituent decoder #1 and constituent decoder #2 respectively.

They become *a priori* information for the corresponding subsequent constituent decoder. Prior to the first iteration, $L_{e2}(x_k)$ is initialized to zero, because constituent decoder #2 has not yet acted on the data. After each complete iteration, the values of $L_{e2}(x_k)$ are updated to reflect beliefs regarding the data propagated from decoder #2 back to decoder #1. Decoder #1 takes into account the extrinsic information from decoder #2 simply by adding $L_{e2}(x_k)$ to the received systematic channel observation y_k^s. This is considered sufficient because of how the branch metrics are computed in equation (2.24). The result is a new variable $L^1_{(a+s)}(x_k)$ that combines systematic data and extrinsic information. The other input to decoder #1 is the sequence of channel symbols y_k^{p1} that correspond to the parity bits from encoder #1. The output of decoder #1 is the LLR $\Lambda_{o1}(x_k)$, where $1 \leq k \leq N$ since information for the tail bits is not shared with the other decoder.

The extrinsic component of the output of decoder #1 is isolated by subtracting $L^1_{(a+s)}(x_k)$ from $\Lambda_{o1}(x_k)$. The result is denoted $L_{e1}(x_k)$ which is then combined with the received systematic channel symbols y_k^s to form a new variable $L^2_{(a+s)}(x_k)$, just as was done to generate $L^1_{(a+s)}(x_k)$. The inputs to decoder #2 become the interleaved version of $L^2_{(a+s)}(x_k)$ and the channel symbols that correspond to the parity bits of encoder #2. The output of decoder #2 is the LLR $\tilde{\Lambda}_{o2}(x_k)$, $1 \leq k \leq N$, which is deinterleaved to form $\Lambda_{o2}(x_k)$. The extrinsic information $L_{e2}(x_k)$ is calculated by subtracting $L^2_{(a+s)}(x_k)$ from $\Lambda_{o2}(x_k)$ before it is fed back to be used by decoder #1 during the next iteration.

Once the iterations have been completed, a hard bit decision $\hat{x}_k$ is generated using $\Lambda_{o2}(x_k)$ according to the rule that $\hat{x}_k = 1$ when $\Lambda_{o2}(x_k) \geq 0$ and $\hat{x}_k = 0$ when $\Lambda_{o2}(x_k) < 0$.

It is important to note that the architecture of this turbo decoder requires that the entire data frame has been received before proceeding with the decoding. Therefore, it is necessary that there exists a large enough buffer to store the received systematic and parity channel observations as well as the tail bits.

3.3 Simplification of Decoding Algorithm

The first step in the implementation of the proposed turbo decoder consists of the simplification in the decoding algorithm used by the constituent decoders. Once the algorithm has been defined, the effects of finite-precision arithmetic on the operation and accuracy of the decoder are considered.

3.3.1 Log-Add Kernel Approximation

As mentioned previously, the Log-MAP algorithm operating strictly in the logarithmic domain addresses most of the implementation issues associated with MAP decoding. However, the log-add kernel introduced by the Log-MAP algorithm still represents a problem. Recall from equation (2.21) that the log-add kernel consists of a maximization operation with the addition of a correction function. Setting the correction term to zero as it is done in the Max-Log-MAP algorithm results in a simple implementation but at the expense of approximately 0.5 dB loss in coding gain. The use of LUTs to store the correction function has been reported to result in performance that approaches that of the Log-MAP algorithm [27] [26], but it has been argued that it may not be the best suited approach for the emerging global 3G mobile communication systems [29]. For hardware based solutions such as fixed-point ASIC implementations, the impact of LUTs is increased gate count because parallelism may require duplicating the table. Furthermore, the LUT will likely require very fast access times in a turbo decoder, and the implementation of high-speed memory circuits for this purpose may be too expensive. In a software-based decoder, the non-sequential memory accesses that occur from indexing can result in pipeline stalls that increase the execution time [30].

The Linear-Log-MAP algorithm described in [29], where the non-linear correction function is approximated as a linear function, was considered as a potential candidate for the implementation considered in this thesis. However, it was determined that it is computationally expensive when one considers the possible use of a SIMD array processor to execute the algorithm. The constant-log-MAP algorithm of Gross and Gulak [28] was instead chosen as a reasonable approximation for the calculation of the correction factor of the log-add kernel. The constant-log-MAP is characterized by two parameters as indicated in equation (3.1)

$$\ln(e^x + e^y) \approx max(x,y) + \begin{cases} 0, & |y-x| > T \\ C, & |y-x| \leq T \end{cases}, \tag{3.1}$$

where T is some predefined threshold. The analysis performed in [30] shows that values such as $C = 0.5$ and $T = 1.5$ provide near-optimal results over a range of frame sizes and channel conditions.

It is important to note that even though the approximation of the constant-Log-MAP algorithm is equivalent to a LUT with two entries, its complexity can be simplified by realizing that the required conditional operations in software or the multiplexors in hardware are trivial to implement.

Recall that the received channel symbols y_k are scaled by the channel reliability factor L_c prior to being sent to the decoder. This new quantity will be denoted as λ_k. The scaling ensures that the channel observation is compatible with the *a priori* and extrinsic components of the *a posteriori* LLR. From an implementation perspective, as explained in [29], the non-linear correction function $f_c(\cdot)$ of the log-add kernel is the only operation in the Log-MAP algorithm that requires the exact magnitude of the quantity λ_k. The other two operations (maximization and addition) are scale-invariant. In fact, if Max-Log-MAP was used as the decoding algorithm, the operations of the algorithm could easily be performed on unscaled soft values because the channel and noise estimates are not required. On the other hand, for turbo decoding which uses the Log-MAP algorithm to operate on the unscaled soft values y_k, the use of the following soft combiner is required.

$$\begin{aligned} max^{\star}(y_1,y_2) &= \frac{1}{L_c} max^{\star}(\lambda_1,\lambda_2) \\ &= max(y_1,y_2) + \frac{1}{L_c} f_c(L_c|y_1 - y_2|). \end{aligned} \tag{3.2}$$

This new definition of the $max^{\star}$ operation has been verified in simulation with a floating-point implementation of the Log-MAP algorithm where the exact calculation of the correction factor is performed.

Following the idea of the modified soft combiner, the constant-Log-MAP algorithm can undergo a slight modification. Instead of using the procedure of (3.2) every time that $max^{\star}$ is invoked, the parameters C and T of the constant-Log-MAP can be scaled by the reciprocal of L_c when a change in SNR is detected. The scaling of C and T removes their constant property, however, assuming that the SNR is constant over the duration of a frame, they behave as constants over each frame.

From a system level point of view, prior to decoding each frame the channel estimator would provide an estimate of the reciprocal of L_c to the decoder; the decoder would then use the estimate to update the values of C and T as long as there is a recognizable change in value when compared to previous estimates.

3.3.2 Normalization of State Metrics

The recursive nature of the computation of the $\alpha(S_k)$ and $\beta(S_k)$ state metrics leads to the almost monotonic increase of their values as the calculations proceed through the trellis. It is common practice to normalize the value of the state metrics at each trellis stage before proceeding to the next stage. A conservative normalization method is applied for the implementation of the constituent decoders in this design. Normalization consists of subtracting a number which is constant with respect to all the states at the k^{th} trellis stage. The constant value selected is the maximum value among all the states, such that

$$\widehat{\alpha(S_k)} = \alpha(S_k) - \max_{S} \alpha(S_k), \tag{3.3}$$

$$\widehat{\beta(S_k)} = \beta(S_k) - \max_{S} \beta(S_k), \tag{3.4}$$

where $\widehat{\alpha(S_k)}$ and $\widehat{\beta(S_k)}$ are used for subsequent computations. The use of normalization helps to ensure that the state metrics do not overflow. Adding a constant to all $\alpha(S_k)$ or $\beta(S_k)$ for a given k has no influence on the soft output since the constant eventually cancels out [20]. Other normalization methods are described in [35] and [36]. Note that the value of the normalized metrics is at most zero, therefore, it is possible to ignore the sign and treat these quantities as unsigned values in future operations. The design choice was made to maintain a two's complement representation so as to remain consistent with the representation of other variables (such as the branch metrics) involved in the decoding procedure.

3.3.3 Backward Recursion Initialization

The tail bits of the received data frame do not include any information bits; they were simply used to return the constituent encoders back to the all-zero state before encoding a new data frame. Therefore, it is not necessary to compute LLR values for these bits. By inspection of the trellis tail, as seen in Figure 3.3, it is observed that it is possible to perform a short backward recursion only on the tail bits to generate initialization values for the β state metrics that can replace the initialization described by equation (2.29). Another important observation is that these initialization values remain constant throughout all the iterations of the turbo decoding loop for any given frame, and therefore, it is only necessary to calculate them once. The approach used in this implementation is to compute the backward recursion initialization values prior to performing turbo decoding of each received frame. In this manner, each constituent decoder initializes the backward recursion with β values that were computed with their corresponding tail bits.

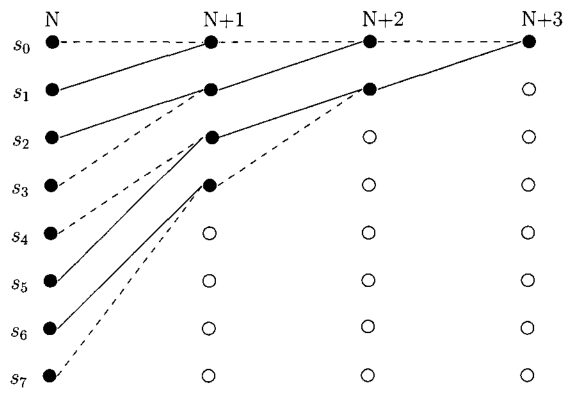


Figure 3.3: Trellis tail for UMTS turbo code.

3.4 Finite Precision Analysis

When designing the constituent decoder based on finite-precision, one faces several issues related to the arithmetic of fixed-precision numbers. The most relevant concerns are:

1. The integer representation of the LLRs, which are inherently real numbers.
2. The internal precision of the decoder arithmetic operations.

As stated previously, one of the design goals for this implementation is to use the same number of fractional bits (fractional precision) for all the quantities that take part in the decoding algorithm. This design constraint simplifies the arithmetic operations by forcing the radix point of all operands to be aligned, and so shifting operations are avoided.

3.4.1 Quantization Method

Our design approach consists of quantizing the unscaled soft-values y_k received from the demodulator and then operating strictly with fixed-point quantities. Based on the concepts covered in section 2.7, fixed-point values are presented in the format (n_{wl}, n_{fwl}), such that n_{wl} is the total number of bits in the entire word length, and n_{fwl} represents the fractional part of the value. The IWL is determined by:

$$\text{IWL} = n_{iwl} = n_{wl} - n_{fwl} - 1 \tag{3.5}$$

The removal of one extra bit to obtain n_{iwl} accounts for the fact that the most significant bit in n_{wl} is the sign bit. All the quantities use signed two's complement representation. It is obvious that if n_{wl} is held constant, then for larger n_{fwl} higher fractional precision can be maintained while the dynamic integral range that can be represented is smaller. On the other hand, if n_{fwl} is reduced, the dynamic integral range is larger at the expense of reduced fractional precision. The range of values for a given fixed point format is the following:

$$\left[-2^{(n_{iwl}-1)}, 2^{(n_{iwl}-1)} - 2^{-n_{fwl}}\right] \tag{3.6}$$

The quantization or rounding mode selected for this design is rounding to infinity. In this mode, the soft-values from the channel are first scaled using binary point-only scaling and then rounded towards positive infinity if positive or negative infinity if negative. The quantization operation is performed according to equation (3.7)

$$y_k^q = \lfloor 2^{n_{fwl}} \cdot y_k \pm 0.5 \rfloor, \tag{3.7}$$

where $\lfloor r \rfloor$ refers to the integer portion of the argument r, and the + sign or - sign is used depending on whether the quantity y_k is positive or negative.

3.4.2 Effect of Fractional Length

All experiments were conducted where the simulator employed binary phase shift keying (BPSK) modulation and an AWGN channel model to simulate transmission of the coded symbols.

Experiments were conducted using a single constituent decoder in order to evaluate the impact of the fractional word length (the number of fractional bits in the fixed-point word) on the BER performance of the turbo decoder. This allowed to isolate the algorithm implementation for the SISO decoders. This decision was based on the observation that the constituent decoder calculates the LLRs, while the structure of the turbo decoder concentrates on the exchange of extrinsic information between the constituent decoders.

Analysis of the effect of n_{fwl} was performed through simulation by using a fixed-point number representation for all the variables in the decoding algorithm such that n_{wl} was set to the maximum value allowed by the fixed-point data type while the value for n_{fwl} was varied. These simulations also allowed validation of the operation of the modified constant-Log-MAP algorithm described previously when used to decode the RSC code of the constituent encoder. The BER performance obtained for the different (n_{wl}, n_{fwl}) configurations was compared against that of a floating-point baseline. Simulation results are displayed in Figures 3.4 - 3.6.

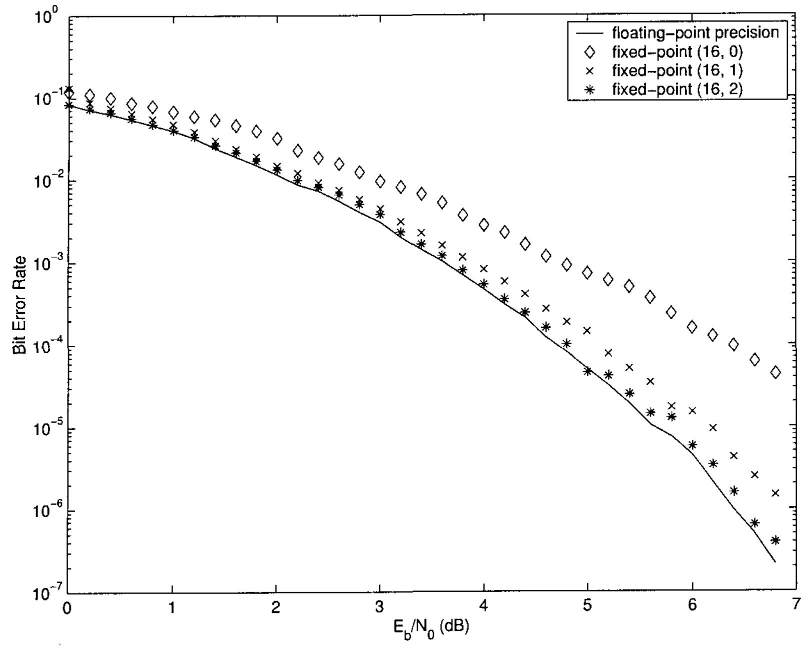


Figure 3.4: Fixed-point BER performance with $0 \leq \text{FWL} \leq 2$.

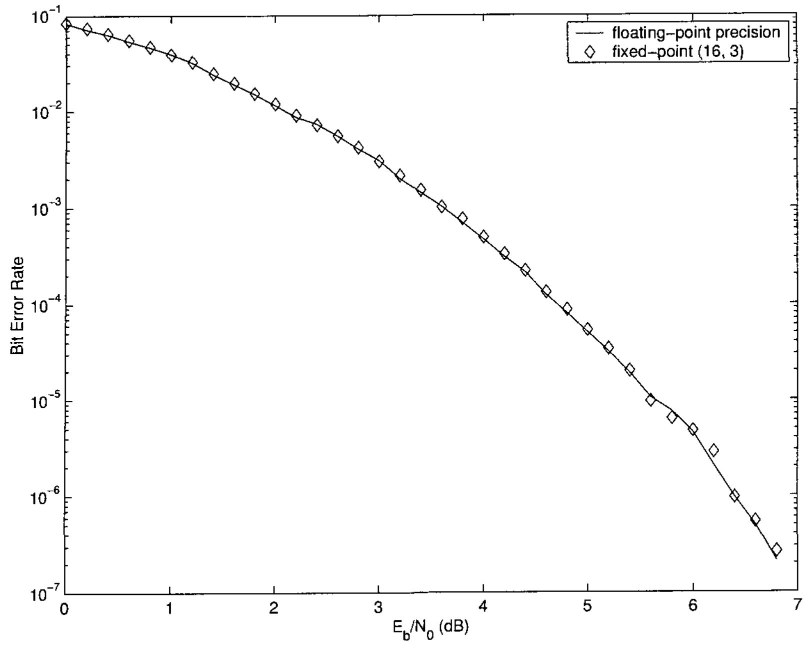
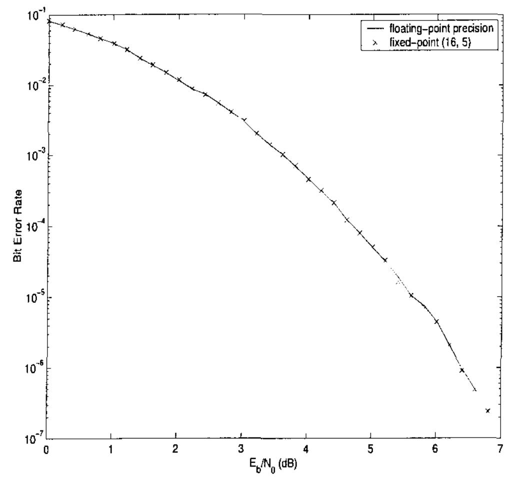
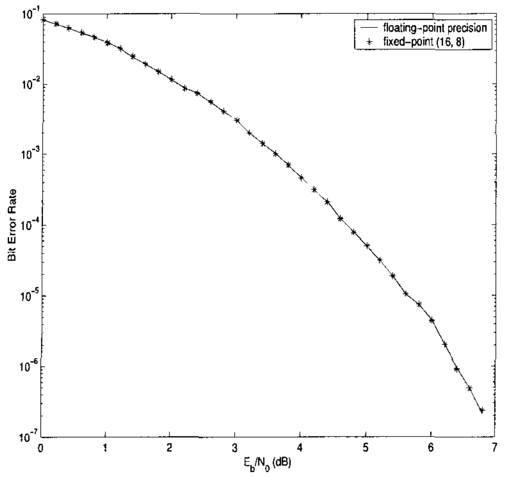


Figure 3.5: Fixed-point BER performance with FWL = 3.

Simulation results in Figure 3.4 clearly show that a minimum of two fractional



(a) FWL = 5 bits.



(b) FWL = 8 bits.

Figure 3.6: BER performance with unnecessarily long FWL.

bits are require to obtain performance that approaches that when using infinite precision such as with a floating-point decoder implementation[1].

Figure 3.5 shows that by setting $n_{fwl} = 3$ the results achieved are almost indistinguishable from those of the floating-point baseline. Finally, the results of Figure 3.6 are presented to indicate that increasing the number of fractional number of bits to larger than three provides no further benefit to the BER performance that can be achieved.

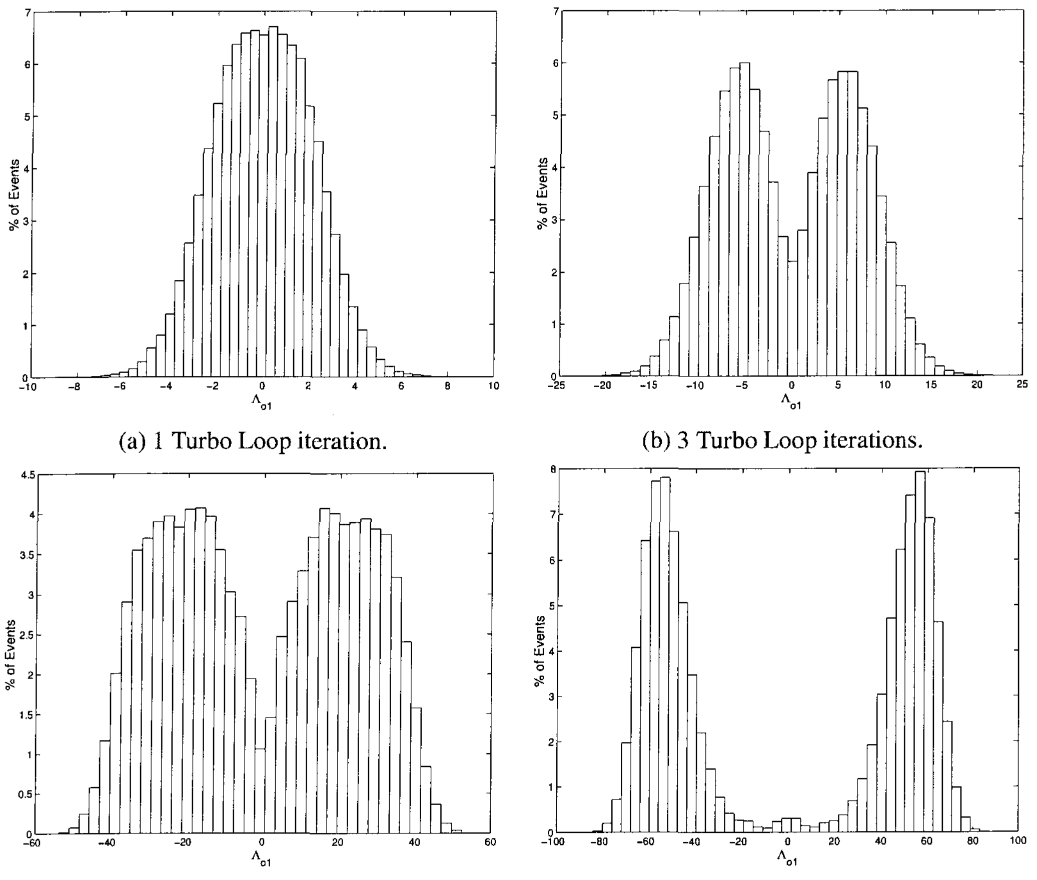
3.4.3 Density Evolution Analysis

When considering the iterative nature of the decoding procedure used in the decoding of turbo codes, there exists one additional issue related to the finite precision arithmetic of a fixed-point implementation. This has to do with the fact that the value of the extrinsic information involved in the operation of the SISO decoders grows as long as the iterations progress. This growth in value has a significant impact in the number of bits that must be reserved for the IWL of the fixed-point representation of the extrinsic LLR.

A different set of experiments was conducted where the floating-point model of

[1]Floating-point representation of real-valued quantities approaches infinite precision as the number of bits in the mantissa increases

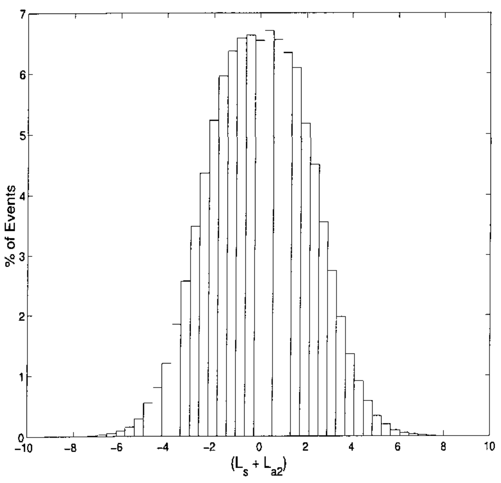
the turbo decoder was used to conduct a density evolution analysis for the quantities involved in the iterative procedure, namely the output LLRs generated by the constituent decoders and the extrinsic information exchanged between decoders. Histograms are used to display the distribution of the different quantities as the iterations of the turbo decoding loop progress. The effect of the SNR level is also investigated by comparing the results for $E_b/N_0 = 0.6$ dB and $E_b/N_0 = 2.0$ dB. Figures 3.7 through 3.10 show the distribution of $\Lambda_{o1}(x_k)$, $L^2_{a+s}(x_k)$, $\Lambda_{o2}(x_k)$ and $L_{e2}(x_k)$ for the case when the channel SNR is 0.6 dB; Figures 3.11 through 3.14 display distributions for the same quantities when the SNR is 2.0 dB.



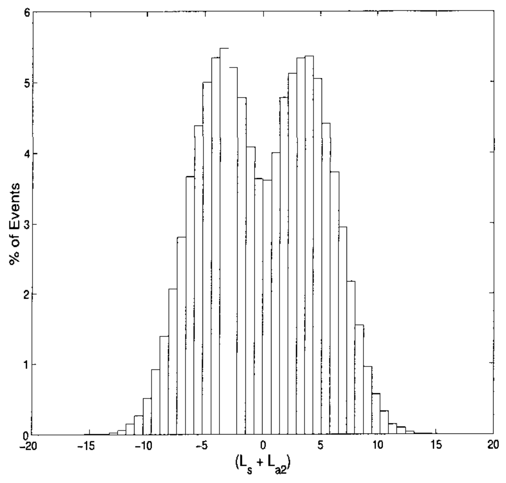
(a) 1 Turbo Loop iteration. (b) 3 Turbo Loop iterations.

(c) 5 Turbo Loop iterations. (d) 10 Turbo Loop iterations.

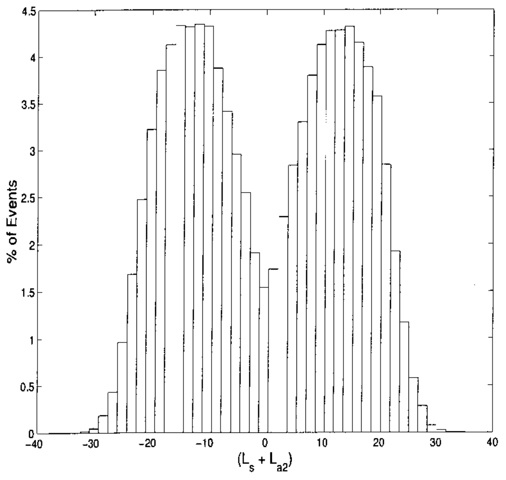
Figure 3.7: Density Evolution of $\Lambda_{o1}(x_k)$, $E_b/N_0 = 0.6$dB.



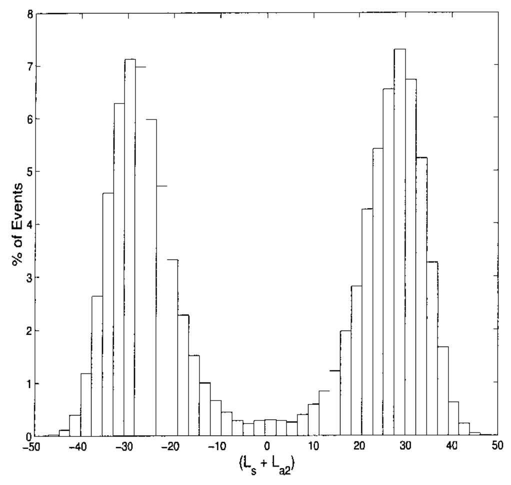
(a) 1 Turbo Loop iteration.



(b) 3 Turbo Loop iterations.

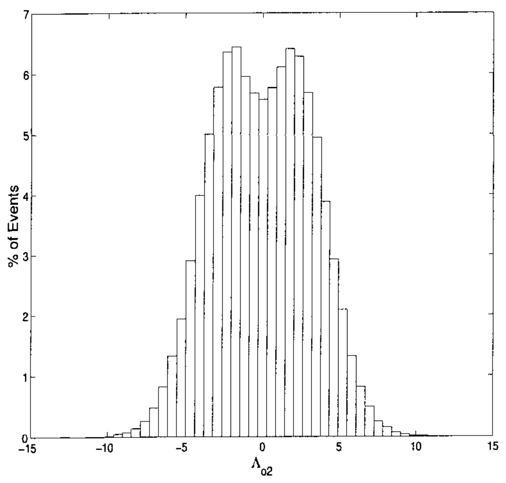


(c) 5 Turbo Loop iterations.

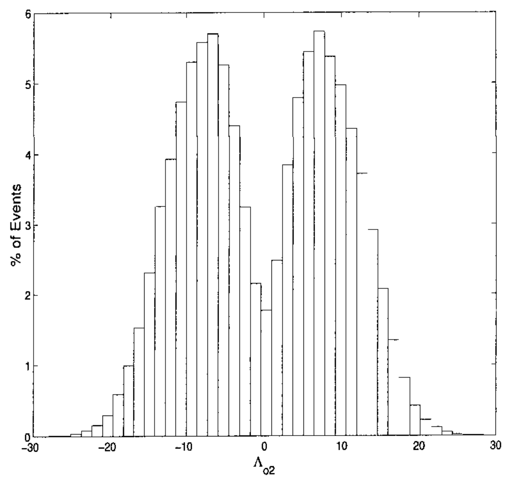


(d) 10 Turbo Loop iterations.

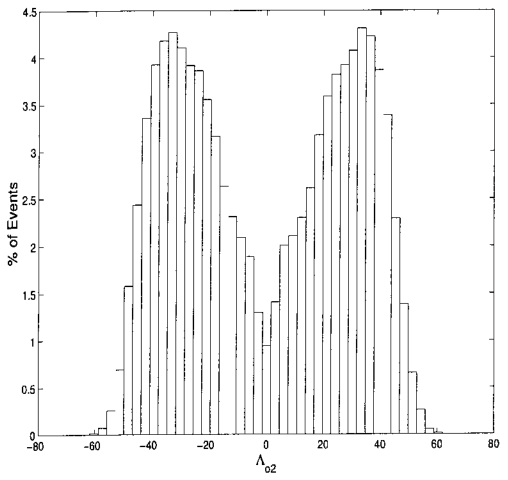
Figure 3.8: Density Evolution of $L^2_{(a+s)}(x_k)$, $E_b/N_0 = 0.6$dB.



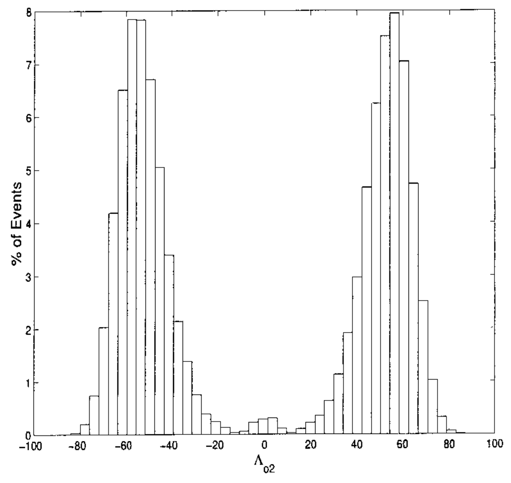
(a) 1 Turbo Loop iteration.



(b) 3 Turbo Loop iterations.

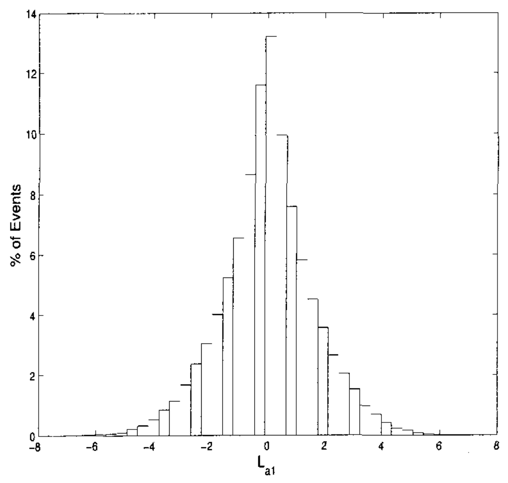


(c) 5 Turbo Loop iterations.

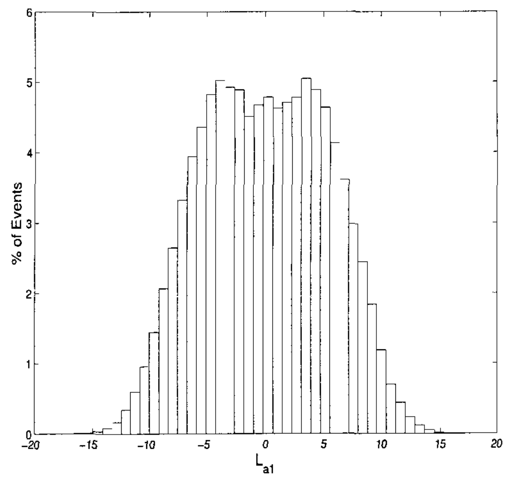


(d) 10 Turbo Loop iterations.

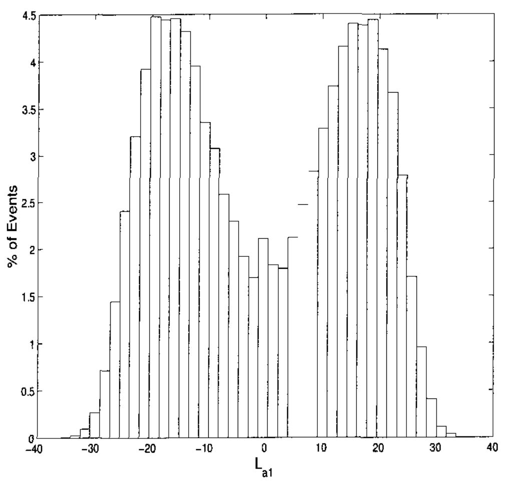
Figure 3.9: Density Evolution of $\Lambda_{o2}(x_k)$, $E_b/N_0 = 0.6$dB.



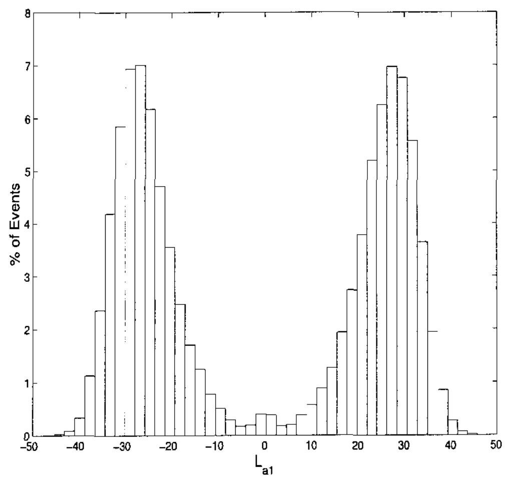
(a) 1 Turbo Loop iteration.



(b) 3 Turbo Loop iterations.

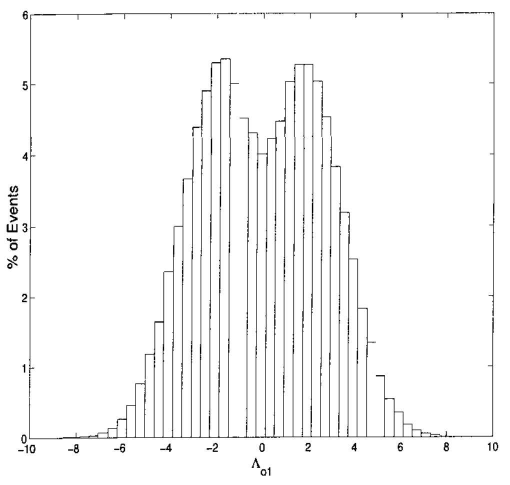


(c) 5 Turbo Loop iterations.

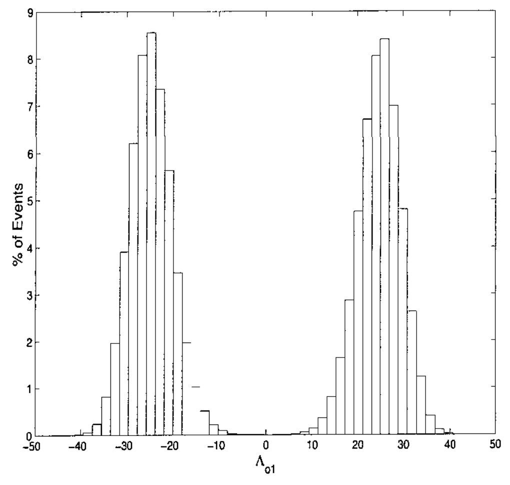


(d) 10 Turbo Loop iterations.

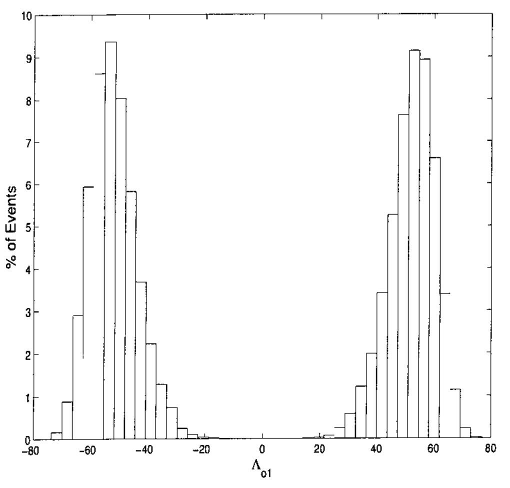
Figure 3.10: Density Evolution of $L_{e2}(x_k)$, $E_b/N_0 = 0.6$dB.



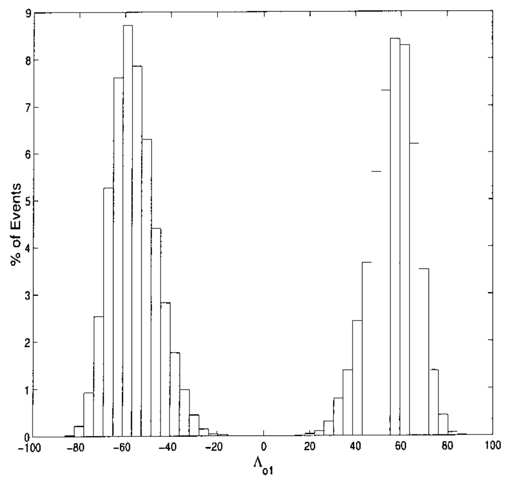
(a) 1 Turbo Loop iteration.



(b) 3 Turbo Loop iterations.

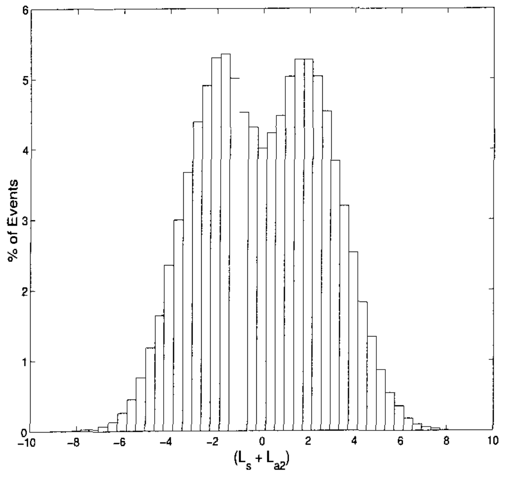


(c) 5 Turbo Loop iterations.

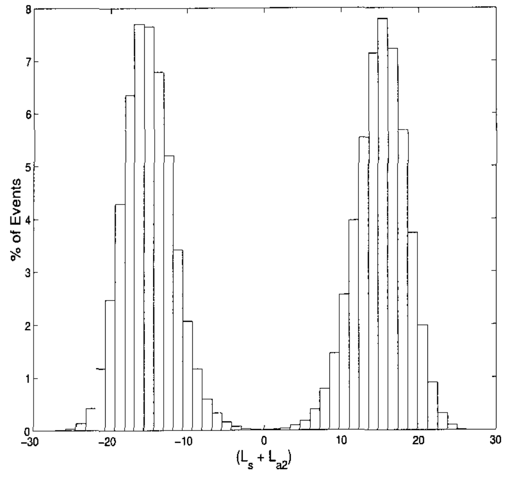


(d) 10 Turbo Loop iterations.

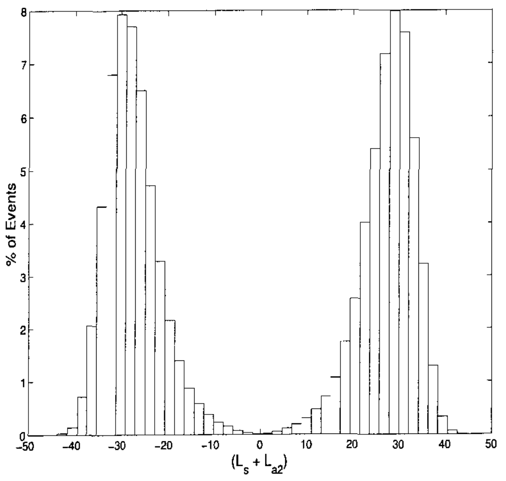
Figure 3.11: Density Evolution of $\Lambda_{o1}(x_k)$, $E_b/N_0 = 2.0$dB.



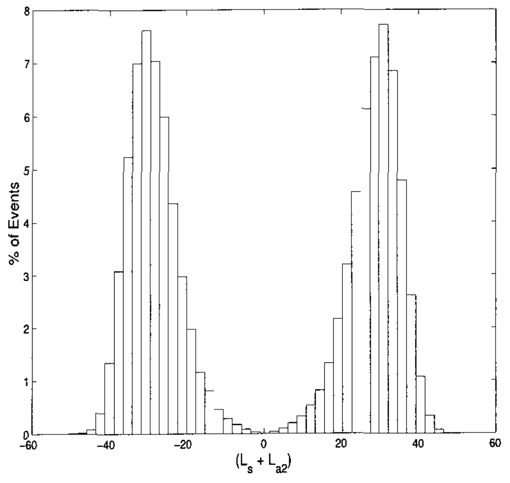
(a) 1 Turbo Loop iteration.



(b) 3 Turbo Loop iterations.

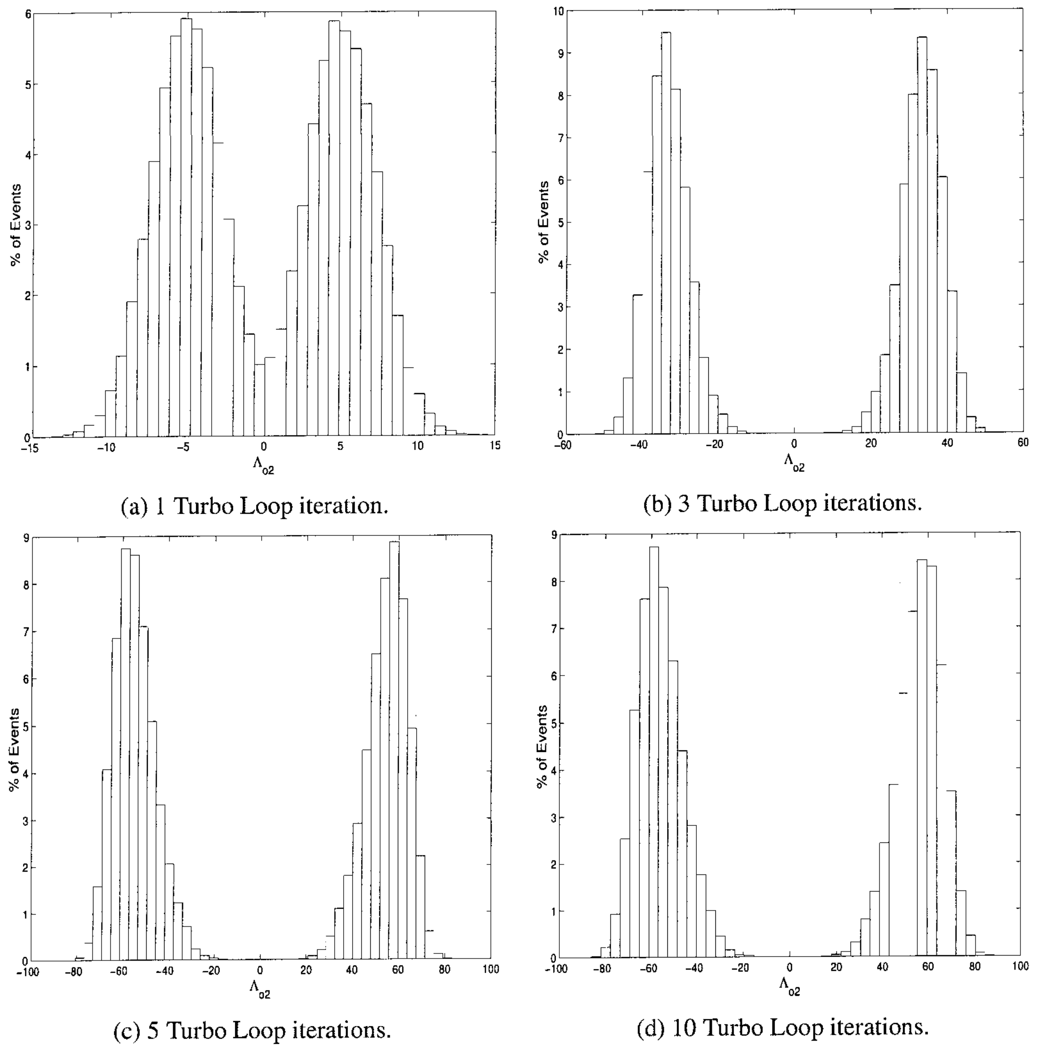


(c) 5 Turbo Loop iterations.



(d) 10 Turbo Loop iterations.

Figure 3.12: Density Evolution of $L^2_{(a+s)}(x_k)$, $E_b/N_0 = 2.0$dB.



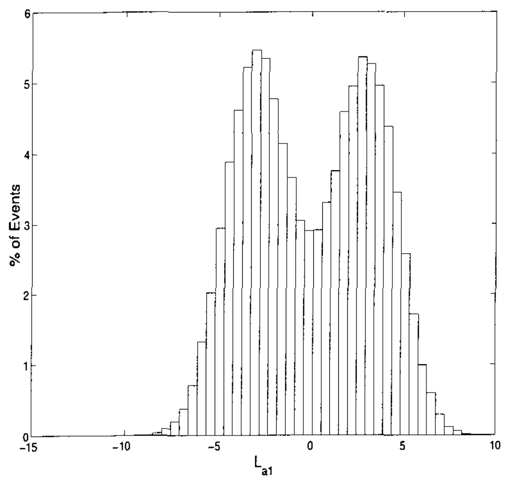
(a) 1 Turbo Loop iteration.

(b) 3 Turbo Loop iterations.

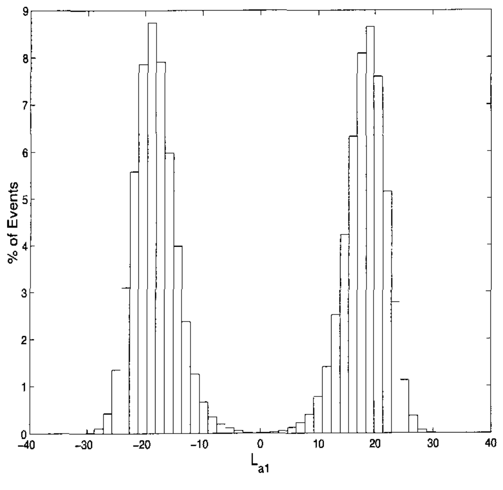
(c) 5 Turbo Loop iterations.

(d) 10 Turbo Loop iterations.

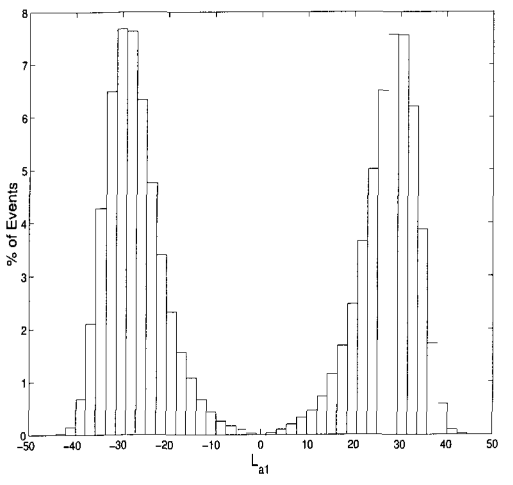
Figure 3.13: Density Evolution of $\Lambda_{o2}(x_k)$, $E_b/N_0 = 2.0$dB.



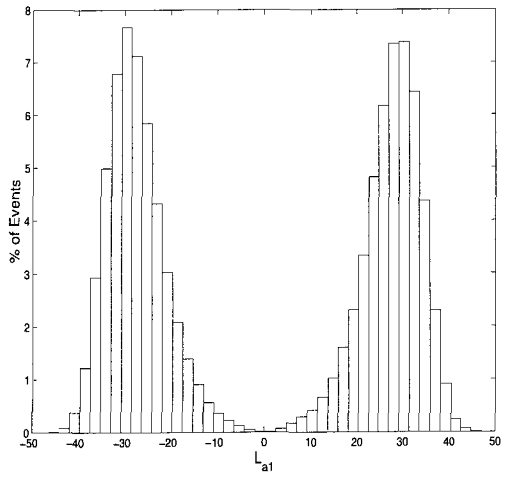
(a) 1 Turbo Loop iteration.



(b) 3 Turbo Loop iterations.



(c) 5 Turbo Loop iterations.



(d) 10 Turbo Loop iterations.

Figure 3.14: Density Evolution of $L_{e2}(x_k)$, $E_b/N_0 = 2.0\text{dB}$.

Notice that for the lower SNR levels it takes more turbo loop iterations before the LLR distributions start showing a more pronounced division between those that estimate the data bit as $u_k = 1$ and those estimating the data bit as $u_k = 0$. [2]

The plots from Figures 3.7, 3.9, 3.11 and 3.13 show that the absolute value of the magnitude of the quantities $\Lambda_{o1}(x_k)$ and $\Lambda_{o2}(x_k)$ may grow to greater than 80 when no constraints are imposed on the floating-point model. Based on these observations, one can argue that a minimum of 7 integer bits in the fixed-point word are required to represent the dynamic integer range of the quantities $\Lambda_{o1}(x_k)$ and $\Lambda_{o2}(x_k)$ for up to ten iterations of the turbo decoder. Using the same criteria, a minimum of 6 integer bits would be required for the representation of the extrinsic information exchanged between constituent decoders. However, as it will be shown by simulation results for the turbo decoder implementation presented in this chapter, fewer bits can be used if saturating arithmetic is applied.

3.4.4 Early Saturation of Extrinsics

Preliminary simulation of the turbo decoder showed that unstable operation of the decoder occurred when the extrinsic information, the quantities $L_{e1}(x_k)$ and $L_{e2}(x_k)$, were allocated the same fixed-point word length as the LLRs $\Lambda_{o1}(x_k)$ and $\Lambda_{o1}(x_k)$. Using the same word length resulted in the variables $L^1_{(a+s)}(x_k)$ and $L^2_{(a+s)}(x_k)$ eventually approaching the same saturation limit imposed by the fixed-point word length as the constituent decoder output LLRs. Recall that these variables are used as input to their corresponding constituent decoders and are subtracted from the output LLR in order to isolate the extrinsic information. With both the input and output variables having similar values in magnitude, sign reversals can result when the extrinsic information is calculated. This sign reversal translates into errors being introduced by the iterative decoding mechanism instead of reducing them. In this case the BER performance of the decoder displays an oscillatory behavior, such that the BER would drop as the iterations progress until both input and output LLRs would

[2]Recall that the sign of the LLR is used to make the hard decision: LLR $< 0 \rightarrow u_k = 0$, and LLR $\geq 0 \rightarrow u_k = 1$.

saturate and then the BER would escalate to a value similar to the one observed after the first decoder iteration. These oscillations would repeat with a period equal to the number of decoder iterations needed for the quantities to reach saturation.

It was then determined that this inappropriate behavior could be avoided by forcing *early saturation*, implemented as the allocation of fewer bits in the fixed-point word representation of the extrinsic quantities $L_{e1}(x_k)$ and $L_{e2}(x_k)$ to prevent sign reversal. The decision to perform early saturation was validated by using the floating-point decoder model and applying uniform soft quantization to the extrinsic variables. The results of simulations for several soft quantization ranges are presented in Figures 3.15 through 3.18 for different iterations of the decoder.

Notice the abnormal behavior (not smooth transition) of the BER curves presented in Figures 3.15 through 3.18 at the 1.2 dB SNR point. This particular result is attributed to methodology used in the simulations to estimate the BER performance at a given SNR level. The SNR range 0.0dB $\leq E_b/N_0 \leq$ 1.8dB was divided into three regimes where different number of bit errors are accumulated in each region to estimate the bit error rate. The three regimes under evaluation were the following:

Regime 1: 0.0dB $\leq E_b/N_0 \leq$ 0.4dB

Simulations were conducted until at least 5000 bit errors were recorded.

Regime 2: 0.6dB $\leq E_b/N_0 \leq$ 1.0dB

Simulations were conducted until at least 1000 bit errors were recorded.

Regime 3: 1.2dB $\leq E_b/N_0 \leq$ 1.8dB

Simulations were conducted until at least 100 bit errors were recorded.

The 1.2 dB point is a transition point where the number of bit errors collected is reduced by an order of magnitude to shorten simulation times. The abrupt change in simulation constraints and the randomness of the procedure leads to simulation results not matching the expected behavior of the decoder.

A recommendation for future experiments is to set simulation constraints where simulations will continue until a minimum number of bit errors and a minimum number of frame errors are recorded.

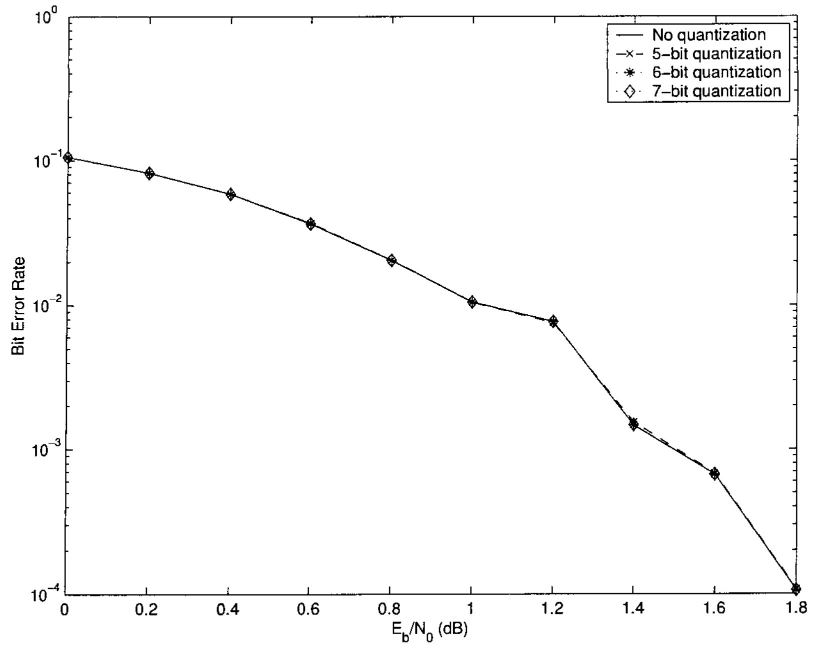


Figure 3.15: Early saturation of extrinsics, 2 iterations.

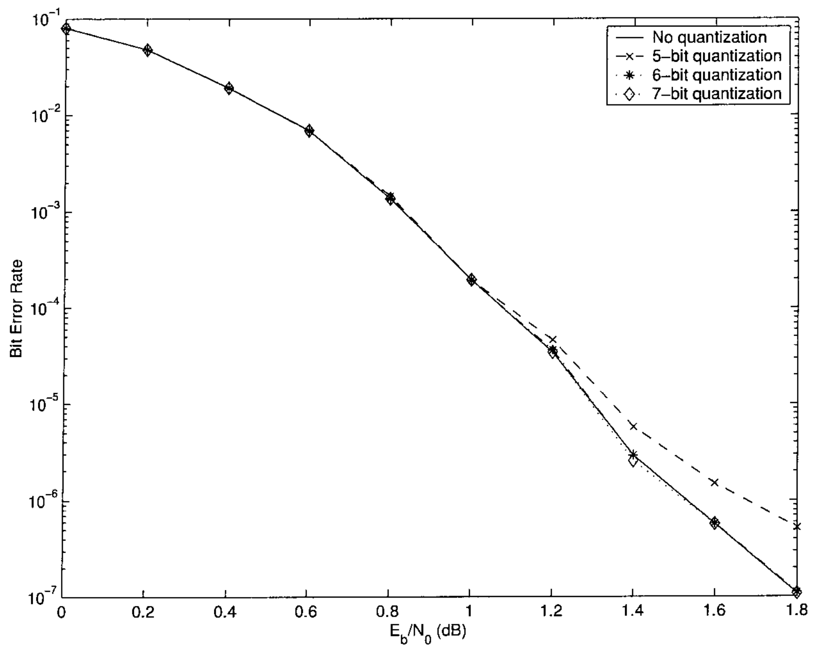


Figure 3.16: Early saturation of extrinsics, 4 iterations.

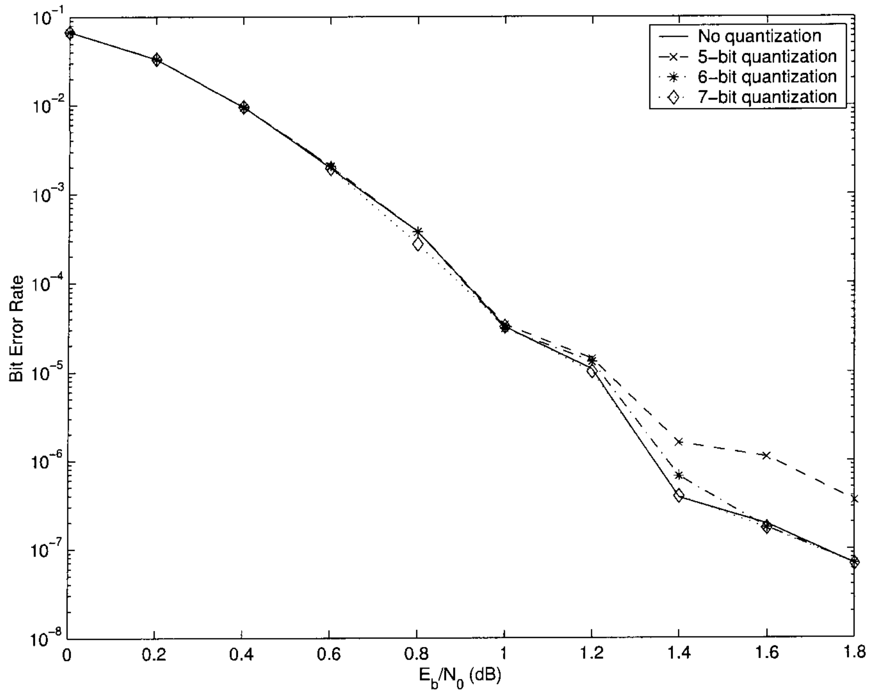


Figure 3.17: Early saturation of extrinsics, 6 iterations.

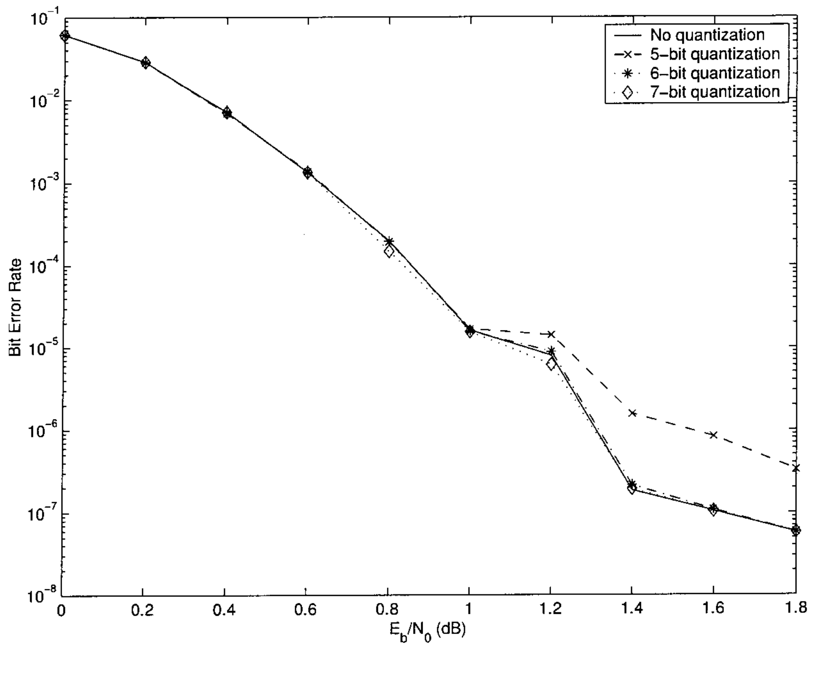


Figure 3.18: Early saturation of extrinsics, 8 iterations.

Observe that early saturation of the extrinsic information has a greater impact on the BER performance for a higher number iterations, and also that it is more pronounced in the *error floor* region. In conclusion, it was determined that limiting the fixed-point word length for the extrinsic variables to one bit less than the fixed-point word length of the output LLRs was sufficient to avoid degradation in the performance of the turbo decoder due to sign reversal during calculation of the extrinsic information.

3.5 Fixed-Point Turbo Decoder Simulations

Simulations were conducted to evaluate the performance of the fixed-point turbo decoder based on the modified constant-Log-MAP algorithm. The simulations were only performed for the AWGN channel. Taking advantage of the results presented in section 3.4.2, the fixed-point representation of all the quantities internal to the SISO decoders as well as that of the quantities external to these decoders have two fractional bits reserved within the finite word length. Simulations evaluate the BER performance of the fixed-point turbo decoder implementation for a range of IWL. Arithmetic operations and intermediate results use the maximum precision allowed by the fixed-point data type. All simulations were performed with the frame length set to 1024 bits.

3.5.1 BER Performance

Figures 3.19 through 3.22 display the BER performance for 2, 4, 6 and 8 decoder iterations, and compare the results against the floating-point baseline. Fixed-point word configurations evaluated are (6, 2), (8, 2) and (12, 2), where the first digit in parenthesis indicates the word length n_{wl} and the second indicates the number of fractional bits n_{fwl}. The error introduced during fixed-point arithmetic decoding is the combination of two main sources: the quantization error associated with the quantized soft values from the channel before the decoder, and the accumulated errors resulting from saturation of the arithmetic operations inside the decoder. In particular, a loss of precision is introduced by the clipping of the extrinsic LLR values at the output of the constituent decoders. It was observed during simulations, that allocating a number of bits greater than four to the integer range of the fixed-point quantity resulted in better performance than when allocating the additional bits to the fractional precision. When allocating an insufficient number of bits to the integer part, the decoder performs satisfactorily for a limited number of iterations (less than four), before the extrinsic information exchanged between constituent decoders starts diverging. As a result the variables involved in the operation of the finite precision turbo decoder are configured with the values of Table 3.2. The

word length (n_{wl})	8
fractional bits (n_{fwl})	2
integer bits (n_{iwl})	5
sign representation	two's complement

Table 3.2: Turbo Decoder Fixed-Point Word Configuration.

exceptions are $L_{e1}(x_k)$ and $L_{e2}(x_k)$ that use $n_{wl} = 7$ and $n_{iwl} = 4$ while still keeping the same number of fractional bits.

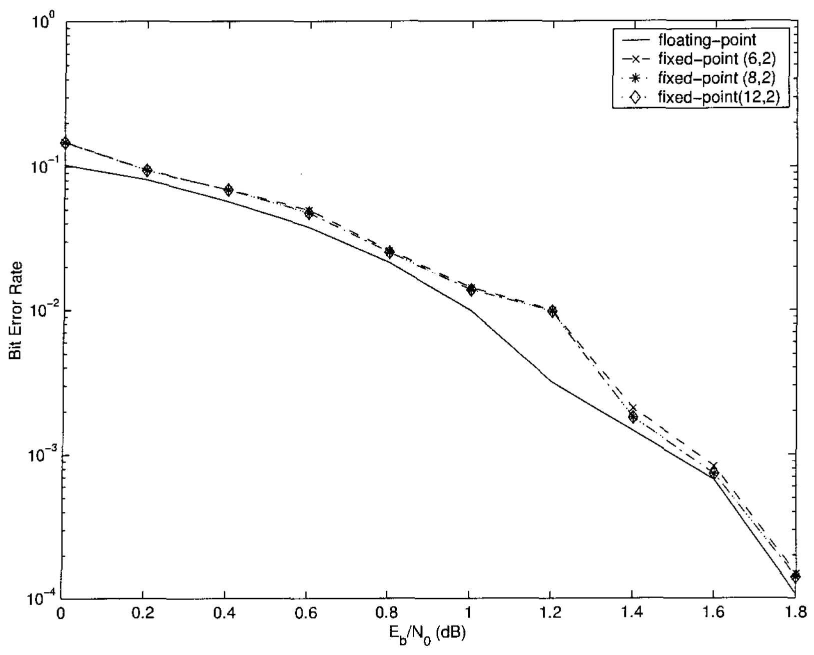


Figure 3.19: Fixed-point Turbo Decoder BER performance, 2 iterations.

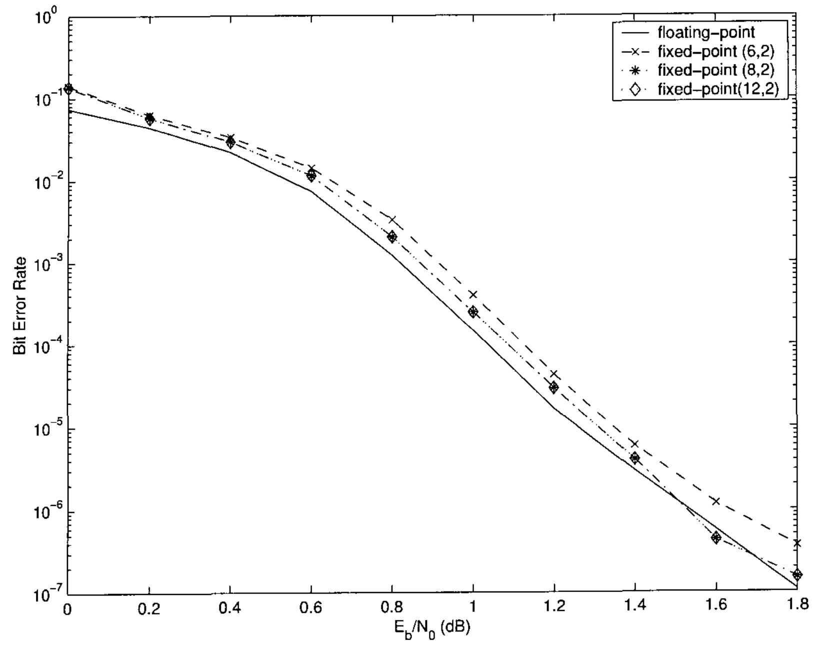


Figure 3.20: Fixed-point Turbo Decoder BER performance, 4 iterations.

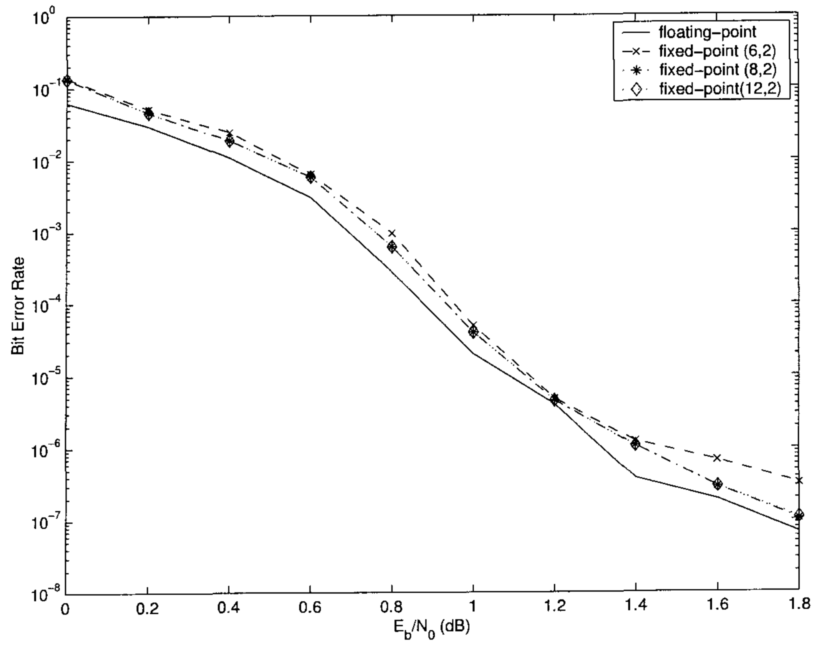


Figure 3.21: Fixed-point Turbo Decoder BER performance, 6 iterations.

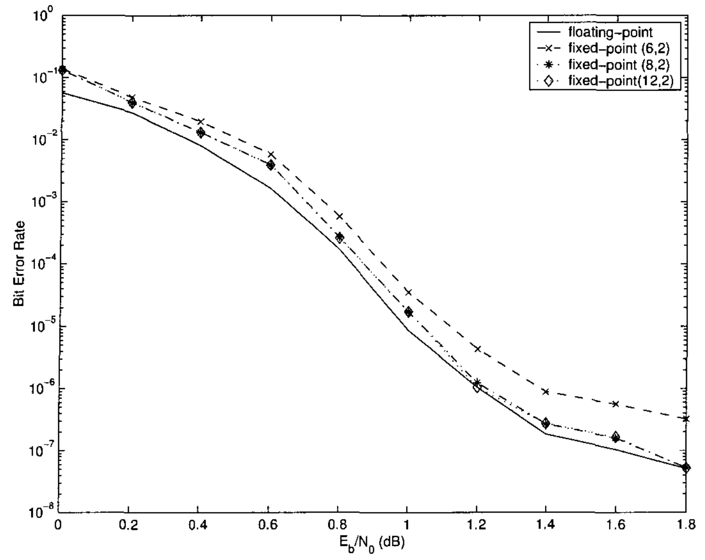


Figure 3.22: Fixed-point Turbo Decoder BER performance, 8 iterations.

3.6 Summary

The design, implementation and evaluation of a fixed-point turbo decoder for the UMTS turbo code has been discussed in this chapter. Of particular importance is the selection of the constant-Log-MAP algorithm for use in the constituent SISO decoders. Slight modifications are applied to the decoding algorithm to be able to work with unscaled soft channel observations. The effects of fractional word length were examined by evaluating the performance of a constituent SISO decoder, and results demonstrate that 2 fractional bits, with an 8-bit total word length, are sufficient to obtain a BER that closely matches that of a floating-point implementation over a range of SNR levels. Selection of the IWL was conducted through a density evolution analysis. It was determined that while most quantities used in the iterative turbo decoding algorithm require an IWL with at least 5 bits, additional clipping is necessary for the extrinsic LLR information in order to help maintain convergence as the number of decoder iterations increases. Evaluation of the fixed-point turbo decoder shows that a fixed-point word configuration of (8, 2) results in a BER performance that lies within 0.1 dB of the performance obtained with the floating-point reference implementation for SNR levels between 0.0 dB and 1.8 dB.

Chapter 4

Data Parallelism in Turbo Decoding

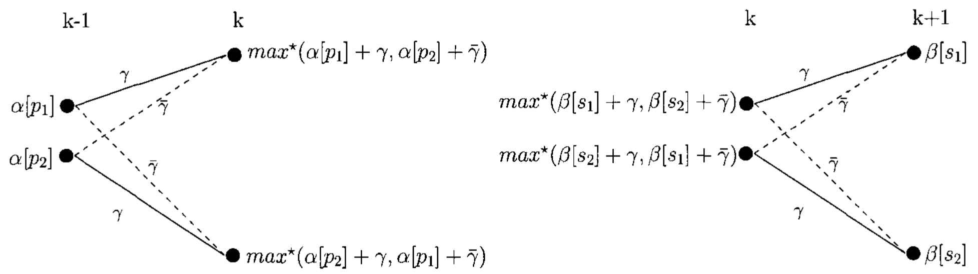
Exploring data parallelism in signal processing algorithms is of great interest because it can lead to a high degree of parallelism involving thousands of concurrent data operations, resulting in high speed-up factors. This chapter presents the exploration of data parallelism in the decoding algorithm for turbo codes. The main objectives are: (a) the identification of suitable parallel data structures that can be distributed among the PEs of a SIMD computer, (b) the distribution of the parallel data among the PEs, and (c) identification of a simple inter-processor communication network that will support the data transfer operations required by the parallel algorithm.

The chapter begins by analyzing the structure of the trellis that describes the turbo code. This analysis allows identification of the relevant parallel data sets and the communications patterns used in the algorithm. An efficient mapping of the trellis structure to SIMD hardware is presented in section 4.1, followed by an estimation of the memory requirements per PE. A high level description of the parallel algorithm, in particular the procedure for the computation of the state metrics, is presented in section 4.2. Finally a brief discussion on the topics of parallelism and efficiency is given.

4.1 Turbo Code Trellis Analysis

It was previously stated that PCCCs use simple RSC constituent codes with short constraint lengths. Rate 1/2 encoder models are popular such as in the case of the UMTS turbo code that was presented in the previous chapter. The trellis diagram representation of the constituent code is a useful means of describing the turbo code.

Analysis of the trellis structure starts by examining its basic building blocks. For a binary code alphabet, the branch transitions appear as butterfly pairs of the form displayed in Figure 4.1. Here the dashed lines are for branches where the systematic bit of the coded symbol $c^s = 0$, and the solid lines are for the branches where the systematic bit $c^s = 1$. The branch metrics are represented by γ and $\bar{\gamma}$ such


k-1
k
$max^\star(\alpha[p_1] + \gamma, \alpha[p_2] + \bar{\gamma})$
$\alpha[p_1]$
$\alpha[p_2]$
γ
$\bar{\gamma}$
$\bar{\gamma}$
γ
$max^\star(\alpha[p_2] + \gamma, \alpha[p_1] + \bar{\gamma})$
k
k+1
$\beta[s_1]$
$max^\star(\beta[s_1] + \gamma, \beta[s_2] + \bar{\gamma})$
$max^\star(\beta[s_2] + \gamma, \beta[s_1] + \bar{\gamma})$
γ
$\bar{\gamma}$
$\bar{\gamma}$
γ
$\beta[s_2]$

(a) Log-MAP forward recursion (b) Log-MAP backward recursion

Figure 4.1: Trellis Butterfly with State Metrics.

that the bits of the coded symbol that correspond to the branch identified by $\bar{\gamma}$ are the negated version of the bits in the coded symbol corresponding the branch labeled with γ. This is a property of RSC codes, and when a rate 1/2 encoder is used, half the butterfly pairs have coded symbols (00) and (11), and the other half have coded symbols (01) and (10) [20].

Figures 4.1(a) and 4.1(b) show the forward and backward recursions respectively for the calculation of the α and β state metrics in the Log-MAP algorithm. It is shown that all the state nodes perform the same data operations of addition and *max** as the recursions progress. The branch metrics can be calculated ahead of time for all trellis stages and stored in a buffer or, to reduce memory usage at the expense of repetitive computations, they can be computed for each trellis stage at every step of the recursions. Also recall that the likelihood of the information bit $u_k = 1$ and the likelihood of the information bit $u_k = 0$ for all states along all time intervals, as well as the LLRs, can be computed during either the forward or backward recursion, whichever is chosen to be the second recursion.

These observations point out the data that can be represented as parallel variables, and they are:

- The branch metrics γ.
- The forward state metrics α.
- The backward state metrics β.
- The log-likelihood of $u_k = 1$.
- The log-likelihood of $u_k = 0$.
- The LLR values.

By realizing that the same data operations are repeated on different data sets due to the recursive nature of the algorithm, data parallelism can be exploited if all these operations can be performed concurrently on all the data sets. If only the individual butterfly pairs are considered, the obvious solution would be to allocate a PE to every state node of the trellis, and Figure 4.1 shows that data communication is such that one node at time interval k needs data from two immediate predecessors at time interval $k-1$ during the forward recursion, and from two immediate successors at time interval $k+1$ during the backward recursion. Such a communication pattern can be expressed as a point-to-point (shift) data routing operation.

However, the communication pattern gets more complicated when one considers the entire trellis sections as illustrated in Figure 4.2, where each trellis section is made up of a number of butterfly pairs that depends on the number of states in the column, and the number of states depends on the constraint length K_c of the constituent code.

Even though the inter-processor data routing operating may be visualized as a shift of data from states at interval $k-1$ or $k+1$ to states at interval k, it is not a

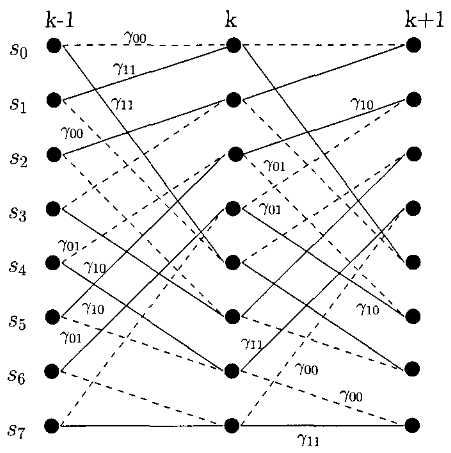


Figure 4.2: Adjacent Trellis Sections for RSC Code of UMTS Turbo Code.

simple left-right/right-left data transfer. There exists a vertical offset for most of the branches, and this vertical offset varies from butterfly pair to butterfly pair.

A potential solution would be to organize the PEs of the array processor as a two dimensional grid with an inter-processor communication network that corresponds exactly to the branches of the trellis of the constituent code. Such a solution requires a SIMD array processor architecture with fine grain data parallelism. SIMD computers with PIM technology offer such fine grain data parallelism, however, this alternative for mapping trellis parallel data structures to SIMD hardware would not result in practical implementations for the following reasons:

1. The number of PEs necessary to support large frame sizes (up to 5114 bits for UMTS) grows very rapidly as indicated by equation (4.1), where N is the length of the data frame, K_c is the constraint length of the constituent code and M_{PE} is the number of processing element nodes.

$$M_{PE} = N \times 2^{K_c - 1} \tag{4.1}$$

2. The inter-processor communication network does not match any of the networks commonly used in parallel processor architectures in spite of the regular structure of the trellis.

The first issue can be alleviated by the use of the latest VLSI technologies, but the amount of distributed memory required for all the PEs may impose other limitations. The exponential increase in the number of PEs can be kept to a minimum by remembering that turbo codes typically use simple constituent codes with constraint length $3 \leq K_c \leq 5$ [37].

The second issue, however, has a greater impact for practical implementations because a customized inter-processor communication network would be tied to exactly one RSC constituent code and would offer no flexibility to support other codes. This holds true even for configurations that specify short frame lengths because the connections between state nodes of adjacent trellis columns are determined by the generator polynomials for the constituent RSC.

4.1.1 Efficient Trellis Structure to SIMD Hardware Mapping

A more efficient and flexible mapping of the RSC code trellis structure to SIMD hardware is portrayed in Figure 4.3. This particular approach recognizes that the

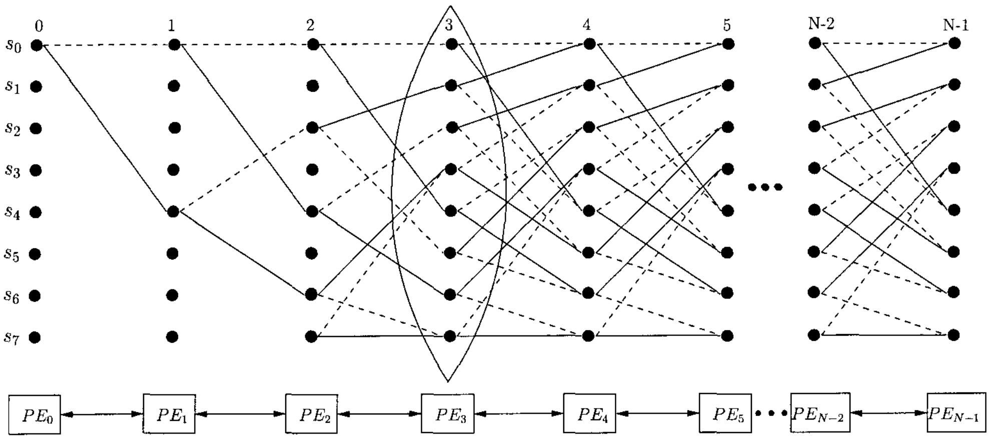


Figure 4.3: PE allocation.

relatively short constraint length of the constituent code may allow for the distribution of the parallel variables such that each PE holds the set of data that corresponds to a column or section of the trellis. In this manner each PE operates sequentially on all the state nodes that correspond to the trellis section allocated to it. However, since all the PEs execute the same instructions in lockstep, all trellis sections are scanned in parallel.

The main advantage of this approach is that the complexity of the inter-processor communication network is reduced to that of a linear array. Since the data routing operations are limited to point-to-point communication between adjacent neighbors, the long network diameter of the linear array network does not represent a problem even for long frame lengths.

4.1.2 Memory Requirements

In addition to PE distribution it is also necessary to determine PE local memory requirements for storage of the parallel variables involved in the decoding algorithm. Figure 4.4 displays a high-level organization of the parallel variables within the local memory for an individual PE.

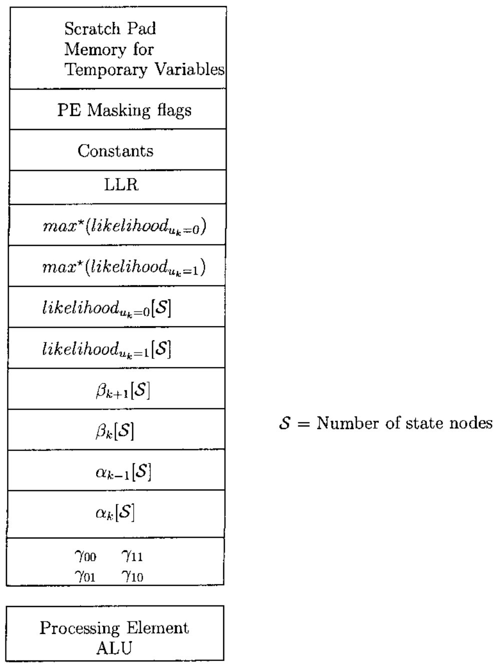

Scratch Pad Memory for Temporary Variables
PE Masking flags
Constants
LLR
$max^*(likelihood_{u_k=0})$
$max^*(likelihood_{u_k=1})$
$likelihood_{u_k=0}[S]$
$likelihood_{u_k=1}[S]$
$\beta_{k+1}[S]$
$\beta_k[S]$
$\alpha_{k-1}[S]$
$\alpha_k[S]$
γ_{00} γ_{11} γ_{01} γ_{10}
Processing Element ALU
S = Number of state nodes

Figure 4.4: Parallel variable distribution per PE.

It can be observed that each PE needs access within its local memory to the branch metrics, and α and β state metrics for the corresponding trellis section. Each PE also requires the normalized α state metrics from the preceding trellis section and normalized β state metrics from the succeeding trellis section, which are used in the calculation of new values for the state metrics as per the algorithm described in the next section. Additional storage is needed for the log-likelihood values for $u_k = 0$ and $u_k = 1$, and for the LLR. Finally, storage is required for constants and flags used for masking of PEs as well as a scratch pad memory region for temporary variables.

4.2 Parallel Window ONE Algorithm

The parallel processor implementation assumes that the number of PEs available in the array processor is at least equal to the number of information bits in the data frame. It is also assumed that it is possible to wait for the entire frame to be received before decoding it. One preliminary step is taken prior to parallel decoding: a short sequential backward recursion, as described in section 3.3.3, is performed to compute an initial β distribution, since it is unnecessary for the tail bits to be included in the parallel decoding. The α distributions for all trellis sections except the first one are initialized as equiprobable; the first trellis section is initialized as per equation (2.28). A similar initialization is performed for the β distributions of all trellis section except the last section which corresponds to the last information bit; this section is initialized with the β values computed by the short sequential backward recursion of the tail bits. This initialization method is similar to the one employed by the sliding window technique [38] for initializing the backward recursion of the individual windows due to lack of information about the distribution of the β state metrics at the end of each window.

Prior to calculating the α and β state metrics, the γ branch metrics are computed in parallel for all trellis sections since it is assumed that the received channel symbols have been distributed among all the PEs in the proper order. The branch metrics γ are calculated using the simplified method as per equation (2.24).

New α and β state metric are calculated as per equations (2.25) and (2.26). These calculations happen in lockstep for all trellis sections. Once the new values are obtained, the α_{k-1} and β_{k+1} are updated in the neighboring PEs. For the case of the α metrics, each PE updates the α_{k-1} values of its immediate neighbor to the right; and for the case of the β metrics, each PE updates the β_{k+1} values of its immediate neighbor to the left. The procedure of calculating new values for the α and β state metrics is repeated using the updated α_{k-1} and β_{k+1} values respectively. It is important to note that the first trellis section and the last trellis section continue using the initialization values for α_{k-1} and β_{k+1} respectively.

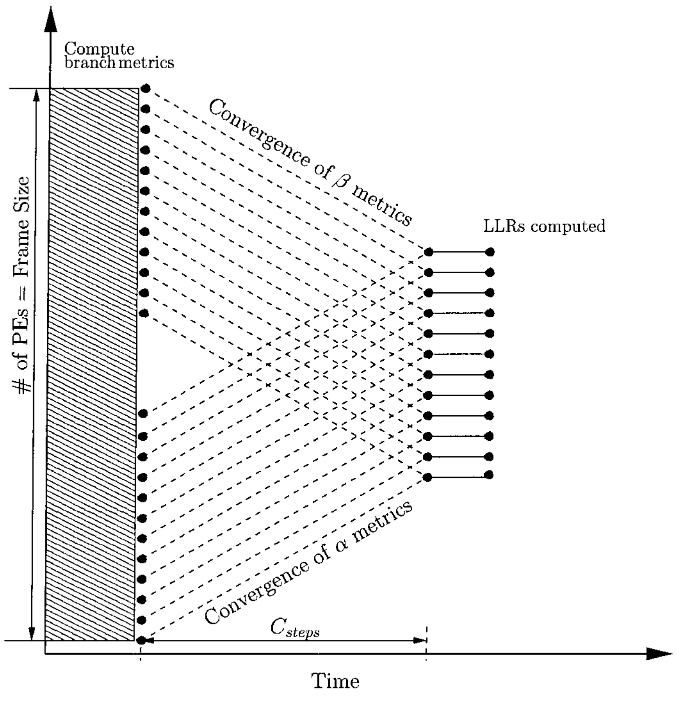


Figure 4.5: Parallel Decoding Method with respect to Time.

The objective is to repeat the parallel evaluation of the α and β values until these values converge to the same value as those calculated using the sequential forward-backward algorithm. Each parallel computation of state metrics will be denoted as a *convergence step*. Therefore, the parallel approach effectively replaces the forward and backward recursions with a predefined number of convergence steps. The number of convergence steps that should be used is considered in the next subsection.

Once the state metrics have been computed, the LLR values for the information bits can be calculated in parallel for all trellis sections because the branch and state metrics already reside in the distributed memory of the SIMD array processor. The diagram of Figure 4.5 depicts the operation of the parallel decoding method with respect to time for an example frame size of 12 data bits.

As discussed above, in this parallel processor implementation of the SISO decoder, one trellis section is allocated per PE. This configuration is equivalent to having multiple windows with window length $W = 1$ where the decoder operates on all the windows in parallel. It is then appropriate to label this new decoding technique *The Parallel Window ONE* or PW-ONE algorithm.

4.2.1 Convergence of α and β State Metrics

Earlier work describing the parallelism in trellis decoding and methods to exploit this parallelism are found in [39] and [40].

The equiprobable β distribution initialization used in the sliding window decoding technique requires that each window includes a 'stabilization' length L such that if the window length is W, the decoder only calculates LLRs values for $D = W - L$ trellis sections. Literature on this topic suggests that the backward recursion using the sliding window technique closely approximates the exact β distribution when the stabilization length L is five to six times the constraint length of the code [22].

The number of convergence steps used in the PW-ONE algorithm perform a similar function as the stabilization region of the sliding window algorithm. The advantage of the parallel algorithm is that upon completion of the predefined number of convergence steps, the α and β distributions will be close to their exact value for all columns along the trellis.

Based on this result, the following hypothesis can be formulated for the PW-ONE algorithm: the number of convergence steps required for the parallel algorithm to compute α and β distributions that closely approximate the exact distributions is five to six times the constraint length of the code. Simulation results that confirm this hypothesis for some code configurations, but belie it for others are presented in the next chapter.

4.3 Parallelism and Efficiency

Two parameters that are commonly used in parallel computing for evaluation of algorithm performance are the *speedup factor* and the *system efficiency*. The speedup factor, or simply speedup, refers to how much faster a parallel algorithm is than a corresponding sequential algorithm [41]. Speedup is defined by the following equation

$$\mathcal{S}_p = \frac{T_1}{T_p}, \tag{4.2}$$

where

p is the number of processors.

T_1 is the execution time of the sequential algorithm.

T_p is the execution time of the parallel algorithm with p processors.

When $\mathcal{S}_p = p$, it is said that the parallel algorithm achieves linear speedup. In this case, if the number of processors is doubled, then the processing speed is doubled.

Efficiency is a performance metric that indicates the actual degree of speedup performance achieved as compared with the maximum value [10]. The system efficiency of a p-processor system is defined by

$$\mathcal{E}_p = \frac{\mathcal{S}_p}{p} = \frac{T_1}{pT_p} \tag{4.3}$$

Given that $1 \leq \mathcal{S}_p \leq p$, then the value of efficiency lies in the range $1/p \leq \mathcal{E}_p \leq 1$.

For the case of the PW-ONE algorithm for a constituent decoder, one can attempt to express the speedup factor and efficiency in terms of the dominant operations in the algorithm rather than in terms of absolute execution time measurements. In both the sequential and parallel SISO decoding algorithms, the dominant operations have been identified to be: (a) addition[1], (b) *max**, and (c) max. Expressions for speedup and efficiency are derived assuming that all three operations are monolithic, that is, implementation details of *max** and max are not considered. Therefore, one does not need to be concerned with the number of native processor

[1]In this thesis, additions and subtractions are grouped together under the same category and simply labeled as addition.

instructions and their execution times. It is reasonably accurate to assume that all operations require equal execution time.

The following notation is used to derive the expressions for speedup and efficiency:

G is the number of state nodes in a column of the trellis.

N is the frame length.

C_{steps} is the number of convergence steps in the PW-ONE algorithm.

Note that the PW-ONE algorithm assumes that the number of PEs available is at least equal to the frame length, therefore, $p = N$.

The numbers of operations performed by the sequential forward/backward algorithm for each column of the trellis are detailed on Table 4.1. The operation for calculation of the branch metrics is included as part of both the forward and backward recursion and consists of a single addition as per equation (2.24). Since the

Variable Calculated	additions	*max**	max	Total Operations
α state metrics	$3\times G+1$	G	G - 1	$5\times G$
β state metrics	$3\times G+1$	G	G - 1	$5\times G$
LLRs	$4\times G+1$	$2\times(G$ - 1)	0	$6\times G$ - 1
				$16\times G$ - 1

Table 4.1: Operations per Trellis Column in Sequential Log-MAP Algorithm.

operations per trellis column are repeated N times to decode one frame, one can express the execution time for the sequential algorithm in terms of the dominant operations as follows:

$$T_1 = (16 \times G - 1) \times N. \tag{4.4}$$

In the PW-ONE algorithm the state nodes of a trellis column are also scanned sequentially because an entire column is allocated to each PE of the SIMD array processor. Therefore, the PW-ONE algorithm requires the same number of operations to calculate the α and β state metrics as in the forward/backward approach. These operations are repeated a number of times equal to the predefined number of convergence steps C_{steps}. However, the operations for the calculation of branch metrics are not included this time because branch metrics are computed prior to starting the convergence procedure. Branch metrics are calculated by all PEs in parallel and, therefore, computing branch metrics counts as a single operation. Similarly, the number of operations to calculate the LLR values is the same as in the forward/backward algorithm. The difference here is that the LLRs are computed after the convergence procedure, and with all the PEs operating in parallel, the LLRs for the entire frame are generated at the same time without the need for repeating the calculations. Table 4.2 details the number of operations in the PW-ONE algorithm.

Variable Calculated	Total Operations
branch metrics	1
α state metrics	$(5\times G - 1)\times C_{steps}$
β state metrics	$(5\times G - 1)\times C_{steps}$
LLRs	$6\times G - 1$
	$\mathbf{(10\times G - 2)\times C_{steps} + 6\times G}$

Table 4.2: Operations per PE in PW-ONE Algorithm.

The execution time for the parallel algorithm with $p = N$ processors can therefore be expressed as:

$$T_p = (10\times G-2)\times C_{steps} + 6\times G. \tag{4.5}$$

Using equations (4.4) and (4.5), the expressions for speedup and efficiency when evaluating the PW-ONE algorithm against the sequential constant-Log-MAP algorithm can be stated as:

$$S_p = \frac{T_1}{T_p} = \frac{(16\times G-1)\times N}{(10\times G-2)\times C_{steps} + (6\times G)} \tag{4.6}$$

$$\mathcal{E}_p = \frac{S_p}{N} = \frac{(16\times G-1)}{(10\times G-2)\times C_{steps} + (6\times G)} \tag{4.7}$$

It is then possible to calculate values for the speedup factor and efficiency based on the expressions of equations (4.6) and (4.7) respectively. Substituting in place of the variable parameters values that correspond to the UMTS turbo code and the PW-ONE implementation evaluated in section 5.4 such that:

- Frame Length $N = 1024$

- Constraint Length $K_c = 4$
- Number of state nodes per trellis column $G = 2^{K_c-1} = 8$
- Number of convergence steps $C_{steps} = 6 \times K_c = 24$,

one obtains the following speedup and efficiency results:

$$\mathcal{S}_p(\mathrm{PW}-\mathrm{ONE}) \approx 67.7 \tag{4.8}$$

$$\mathcal{E}_p(\mathrm{PW}-\mathrm{ONE}) \approx 0.066 \tag{4.9}$$

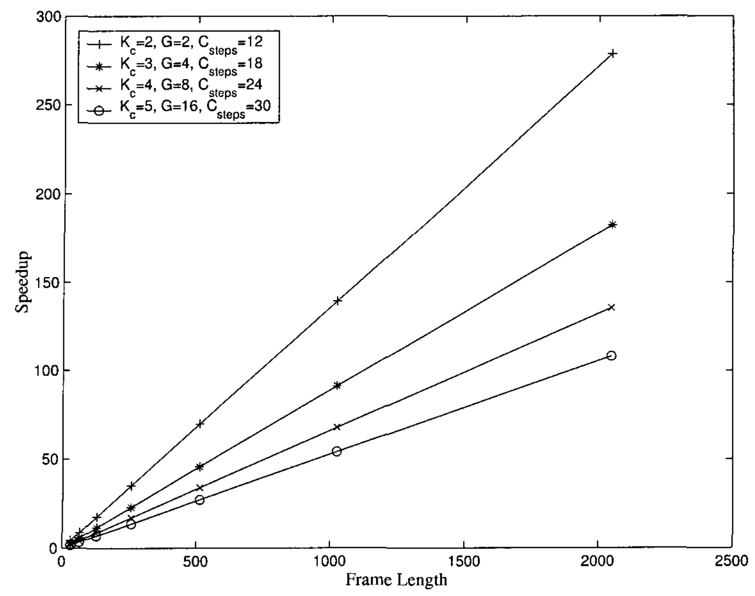
Note that the computed speedup factor is a hypothetical speedup for a uni-processor versus N identical processor, neglecting inter-processor communication. For the speedup of a real SIMD array processor, relative to a real embedded processor, refer to section 5.5.

The low value of efficiency $\mathcal{E}_p(\mathrm{PW}-\mathrm{ONE})$ implies that it is better to use the sequential implementation of the algorithm when the goal is to reduce the number of operations. However, the parallel algorithm is the better choice for reducing the latency of computing the end results at the expense of more hardware.

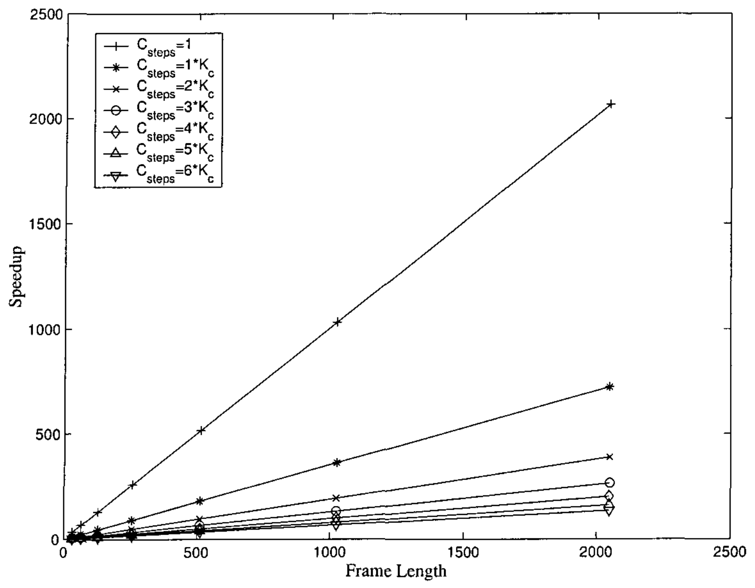
It is important to note that when using a SIMD array processor with PIM technology, the incremental hardware cost is low when one considers the gains in execution time performance and energy consumption performance.

The plots of Figure 4.6 show how the speedup factor is affected when code parameters are changed. Realizing that the number of state nodes G and the number of convergence steps C_{steps} are determined by the constraint length K_c of the constituent code, one can observe in Figure 4.6(a) the impact on speedup for constraint length values in the range $2 \leq K_c \leq 5$, and with the number of convergence steps set to $C_{steps} = 6 \times K_c$ for all cases. Figure 4.6(b) displays the effect of varying the number of convergence steps for a given constraint length, in this case $K_c = 4$. The plot of Figure 4.7 portrays the efficiency of the data-parallel algorithm when the parameters of equation (4.7) take on different values.

Based on these results, the main observations are:



(a) Impact of Constraint Length on Speedup



(b) Impact of Convergence Steps on Speedup

Figure 4.6: Effect of Code Parameters on Speedup.

- Speedup achieved by the PW-ONE algorithm is proportional to the frame length N, and inversely proportional to the number of convergence steps C_{steps}.

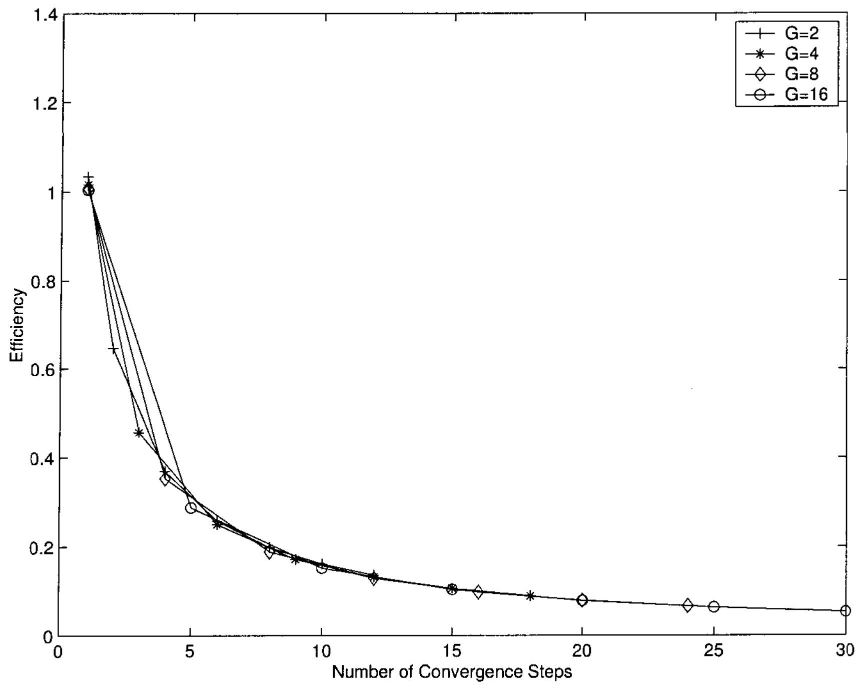


Figure 4.7: Efficiency for different Code Parameters.

- To maintain equivalent coding gain when compared to the sequential algorithm, however, C_{steps} needs to increase from $6 \times K_c$ to $7 \times K_c$ as the frame size increases to greater than 512 data symbols (see section 5.4).
- The efficiency of the PW-ONE algorithm is determined by the number of convergence steps for a given constituent code.

4.4 Summary

After careful analysis of the trellis structure that describes RSC constituent codes used in turbo codes, it was determined that a flexible, efficient and relatively inexpensive SIMD array processor that satisfies the requirements for parallel decoding of an RSC code is one where the PEs are organized as a linear array. The storage requirements for the individual PEs was presented.

The concept of the convergence step was introduced as the basis for the computation of the state metrics in the newly defined Parallel Window ONE algorithm. In this algorithm, the α and β metrics are evaluated in parallel, and after a predefined number of convergence steps, converge to the true metric values. This algorithm will be used as the decoding algorithm in the constituent decoders for the parallel processor implementation of the turbo decoder discussed in the next chapter.

Chapter 5

Parallel Processor Decoder Implementation

This chapter presents the implementation details of the data-parallel turbo decoding algorithm for execution by the SIMD array processor in the J2210 Media processor. It also describes the experimental methodology for evaluation of the algorithm and summarizes the results obtained from this evaluation.

The chapter begins by describing the interleaver and explaining why it is implemented as set of functions in sequential software. Section 5.2 describes a data communication topology that results in reduced inter-processor communication overhead when using the inter-processor communication network of the Array Processor in the J2210. It also presents the pseudo-code for the implementation of the PW-ONE algorithm. Scalability of the algorithm when a greater number of PEs is available, and the impact of the number of convergence steps on the BER performance of the turbo decoder, are the primary topics addressed in Section 5.4. Section 5.5 presents results related to execution time of the sequential and data parallel algorithms and highlights the speedup factor achieved by this implementation of the PW-ONE algorithm. The chapter concludes by considering the low energy consumption advantage offered by the PIM technology of the Array Processor.

5.1 Sequential Interleaver

As stated in Chapter 3, the turbo code selected for development and verification of the turbo decoder addressed in this thesis is the UMTS turbo code. Consequently, the interleaver/de-interleaver implemented for use by the turbo decoder is the one described in the UMTS specification.

Early in the development it was decided to implement the UMTS prime interleaver in sequential software to be executed by the ARM922T$^{\text{TM}}$ RISC engine in the J2210 Media processor. The inter-processor communication network topology of the Array Processor in the J2210 does not provide the flexibility to implement the interleaver efficiently, resulting in a large number of data transfers between PEs to achieve the required permutation of the data. Therefore, a parallel processor implementation of the interleaver may not offer any speedup advantage over its sequential implementation. Also, the limited number of CUs restricts the size of the interleaver.

The UMTS prime interleaver can be visualized as a matrix with 5, 10, or 20 rows and between 8 and 256 columns (inclusive), depending on the frame length. The data is written into the interleaver in a row-wise fashion, with the first data bit placed in the upper-left position of the matrix. Permutation of the data proceeds as follows: first, each row of the matrix undergoes intra-row permutations in accordance with the algorithm described in [2]; next, inter-row permutations are performed to change the ordering of rows without changing the order of the elements within each row. After the intra-row and inter-row permutations, data is read from the interleaver in a column-wise fashion.

The algorithm for the generation of the interleaver has been implemented as a C++ module. This allows for a flexible interleaver design that can support the range of data frame sizes stated in the UMTS specification. Memory for the interleaver is allocated dynamically as part of the initialization procedure. The System Memory block of the J2210 Media processor is used for storing the interleaver indexes. The System Memory is a 32 KByte block of fast static RAM that is connected to the

Advanced High-performance Bus (AHB) of the ARM processor for fast data transfers. Two functions are provided to the programmer to perform interleaving and de-interleaving of data. The interleaving function simply uses the indexes provided by the interleaver generator to re-arrange the data elements of the input array and stores them in the permuted order in the output array. Similarly, the de-interleaving function uses the indexes to reverse the order of the permutation. Using an embedded memory block helps to speed up execution time of the interleaving and de-interleaving of data. The source code for the implementation of the UMTS interleaver generator is included in section A.2 of Appendix A.

5.2 Implementation using the J2210 Array Processor

This section presents implementation details for execution of the Parallel Window ONE algorithm on the Array Processor of the J2210 Media processor. These details include a description of the inter-processor communication topology that results in reduced data communication overhead, and the pseudo-code for the three main components of the algorithm.

5.2.1 Inter-Processor Communication Topology

The first step to implement the PW-ONE algorithm to target the Array Processor in the J2210 SoC is to select a topology for the inter-processor communications that resembles the linear array network assumed by the PW-ONE algorithm. Recall that the CUs of the Array Processor are organized as a 4 x 24 2-D mesh, and the inter-processor communication network enables each CU to obtain data from its eight nearest neighbors as shown in Figure 5.1. This results in the additional benefit that the instruction set of the Array Processor supports diagonal data transfers. The rest of this chapter will continue to use the notation adopted in [19] to specify directions (NW,N,NE,E,W,SW,S,SE) of inter-processor communication. A key factor to keep in mind in the selection of the inter-processor communications topology with the communication network present in the Array Processor of the J2210 is that it does not provide the wrap-around feature available in other 2-D meshes such as the

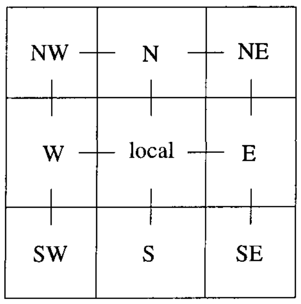


Figure 5.1: A CU and its Nearest Neighbors.

Spiral, Torus and Toroid of Figure 2.2. The two network topologies that have been considered are:

West-East Data Transfer

Use of this topology results in data transfer between PEs in the W $\longrightarrow$ E direction during the computation and convergence of α state metrics, and in the W $\longleftarrow$ E during the computation and convergence of the β state metrics. Handling an *edge condition* here refers to transferring data between CUs at opposite edges of two adjacent rows.

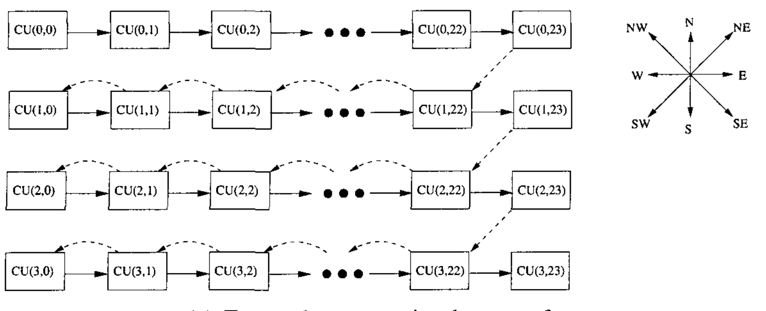
North-South Data Transfer

Use of this topology results in data transfer between PEs in the N $\longrightarrow$ S direction during the computation and convergence of α state metrics, and in the N $\longleftarrow$ S during the computation and convergence of the β state metrics. Handling an *edge condition* here refers to transferring data between CUs at opposite edges of two adjacent columns.

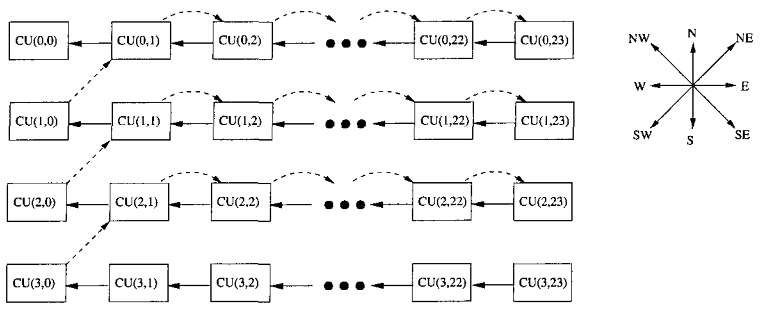
Inter-processor data communication using the West-East Data Transfer topology is shown in Figure 5.2, where the solid arrows indicate the data communication pattern that is normally required[1] while the dashed arrows indicate how the edge condition is handled. These two data communication patterns do not occur in parallel but in a sequential manner. Additional storage per CU is required for a buffer

[1]This would be the only data communication pattern needed if wrap-around with offset between adjacent rows were available.

that holds the data being transferred, plus masking[2] of CUs is imperative to prevent data from being corrupted.



(a) Forward state metrics data transfer



(b) Backward state metrics data transfer

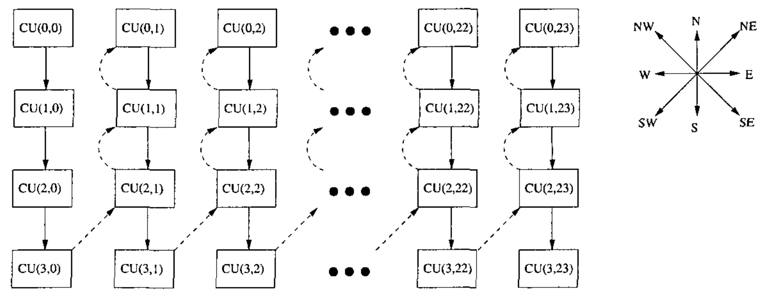
Figure 5.2: West-East Data Transfer inter-PE Communication Pattern.

In contrast, Figure 5.3 displays the inter-processor data communication pattern using the North-South Transfer topology. Again, dashed arrows refer to how the data is transferred between CUs at opposite edges of two adjacent columns, and solid arrows show the data communication pattern with all CUs enabled[3] (the only communication pattern needed if wrap-around with offset between adjacent columns were available).

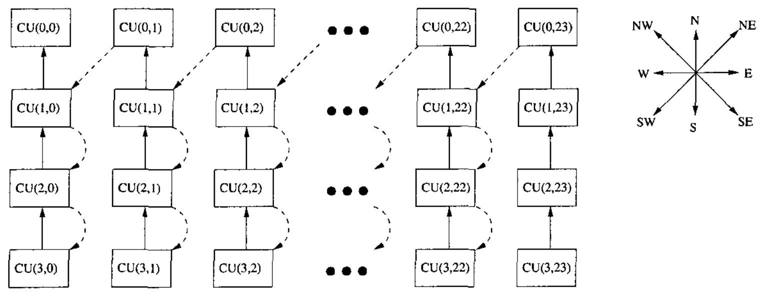
Note that only one CU is enabled per row during each data hop between CUs when handling the edge condition in the West-East Data Transfer topology. Special consideration is given to the outer-most rows of the Array Processor. Similarly only one CU is enabled per column during each data hop between CUs for the case of the North-South Data Transfer topology, and the outer-most columns of the Array

[2]Masking a CU is achieved by clearing the contents of the Write Enable (WE) register, thereby disabling the ability of the CU to write to its local memory.

[3]In this context, a CU being enabled refers to it being enabled to write to its local memory.



(a) Forward state metrics data transfer



(b) Backward state metrics data transfer

Figure 5.3: North-South Data Transfer inter-PE Communication Pattern.

Processor are considered special cases. Therefore, handling edge conditions can be expensive in terms of inter-processor communication overhead.

For the case of an asymmetrical 2-D mesh of CUs such as the one in the Array Processor of the J2210, the choice of the North-South Data Transfer topology offers an advantage in terms of reduced inter-processor communication overhead because only three data hops are required to transfer data between edges of adjacent columns as compared to twenty three hops to transfer data between edges of adjacent rows if the West-East Data Transfer topology is used. The main drawback of the North-

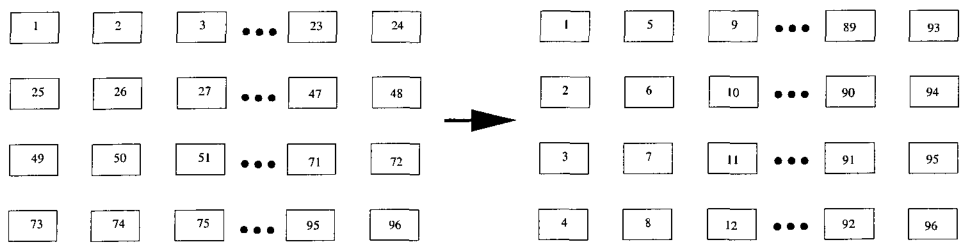


Figure 5.4: Row-wise to Column-wise re-ordering of data.

South Data Transfer topology is that the elements of the data arrays must be rearranged so that they exhibit a sequential column-wise order before they are written to the distributed memory of the Array Processor. The reverse is true when reading data from the Array Processor memory. Figure 5.4 displays the change in order for the CU arrangement in the Array Processor.

Section 5.5 compares the execution time of the emulated spiral topology (Torus with offset) of Figure 5.5 against the execution time of the West-East and North-South Data Transfer topologies when evaluating the PW-ONE algorithm with different convergence step configurations.

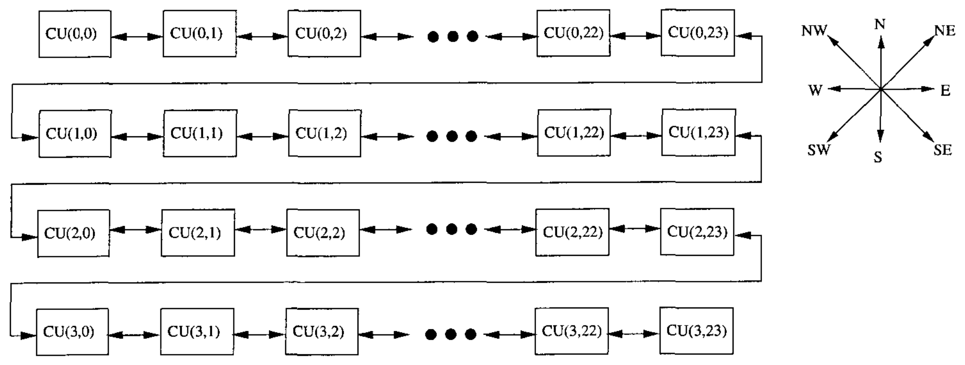


Figure 5.5: Emulation of Linear Array network using a 2-D Mesh.

5.2.2 Description of Algorithm Implementation

The implementation of the parallel processor SISO constituent decoder using the PW-ONE algorithm can be divided into three main components. These components are:

1. Calculation of branch metrics.
2. Calculation and convergence of α and β state metrics.
3. Calculation of LLR values.

The use of pseudo-code is preferred when describing the different components of the algorithm and helps to avoid the intricate details of SEL programming. The notation used in the pseudo-code assumes that there are as many PEs as data elements, and the block **for all** k **in parallel do** $\cdots$ **end for** causes all PEs to execute the statements inside the block in synchrony.

Branch Metric Computation

The computation of the branch metrics requires that the systematic (data) and parity channel observations are distributed among all the CUs of the Array Processor. Recall that for a rate 1/2 RSC constituent code there are four possible branch metrics. These branch metrics will be denoted as $\gamma_k^{(00)}$, $\gamma_k^{(01)}$, $\gamma_k^{(10)}$ and $\gamma_k^{(11)}$, where the ij superscript is such that i refers to the value of systematic bit c^s of the coded symbol, and j refers to the value of the corresponding parity bit c^p. The branch metrics are then calculated in parallel as follows:

```
for all k in parallel do
    $\gamma_k^{(00)} = 0$
    $\gamma_k^{(10)} =$ data channel symbol
    $\gamma_k^{(01)} =$ parity channel symbol
    $\gamma_k^{(11)} = \gamma_k^{(10)} + \gamma_k^{(01)}$
end for
```

Notice that since $\gamma_{00} = 0$ always, it is possible to omit it from all operations that require it. It is assumed that overflow checking is implied in all arithmetic operations.

Computation of α and β State Metrics

The component of the PW-ONE algorithm responsible for calculation of the state metrics uses the convergence step concept. Inter-processor communication also takes place during these computations. For the implementation targeting the J2210 Array Processor, the North-South Data Transfer topology described in the previous section is used.

Each convergence step consists of two half cycles. The first half cycle corresponds to the computation of the α state metrics. The state nodes corresponding to each processing element (also referred to as computational unit CU) are scanned sequentially using the branch metrics to calculate new values for the α metrics. The following pseudo-code describes the operations performed during the first half cycle of a convergence step.

```
for all $k$ in parallel do
    for Number of Convergence Steps do
            // First half of convergence step: Calculate $\alpha$ state metrics
        for i = 0 to $G$-1 do
            $\alpha_k^1[i] = \gamma_k^{(1x)} + \bar{\alpha}_k^1[i]$;        // branch with data bit $u_k = 1$
            $\alpha_k^0[i] = \gamma_k^{(0x)} + \bar{\alpha}_k^0[i]$;        // branch with data bit $u_k = 0$
            $\alpha_k[i] = max^\star(\alpha_k^1[i], \alpha_k^0[i])$;
        end for
            // Find maximum out of $\alpha_k$ values
        $\alpha_k^{max} = \max(\alpha_k[0], \ldots, \alpha_k[G-1])$;

            // Normalized $\alpha_k$ values are used to update $\bar{\alpha}_k$ values
        for i = 0 to $G$-1 do
            $\bar{\alpha}_k[i] = \alpha_k[i].N - \alpha_k^{max}.N$;
        end for

            // Handle edge condition
        Mask CUs not needing to update their local memory
        as edge condition proceeds;
        One NW shift moves data to adjacent column;
        2 vertical shifts move data up the column to top row;
        Update $\bar{\alpha}_k$ values of CUs in the top row;

            // Restore $\bar{\alpha}_k$ values of CU at top-left corner of grid
            // to initialization values
```

The pseudo-code uses G to denote the number of state nodes per trellis column. Inter-processor data communication is observed during the calculation and storage of the normalized $\bar{\alpha}$ metrics. The updated values for $\bar{\alpha}$ are saved to be used during the next convergence step. $\bar{\alpha}$ corresponds to α_{k-1} of the sequential algorithm. Handling the edge condition, that is, moving data from one edge of a column in the Array Processor to the opposite edge in the adjacent column involves multiple point-to-point shifts. The pseudo-code omits details of loading and storing of data during each point-to-point shift, and only presents a very simple description of the steps involved in the procedure.

The second half cycle of each convergence step focuses on computation of the β state metrics. The procedure is identical to the one used in the calculation of the α metrics with the exception that the data flow is in the opposite direction. The following pseudo-code insert displays the operations and other details that are required during the second half cycle.

```
            // Second half of convergence step: Calculate $\beta$ state metrics
        **for** i = 0 **to** G-1 **do**
            $\beta_k^1[i] = \gamma_k(1x) + \bar{\beta}_k^1[i]$;      // branch with data bit $u_k = 1$
            $\beta_k^0[i] = \gamma_k(0x) + \bar{\beta}_k^0[i]$;      // branch with data bit $u_k = 0$
            $\beta_k[i] = max^*(\beta_k^1[i], \beta_k^0[i])$;
        **end for**
            // Find maximum out of $\beta_k$ values
        $\beta_k^{max} = \max(\beta_k[0], \ldots, \beta_k[G-1])$;

            // Normalized $\beta_k$ values are used to update $\bar{\beta}_k$ values
        **for** i = 0 **to** G-1 **do**
            $\bar{\beta}_k[i] = \beta_k[i].S - \beta_k^{max}.S$;
        **end for**

            // Handle edge condition
        **Mask CUs not needing to update their local memory**
        **as edge condition proceeds;**
        **One SE shift moves data to adjacent column;**
        **2 vertical shifts move data down the column to bottom row;**
        **Update $\bar{\beta}_k$ values of CUs in the bottom row;**

            // Restore $\bar{\beta}_k$ values of CU at bottom-right corner of grid
            // to initialization values
    **end for**
**end for**
```

LLR Calculation

The third and last component of the PW-ONE algorithm implementation is the computation of the LLR values. At this point in the algorithm, it is assumed that the α and β metrics have converged to their exact values. Since the values for the branch metrics and for both state metrics are already present in the local memory of the corresponding PE, calculation of the log-likelihood values for source data bits $u_k = 1$ and $u_k = 0$ proceeds synchronously for all PEs. There is a small element of inter-processor data communication in this component of the algorithm which involves transferring the values of the α metrics to the PEs holding the corresponding γ and β metrics in order to compute the correct log-likelihood values. Details of this last section of the algorithm are presented in the following pseudo-code.

```
for all k in parallel do
    Move $\alpha_k$ values to neighbor PE to the South
    and handle edge condition;

    for i = 0 to G-1 do
        $\Lambda_k^1[i] = \alpha_k[i] + \gamma_k^{(1x)} + \bar{\beta}_k[i]$;
        $\Lambda_k^0[i] = \alpha_k[i] + \gamma_k^{(0x)} + \bar{\beta}_k[i]$;
    end for

        // Compute max* value for $\Lambda_k^1$ and $\Lambda_k^0$
    $\Lambda_k^1 = \Lambda_k^1[0]$;
    $\Lambda_k^0 = \Lambda_k^0[0]$;
    for i = 1 to G-1 do
        $\Lambda_k^1 = max^\star(\Lambda_k^1[i], \Lambda_k^1)$;
        $\Lambda_k^0 = max^\star(\Lambda_k^0[i], \Lambda_k^0)$;
    end for

        // Calculate the LLR
    LLR $= \Lambda_k^1 - \Lambda_k^0$;
end for
```

5.3 Communication with the Host Processor

The ARM922T^{TM} RISC engine in the J2210 acts as the host or master processor. In this application, communication with the host processor happens at two levels: (a) sending commands to the Array Controller, and (b) writing to and reading from the memory of the Array Processor[4]. Sending commands to the Array Controller is accomplished by using the set of *ACSendCommand* x( ) Applications Programming Interface (API) function calls. The host processor is responsible for writing the simulated systematic and parity channel observations to the memory of the Array Processor, sending the command to execute the parallel algorithm and then reading the LLRs values from the memory of the Array Processor. Reading from and writing to the Array Processor memory is accomplished by using the *ReadMem32()* and *WriteMem32()* API function calls respectively.

The ARM922T™ performs 32-bit word (4 bytes) accesses to and from the Array Processor memory using the above mentioned API calls. The 32-bit word access is seen as a 4-byte packet that get distributed across four horizontally adjacent CUs. When reading data from the Array Processor memory, the 4-byte packet is unpacked in software if necessary. Direct Memory Access (DMA) channels are available to support higher rate data transfer such as storing multiple bytes per CU. However, it was determined that the single word accesses provided by the *ReadMem32()* and *WriteMem32()* functions satisfied the data rate requirements of the data-parallel turbo decoder implementation due to the relatively small amount of data written to and read from the Array Processor memory during the decoding of each frame.

Descriptions of and examples on how to use the API function calls are provided in the J2210 Software Tools User's Manual [19].

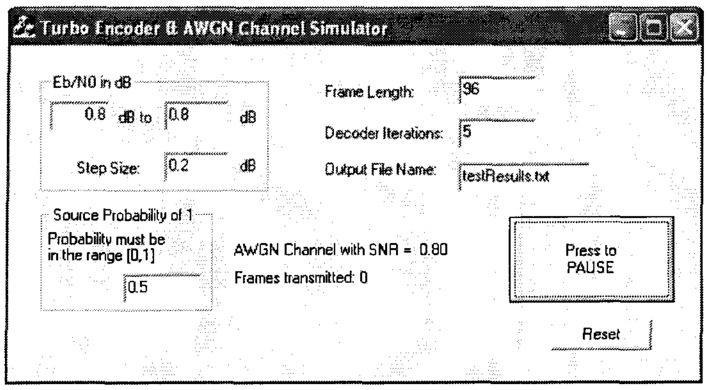
5.4 BER Performance Evaluation

5.4.1 Simulator and Experimental Method

Verification of the data-parallel turbo decoder requires properly encoded data with additive noise that can serve as input vectors to the decoder. A simulator was designed as a visual application to run on a desktop computer for the purpose of generating the test data for the turbo decoder implemented in the embedded target.

[4]Atsana's documentation uses the term Computational Memory (CMEM) when referring to the Array Processor.

Figure 5.6 shows a picture of the user interface of the turbo encoder/channel model simulator.



Eb/N0 in dB
0.8 dB to 0.8 dB
Step Size: 0.2 dB
Source Probability of 1
Probability must be in the range [0,1]
0.5
Frame Length: 96
Decoder Iterations: 5
Output File Name: testResults.txt
AWGN Channel with SNR = 0.80
Frames transmitted: 0

Figure 5.6: Turbo Encoder and AWGN Channel Simulator GUI.

The simulator generates random binary data in packets with a frame length specified by the user. Turbo encoding is applied to the data frame using the UMTS turbo encoder. The simulator then adds noise to the encoded data frame assuming an AWGN channel model. Demodulation of the data is assumed to be perfect soft-decision. The data with additive noise is then sent through the desktop PC serial port to the J2210 CEB for decoding. The turbo decoder application executed by the J2210 generates the LLR values for the information bits and sends that data back to the simulator application on the desktop PC. The simulator then performs the hard decision of the received bits and compares it against the value of the original binary data. At least one thousand bit errors for SNR $\leq$ 1.0dB, or one hundred bit errors for SNR $>$ 1.0dB are recorded to obtain a good estimate of the BER for the given conditions. BER performance results are compared against the sequential, fixed-point implementation of the turbo decoding algorithm.

5.4.2 Scalability of Parallel Algorithm

There was interest in verifying the scalability of the PW-ONE algorithm when more PEs are available. Rather than verifying the BER performance of the turbo decoder for the short frame size of 96 bits over a range of SNR levels, it was decided to evaluate the BER performance when the frame length of the data packet is varied while keeping a fixed SNR value.

The SNR of $E_b/N_0 = 0.8$dB was selected to conduct the evaluation. This choice of SNR level has no particular significance, but examination of the results presented in Chapter 3 reveal that the *turbo cliff* region of the UMTS turbo code is observed in the range $0.6\text{dB} \leq E_b/N_0 \leq 1.2\text{dB}$.

Frames with lengths greater than the number of PEs available in the Array Processor are decoded by applying a modified version of the sliding window technique. The received frame is divided into windows of length $W = 96$, and each window is decoded using the PW-ONE algorithm. The following conditions are used in the application of the modified sliding window technique in order that the results of the decoding are exactly equal to what would be obtained with operation of the PW-ONE algorithm with $p = N$ PEs: (a) there is no information contributed from one window to the next for calculating α metrics, and the same holds true for the computation of β metrics, (b) the overlap between windows is equal to twice the number of convergence steps predefined in the PW-ONE algorithm. Note that this modified algorithm is not as efficient (in terms of total number of computations) as a true sliding window algorithm; it was instead designed in this fashion to accurately emulate the PW-ONE algorithm.

Scalability of the algorithm for variable frame lengths was also verified by using three different values for the convergence steps parameter. The number of convergence steps that were used in the constituent SISO decoder were: $(5 \times K_c)$, $(6 \times K_c)$ and $(7 \times K_c)$, where K_c is the constraint length of the rate 1/2 RSC constituent code used in the turbo code. These particular choices for the number of convergence steps were made so as to test the hypothesis of the PW-ONE algorithm made in section 4.2 of Chapter 4, which was that the number of convergence steps required for the parallel algorithm to compute α and β distributions that closely approximate the exact distributions would be five to six times the constraint length of the code. Consequently a value for the number of convergence steps less than $(5 \times K_c)$ was not considered.

Test results for four, seven and ten turbo decoder iterations are displayed in Figures 5.7, 5.8 and 5.9 respectively. These test results indicate a significant degradation in performance when the number of convergence steps is set to $(5 \times K_c)$. This degradation is more pronounced when the length of the frame increases.

The measured results presented here agree with simulation results for a frame size $N = 1024$ bits and SNR level of $E_b/N_0 = 0.8$dB presented in section 3.5.

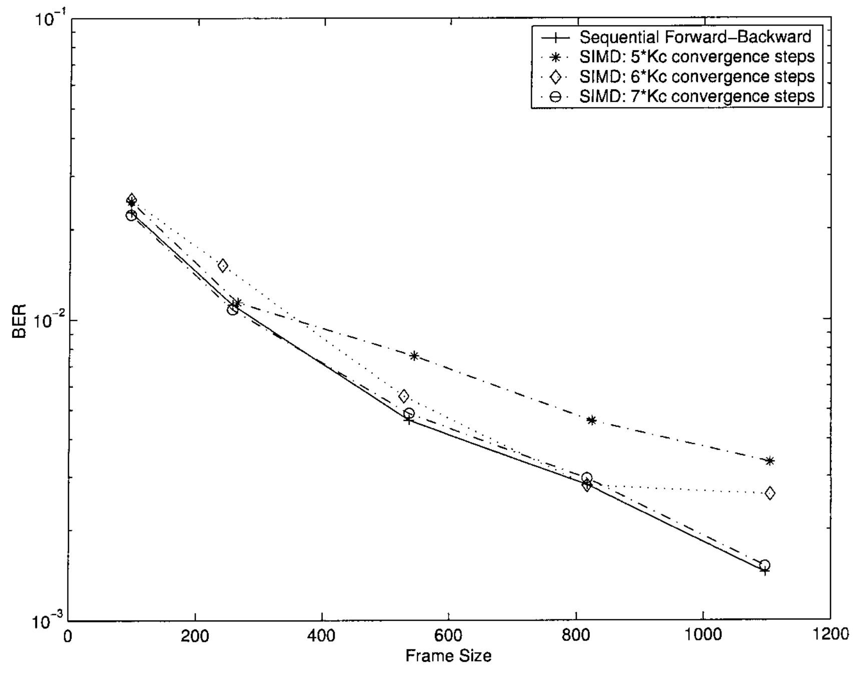


Figure 5.7: BER performance vs. Frame Length, 4 Turbo Iterations.

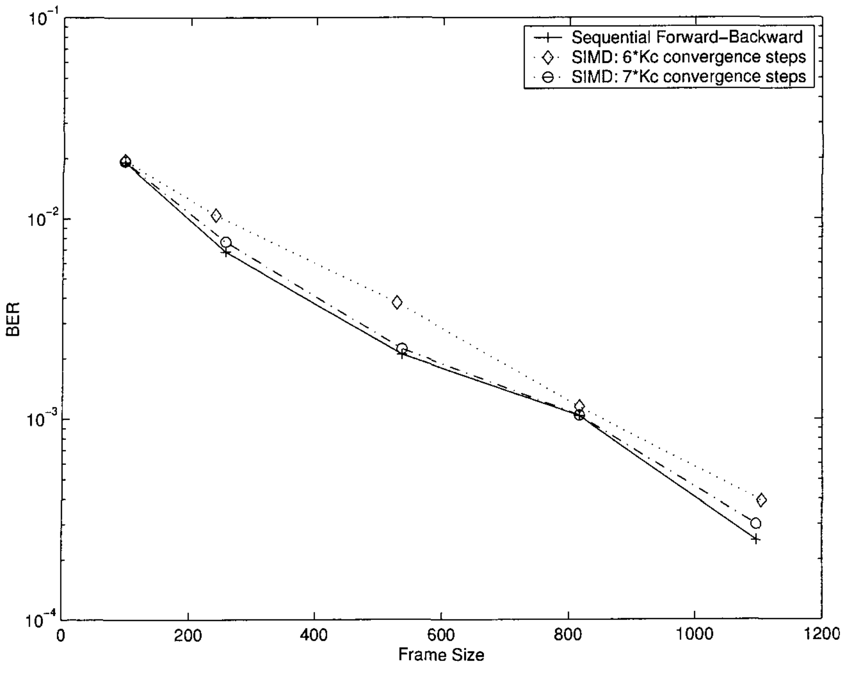


Figure 5.8: BER performance vs. Frame Length, 7 Turbo Iterations.

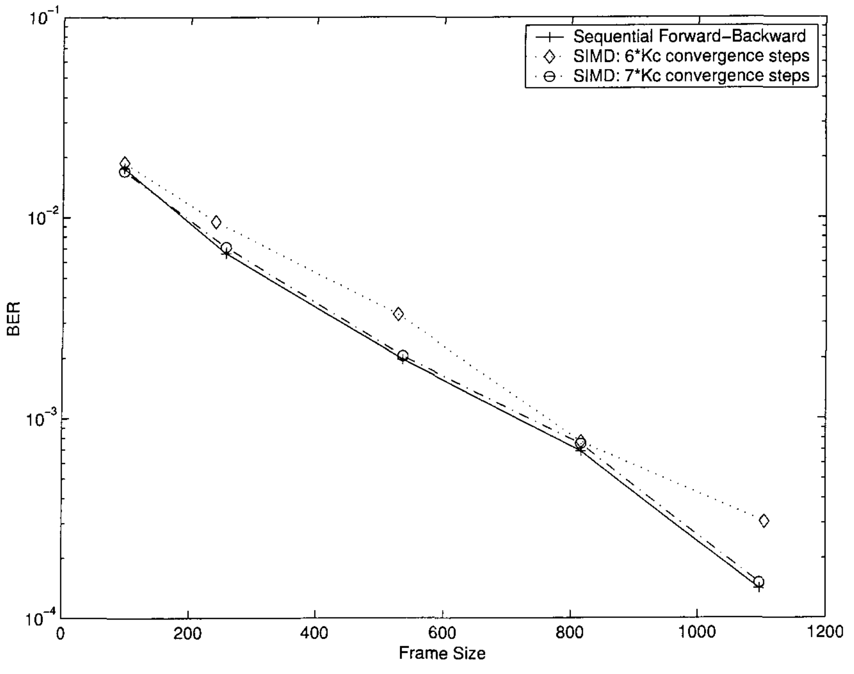


Figure 5.9: BER performance vs. Frame Length, 10 Turbo Iterations.

The BER performance displayed in Figures 5.7 through 5.9 show that the turbo decoder with PW-ONE based constituent decoders works correctly with increasing number of turbo loop iterations, and that better performance is achieved when the number of convergence steps is set to $(7 \times K_c)$.

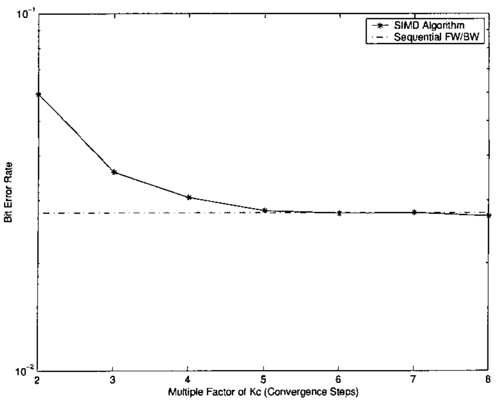
5.4.3 Convergence Steps Analysis

Results from the evaluation of the scalability of the data-parallel turbo decoding algorithm suggest that the degradation in BER performance cannot be ignored even when the number of convergence steps used by the constituent SISO decoders is set to five times the constraint length of the code. Therefore, it is necessary to examine in more detail the impact of the number of convergence steps on the BER performance of the turbo decoder.

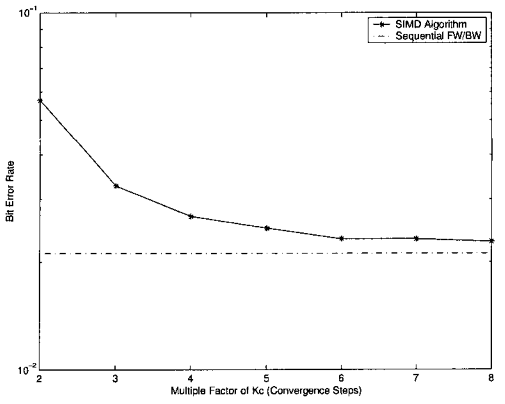
The test methodology consists of selecting two different frame lengths, one short and one relatively long, maintaining a fixed SNR level, and varying the number of convergence steps over a wider range than the range considered previously. The two frame sizes selected for further analysis are: (a) 96 data symbols, and (b) 1056 data symbols. Note that the choice for the short frame size is such that it matches exactly the number of PEs available in the Array Processor.

Short Frame Size (96 data bits)

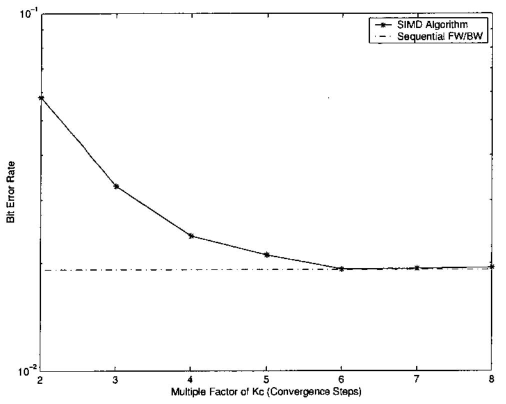
The set of four plots in Figure 5.10 compares the BER performance of the parallel processor turbo decoder against that of the fixed-point, sequential implementation when the frame length is fixed to 96 data symbols. Figure 5.11 displays the effect of turbo loop iterations and Figure 5.12 presents the combined effect of convergence steps and turbo loop iterations as a three dimensional plot.



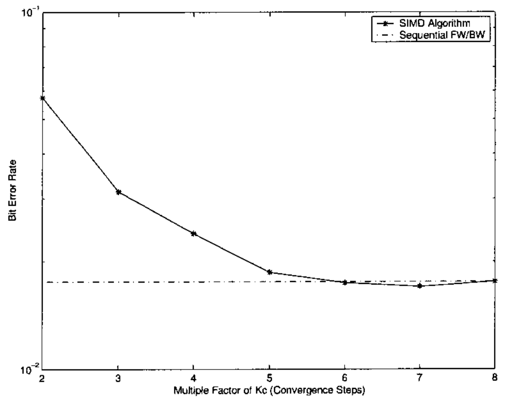
(a) 3 Turbo Loop Iterations.



(b) 5 Turbo Loop Iterations.



(c) 7 Turbo Loop Iterations.



(d) 10 Turbo Loop Iterations.

Figure 5.10: Bit Error Rate versus Number of Convergence Steps (Short Frame Size).

One can observe from the plots of Figures 5.11 and 5.12 that there exists a trade-off between number of convergence steps and the number of turbo loop iterations. A lower BER can be achieved for a given number of convergence steps if more turbo iterations are employed.

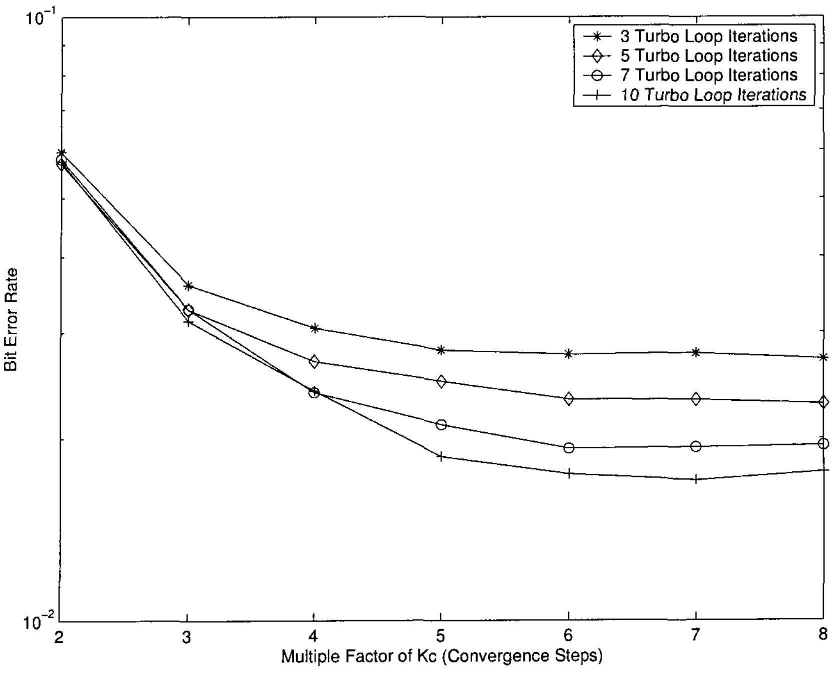


Figure 5.11: Effect of Decoder Iterations (Short Frame Size).

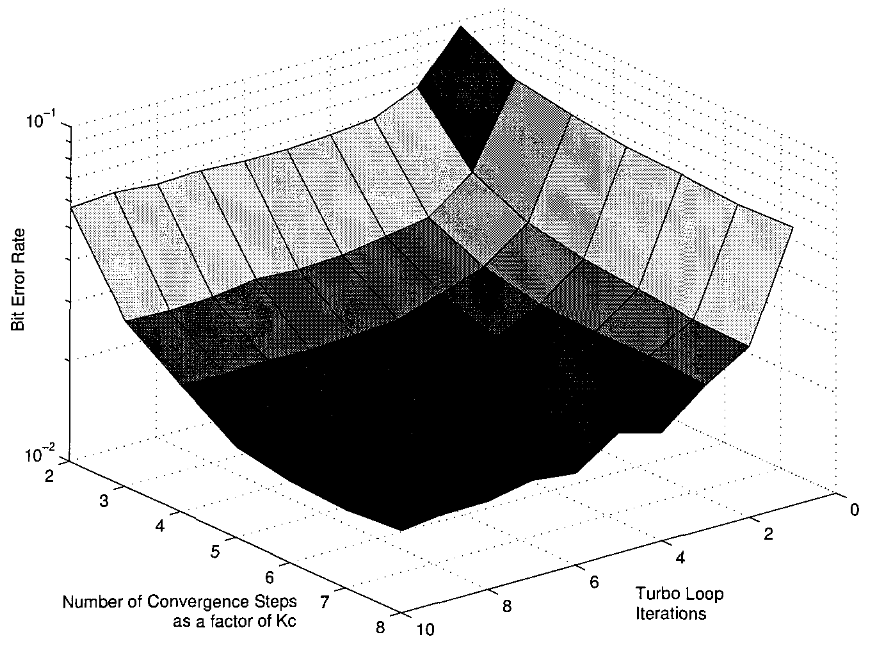
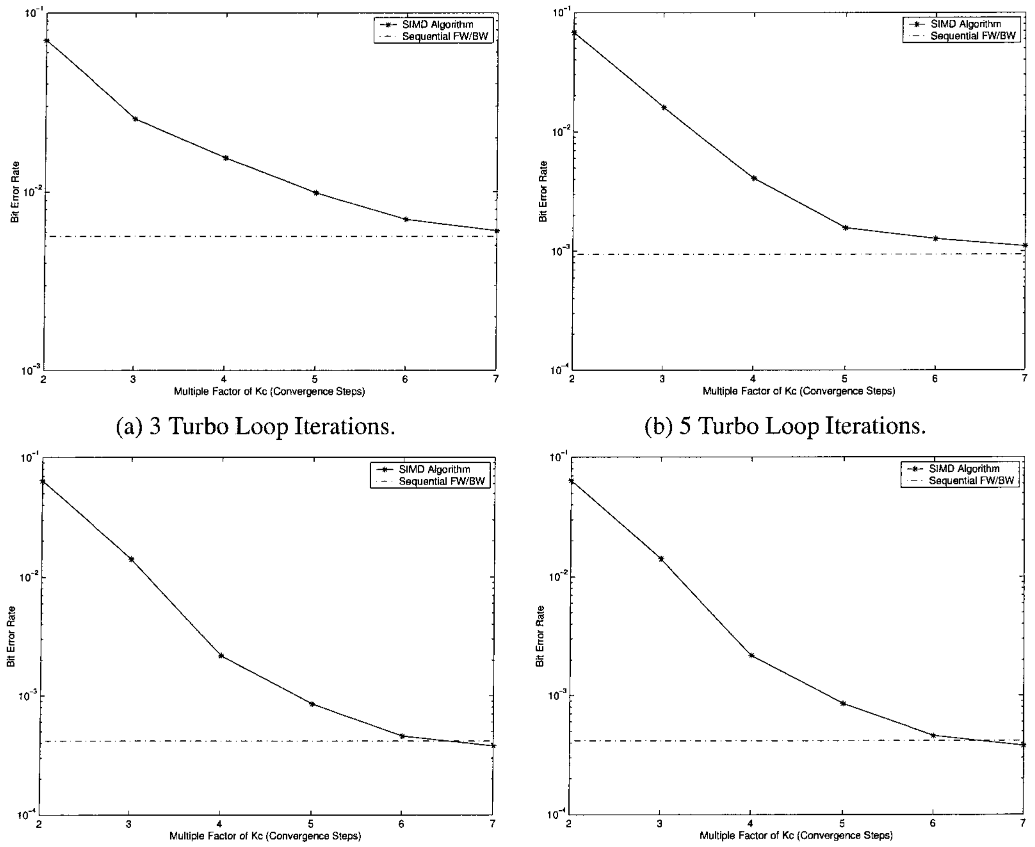


Figure 5.12: BER versus Decoder Iterations versus Convergence Steps (Short Frame Size).

Long Frame Size (1056 data bits)

Similar to the case of the short frame length, the BER performance is evaluated over a wider range for the number of convergence steps, in this case $2 \leq C_{steps} \leq 7$. Results are recorded and plotted in the same format as before. The set of four plots in Figure 5.13 display the BER performance for a fixed number of turbo loop iterations and a variable number of convergence steps in each case. Figure 5.14 and Figure 5.15 display the combined effect as a 2-D plot and as a 3-D plot respectively.



(a) 3 Turbo Loop Iterations. (b) 5 Turbo Loop Iterations.

(c) 7 Turbo Loop Iterations. (b) 10 Turbo Loop Iterations.

Figure 5.13: Bit Error Rate versus Number of Convergence Steps (Long Frame Size).

The plots of Figure 5.13 show how the BER performance of the turbo decoder with PW-ONE based constituent decoders approaches the BER performance of the sequential turbo decoder as the number of convergence steps increases.

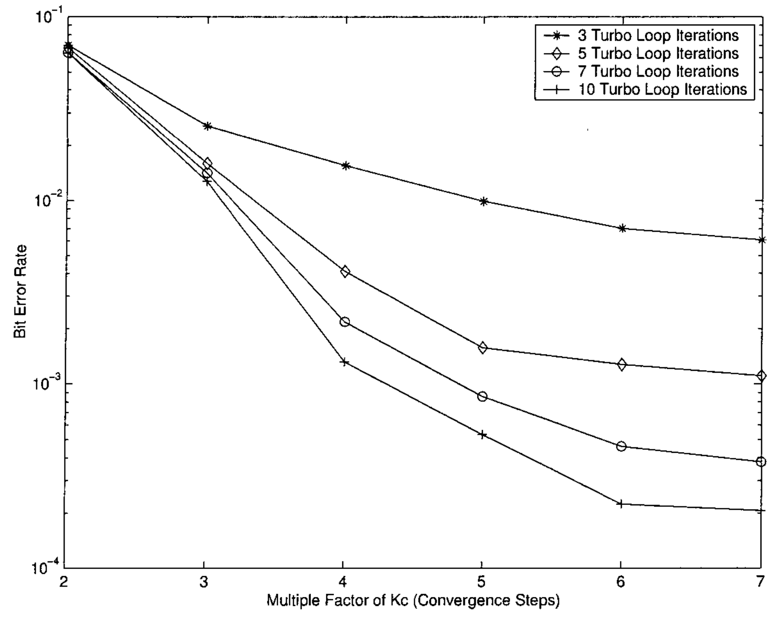


Figure 5.14: Effect of Decoder Iterations (Long Frame Size).

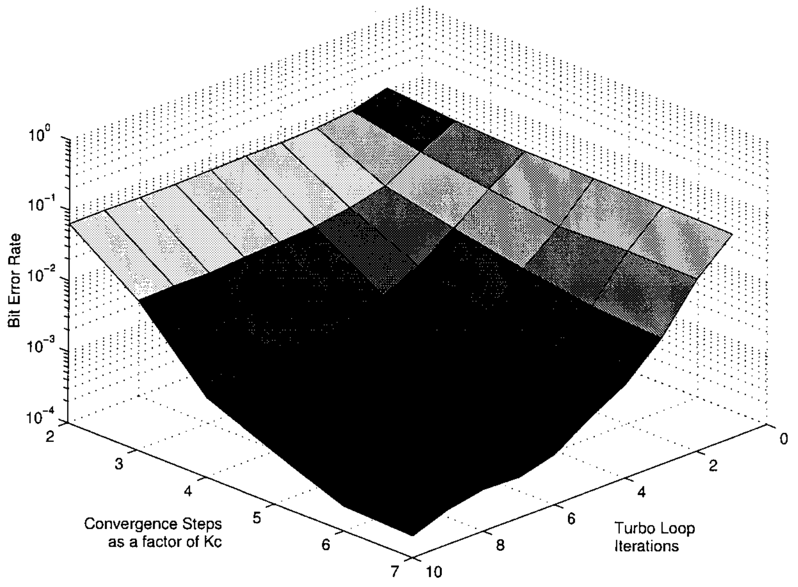


Figure 5.15: BER versus Decoder Iterations versus Convergence Steps (Long Frame Size).

Figures 5.14 and 5.15 again display the BER performance trade-off between convergence steps and turbo loop iterations.

It can be concluded from these observations that loss in BER performance exhibited by the data-parallel turbo decoder is reduced if the number of convergence steps is set to at least five times the constraint length for the case of short frames (for example, 96 bits), but seven times or greater may be required for frames whose length is greater than 1024 bits.

5.5 Execution Time and Speedup

Execution time measurements for both the fixed-point, sequential constituent SISO decoding algorithm and the SIMD implementation of the PW-ONE algorithm make it possible to determine the speedup factor as per equation (4.2). Further assumptions are made about future targeted architectural changes such as a 1-D linear array inter-processor communication network and more PEs, and accurate execution times are obtained when possible by simulating execution on real hardware. Note that the execution time for the data-parallel SISO decoder remains constant for any number of PEs. Measurements obtained when using the J2210 development platform demonstrate this, even though the incorrect decoded data is generated. However, execution time remains constant because the same number of SIMD instructions would be executed if proper hardware was available.

Experiments are conducted to obtain timing measurements that allow calculation of the speedup factor not only for the data-parallel constituent decoder but also for the turbo decoder where multiple instances of this constituent decoder are used. Computing the speedup factor for the case of the turbo decoder requires measuring execution time for the sequential interleaver, and the amount of time taken by the RISC engine to move data into and out of the Array Processor memory to perform sequential interleaving and de-interleaving.

The test methodology for execution time measurement is the following. Time stamping is performed by using the hardware timer facilities in the J2210 SoC. Hardware Timer 1 is used for this purpose. This is a programmable, free-running counter/timer whose tic source is derived from the system clock by sending the system clock through a divide by 4 or divide by 16 block. With a system clock frequency of 96 MHz, and a divider factor of 4, the tic source for Timer 1 is 24 MHz, thereby providing a resolution of 41.667 nanoseconds. Prior to invoking the decoder routine (sequential or parallel), the Counter Load Value is loaded with the maximum value of 0xFFFFFFFF by writing it to the counter/timer data register. As the decoder routine executes, the counter value is decremented on every cycle of the tic source. When the decoding routine finishes executing, the current counter value is saved by reading it from the counter/timer data register. Knowing the difference of the counter/timer value with respect to the pre-loaded value and the resolution of the timer makes it possible to obtain a precise measurement of the execution time.

The turbo encoder/channel model simulator is used to generate and send a predefined number of frames to the embedded target for decoding. A time measurement is recorded every time a frame is decoded, and the process is repeated for the predefined number of frames. This approach guarantees that time stamping is performed during steady state operation of the decoder. In order to obtain a precise execution time estimate, measurements are recorded for 1000 frames and the average is calculated.

Run-time results for moving data into and out of the Array Processor memory are based on a frame size of 96 data bits, but these results can be linearly scaled for longer frame sizes because the ARM^{TM} RISC engine uses sequential word accesses to perform these data transfers to and from memory. Read access times are the same for the simulated 1-D linear array and for the 2-D North-South topologies; however, the write access time for the case of the 2-D North-South topology includes the overhead incurred when re-ordering of the data before storing it in the Array Processor memory as per Figure 5.4. The data transfer times for a frame size with 96 data bits are the following:

- ARM $\leftarrow$ SIMD read access: 11 μs.
- ARM $\rightarrow$ SIMD write access (1-D topology): 9 μs.
- ARM $\rightarrow$ SIMD write access (2-D N-S): 16 μs.

With the previous results it is then possible to calculate an estimate for the time spent by the ARM^{TM} processor executing sequential instructions between the data-parallel SISO stages of the turbo decoder. These execution time estimates[5] are summarized in Table 5.1.

Frame Length	Read SIMD Time	Interleaver Time	Write SIMD (1-D)	Total Time	Write SIMD (2-D N-S)	Total Time
96	11	13	9	**33**	16	**40**
128	15	18	12	**45**	22	**55**
256	30	36	24	**90**	43	**109**
512	60	79	48	**187**	86	**225**
1024	120	159	96	**375**	172	**451**
2048	239	320	192	**751**	344	**903**
Execution time in microseconds (μs)						

Table 5.1: Memory Accesses and Interleaver Run-Times

The next set of measurements have been recorded to compare the execution time of the PW-ONE algorithm when using the North-South and West-East Data Transfer topologies of the inter-processor communication network against either a 1-D array or a spiral topology that assumes wrap-around with offset at the West and East edges of the 2-D grid. These results are frame size independent since execution time remains constant. The spiral topology is emulated by ignoring the edge condi-

C_{steps}	1-D	North-South		East-West	
$K_c = 4$	Time	Time	% Overhead	Time	% Overhead
$1 \times K_c$	0.171	0.217	26.90	0.378	121.05
$2 \times K_c$	0.321	0.411	28.04	0.716	123.05
$3 \times K_c$	0.472	0.606	28.39	1.054	123.31
$4 \times K_c$	0.622	0.800	28.62	1.391	123.63
$5 \times K_c$	0.772	0.994	28.76	1.729	123.96
$6 \times K_c$	0.923	1.188	28.71	2.066	123.84
$7 \times K_c$	1.073	1.383	28.89	2.404	124.04
$8 \times K_c$	1.224	1.577	28.84	2.741	123.94
Execution time measured in milliseconds					

Table 5.2: Execution Time of PW-ONE for Different Topologies

tion that must be handled in the other two cases. The results displayed in Table 5.2

[5]Time measurements and estimates are rounded up to the nearest microsecond even though the resolution of Timer 1 allows to approximate results to $\frac{1}{10}$ of a microsecond.

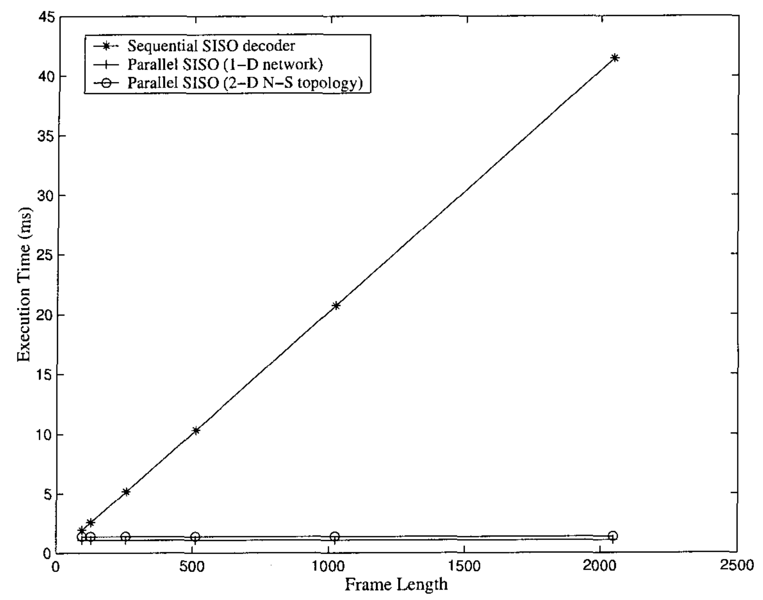
show that the data communication overhead of handling the edge condition for the North-South Data Transfer topology accounts for an extra 27% to 29% increase in execution time as compared to the execution time observed with the spiral network topology. In contrast, the data communication overhead incurred when handling the edge condition of the West-East Data Transfer topology makes execution of the PW-ONE algorithm more than two times slower relative to the using the spiral network.

Taking advantage of the scalability results presented in the previous section and assuming that there are enough PEs available to support variable frame lengths in the range $96 \leq N \leq 2048$, it is possible to compute the speedup factor achieved by executing the PW-ONE algorithm in the Array Processor of the J2210. Table 5.3 presents the speedup factor results for the constituent SISO decoder when comparing the sequential, fixed-point forward/backward algorithm against the parallel implementation with the more efficient data communication topologies. The execution time measurements used to compute the speedup factor are those where $7 \times K_c$ convergence steps were used because they exhibited BER performance closest to that of the sequential algorithm.

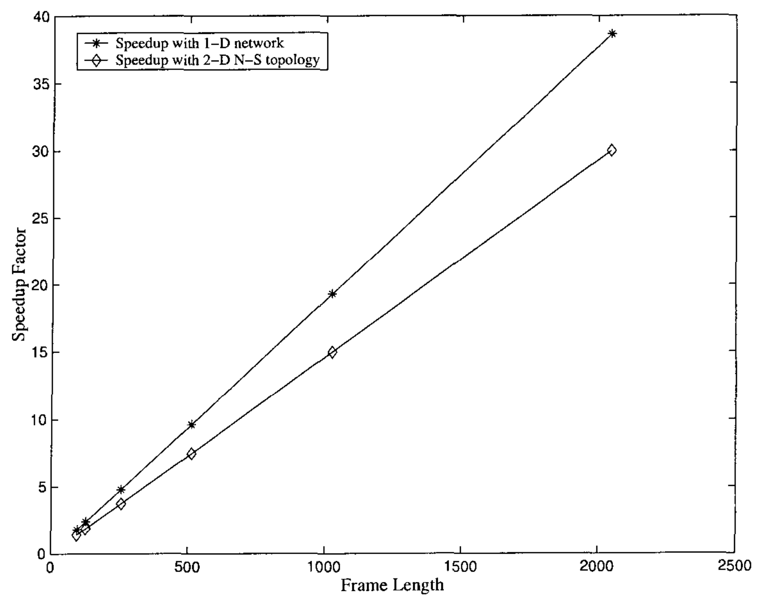
Frame Length	Sequential Time	SIMD 1-D Time	SIMD 1-D Speedup	SIMD 2-D (N-S) Time	SIMD 2-D (N-S) Speedup
96	1.925 ms	1.073 ms	**1.79**	1.383 ms	**1.39**
128	2.570 ms	1.073 ms	**2.40**	1.383 ms	**1.86**
256	5.136 ms	1.073 ms	**4.79**	1.383 ms	**3.71**
512	10.327 ms	1.073 ms	**9.62**	1.383 ms	**7.47**
1024	20.713 ms	1.073 ms	**19.30**	1.383 ms	**14.98**
2048	41.457 ms	1.073 ms	**38.64**	1.383 ms	**29.98**

Table 5.3: Speedup Factor Comparison for Constituent SISO Decoder

The plots of Figure 5.16 display graphically a comparison of execution time for the sequential and parallel implementation of the SISO decoding algorithm as a function of frame length. Note that the execution time of the data-parallel constituent decoder task for the case where frame length $N = 96$ is the only measured value since this execution time would remain constant for other frame sizes. The speedup factor behavior as displayed in Figure 5.16(b) is comparable to the estimated behavior based on equation (4.6) and plotted in Figure 4.6.



(a) Execution Time vs. Frame Length



(b) Speedup vs. Frame Length

Figure 5.16: Execution Time and Speedup Factor of Single Constituent Decoder.

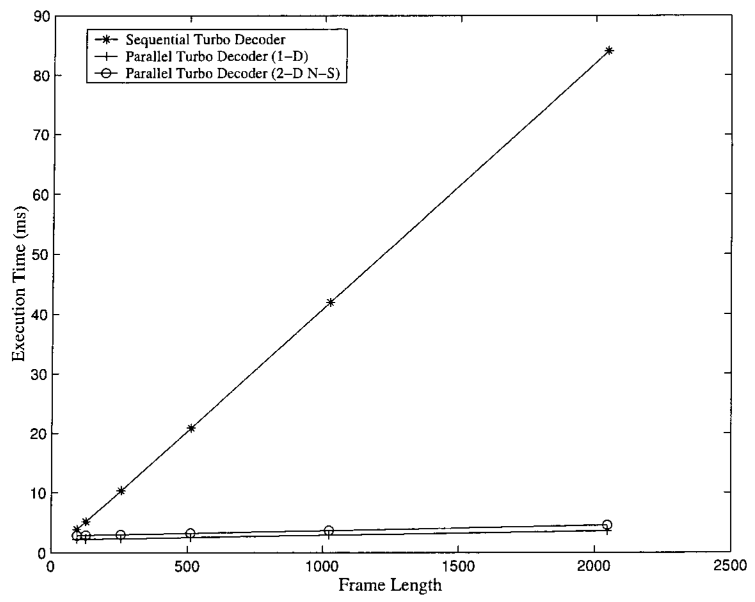
When reviewing the speedup factor results, it is important to note that the workload is proportional to the number of processing elements used. In other words, the underlying assumption is that the number of PEs available in the array processor is proportional to the length of the data frame.

Speedup factors achieved when evaluating a turbo decoder whose constituent SISO decoders are based on the SIMD implementation of the PW-ONE algorithm are presented in Table 5.4. The execution time of a single turbo decoder iteration is the parameter used to compute the speedup factor. Note that for the case of the turbo decoder with data-parallel components, the execution time for one turbo loop iteration is the sum of the execution time of two SIMD constituent decoders plus twice the time taken by the interleaver/de-interleaver including the associated read and write memory accesses to the Array Processor memory as indicated in Table 5.1.

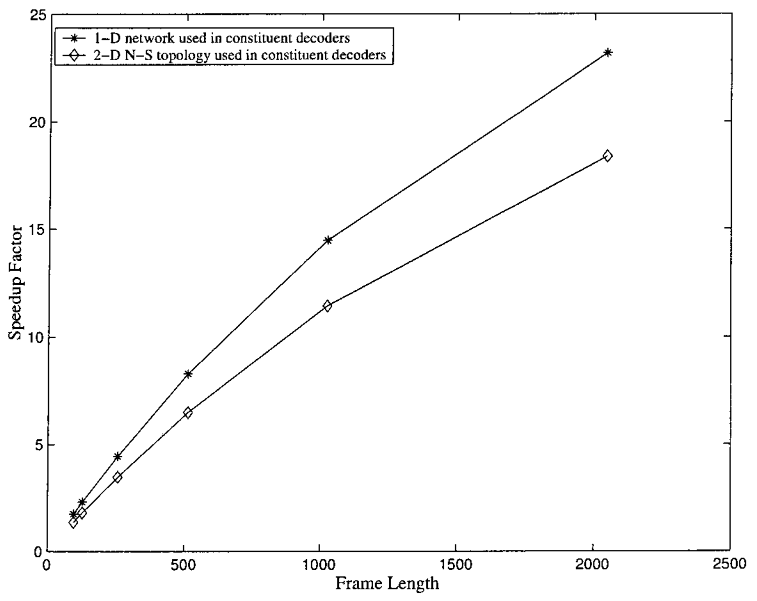
Frame Length	Sequential Time	SIMD 1-D Time	SIMD 1-D Speedup	SIMD 2-D (N-S) Time	SIMD 2-D (N-S) Speedup
96	3.885 ms	2.218 ms	**1.75**	2.852 ms	**1.36**
128	5.185 ms	2.240 ms	**2.31**	2.879 ms	**1.80**
256	10.366 ms	2.330 ms	**4.45**	2.987 ms	**3.47**
512	20.883 ms	2.522 ms	**8.28**	3.218 ms	**6.49**
1024	41.962 ms	2.901 ms	**14.47**	3.671 ms	**11.43**
2048	84.057 ms	3.652 ms	**23.02**	4.575 ms	**18.37**

Table 5.4: Speedup Factor Comparison for Turbo Decoder

The results of Table 5.4 have been plotted as a function of frame length and they can be observed in Figure 5.17. Note that for this case too, the workload is proportional to the number of processing elements. The impact of moving data into and out of the Array Processor memory to perform the sequential interleaving/de-interleaving can be observed as the decrease in the slope of the speedup graphs as the frame length increases. One would choose to use a point-to-point hardware interleaver for faster execution time per turbo decoder iteration. Such a hardware interleaver could be difficult to tune for different frame sizes and would add to the overall hardware cost.



(a) Execution Time vs. Frame Length



(b) Speedup vs. Frame Length

Figure 5.17: Execution Time and Speedup Factor of Turbo Decoder.

5.6 Power and Energy Considerations

Energy consumption is a main concern for many digital signal processing applications, primarily for portable applications where battery life is paramount. Energy per task is an important and useful metric, especially for mobile devices whose batteries have a finite energy capacity. The average power consumed by a portable device can be reduced by using a power management strategy where the device goes into low-power or standby mode after completion of computationally intensive tasks. However, the energy drained from the energy source (batteries) during execution of a given task can be calculated using equation (5.1).

$$Energy = Power \times Time \tag{5.1}$$

Therefore, both power and execution time per task are measured in order to calculate energy consumption per task. Execution time measurements for the sequential and parallel decoding algorithms were presented in section 5.5; in this section the methodology to obtain power measurements is described and energy consumption estimates are presented.

The power measurement method used to obtain accurate power and energy estimates for the ARM922T$^{\text{TM}}$ RISC engine and the Array Processor of the J2210 SoC takes advantage of the power distribution circuitry on the Atsana J2210 CEB. The power distribution on the CEB is illustrated in Figure 5.18. It consists of three

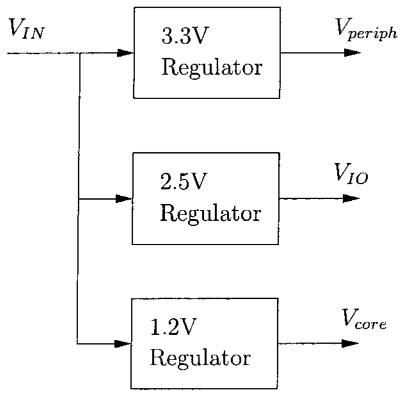


Figure 5.18: J2210 Customer Evaluation Board power distribution.

voltage regulators to supply the three voltage rails of the CEB design. The 3.3V regulator supplies power to the external memory and peripherals on the board. The I/O and core of the J2210 SoC are powered by the 2.5V and 1.2V regulators respectively. This is a convenient configuration because one can simply connect an ammeter in series between the output of the 1.2V regulator and its load to measure current consumption of the J2210 SoC.

The experimental setup was selected so that it bypasses the 1.2V regulator completely by desoldering the output pin of the 1.2V regulator from the board. An Agilent E3647A programmable, dual output, DC power supply was used to provide the necessary voltage sources. The experimental setup is depicted in Figure 5.19. Observe that one of the power supply outputs provides the primary input voltage V_{IN} to the CEB, while the second output replaces the 1.2V regulator. The ammeter

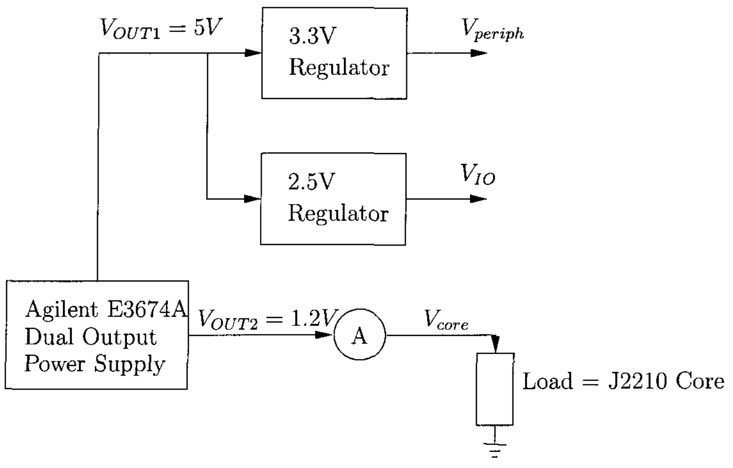


Figure 5.19: Experimental Setup using E3647A Power Supply.

shown in Figure 5.19 is built into the E3647A equipment and current measurement readings with a resolution of 1 mA are provided on the display of the power supply. The dual output capabilities of the E3647A power supply allow correct powering of the CEB in the test configuration while still meeting the power-up supply sequencing required by the J2210.

Power measurement inconsistencies usually result when on-chip peripherals and off-chip I/O are included as part of the measurement. The previously described experimental setup excludes any off-chip I/O factors because only the current consumption of the J2210 core is monitored, which still includes the current consumption contributions of on-chip peripherals. The J2210 SoC provides a power management strategy where various subsystems can be placed in a sleep mode via clock gating techniques. This technique is applied to power down on-chip peripherals that should remain inactive during the experiments.

Several test cases were identified that in order to isolate the current consumption of the ARM922T$^{\text{TM}}$ RISC engine, the Array Controller and the Array Processor. These test cases were the following:

Test Case A - Quiescent or Leakage Current

It is possible to obtain an estimate of the leakage current of the J2210 by asserting the power down input pin on the chip. When power down is asserted the DLL is disabled and all clocks are stopped.

Test Case B - ARM922T halted

Using this test case, it is possible to obtain a measurement that can be used as a baseline to determine the current consumption of the RISC engine when executing the sequential SISO decoding algorithm. The Array Controller (AC), CMEM Interface Unit (CIU), and the Array Processor are kept in sleep mode in this experiment[6].

Test Case C - ARM922T active

Here the AC, CIU, and the Array Processor remain in sleep mode while the RISC engine executes the sequential decoding algorithm. This operation is repeated in a loop that lasts approximately 30 seconds.

Test Case D - ARM922T, AC and CIU active

The AC architecture allows it and the RISC engine to execute asynchronously. Therefore, the Command Queue Unit of the Array Controller is first filled with commands, and then it executes these commands while

[6]Other on-chip peripherals that are put in sleep mode for all test cases are the USART and Block Encoder.

the RISC engine executes the sequential algorithm. The two processors remain executing their commands in a loop for a period of approximately 30 seconds. The Array Processor is kept in sleep mode.

Test Case E - Array Processor active

This test case is similar to case D expect that now the Array Processor is taken out of sleep mode.

Raw current measurements for each test case are summarized on Table 5.4. Isolating

Test Case	I_{core} Reading
A	3 mA
B	20 mA
C	57 mA
D	102 mA
E	141 mA

Table 5.5: Raw Current Consumption Measurements

the current consumption of the ARM922T™ , the AC and CIU combination, and the Array Processor, and assuming a constant core voltage of $V_{core} = 1.2V$, one can calculate the power consumption for each subsystem. These power measurements were obtained using a data set where the SNR level was set to $E_b/N_0 = 1.0$ dB.

Subsystem Name	ARM922T	AC + CIU	Array Processor
Current Consumption	34 mA	42 mA	36 mA
Power	40.8 mW	50.4 mW	43.2 mW

Table 5.6: Power Estimates for Individual Subsystems

Assuming that the power of the Array Controller remains constant, the power of the Array Processor can be scaled linearly with the number of PEs[7]. Then, one can compare energy consumed by the Array Processor against the energy consumed by the ARM™ core when executing their corresponding SISO decoding algorithms

[7]There exist some quantization effects that have been neglected. For instance, if the addition of one more PE results in the addition of a new memory bank, then power consumption would increase by a step greater than a linear step, but larger increments in the number of PEs should result in a close to linear increase in power consumption.

for various frame lengths. Execution time measurements from Table 5.1, the execution time for $7 \times K_c$ convergence steps, and power consumption measurements from Table 5.6 are used to estimate the energy consumption.

Frame Length	ARM922T Energy	Array Processor Energy	% Energy Savings
96	78.54 μJ	59.75 μJ	23.9
128	104.86 μJ	79.66 μJ	24.0
256	209.55 μJ	159.32 μJ	24.0
512	421.34 μJ	318.64 μJ	24.4
1024	845.09 μJ	637.29 μJ	24.6
2048	1691.45 μJ	1274.57 μJ	24.6

Table 5.7: Energy Consumption Comparison

The results of Table 5.6 clearly show that PIM technology offers an energy consumption advantage over a low-power, high performance embedded microprocessor such as the ARM922T$^{\text{TM}}$.

5.7 Summary

The implementation of the PW-ONE algorithm for execution on commercially available hardware has been described in this chapter. Implementation details such as the selection of the inter-processor communication topology were explained. The constituent data-parallel SISO decoders have been combined with a sequential implementation of the interleaver to form the complete turbo decoder. Evaluation of the turbo decoder has demonstrated that the concepts of the PW-ONE algorithm are scalable to accommodate the use of more PEs so that the decoding of longer data frames can be supported. Results also indicate that $7 \times K_c$ or more convergence steps may be required when decoding longer frames.

Considering the constituent SISO decoder alone, speedup factors greater than 10 can be obtained when applying the PW-ONE decoding technique to frames whose length exceeds 512 data symbols. Finally, the use of PIM technology results in approximately 24% energy savings, and this has been demonstrated with working hardware.

Chapter 6

Conclusions

This research work presented the design and implementation of a fixed-point, data-parallel algorithm for the decoding of turbo codes. The algorithm has been developed to be executed by the SIMD array processor found in the Atsana Semiconductor J2210 Media processor. Embedded systems for wireless applications require communications algorithms where fixed-point arithmetic is used as efficiently as possible, and where parallelism is exploited in both software and hardware in order to support the throughput demanded by high data rate applications. This work shows that a SIMD computer with PIM technology provides the flexibility of software with the performance of dedicated hardware to support the implementation of a turbo decoder in a 3G mobile platform. The design of the fixed-point, data-parallel algorithm required exhaustive analysis of the impact of finite precision on the turbo decoding algorithm and the exploration of data parallelism in the constant-Log-MAP algorithm used by the constituent SISO decoders of a turbo decoder.

Chapters 1 and 2 presented an introduction to this thesis and provided background material necessary for understanding the contributions of this work. In Chapter 3, the implementation of a fixed-point turbo decoder was presented. The constant-Log-MAP algorithm was selected for the constituent SISO decoders because of the simple method it uses to approximate the correction function of the *max** operation. The turbo decoder implementation described in Chapter 3 successfully minimized the fixed-point word length for all the variables used in the algorithm by operating on unscaled channel symbols and by limiting the saturation threshold of results from arithmetic operations. The effect of fractional precision was examined by evaluating the performance of the individual constituent decoders, while a density evolution analysis provided an indication of the number of integer bits required. Empirical results demonstrated that a turbo decoder with a fixed-point configuration of (8, 2), where the first digit represents the word length and the second digit indicates the number of fractional bits, achieves a BER performance that lies within 0.1 dB of the performance of a floating-point implementation. However, this is only achieved by limiting the fixed-point word length for the extrinsic information to one bit less than the fixed-point representation of the LLR values at the output of the SISO decoders.

Chapter 4 explored the possibilities for data parallelism in the trellis-based decoding algorithm for turbo codes. This exploration resulted in the identification of parallel data structures and inter-processor communication requirements. The trellis structure that describes the RSC constituent code can be mapped efficiently to a SIMD array processor where the PEs are arranged as a linear array. The requirements for the inter-processor communication network are simplified by this organization because data transfers between the state nodes of the trellis at different time intervals can be visualized as point-to-point data shifts. The Parallel Window ONE (PW-ONE) algorithm is also described in Chapter 4, where state metric distributions are evaluated in parallel for the entire trellis, and they converge to their true metric value after a predefined number of convergence steps. The number of convergence steps required by the PW-ONE algorithm affects the speedup and efficiency of the data-parallel algorithm.

Chapter 5 presents the implementation details of the PW-ONE algorithm when the SIMD target architecture is the Array Processor of the J2210 SoC. An efficient implementation depends on the selection of an inter-processor data transfer topology that reduces the communication overhead given the available inter-processor communication network. The communication overhead of the North-South Data Transfer topology described in Chapter 5 represents no more than 29% of the total execution time.

BER performance test results show that the PW-ONE algorithm is scalable to any size of frame length. However, analyzing the effect of the number of convergence steps on the BER performance demonstrates that five times the constraint length of the code is sufficient for short frames while seven times or greater may be required for longer frames. The results then confirm the PW-ONE algorithm hypothesis for some code configurations but contradict it for others. When the predefined number of convergence steps is fixed to seven times the constraint length of the code, execution time measurements on the hardware target show that the execution of the PW-ONE algorithm by a SIMD computer, as compared to execution of the sequential forward/backward algorithm by a high performance ARM^{TM} processor, can achieve speedup factors greater than ten when the frame length is greater than 512 data symbols. Experimental results also indicate that processing in memory offers 24% savings in energy consumption when compared to the popular low-power ARM9^{TM} architecture.

6.1 Future Research Directions

Future research work can be divided into three categories: algorithm development, architectural changes of the SIMD array processor hardware and VLSI implementation of the PW-ONE algorithm.

Algorithm Development

Regarding algorithm development and optimization, future research tasks may include:
1) In an effort to reduce the latency of the decoding procedure in the constituent SISO decoders, a different method for the computation of the α and β state metrics can be employed. The method applied by the current implementation initializes the state metric distributions as equiprobable at the start of the decoding procedure in each constituent SISO decoder. A new approach could take advantage of available memory resources so that each constituent decoder stores the state metrics values computed during it previous active state and then use them to initialize the state

metric distributions at the beginning of a new decoding iteration. Conducting experiments to evaluate this approach would demonstrate the possibility of reducing the number of convergence steps required by the constituent SISO decoder down to one constraint length of the constituent code or lower before computing the LLRs and sending them through the interleaver blocks.

2) Under the assumption that the Array Processor in the J2210 Media Processor continues to be the target SIMD architecture, develop custom micro-instructions to reduce execution time. For example, the MAX() micro-instruction provided with the J2210 AP SDK works well for unsigned numbers. However, to support the general case of finding the maximum between two signed or unsigned numbers, three additional SIMD instructions need to be executed including their corresponding load and store cycles.

3) Increase the fixed-point precision of the PW-ONE algorithm from 8-bit to 16-bit. It is expected that the increase in precision will reduce the frequency with which overflow conditions are checked. Reducing the time spent checking for arithmetic overflow may allow to achieve a higher speedup factors at the expense of an increase in storage requirements.

4) Determine if normalization of the α and β state metrics can be omitted from the operations performed by the PW-ONE algorithm. If normalization is still required, investigate other techniques that can potentially reduce the inter-processor communication overhead.

SIMD Architecture

Research tasks involving hardware modifications to the SIMD array processor architecture may include the following:

1) Develop a SIMD array processor with more PEs (CUs). To satisfy area restriction imposed by an SoC, the amount of distributed memory in the array processor may have to be reduced to make room for the logic circuits of the additional CUs. The main objective of this task would be to determine the point of equilibrium between the amount of distributed memory and the number of PEs that is practical for an

application involving the decoding of turbo codes. A 2-D mesh arrangement of CUs with wrap-around at the edges for efficient data transfer would require analyzing the latency issues introduced by the long interconnections.
2) Investigate the design of a dedicated communication network for interleaving the data that becomes the extrinsic input between constituent SISO decoding stages. This task would quantify the time delay imposed by this network as a factor equivalent to a given number of SIMD instructions.
3) Investigate the implementation of a CU with a custom word length other than 8-bit wide. One would determine the word length to satisfy the finite precision requirements of the algorithm and to reduce the occurrence of overflow from arithmetic operations without having to double the number of bits used by the existing architecture.

VLSI Implementation

A different research direction could see research efforts focused on the implementation of the PW-ONE algorithm in dedicated hardware using FPGA or VLSI technologies. Such an implementation could be coupled with a parallel hardware interleaver. With a completely data-parallel turbo decoder, researchers could determine the number of turbo loop iterations required to achieve the same BER performance as the sequential forward-backward algorithm if the PW-ONE algorithm does not wait for convergence of the α and β state metrics before estimating LLRs for the next decoding stage. A true data-parallel turbo decoder could achieve reduced latency, high data throughput and lower energy consumption per decoded bit at the expense of more arithmetic calculations.

SIMD Implementation of LDPC Decoders

The architecture of low density parity check (LDPC) decoders where there is no dependencies between neighboring data bits during a half-iteration of the decoding procedure lends itself nicely for implementation using a SIMD array processor. A final suggestion regarding potential future work involves investigating the design requirements for such an implementation.

Bibliography

[1] C. Berrou, A. Glavieux, and P. Thitimasjshima, "Near Shannon limit error-correcting coding and decoding: Turbo-codes(1)," in *Proc. IEEE International Conference on Communications*, Geneva, Switzerland, May 1993, pp. 1064–1070.

[2] 3GPP, "Universal mobile telecommunications system (UMTS): Multiplexing and channel coding (FDD)," European Telecommunications Standards Institute, TS 125.212 version 5.9.0 Release 5, June 2004.

[3] J. G. Harrison. "Implementation of a 3GPP Turbo Decoder on a Programmable DSP Core". 3DSP Corporation. [Online]. Available: http://www.3dsp.com/pdf/3dspTurboWhitePaper.pdf

[4] D. Gnaedig, M. Lapeyre, F. Mouchoux, and E. Boutillon, "Efficient SIMD technique with parallel Max-Log-MAP Algorithm for Turbo Decoders," in *GSPx2004 Embedded Applications Software & Hardware*, Santa Clara, CA, Sept. 2004.

[5] *S3000 3GPP (WCDMA) Compliant Turbo Decoder*, 1st ed., iCODING Technology Inc., San Diego, CA, Jan. 2002.

[6] *PCD03V 3GPP/3GPP2 8 state turbo decoder*, Small World Communications, Payneham South, Australia.

[7] V. C. Gaudet and P. G. Gulak, "A 13.3-Mb/s 0.35-μm CMOS Analog Turbo Decoder IC with a Configurable Interleaver," *IEEE J. Solid-State Circuits*, vol. 38, no. 11, pp. 2010–2015, Nov. 2003.

[8] TMS320C6416 Fixed-Point Digital Signal Processor Data Sheet. tms320c6416.pdf. Texas Instruments Inc. Houston, Texas. [Online]. Available: http://focus.ti.com/docs/prod/folders/print/tms320c6416t.html

[9] J. Hennessy and D. Patterson, *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann, 1994.

[10] K. Hwang, *Advanced Computer Architecture - Parallelism, Scalability, Programmability*, ser. Computer Engineering. McGraw-Hill, 1993.

[11] N. Aklilu, "Integrating computational RAM (C•RAM) into a system architecture," Master's thesis, University of Alberta, Edmonton, Alberta, July 2001.

[12] D. G. Elliott, "Computational RAM: A memory-SIMD hybrid," Ph.D. dissertation, University of Toronto, Toronto, 1998.

[13] D. Leder, "C•RAM with Fault-tolerant Reconfigurable 1D, 2D and 3D Communication Network," Master's thesis, University of Alberta, Edmonton, Alberta, 2004.

[14] S. Dillen, "Quantitative analysis of the SIMD DSP-RAM architecture," Master's thesis, University of Alberta, Edmonton, Alberta, July 2003.

[15] D. G. Elliott, M. Stumm, W. M. Snelgrove, C. Cojocaru, and R. Mckenzie, "Computational RAM: Implementing Processors in Memory," *IEEE Des. Test. Comput.*, vol. 16, no. 1, pp. 32–41, Jan. 1999.

[16] T. E. Le, W. M. Snelgrove, and S. Panchanathan, "SIMD processor arrays for image and video processing: a review," in *Proc. of SPIE*, vol. 3311 - Multimedia Hardware Architectures, San Jose, CA, Jan. 1998, pp. 30–41.

[17] M. Gokhale, B. Holmes, and K. Iobst, "Processing in Memory: The Terasys Massively Parallel PIM Array," *IEEE Computer*, vol. 28, no. 3, pp. 23–31, Apr. 1995.

[18] C. Cojocaru, "Computational RAM: Implementation and bit-parallel architecture," M.Eng. thesis, Carleton University, Ottawa, Ontario, Jan. 1995.

[19] *J2210 Software Tools User Manual*, Atsana Semiconductor Corporation, Ottawa, Ontario, July 2003, Document number: SWE-001-04.

[20] Y. Wu, "Implementation of Parallel and Serial Concatenated Convolutional Codes," Ph.D. dissertation, Virginia Polytechnic Institute and State University, Blacksburg, VA, Apr. 2000.

[21] J. Hagenauer, E. Offer, and L. Papke, "Iterative decoding of binary block and convolutional codes," *IEEE Trans. Inform. Theory*, vol. 42, no. 2, pp. 429–445, Mar. 1996.

[22] S. Benedetto, D. Divsalar, G. Montorsi, and F. Pollara, "Soft-output decoding algorithms in iterative decoding of turbo codes," NASA Jet Propulsion Laboratory (JPL), TDA Progress Report 42-124, Feb. 1996.

[23] J. Hagenauer and P. Hoeher, "A Viterbi algorithm with soft-decision outputs and its applications," in *IEEE GLOBECOM'89*, Nov. 1989, pp. 1680–1686.

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

[25] L. R. Bahl, J. Cocke, F. Jelinek, and J. Raviv, "Optimal decoding of linear codes for minimizing symbol error rate," *IEEE Trans. Inform. Theory*, vol. 20, pp. 284–287, Mar. 1974.

[26] P. Robertson, E. Villebrun, and P. Hoeher, "A comparison of optimal and sub-optimal MAP decoding algorithms operating in the log domain," in *Proc. IEEE International Conference on Communications*, vol. 2, Seattle, WA, June 1995, pp. 1009–1013.

[27] J. A. Erfanian and S. Pasupathy, "Low-complexity parallel-structure symbol-by-symbol detection for ISI channels," in *IEEE Pacific Rim Conf. Communications, Computers and Signal Processing*, June 1989, pp. 350–353.

[28] W. J. Gross and P. G. Gulak, "Simplified MAP algorithm suitable for implementation of turbo decoders," *Electronic Letters*, vol. 34, no. 16, pp. 1577–1578, Aug. 1998.

[29] J. F. Cheng and T. Ottosson, "Linearly approximated log-MAP algorithms for turbo decoding," in *Proc. IEEE Veh. Tech. Conf. (VTC)*, Houston, TX, May 2000, pp. 2252–2256.

[30] B. Classon, K. Blankenship, and V. Desai, "Turbo decoding with the constant-log-MAP algorithm," in *Proc. Second Int. Symp. Turbo Codes and Related Appl.*, Brest, France, Sept. 2000, pp. 467–470.

[31] H. Michel, A. Worm, M. Munch, and N. Wehn, "Hardware/Software tradeoffs for advanced 3G channel coding," in *Design, Automation and Test in Europe Conference and Exhibition*, Mar. 2002, pp. 396–401.

[32] X. Zeng and Z. Hong, "Design and Implemenation of a Turbo Decoder for 3G W-CDMA systems," *IEEE Trans. Consumer Electron.*, vol. 48, no. 2, pp. 284–291, May 2002.

[33] *SystemC 2.0.1 Language Reference Manual*, 1st ed., Open SystemC Initiative, San Jose, CA, 2003.

[34] IT++ library: Fixed-point data types. [Online]. Available: http://itpp.sourceforge.net/latest/group__fixtypes.html

[35] M. Valenti and J. Sun, "The UMTS turbo code and an efficient decoder implementation suitable for software defined radios," *International Journal on Wireless Information Networks*, vol. 8, no. 4, pp. 203–216, Oct. 2001.

[36] Z. Wang, H. Suzuki, and K. K. Parhi, "VLSI implementation issues of TURBO decoder design for wireless applications," in *IEEE Workshop on Signal Processing Systems, SiPS 99*, Oct. 1999, pp. 503–512.

[37] M. C. Valenti, "Iterative Detection and Decoding for Wireless Communications," Ph.D. dissertation, Virginia Polytechnic Institute and State University, Blacksburg, VA, July 1999.

[38] A. J. Viterbi, "An intuitive justification and a simplified implementation of the MAP decoder for convolutional codes," *IEEE J. Select. Areas Commun.*, vol. 16, no. 2, pp. 260–264, Feb. 1998.

[39] M. Bickerstaff, L. Davis, C. Thomas, D. Garrett, and C. Nicol, "A 24Mb/s radix-4 logMAP turbo decoder for 3GPP-HSDPA mobile wireless," in *ISSCC 2003*, vol. 1 - Digest of Technical Papers, 2003.

[40] D. Gnaedig, E. Boutillon, M. Jézéquel, V. C. Gaudet, and P. G. Gulak, "On multiple slice turbo codes," in *Proc. 3rd Int. Symp. Turbo Codes and Related Appl.*, Brest, France, Sept. 2003.

[41] WIKIPEDIA, free online encyclopedia. [Online]. Available: http://en.wikipedia.org/wiki/Speedup

Appendix A

Turbo Encoder/Decoder C++ Source Code

A.1 Fixed-Point Data Type Definition

Listing A.1: Fixed-Point Data Type Class Definition

```
/**
 * fixedpoint.h
 *
 * Copyright (c) 2004, 2005 Marco Castellon
 * University of Alberta, Edmonton, CANADA
 * All rights reserved.
 *
 * This software may be used for non-profit university research if
 * given the author's expressed permission. An executed license
 * agreement with the author is required for all other uses of this
 * software. Redistribution of this software is not permitted without
 * the author's expressed permission. This copyright notice must
 * remain intact. Derivative works may contain additional notices.
 *
 * THIS SOFTWARE IS PROVIDED BY THE AUTHOR "AS IS" AND COMES WITH
 * NO WARRANTY.
 *
 * Description: Class definition for fixed-point data types.
 *
 *
 */

#ifndef FIXEDPOINT_H
#define FIXEDPOINT_H

#include "fixassert.h"

// Typedef for signed 16-bit integer
typedef short int INT16;

// Typedef for unsigned 16-bit integer
typedef unsigned short int UINT16;

// Type to represent fixed-point word
typedef INT16 fixword;

// Maximum word length
const int MAX_WORDLEN = 16;

// Table for fast multiplication or division by 2^n
const UINT16 POW_OF_2[16] = {
  0x0001, 0x0002, 0x0004, 0x0008, 0x0010,
  0x0020, 0x0040, 0x0080, 0x0100, 0x0200,
  0x0400, 0x0800, 0x1000, 0x2000, 0x4000,
  0x8000
};

// Table for fast multiplication by 2^(n-16)
const double DOUBLE_POW2[32] = {
  1.525878906e-05, 3.051757812e-05, 6.103515625e-05, 1.220703125e-04,
  2.44140625e-04,  4.8828125e-04,   9.765625e-04,    0.001953125,
  0.00390625,      0.0078125,       0.015625,        0.03125,
  0.0625,          0.125,           0.25,            0.5,
  1.0,             2.0,             4.0,             8.0,
  16.0,            32.0,            64.0,            128.0,
  256.0,           512.0,           1024.0,          2048.0,
  4096.0,          8192.0,          16384.0,         32768.0
};

// Overflow modes
enum Ovf_Mode {
  SAT,                          // Saturation
  WRAP                          // Wrap-around
};

/*
  Fixed-point data type.
  Only Two's complement sign encoding is supported, and
  only quantization mode supported is "Rounding to Infinity" as
  described by the SC_RND_INF quantization mode in systemC.
*/

struct RawBits {
    fixword word_bits;
    int fraction_bits;
    };

class FixPoint {

public:
  // Default constructor.
  explicit FixPoint (double realValue = 0.0, int wl = MAX_WORDLEN, int fl = 0);

  // Copy constructor.
  FixPoint (const FixPoint &fix);

  // Constructor to handle the unconstrained results of arithmetic operations.
  //FixPoint (fixword raw, int fl);
  FixPoint (const RawBits &bits);

  // Destructor
  virtual ~FixPoint() {}

  /**** ACCESSOR AND MUTATOR MEMBER METHODS.****/

  // Set fractional length (without shifting).
  void set_fractionLength(int fl);

  // Set overflow mode for all objects of this class.
  static void set_OverflowMode(Ovf_Mode o_mode);

  // Get fractional length.
  int get_fractionLength(void) const;

  // Get word length.
  int get_wordLength(void) const;

  // Get overflow mode.
  static Ovf_Mode get_OverflowMode(void);

  // Get maximum value of data representation
  fixword get_max() const {return max_val;}

  // Get minimum value of data representation
  fixword get_min() const {return min_val;}

  /**** OVERLOADED ASSIGNMENT AND COMBINED BINARY/ASSIGNMENT OPERATORS .****/

  // Assignment from Fix
  FixPoint& operator=(const FixPoint &x);

  // Addition of Fix
  FixPoint& operator+=(const FixPoint &x);

  // Subtraction of Fix
  FixPoint& operator-=(const FixPoint &x);

  // Unary negative of Fix
  FixPoint operator-() const;

  // Left shift n bits
  FixPoint& operator<<=(const int n);

  // Right shift n bits using quantization mode RND_INF
  FixPoint& operator>>=(const int n);

  // Overloaded binary addition operator (FixPoint + FixPoint)
  friend const FixPoint operator+(const FixPoint &x, const FixPoint &y);

  // Overloaded binary subtraction operator (FixPoint + FixPoint)
```

```
    friend const FixPoint operator-(const FixPoint &x, const FixPoint &y);

    // Absolute value
    friend const FixPoint absolute(const FixPoint &x);

    // Maximum of two values
    friend const FixPoint maximum(const FixPoint &x, const FixPoint &y);

    // Set to x * pow2(n) using quantization mode
    void set(double x, int n);

    // Set data representation (mainly for internal use since it reveals the representation type)
    void set_rawbin(fixword x) {raw_bin_num = handle_overflow(x);}

    // Left shift \c n bits
    void lshift(int n);

    // Right shift \c n bits using quantization mode \c qmode (constructor argument)
    void rshift(int n);

    // Get raw binary number used in the data representation.
    // (mainly for internal use since it reveals the representation type)
    fixword get_rawbin() const {return raw_bin_num;}

    // Conversion to double
    double unfix() const;

    // Conversion to double
    operator double() const
    {
      FXP_ASSERT(fraction_len>=-15 && fraction_len<=16, "FixPoint::operator double: illegal fractional length!")
          ;
      return double(raw_bin_num)*DOUBLE_POW2[16 - fraction_len];
    }

    // Check that x.fractionlen==y.fractionlen OR x==0 OR y==0 and return the
    // the fractional length (for the non-zero argument)
    friend int assert_fractionlen(const FixPoint &x, const FixPoint &y);

  protected:

    // Raw binary number used for Data representation
    fixword raw_bin_num;

    // The number of bits to the right of binary point.
    // This value determines the scaling and interpretation of the fixed-point number.
    int fraction_len;

    // Word length
    int word_len;

    // Overflow mode
    static Ovf_Mode ovfmode;

    // Minimum allowed value (help variable to speed up calculations)
    fixword min_val;

    // Maximum allowed value (help variable to speed up calculations)
    fixword max_val;

    // Number of unused (MSB) bits (help variable to speed up calculations)
    int n_unused_bits;

    // Calculate help variables min, max and n_unused_bits
    void initialize (void);

    // Handle overflows using overflow mode.
    fixword handle_overflow(fixword x) const;

    // Convert from double to raw binary representation using power-of-two scaling
    // and apply quantization.
    fixword scale_and_quantize(double x) const;

    // Right shift  n bits using quantization mode.
    fixword rshift_and_quantize(fixword x, int n) const;

  };


  /*
      brief Templated fixed-point data type

    */
  template<int wl>
  class Fixed : public FixPoint {
  public:
    // Default constructor
    Fixed(double real=0.0, int fl=0)
      : FixPoint(real, wl, fl) {}
  // Constructor
  Fixed(const FixPoint &x)
    : FixPoint(x) {}

  // Destructor
  virtual ~Fixed() {}

  // Assignment from Fix
  Fixed& operator=(const FixPoint &x)
  {
    fraction_len = x.get_fractionLength();
    raw_bin_num = handle_overflow(x.get_rawbin());
    return *this;
  }

protected:
};

  // Typedefs for saturated Fixed
  typedef Fixed<1> sfixed1;
  typedef Fixed<2> sfixed2;
  typedef Fixed<3> sfixed3;
  typedef Fixed<4> sfixed4;
  typedef Fixed<5> sfixed5;
  typedef Fixed<6> sfixed6;
  typedef Fixed<7> sfixed7;
  typedef Fixed<8> sfixed8;
  typedef Fixed<9> sfixed9;
  typedef Fixed<10> sfixed10;
  typedef Fixed<11> sfixed11;
  typedef Fixed<12> sfixed12;
  typedef Fixed<13> sfixed13;
  typedef Fixed<14> sfixed14;
  typedef Fixed<15> sfixed15;
  typedef Fixed<16> sfixed16;

/********** BEGINNING OF FUNCTIONS ****************/

  // Set y = x * pow2(n) using the quantization mode of y
  inline void set_fix(FixPoint &y, double x, int n) {y.set(x, n);}

  // Left shift n bits
  inline void lshift_fix(FixPoint &y, int n) {y.lshift(n);}
  // Right shift n bits using the quantization mode of y
  inline void rshift_fix(FixPoint &y, int n) {y.rshift(n);}


  // Convert Fix to double by multiplying the bit representation with pow2(-shift)
  inline double unfix(const FixPoint &x) {return x.unfix();}


#endif // FIXEDPOINT_H
```

Listing A.2: Fixed-Point Data Type Class Implementation

```
/**
 * fixedpoint.cpp
 *
 * Copyright (c) 2004, 2005 Marco Castellon
 * University of Alberta, Edmonton, CANADA
 * All rights reserved.
 *
 * This software may be used for non-profit university research if
 * given the author's expressed permission. An executed license
 * agreement with the author is required for all other uses of this
 * software. Redistribution of this software is not permitted without
 * the author's expressed permission. This copyright notice must
 * remain intact. Derivative works may contain additional notices.
 *
 * THIS SOFTWARE IS PROVIDED BY THE AUTHOR "AS IS" AND COMES WITH
 * NO WARRANTY.
 *
 * Description: Class Implementation for fixed-point data types.
 *
 */

#include "fixedpoint.h"

Ovf_Mode FixPoint::ovfmode = SAT;  // Default overflow handling mode is saturation.

/**
 * Definition of default constructor.
 */
FixPoint::FixPoint(double realValue, int wl, int fl)
  : word_len(wl), fraction_len(fl)
{
  initialize();
  raw_bin_num = scale_and_quantize(realValue);
}

/**
 * Definition of copy constructor.
 */
FixPoint::FixPoint(const FixPoint &fix)
{
  word_len = fix.word_len;
  fraction_len = fix.fraction_len;

  initialize();

  raw_bin_num = fix.raw_bin_num;
}

/**
 * Definition of constructor for unrestricted arithmetic operations.
 */
FixPoint::FixPoint(const RawBits &bits)
{
  word_len = MAX_WORDLEN;

  initialize();

  fraction_len = bits.fraction_bits;
  raw_bin_num = bits.word_bits;
}

/**
 * initialize()
 * Based on the specified word length, determine the minimum and maximum
 * raw binary numbers that can be represented using 2's complement sign encoding.
 */
void FixPoint::initialize (void)
{
  FXP_ASSERT(word_len >= 1 && word_len <= 16, "FixPoint::initialize: Illegal word length!");
  max_val = fixword(POW_OF_2[word_len - 1] - 1);
  min_val = static_cast<fixword>(-max_val - 1);

  n_unused_bits = MAX_WORDLEN - word_len;
}

/**
 * handle_overflow()
 */
fixword FixPoint::handle_overflow(fixword x) const
{
  fixword tmp = x;
  bool overflow = false;

  if (tmp < min_val) {
    overflow = true;
    switch (FixPoint::ovfmode) {
    case WRAP:
      tmp = fixword((static_cast<INT16>(tmp) << n_unused_bits) >> n_unused_bits);
```

```
        break;
      case SAT:
        tmp = min_val;
        break;
      default:
        break;
      }
    }
  else if (tmp > max_val) {
    overflow = true;
    switch (FixPoint::ovfmode) {
    case WRAP:
      tmp = fixword((static_cast<INT16>(tmp) << n_unused_bits) >> n_unused_bits);
      break;
    case SAT:
      tmp = max_val;
      break;
    default:
      break;
    }
  }

  return tmp;
}

fixword FixPoint::scale_and_quantize(double x) const
{
  FXP_ASSERT(fraction_len>=-16 && fraction_len<=15, "FixPoint::scale_and_quantize: Illegal shift!");

  fixword temp = 0;

  // Apply power-of-two scaling to the real value.
  double scaled_value = x*DOUBLE_POW2[fraction_len + 16];

  // If the number is negative round towards minus infinity, and if
  // it is positive round to positive infinity. systemC calls this
  // SC_RND_INF quantization mode.
  if (x < 0)
    temp = handle_overflow(static_cast<fixword>(scaled_value - 0.5));
  else
    temp = handle_overflow(static_cast<fixword>(scaled_value + 0.5));

  return temp;
}

/**
 * rshift_and_quantize()
 */
fixword FixPoint::rshift_and_quantize(fixword x, int n) const
{
  FXP_ASSERT(n >= 0, "FixPoint::rshift_and_quantize: n cannot be negative!");

  fixword temp = 0;

  if (n == 0)
    {
      temp = x;
    }
  else
    {
      // If the most significant deleted bit is 1,
      // and either the inverted value of the sign bit or at least one other deleted bit is 1,
      // add 1 to the remaining bits
      if ((x & (static_cast<fixword>(1) << (n - 1))) &&
          ((x >= 0) || (x & ((static_cast<fixword>(1) << (n - 1)) - 1))))
        temp = fixword((x >> n) + 1);
      else
        temp = fixword(x >> n);
    }

  return temp;
}

/**
 * set_fractionLength()
 */
void FixPoint::set_fractionLength (int fl) { fraction_len = fl; }

/**
 * set_OverflowMode()
 */
void FixPoint::set_OverflowMode(Ovf_Mode o_mode) { FixPoint::ovfmode = o_mode; }

/**
 * get_fractionLength()
 */
int FixPoint::get_fractionLength(void) const { return(fraction_len); }

/**
```

```
 * get_wordLength()
 */
int FixPoint::get_wordLength(void) const { return(word_len); }

/**
 * get_OverflowMode()
 */
Ovf_Mode FixPoint::get_OverflowMode(void) { return(FixPoint::ovfmode); }


/**************** BEGINNING OF Fix class implementation *************************/

FixPoint& FixPoint::operator=(const FixPoint &x)
{
  fraction_len = x.fraction_len;
  raw_bin_num = handle_overflow(x.raw_bin_num);
  return *this;
}


FixPoint& FixPoint::operator+=(const FixPoint &x)
{
  fraction_len = assert_fractionlen(*this, x);
  raw_bin_num = handle_overflow(fixword(raw_bin_num + x.raw_bin_num));
  return *this;
}


FixPoint& FixPoint::operator-=(const FixPoint &x)
{
  fraction_len = assert_fractionlen(*this, x);
  raw_bin_num = handle_overflow(fixword(raw_bin_num - x.raw_bin_num));
  return *this;
}


FixPoint FixPoint::operator-() const
{
  RawBits bits;

  bits.word_bits = static_cast<fixword>(-raw_bin_num);
  bits.fraction_bits = fraction_len;
  return FixPoint(bits);
}

FixPoint& FixPoint::operator<<=(const int n)
{
  FXP_ASSERT(n >= 0, "FixPoint::operator<<=: n cannot be negative!");
  fraction_len += n;
  raw_bin_num = handle_overflow(fixword(raw_bin_num << n));
  return *this;
}

FixPoint& FixPoint::operator>>=(const int n)
{
  fraction_len -= n;
  raw_bin_num = rshift_and_quantize(raw_bin_num, n);
  return *this;
}

void FixPoint::set(double x, int n)
{
  fraction_len = n;
  raw_bin_num = scale_and_quantize(x);
}


void FixPoint::lshift(int n)
{
  FXP_ASSERT(n >= 0, "FixPoint::lshift: n cannot be negative!");
  fraction_len += n;
  raw_bin_num = handle_overflow(fixword(raw_bin_num << n));
}

void FixPoint::rshift(int n)
{
  fraction_len -= n;
  raw_bin_num = rshift_and_quantize(raw_bin_num, n);
}


double FixPoint::unfix() const
{
  FXP_ASSERT(fraction_len>=-15 && fraction_len<=16, "FixPoint::unfix: illegal shift!");
  return (static_cast<double>(raw_bin_num)*DOUBLE_POW2[16 - fraction_len]);
}


int assert_fractionlen(const FixPoint &x, const FixPoint &y)
```

```
{
  int temp = 0;

  if (x.fraction_len == y.fraction_len)
    temp = x.fraction_len;
  else if (x.fraction_len == 0)
    temp = y.fraction_len;
  else if (y.fraction_len == 0)
    temp = x.fraction_len;
  else
    FXP_ERROR("assert_fractionlen: Different shifts not allowed!");

  return temp;
}

/***** OTHER FUNCTIONS ********/

const FixPoint absolute(const FixPoint &x)
{
  RawBits bits;
  fixword temp = x.raw_bin_num;

  bits.word_bits = static_cast<fixword>(temp >= 0 ? temp : -temp);    // Risk for overflow.
  bits.fraction_bits = x.fraction_len;

  return FixPoint(bits);
}

const FixPoint maximum(const FixPoint &x, const FixPoint &y)
{
  RawBits bits;
  fixword operand1, operand2;

  if(x.fraction_len != y.fraction_len)
    FXP_ERROR("maximum: Different fractional lengths not allowed!");
  else
    bits.fraction_bits = x.fraction_len;

  operand1 = x.raw_bin_num;
  operand2 = y.raw_bin_num;

  bits.word_bits = (operand1 >= operand2 ? operand1 : operand2);

  return FixPoint(bits);
}

/***** THE OPERATORS ********/
  /////////////////////////////////
  // Operators for Fix and Fixed //
  /////////////////////////////////

// Declared as friend, therefore, can access private members.
const FixPoint operator+(const FixPoint &x, const FixPoint &y)
{
  RawBits bits;

  if (x.fraction_len != y.fraction_len)
    FXP_ERROR("addition operator: Different fractional lengths not allowed!");
  else
    bits.fraction_bits = x.fraction_len;

  bits.word_bits = static_cast<fixword>(x.raw_bin_num + y.raw_bin_num);

  return FixPoint(bits);
}

// Declared as friend, therefore, can access private members.
const FixPoint operator-(const FixPoint &x, const FixPoint &y)
{
  RawBits bits;

  if (x.fraction_len != y.fraction_len)
    FXP_ERROR("subtration operator: Different fractional lengths not allowed!");
  else
    bits.fraction_bits = x.fraction_len;

  bits.word_bits = static_cast<fixword>(x.raw_bin_num - y.raw_bin_num);
  return FixPoint(bits);
}
```

A.2 UMTS Interleaver Generator Method

Listing A.3: Method that generated interleaver map

```
/**
 * umtsInterleaver.cpp
 *
 * Copyright (c) 2004,2005 Marco Castellon
 * University of Alberta, Edmonton, CANADA
 * All rights reserved.
 *
 * Description:
 * This is the implementation of the algorithm for the generation of the
 * Turbo code internal interleaver defined UMTS specification. Reference is
 * document number: ETSI TS 25.212 V5.9.0 (2004-06), pages 17 - 20. This
 * function is a member method of the turboCodec class defined in
 * turboCodec.h . The class member variables used here are "frameLength"
 * and "pInterleaverMap". The pInterleaverMap stores the permutations
 * generated by this function, and they are used by the interleaver/
 * de-interleaver methods.
 *
 */

#include <cstdlib>
#include "turboCodec.h"

using namespace std;


struct PrimeTableElement {
  int prime_number;
  int primitive_root;
};
/**
 * Table 2, page 19 of 3GPP Specification. Document Number: TS 25.212
 * List of prime number p and associated primitive root v.
 */
const PrimeTableElement tableII_3GPP[52] = {
  {7,3},{11,2},{13,2},{17,3},{19,2},{23,5},{29,2},{31,3},{37,2},{41,6},{43,3},
  {47,5},{53,2},{59,2},{61,2},{67,2},{71,7},{73,5},{79,3},{83,2},{89,3},{97,5},
  {101,2},{103,5},{107,2},{109,6},{113,3},{127,3},{131,2},{137,3},{139,2},{149,2},{151,6},
  {157,5},{163,2},{167,5},{173,2},{179,2},{181,2},{191,19},{193,5},{197,2},{199,3},{211,2},
  {223,3},{227,2},{229,6},{233,3},{239,7},{241,7},{251,6},{257,3}
};

/**
 * Table of Prime numbers. Also from Table 2.
 */
const int primes[55] = {
  2, 3, 5, 7, 11, 13, 17, 19, 23, 29,
  31, 37, 41, 43, 47, 53, 59, 61, 67, 71,
  73, 79, 83, 89, 97, 101, 103, 107, 109, 113,
  127, 131, 137, 139, 149, 151, 157, 163, 167, 173,
  179, 181, 191, 193, 197, 199, 211, 223, 227, 229,
  233, 239, 241, 251, 257
 };

/**
 * GCD - Function to compute the Greatest Common Divider between
 *       2 integer numbers.
 */
int gcd (int a, int b)
{
  register int c;
  while (a)
  {
    c = a;
    a = b%a;
    b = c;
  }
  return b;
}

/**
 * umtsInterleaverMap()
 *
 * Implementation of the Turbo code internal interleaver as defined in the UMTS 3GPP
 * specification (Document TS 125.212 version 5.9.0), pages 17-20.
 */
void TurboCodec::umtsInterleaverMap (void)
{
  int rows; // Number of rows of rectangular matrix.
  int columns; //Number of columns of rectangular matrix.
  int primeNumber;
  int primitiveRoot;

  /** Inter-row permutation patterns */
```

```
int interRowInterleaver_I[5]     = {4, 3, 2, 1, 0};
int interRowInterleaver_II[10]   = {9, 8, 7, 6, 5, 4, 3, 2, 1, 0};
int interRowInterleaver_III[20]  = {19,9,14,4,0,2,5,7,12,18,16,13,17,15,3,1,6,11,8,10};
int interRowInterleaver_IV[20]   = {19,9,14,4,0,2,5,7,12,18,10,8,13,17,3,1,16,6,15,11};

int i = 0;
int j = 0;
int position = 1;
int gcd_val = 0;
int **originalMatrix;
int **permutedMatrix;

int *baseSequence = NULL; // Length is equal to the number of columns of the rectangular matrix.
int *qSequence = NULL;    // Length is equal to the number of rows of the rectangular matrix.
int *rSequence = NULL;
int *interRowPermutation = NULL;


/* Determine the number of rows of the rectangular matrix.*/
if ((frameLength >= 40) && (frameLength <= 159))
  rows = 5;
else if (((frameLength >= 160) && (frameLength <= 200)) || ((frameLength >= 481) && (frameLength <= 530)))
  rows = 10;
else
  rows = 20;

/* Find the prime number p to be used in the intra-row permutations.*/
do {
  primeNumber = tableII_3GPP[i].prime_number;
  primitiveRoot = tableII_3GPP[i].primitive_root;
  i++;
} while((rows*(primeNumber+1)) < frameLength);

/* Determine the number of columns of the rectangular matrix. */
if (frameLength <= (rows*(primeNumber-1)))
  columns = primeNumber - 1;
else if ((frameLength > (rows*(primeNumber-1))) && (frameLength <= (rows*primeNumber)))
  columns = primeNumber;
else if (frameLength > (rows*primeNumber))
  columns = primeNumber + 1;


/* Dynamically allocate memory for the matrix, and initialize it.*/
originalMatrix = new int *[rows];
for(i = 0; i < rows; i++)
{
  originalMatrix[i] = new int[columns];
  for(j = 0; j < columns; j++)
  {
    if (((columns*i)+j) < frameLength)
      originalMatrix[i][j] = position++;
    else
      originalMatrix[i][j] = 0;
  }
}


// Allocate memory for the base sequence and the prime integer sequence.
baseSequence = new int[columns];
baseSequence[0] = 1;
qSequence = new int[rows];
qSequence[0] = 1;

/* Construct the base sequence for intra-row permutations*/
for (i = 1; i < columns; i++)
{
  baseSequence[i] = (primitiveRoot*baseSequence[i-1])%primeNumber;
}

/* Construct the q sequence (prime integer sequence).*/
for (i = 1; i < rows; i++)
{
  j = 0;
  do {
    qSequence[i] = tableII_3GPP[j].prime_number;
    gcd_val = gcd(qSequence[i], (primeNumber-1));
    j++;
  } while((gcd_val != 1) || (qSequence[i] <= qSequence[i-1]));
}

/* Permute the q sequence into the r sequence.*/
rSequence = new int[rows];
/* Select the appropriate inter-row permutation pattern to use.*/
switch (rows)
  {
    case 5:
      interRowPermutation = interRowInterleaver_I;
      break;
    case 10:
        interRowPermutation = interRowInterleaver_II;
        break;
      case 20:
        if (((frameLength >= 2281) && (frameLength <= 2480)) || ((frameLength >= 3161) && (frameLength <=
             3210)))
          interRowPermutation = interRowInterleaver_III;
        else
          interRowPermutation = interRowInterleaver_IV;
        break;
      default:
        break;
    }

  for (i = 0; i < rows; i++)
    *(rSequence+interRowPermutation[i]) = *(qSequence+i);

  /* Perform the intra-row permutations.*/
  // Store the results in a dynamically allocated matrix.
  permutedMatrix = new int *[rows];

  for (i = 0; i < rows; i++)
    {
      permutedMatrix[i] = new int[columns];

      if (columns == primeNumber)
        {
          for(j = 0; j < (primeNumber-1); j++)
            {
              int index1, index2;

              index1 = (j*rSequence[i])%(primeNumber-1);
              index2 = baseSequence[index1];
              permutedMatrix[i][index2] = originalMatrix[i][j];
            }
          permutedMatrix[i][0] = originalMatrix[i][(primeNumber-1)];
        }

      else if (columns == (primeNumber+1))
        {
          int savedIndex;
          for(j = 0; j < (primeNumber-1); j++)
            {
              int index1, index2;

              index1 = (j*rSequence[i])%(primeNumber-1);
              index2 = baseSequence[index1];
              if (!j)
                savedIndex = index2;
              permutedMatrix[i][index2] = originalMatrix[i][j];
            }
          permutedMatrix[i][0] = originalMatrix[i][(primeNumber-1)];
          permutedMatrix[i][primeNumber] = originalMatrix[i][primeNumber];

          if ((i == (rows-1)) && (frameLength == (rows*columns)))
            {
              int tmp;
              tmp = permutedMatrix[i][savedIndex];
              permutedMatrix[i][savedIndex] = permutedMatrix[i][primeNumber];
              permutedMatrix[i][primeNumber] = tmp;
            }
        }

      else if (columns == (primeNumber-1))
        {
          for(j = 0; j < (primeNumber-1); j++)
            {
              int index1, index2;

              index1 = (j*rSequence[i])%(primeNumber-1);
              index2 = baseSequence[index1] - 1;
              permutedMatrix[i][index2] = originalMatrix[i][j];
            }
        }
    }


  /* Perform the inter-row permutation.*/
  for (i = 0; i < rows; i++)
    {
      for (j = 0; j < columns; j++)
        {
          originalMatrix[i][j] = permutedMatrix[interRowPermutation[i]][j];
        }
    }

  /* Read the permutted matrix one column at a time from top to bottom, and
   * store the interleaver mapping configuration in the corresponding array.*/
  int frameIndex = 0;
  for (i = 0; i < columns; i++)
```

```
    {
      for(j = 0; j < rows; j++)
        {
          if (originalMatrix[j][i])
            pInterleaverMap[frameIndex++] = originalMatrix[j][i] - 1;
        }
    }

  /* Return all dynamically allocated memory back to the heap.*/
  delete [] baseSequence;
  delete [] qSequence;
  delete [] rSequence;

  for (i = 0; i < rows; i++)
    {
      delete [] permutedMatrix[i];
      delete [] originalMatrix[i];
    }

  delete [] permutedMatrix;
  delete [] originalMatrix;

}
```

A.3 Turbo Encoder/Decoder Class

Listing A.4: Turbo Encoder/Decoder Class Definition

```
/**
 * turboCodec.cpp
 *
 * Copyright (c) 2004,2005 Marco Castellon
 * University of Alberta, Edmonton, CANADA
 * All rights reserved.
 *
 * Description:
 * Definition of the turbo encoder/decoder class.
 *
 */
#ifndef TURBOCODEC_H
#define TURBOCODEC_H

#include "fixedpoint.h"

#define NUM_OF_STATES 8
#define FRACTION_LEN  2

#define WORD_LEN_8BITS

#if defined(WORD_LEN_6BITS)
  #warning Total word length of fixed-point type will be 6-bits.
  #define WORD_LEN      6
  typedef sfixed6 sc_fixed;       // Define new types for fixed-point variables, and
  typedef sfixed5 sc_fixedml;     // use the same notation as in SystemC.
#elif defined(WORD_LEN_7BITS)
  #warning Total word length of fixed-point type will be 7-bits.
  #define WORD_LEN      7
  typedef sfixed7 sc_fixed;       // Define new types for fixed-point variables, and
  typedef sfixed6 sc_fixedml;     // use the same notation as in SystemC.
#elif defined(WORD_LEN_8BITS)
  #define WORD_LEN      8
  typedef sfixed8 sc_fixed;       // Define new types for fixed-point variables, and
  typedef sfixed7 sc_fixedml;     // use the same notation as in SystemC.
#elif defined(WORD_LEN_12BITS)
  #warning Total word length of fixed-point type will be 12-bits.
  #define WORD_LEN      12
  typedef sfixed12 sc_fixed;       // Define new types for fixed-point variables, and
  typedef sfixed11 sc_fixedml;     // use the same notation as in SystemC.
#elif defined(WORD_LEN_16BITS)
  #warning Total word length of fixed-point type will be 16-bits.
  #define WORD_LEN      16
  typedef sfixed16 sc_fixed;       // Define new types for fixed-point variables, and
  typedef sfixed15 sc_fixedml;     // use the same notation as in SystemC.
#endif

/**
 * Declaration of Turbo Encoder/Decoder class.
 */
class TurboCodec
{

private:
  bool puncturedCode;

  short constraintLength;
```

```
    short feedForwPolyOctal;
    short feedBackPolyOctal;
    short codeWordLength;

    int frameLength;
    int *pInterleaverMap;

    sc_fixed clm_T;
    sc_fixed clm_C;

    void umtsInterleaverMap(void);

    const FixPoint maxStar(const FixPoint &operandX, const FixPoint &operandY);

public:

    // Declare the default constructor.
    TurboCodec();

    // Optional constructor.
    TurboCodec(short Kc, short g0D, short g1D, int frameSize);

    // Declare the destructor.
    ~TurboCodec();

    // Declaration of mutator methods.
    void setConstraintLength(short Kc);
    void setPolynomials(short g0D, short g1D);
    void setFrameLength(int lengthOfFrame);
    void setPuncturing( char puncture);
    void setCodeWordLength(void);

    // Declaration of accessor methods.
    short getCodeWordLength(void) const;
    bool isPunctured(void) const;

    // Declaration of methods related to the Interleaver/DeInterleaver.
    void generateInterleaverMap(void);
    void Interleaver(const char *pXk, char *pXk_I);
    void Interleaver(const FixPoint pXk[], FixPoint pXk_I[]);
    void DeInterleaver(const FixPoint pXk_I[], FixPoint pXk[]);

    // Declaration of method(s) related to the Encoder.
    void EncodeWithTail(const char *dataBit, char *parityBit, char tailBits[][2]);

    // Declaration of methods related to the Decoder.
    void setConstantLogMapParams(double Lc);
    void decodeTail(sc_fixed trellisTail[][2], FixPoint betakInit[]);
    void decodeFrame(FixPoint systematicBits[], FixPoint parityBits[],
                     FixPoint betakInit[], FixPoint lambda_Au[]);

};

#endif // TURBOCODEC_H
```

Listing A.5: Turbo Encoder/Decoder Class Implementation

```
/**
 * turboCodec.cpp
 *
 * Copyright (c) 2004, 2005 Marco Castellon
 * University of Alberta, Edmonton, CANADA
 * All rights reserved.
 *
 * This software may be used for non-profit university research if
 * given the author's expressed permission. An executed license
 * agreement with the author is required for all other uses of this
 * software. Redistribution of this software is not permitted without
 * the author's expressed permission. This copyright notice must
 * remain intact. Derivative works may contain additional notices.
 *
 * THIS SOFTWARE IS PROVIDED BY THE AUTHOR "AS IS" AND COMES WITH
 * NO WARRANTY.
 *
 * Description:
 * Implementation of the Turbo Encoder/Decoder Class.
 * The turbo encoder is generic in that the constituent encoder
 * takes in as parameters the generator polynomials. The frame length
 * is also an input parameter and is only limited by the restrictions
 * imposed by the UMTS specification.
 *
 * The constituent SISO decoder is specific to the UMTS turbo code,
 * future work may involve the implementation of a generic SISO decoder.
 *
 */

//include header files
#include <cstdlib>
#include "turboCodec.h"


// Declare a few constants for the limits of the Decoder.
const double CLM_CORRECTOR = 0.5;
const double CLM_THRESHOLD = 1.5;

#if defined(WORD_LEN_6BITS)
const double N_INFINITY = -8.0;
#elif defined(WORD_LEN_7BITS)
const double N_INFINITY = -16.0;
#elif defined(WORD_LEN_8BITS)
const double N_INFINITY = -32.0;
#elif defined(WORD_LEN_12BITS)
const double N_INFINITY = -512.0;
#elif defined(WORD_LEN_16BITS)
const double N_INFINITY = -8192.0;
#endif


/**
 * Define the value of negative infinity in terms of the
 * range available for the FixPointed-point implementation.
 */
const FixPoint NEG_INFINITY(N_INFINITY, WORD_LEN, FRACTION_LEN);
const FixPoint FIX_ZERO(0.0, WORD_LEN, FRACTION_LEN);


// Definition of default constructor.
TurboCodec::TurboCodec()
{
  constraintLength = 4;
  feedBackPolyOctal = 013;
  feedForwPolyOctal = 015;
  puncturedCode = false;
  pInterleaverMap = NULL;

  setCodeWordLength();
  frameLength = 1024;

}

// Definition of constructor with given parameters.
TurboCodec::TurboCodec(short Kc, short g0D, short g1D, int frameSize)
{
  setConstraintLength(Kc);
  setPolynomials(g0D, g1D);;
  puncturedCode = false;
  pInterleaverMap = NULL;

  setCodeWordLength();
  setFrameLength(frameSize);

}

TurboCodec::~TurboCodec()
{
  // Return back to heap memory allocated for Interleaver Map pointer.

  if(pInterleaverMap != NULL)
    {
      delete [] pInterleaverMap;
      pInterleaverMap = NULL;
    }
}

void TurboCodec::setConstraintLength(short Kc)
{
  constraintLength = Kc;
}

void TurboCodec::setPolynomials(short g0D, short g1D)
{
  feedBackPolyOctal = g0D;
  feedForwPolyOctal = g1D;
}

void TurboCodec::setFrameLength(int lengthOfFrame)
{
  frameLength = lengthOfFrame;
}

void TurboCodec::setPuncturing(char puncture)
{
  if(puncture == 'Y' || puncture == 'y')
    puncturedCode = true;
  else
    puncturedCode = false;
}

void TurboCodec::setCodeWordLength(void)
{
  if (isPunctured())
    codeWordLength = 2;
  else
    codeWordLength = 3;
}

short TurboCodec::getCodeWordLength(void) const
{
  return (codeWordLength);
}

bool TurboCodec::isPunctured(void) const
{
  return (puncturedCode);
}


/**
 * RECURSIVE SYSTEMATIC CONVOLUTIONAL ENCODER                          *
 *                                                                     *
 * Purpose: To encode a data sequence using a recursive systematic     *
 *          convolutional encoder. Also terminates the trellis and     *
 *          generates tails bits to return encoder to zero state.      *
 **/
void TurboCodec::EncodeWithTail(const char *dataBit, char *parityBit, char tailBits[][2])
{
  int membits;                /* Number of memory elements of the encoder = ConstraintLength - 1 */
  int i, j;
  int g0D, g1D;

  unsigned short shiftRegister;   /* the encoder shift register (enough for 16-bits) */
  short sumNode;                  /* Output of the sum node using GF(2) addition.*/

  short *g0;                  // Generator polynomial g0 (feedback).
  short *g1;                  // Generator polynomial g1 (feedforward).

  membits = constraintLength - 1;

  // Allocate memory for the generator polynomials and shift register.
  g0 = new short[constraintLength];
  g1 = new short[constraintLength];


  // Configure the polynomials.
  for (i = 0, g0D=feedBackPolyOctal, g1D=feedForwPolyOctal; i < constraintLength; i++)
    {
      g0[i] = g0D & 1;
      g1[i] = g1D & 1;
      g0D >>= 1;
      g1D >>= 1;

    }

  // Make sure the contents of the shift register are cleared before beginning.
  shiftRegister = 0;

  /* Now start the encoding process for the data bits.*/
  /* compute the upper and lower mod-two adder outputs, one bit at a time */
  for (i = 0; i < frameLength; i++)
    {
      shiftRegister |= (dataBit[i] << membits);
      sumNode = 0;

      // Division by the feed-back polynomial.
      for (j = 0; j < constraintLength; j++)
        {
          if (g0[j])
            sumNode ^= ((shiftRegister & (1 << j)) >> j);
        }

      shiftRegister &= ~(1 << membits);         // First clear the MSB of the shift register.
      shiftRegister |= (sumNode << membits);    // Assign a new value to the MSB.
      sumNode = 0;

      // Multiplication by the feed-forward polynomial.
      for (j = (constraintLength-1); j >=0; j--)
        {
          if (g1[j])
            sumNode ^= ((shiftRegister & (1 << j)) >> j);
        }

      /* write the parity output bit */
      parityBit[i] = (char)sumNode;

      /* Shift the contents of the shift register to the right by one.*/
      shiftRegister >>= 1;
    }

  //cout << "shift register status before tail: " << shiftRegister << endl;
  /* Now generate the tail bits to return the encoder to zero state.*/
  for (i = 0; i < membits; i++)
    {
      sumNode = 0;
      for (j = 0; j < (constraintLength-1); j++)
        {
          if (g0[j])
            sumNode ^= ((shiftRegister & (1 << j)) >> j);
        }

      shiftRegister |= (sumNode << membits);
      tailBits[i][0] = (char)sumNode;

      sumNode = 0;

      // Division by the feed-back polynomial.
      for (j = 0; j < constraintLength; j++)
        {
          if (g0[j])
            sumNode ^= ((shiftRegister & (1 << j)) >> j);
        }

      shiftRegister &= ~(1 << membits);
      shiftRegister |= (sumNode << membits);
      sumNode = 0;

      // Multiplication by the feed-forward polynomial.
      for (j = (constraintLength-1); j >= 0; j--)
        {
          if (g1[j])
            sumNode ^= ((shiftRegister & (1 << j)) >> j);
        }

      /* write the parity output bit for the corresponding tail bit */
      tailBits[i][1] = (char)sumNode;

      /* Shift the contents of the shift register to the right by one.*/
      shiftRegister >>= 1;

    }
  //cout << "shift register status after tail: " << shiftRegister << endl;
  delete [] g0;
  delete [] g1;

} // end EncoderWithTail()


/****************intl*************
Interleaving
*****************intl*************/
void TurboCodec::Interleaver(const char *pXk, char *pXk_I)
{

  for(int n = 0, k ; n < frameLength; n++)
```

```
        {
          k = pInterleaverMap[n];      //address mapping
          pXk_I[n] = pXk[k];          //value mapping

        }
}//end Interleaver

/**
 * Overloaded definition of Interleaving method.
 * Since it only needs to support short and FXPT_TYPE types, no need
 * for templated version.
 */
void TurboCodec::Interleaver(const FixPoint pXk[], FixPoint pXk_I[])
{

  for(int n = 0, k ; n < frameLength; n++)
    {
      k = pInterleaverMap[n];      //address mapping
      pXk_I[n] = pXk[k];          //value mapping

    }
}//end Interleaving

/**
 * Definition of Deinterleaving method.
 * Only supports FXPT_TYPE parameters at the moment.
 */
void TurboCodec::DeInterleaver(const FixPoint pXk_I[], FixPoint pXk[])
{
  for(int n = 0, k ; n < frameLength; n++)
    {
      k = pInterleaverMap[n];      //address mapping
      pXk[k] = pXk_I[n];          //value mapping

    }
}


/**
 * generateInterleaverMap ()
 */
void TurboCodec::generateInterleaverMap(void)
{
  // Deallocate memory in order to avoid memory leaks.
  if(pInterleaverMap != NULL)
    {
      delete [] pInterleaverMap;
      pInterleaverMap = NULL;
    }

  pInterleaverMap = new int[frameLength];

  // Create the interleaver map based on the frameLength.
  umtsInterleaverMap();
}

/**
 * decodeTail()
 */
void TurboCodec::decodeTail(sc_fixed trellisTail[][2], FixPoint betakInit[])
{
  register int i;

  sc_fixed gammak_01;
  sc_fixed gammak_10;
  sc_fixed gammak_11;
  sc_fixed maxBetak;

  sc_fixed betakp1[NUM_OF_STATES];
  sc_fixed betak[NUM_OF_STATES];

  betakp1[0] = FIX_ZERO;
  for (i = 1; i < NUM_OF_STATES; i++)
    {
      betakp1[i] = NEG_INFINITY;
    }

  /* DECODE THE FIRST TAIL SECTION */

  gammak_11 = trellisTail[2][0] + trellisTail[2][1]; // branch metric required.

  // Calculate beta(k) values.
  betak[0] = betakp1[0];
  betak[1] = betakp1[0] + gammak_11;

  // Update beta(k+1) values.
  betakp1[0] = betak[0];
  betakp1[1] = betak[1];
```

```
   /* DECODE THE SECOND TAIL SECTION */

   // Determine the value of the branch metrics (gamma).
   gammak_10 = trellisTail[1][0];
   gammak_01 = trellisTail[1][1];
   gammak_11 = trellisTail[1][0] + trellisTail[1][1];

   // Calculate beta(k) values.
   betak[0] = betakp1[0];
   betak[1] = betakp1[0] + gammak_11;
   betak[2] = betakp1[1] + gammak_10;
   betak[3] = betakp1[1] + gammak_01;

   // Update beta(k+1) values.
   betakp1[0] = betak[0];
   betakp1[1] = betak[1];
   betakp1[2] = betak[2];
   betakp1[3] = betak[3];

   /* DECODE THE THIRD AND LAST TAIL SECTION */

   // Determine the value of the branch metrics (gamma).
   gammak_10 = trellisTail[0][0];
   gammak_01 = trellisTail[0][1];
   gammak_11 = trellisTail[0][0] + trellisTail[0][1];

   // Calculate beta(k) values.
   betak[0] = betakp1[0];
   betak[1] = betakp1[0] + gammak_11;
   betak[2] = betakp1[1] + gammak_10;
   betak[3] = betakp1[1] + gammak_01;
   betak[4] = betakp1[2] + gammak_01;
   betak[5] = betakp1[2] + gammak_10;
   betak[6] = betakp1[3] + gammak_11;
   betak[7] = betakp1[3];

   // Find the maximum of betak state probabilities for the last tail interval.
   // It will be used for normalize the values.
   maxBetak = maximum(betak[0], betak[1]);
   maxBetak = maximum(maxBetak, betak[2]);
   maxBetak = maximum(maxBetak, betak[3]);
   maxBetak = maximum(maxBetak, betak[4]);
   maxBetak = maximum(maxBetak, betak[5]);
   maxBetak = maximum(maxBetak, betak[6]);
   maxBetak = maximum(maxBetak, betak[7]);

   for (i = 0; i < NUM_OF_STATES; i++)
     {
       betakInit[i] = betak[i] - maxBetak;
     }

}

void TurboCodec::decodeFrame(FixPoint systematicBits[], FixPoint parityBits[],
                              FixPoint betakInit[], FixPoint lambda_Au[])
{
  register int i;
  int interval, cw_index;

  sc_fixed gammak_01;
  sc_fixed gammak_10;
  sc_fixed gammak_11;

  sc_fixed alphak[NUM_OF_STATES];

  sc_fixed alphakm1[NUM_OF_STATES];

  sc_fixed betak[NUM_OF_STATES];

  sc_fixed betakp1[NUM_OF_STATES];

  sc_fixed alphak_Pre[2];
  sc_fixed betak_Pre[2];

  sc_fixed lambda_Xk1[NUM_OF_STATES];
  sc_fixed lambda_Xk0[NUM_OF_STATES];

  sc_fixed maxLambda_Xk0;
  sc_fixed maxLambda_Xk1;

  sc_fixed *alphaTime[NUM_OF_STATES];

  sc_fixed maxAlphak;
  sc_fixed maxBetak;


  // Dynamically allocate memory for the values to be stored during the forward recursion.
  // Initialize state metrics.
  for (i = 0; i < NUM_OF_STATES; i++)
```

```
    {
      alphaTime[i] = new sc_fixed[(frameLength+1)];
      alphakm1[i] = NEG_INFINITY;
      betak[i] = NEG_INFINITY;
    }
  alphakm1[0] = FIX_ZERO;

  /**
   * Beginning of forward recursion.
   */

  // Initialize the values alphaTime that correspond to time interval 0.
  alphaTime[0][0] = FIX_ZERO;
  for (i = 1; i < NUM_OF_STATES; i++)
    {
      alphaTime[i][0] = NEG_INFINITY;
    }


  /**
   * Computing alpha values within the body of the frame.
   */
  for (interval = 1, cw_index = 0; interval <= frameLength; interval++, cw_index++)
    {
      // Due to the relatively small number of states, the inner loop
      // will be un-rolled.

      // Calculate the branch metric values used for this interval.
      /* IMPORTANT NOTE: gammak_00 = 0 ALWAYS; THEREFORE, IT IS SIMPLY OMITTED FROM THE CALCULATION */
      gammak_01 = parityBits[cw_index];              // encoded branch = 01
      gammak_10 = systematicBits[cw_index];          // encoded branch = 10
      gammak_11 = gammak_10 + gammak_01;             // encoded branch = 11

      // Calculate the alpha state metric for each of the possible transitions.
      // Use the MAX* operator to determine the alpha metric.
      // Determine the max(Alpha) as each state is visited.

        /* State #0 */
      alphak_Pre[0] = alphakm1[0];
      alphak_Pre[1] = alphakm1[1] + gammak_11;
      alphak[0] = maxStar(alphak_Pre[0], alphak_Pre[1]);

      /* State #1 */
      alphak_Pre[0] = alphakm1[2] + gammak_10;
      alphak_Pre[1] = alphakm1[3] + gammak_01;
      alphak[1] = maxStar(alphak_Pre[0], alphak_Pre[1]);

      maxAlphak = maximum(alphak[0], alphak[1]);

      /* State #2 */
      alphak_Pre[0] = alphakm1[4] + gammak_01;
      alphak_Pre[1] = alphakm1[5] + gammak_10;
      alphak[2] = maxStar(alphak_Pre[0], alphak_Pre[1]);

      maxAlphak = maximum(maxAlphak, alphak[2]);

      /* State #3 */
      alphak_Pre[0] = alphakm1[6] + gammak_11;
      alphak_Pre[1] = alphakm1[7];
      alphak[3] = maxStar(alphak_Pre[0], alphak_Pre[1]);

      maxAlphak = maximum(maxAlphak, alphak[3]);

      /* State #4 */
      alphak_Pre[0] = alphakm1[0] + gammak_11;
      alphak_Pre[1] = alphakm1[1];
      alphak[4] = maxStar(alphak_Pre[0], alphak_Pre[1]);

      maxAlphak = maximum(maxAlphak, alphak[4]);

      /* State #5 */
      alphak_Pre[0] = alphakm1[2] + gammak_01;
      alphak_Pre[1] = alphakm1[3] + gammak_10;
      alphak[5] = maxStar(alphak_Pre[0], alphak_Pre[1]);

      maxAlphak = maximum(maxAlphak, alphak[5]);

      /* State #6 */
      alphak_Pre[0] = alphakm1[4] + gammak_10;
      alphak_Pre[1] = alphakm1[5] + gammak_01;
      alphak[6] = maxStar(alphak_Pre[0], alphak_Pre[1]);

      maxAlphak = maximum(maxAlphak, alphak[6]);

      /* State #7 */
      alphak_Pre[0] = alphakm1[6];
      alphak_Pre[1] = alphakm1[7] + gammak_11;
      alphak[7] = maxStar(alphak_Pre[0], alphak_Pre[1]);
      maxAlphak = maximum(maxAlphak, alphak[7]);

      // Assign new values for alphakm1 and normalize if necessary.
      for (i = 0; i < NUM_OF_STATES; i++)
        {
          alphaTime[i][interval] = alphak[i]; // Save alpha values for future use.

          // Apply normalization rule.
          alphakm1[i] = alphak[i] - maxAlphak;
        }

    } // End of outer loop for forward recursion (calculation of alpha values).

  /**
   * Beginning of backward recursion.
   */

  // Initialize the values for betakp1 based on the decoding of the tail bits.
  for (i = 0; i < NUM_OF_STATES; i++)
    {
      betakp1[i] = betakInit[i];
    }

  /**
   * Computing the beta values within the body of the frame.
   */
  for (interval = (frameLength-1); interval >= 0; interval--)
    {

      // Un-roll the inner loop that would normally iterate over the number of states.

      // Calculate the branch metric values used for this interval.
      /* IMPORTANT NOTE: gammak_00 = 0 ALWAYS; THEREFORE, IT IS SIMPLY OMITTED FROM THE CALCULATION */

      gammak_01 = parityBits[interval];              // encoded branch = 01
      gammak_10 = systematicBits[interval];          // encoded branch = 10
      gammak_11 = gammak_10 + gammak_01;             // encoded branch = 11

      // Calculate the beta state metric for each of the possible transitions.
      // Use the MAX* operator to determine the beta metric.
      // Determine the max(Beta) as each state is visited.

      /* State #0 */
      betak_Pre[0] = betakp1[0];
      betak_Pre[1] = betakp1[4] + gammak_11;
      lambda_Xk0[0] = alphaTime[0][interval] + betakp1[0];
      lambda_Xk1[0] = alphaTime[0][interval] + gammak_11 + betakp1[4];

      betak[0] = maxStar(betak_Pre[0], betak_Pre[1]);

      /* State #1 */
      betak_Pre[0] = betakp1[0] + gammak_11;
      betak_Pre[1] = betakp1[4];
      lambda_Xk1[1] = alphaTime[1][interval] + gammak_11 + betakp1[0];
      lambda_Xk0[1] = alphaTime[1][interval] + betakp1[4];

      betak[1] = maxStar(betak_Pre[0], betak_Pre[1]);
      maxBetak = maximum(betak[0], betak[1]);

      /* State #2 */
      betak_Pre[0] = betakp1[1] + gammak_10;
      betak_Pre[1] = betakp1[5] + gammak_01;
      lambda_Xk1[2] = alphaTime[2][interval] + gammak_10 + betakp1[1];
      lambda_Xk0[2] = alphaTime[2][interval] + gammak_01 + betakp1[5];

      betak[2] = maxStar(betak_Pre[0], betak_Pre[1]);
      maxBetak = maximum(maxBetak, betak[2]);

      /* State #3 */
      betak_Pre[0] = betakp1[1] + gammak_01;
      betak_Pre[1] = betakp1[5] + gammak_10;
      lambda_Xk0[3] = alphaTime[3][interval] + gammak_01 + betakp1[1];
      lambda_Xk1[3] = alphaTime[3][interval] + gammak_10 + betakp1[5];

      betak[3] = maxStar(betak_Pre[0], betak_Pre[1]);
      maxBetak = maximum(maxBetak, betak[3]);

      /* State #4 */
      betak_Pre[0] = betakp1[2] + gammak_01;
      betak_Pre[1] = betakp1[6] + gammak_10;
      lambda_Xk0[4] = alphaTime[4][interval] + gammak_01 + betakp1[2];
      lambda_Xk1[4] = alphaTime[4][interval] + gammak_10 + betakp1[6];

      betak[4] = maxStar(betak_Pre[0], betak_Pre[1]);
      maxBetak = maximum(maxBetak, betak[4]);

      /* State #5 */
      betak_Pre[0] = betakp1[2] + gammak_10;
      betak_Pre[1] = betakp1[6] + gammak_01;
```

```
        lambda_Xk1[5] = alphaTime[5][interval] + gammak_10 + betakp1[2];
        lambda_Xk0[5] = alphaTime[5][interval] + gammak_01 + betakp1[6];

        betak[5] = maxStar(betak_Pre[0], betak_Pre[1]);
        maxBetak = maximum(maxBetak, betak[5]);

        /* State #6 */
        betak_Pre[0] = betakp1[3] + gammak_11;
        betak_Pre[1] = betakp1[7];
        lambda_Xk1[6] = alphaTime[6][interval] + gammak_11 + betakp1[3];
        lambda_Xk0[6] = alphaTime[6][interval] + betakp1[7];

        betak[6] = maxStar(betak_Pre[0], betak_Pre[1]);
        maxBetak = maximum(maxBetak, betak[6]);

        /* State #7 */
        betak_Pre[0] = betakp1[3];
        betak_Pre[1] = betakp1[7] + gammak_11;
        lambda_Xk0[7] = alphaTime[7][interval] + betakp1[3];
        lambda_Xk1[7] = alphaTime[7][interval] + gammak_11 + betakp1[7];

        betak[7] = maxStar(betak_Pre[0], betak_Pre[1]);
        maxBetak = maximum(maxBetak, betak[7]);

        // Assign new values for betakp1 and normalize if necessary.
        for (i = 0; i < NUM_OF_STATES; i++)
          {
            betakp1[i] = betak[i] - maxBetak; // Apply Normalization rule.
          }

        // Compute the value of the LLR for this interval of the trellis.
        maxLambda_Xk0 = maxStar (lambda_Xk0[0], lambda_Xk0[1]);
        maxLambda_Xk1 = maxStar (lambda_Xk1[0], lambda_Xk1[1]);

        for (i = 2; i < NUM_OF_STATES; i++)
          {
            maxLambda_Xk0 = maxStar (lambda_Xk0[i], maxLambda_Xk0);
            maxLambda_Xk1 = maxStar (lambda_Xk1[i], maxLambda_Xk1);
          }

        lambda_Au[interval] = maxLambda_Xk1 - maxLambda_Xk0;

      } // End of beta value calculation within body of frame.

    // Deallocate the memory reserved for all the alpha values of the entire frame.
    for (i = 0; i < NUM_OF_STATES; i++)
      delete [] alphaTime[i];
}

/**
 * Definition of maxStar() member method.
 */
const FixPoint TurboCodec::maxStar(const FixPoint &operandX, const FixPoint &operandY)
{
  sc_fixed maxXY;
  sc_fixed difference;
  sc_fixed correctionTerm;
  sc_fixed absXY;

  maxXY = maximum(operandX, operandY);
  difference = operandX - operandY;

  // absolute value of the difference.
  absXY = absolute (difference);

  // Compute the correction term according to Constant-Log-MAP rule.
  if(absXY.get_rawbin() <= clm_T.get_rawbin())
    {
      maxXY += clm_C;
    }

  return (FixPoint(maxXY));
}

void TurboCodec::setConstantLogMapParams(double Lc)
{
  double correctionFactor = 0.0;
  double thresholdLevel = 0.0;

  // Quantize and scale the CLM threshold level appropriately.
  thresholdLevel = CLM_THRESHOLD / Lc;
  clm_T.set(thresholdLevel, FRACTION_LEN);

  // Quantize and scale the CLM correction factor appropriately.
  correctionFactor = CLM_CORRECTOR / Lc;
  clm_C.set(correctionFactor, FRACTION_LEN);
}
```